POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

A HOLISTIC APPROACH TOWARDS FUTURE SELF-TUNING APPLICATIONS IN HOMOGENEOUS AND HETEROGENEOUS ARCHITECTURES

Doctoral Dissertation of:
Emanuele Vitali

Supervisor:
Prof. Gianluca Palermo

Tutor:
Prof. Cristina Silvano

The Chair of the Doctoral Program:
Prof. Barbara Pernici

Year 2021 – Cycle XXXII

Abstract

WITH the beginning of the dark silicon era, application optimization, even with the exploitation of heterogeneity, has become an important topic of research. One methodology to obtain optimized applications for different architectures is application autotuning. Indeed, applications can obtain the same result with different codes. However, different codes have different extra-functional properties, such as execution time or energy consumption which may change across different architectures. To obtain the best, application autotuning techniques have been proposed in literature. It is very difficult for the original application developer to select the best configuration that can enforce the constraints across different machines, with unknown input and varying configurations.

Given this background, I envision future applications not as monolithic code but as a sequence of modules that are capable of autotuning themselves and can exploit platform heterogeneity. This thesis consists of a collection of methodologies that were developed during my Ph.D. which aim at giving the programmers ways to create these self-tuning modules.

I divided my Ph.D. thesis into two sections, the first one is dedicated to general application autotuning techniques, while the second will be focused on a single application, GeoDock, which has been an industrial use case that I used to develop and validate the proposed techniques. In the first half, we will see the benefit that can be introduced by run-time dynamic autotuning focusing on the condition of the machine, constraints given to the application, or characteristics of the input data. In the second half, we will see the developement of GeoDock from a monolithic non tunable ap-

plication to an heterogeneous and tunable one, and we will see how this has dramatically improved its performances (from tens of ligands per second processed on a single node to thousands).

Contents

CHAPTER 1

Introduction

With the end of Dennard scaling and the beginning of the Dark Silicon era [1], power consumption has become the limit of modern systems. Multicore processors have been the first answer to the end of Dennard scaling, however dark silicon limits even this approach. For this reason, in the latest years, heterogeneous architectures have become always more widespread, thanks to their lower cost of FLOPs per watt [2].

This shift of paradigm introduces a change in application development, since writing code while targeting heterogeneity is more difficult. The programmer needs to consider more details when designing the application, such as data movement between processor and co-processor. Moreover, it has become fundamental to consider extra-functional properties (EFP) such as energy efficiency or time-to-solution. In particular energy efficiency, which was a property mainly related to embedded systems, is now considered fundamental in a wider range of contexts up to High-Performance Computing (HPC).

Among all the possibilities, this thesis focuses on two software aspects. The first aspect is Application Parameterization. When writing an application, it is possible to obtain the same result with different EFPs. It is good practice for programmers to expose some implementation parameters whenever they may alter EFP without altering the behavior of the code. Examples of these parameters may be the number of worker threads, or the algorithm used for a specific operation (e.g. the sorting algorithm). Another possible parameter is the hardware used for the execution of a function. These parameters in literature are called *software knobs* if they can be modified only at compile time. If they can be modified at run-time, they are called *dynamic knobs*.

The second aspect is Approximate Computing. In this approach, the objective of an application is to compute a result that is *good enough* for the user and not the exact one [3]. This allows avoiding some computation, which means energy, to obtain the result. This approach is commonly used to expose accuracy-throughput software knobs. It is common in multimedia applications or where is possible to use techniques such as task skipping [4] or loop perforation [5]. It has been shown in literature [6] that this technique can exchange the accuracy of the result for throughput. For this reason, approximate hardware has also been explored [7, 8].

Given all these aspects, optimizing applications across different systems is becoming a complex task. Indeed, the tradeoffs exposed by approximate computing or by the software knobs make applications difficult to set up for the end-user. Moreover, requirements might contain constraints on extra-functional properties, or some input properties can create some optimization opportunities.

In this context, to help developers, the autonomic computing approach has been proposed [9], where the applications are enhanced with a set of *self-** properties, such as *self-healing*, *self-optimization* or *self-protection*. This thesis will focus on the *self-optimization* property. This property aims at enhancing the application by enabling it to find and exploit optimization opportunities given by the system evolution.

1.1 Thesis Motivations

With the rise of heterogeneous platforms, the already difficult task of optimizing an application has become even more difficult. Indeed, the amount of possibilities to tune extra-functional properties has increased, since we also need to consider which component we are going to run the application (or even part of it) on.

In order to obtain the best, application autotuning techniques have been proposed in literature. The importance of autotuning lies in the fact that it is very difficult for the original application developer to select the best configuration that can enforce the constraints across different machines, with unknown input and varying configurations.

In this context, I want to insert the work of my thesis. I envision future application not as monolithic code, but as a sequence of self-tuning modules, capable of adapt themselves in two orthogonal directions. The first is adaptivity to the input, intended as being able to change how the application performs its work according to some characteristics of the input data. The second is adaptivity to the platform, intended as being capable of changing according to varying runtime constraints and exploit, whenever available, the heterogeneity of present and future platforms.

This thesis consists of a collection of methodologies that aim at advancing the state of the art in the field of application autotuning with a focus on heterogeneity.

1.2 Thesis Contributions

The main contribution of this thesis is a collection of techniques to enhance a target application with autotuning capabilities, with a focus on heterogeneous contexts in the second part of the thesis. As these techniques are strongly tailored to the target application, they are not implemented in a single framework. However, the methodology behind them is general and can be easily ported into similar contexts. Furthermore, these methodologies have been developed inside two European Projects, ANTAREX (AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems) and E4C (Exscalate4Cov).

In particular, the contributions are the following:

1. A framework to automatically tune compiler flags or library parameters at function level. The framework exploits different tools (mARGOt [10], COBAYN [11], LARA [12], MilepostGCC [13]) in a joint effort to ease the programmer job and automatically and seamlessly obtain the best possible configuration according to the underlying architecture for every different hotspot kernel in the code.

2. Analysis of applications to find and expose autotuning possibilities in a reactive way. Applications have been made capable to react to changes in the underlying configurations or changing requirements provided by the user.

3. A methodology has been developed to respect a requirement on time to solution of an application that has been previously enriched with autotuning capabilities.

Table 1.1: *Techniques developed in this thesis and application for which they were tailored. Global means that the technique does not have a specific application.*

Technique	Global	GeoDock	PTDR	Object Detection	Multiplication
Function Level Autotuning	Chapter 4				
Reactive Autotuning		Chapter 9		Chapter 7	
Time to Solution		Chapter 9			
Proactive Autotuning		Chapter 12	Chapter 6	Chapter 7	
Heterogeneity Parameters Tuning		Chapter 10			
Hybrid approach		Chapter 11			
Tunable Library					Chapter 5

4. A methodology to proactively autotune applications according to the input data.The objective of this methodology is to increase computation efficiency and thus save energy and time.

5. Porting of an application to the heterogeneous context, using the OpenACC language, and optimization of its computation organization on the GPU.

6. Optimization of the distribution of the computation across CPU and GPU, mapping the kernels on the most suitable hardware according to the characteristics of the kernel itself.

7. Creation of a parametric library for the synthesis of long unsigned multipliers on FPGA. The library allows the programmers to create hardware accelerators to perform this important operation with a focus on parametrization and the offering of several trade-offs in performance and cost.

As previously mentioned, these methodologies have been studied in different use cases and application. Table 1.1 reports for all the techniques previously described on which application they have been ported and in which chapter this will be discussed.

In the remainder of this thesis, I will write using the first-person plural to acknowledge the support from my advisor and colleagues. However, I take responsibility for all the decisions and choices described in this thesis, since I was the main investigator. The only exceptions are the works described, Chapter 4 and Chapter 9, which are a joint effort with other colleagues.

1.3 Thesis Outline

This thesis is organized into three main sections. The first section focuses on the state of the art in application autotuning. Firstly we describe the research done up to now, then a background section focuses on the mARGOt autotuner, a tool that I contributed to develop.

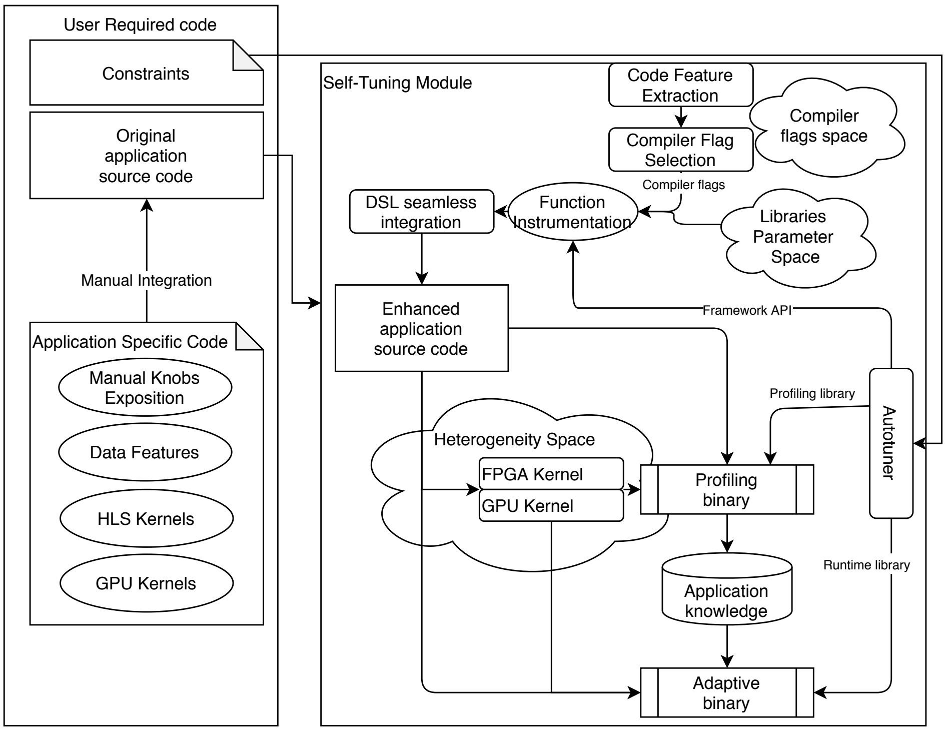


Figure 1.1: *Global architecture of the proposed framework.*

The second section is divided into 4 chapters, and focuses on different methodologies for autotuning applications, at compiler, library or application level. The third section focuses on a single application, a molecular docking tool, that has been used as a use case several times during my Ph.D. to develop and validate new techniques. In last section we will also describe its evolution and we will refer to some experiments that were made possible thanks to the introduced innovation.

Finally, Chapter 13 summarizes the proposed approach and states recommendations for future works.

The work of this thesis has not been implemented in a single framework but is a collection of methodologies. However, they aim at suggesting a global collective paradigm for writing future applications.

Figure 1.1 shows the global vision of a self-tuning module. As we can see, a lot of different components are added to the original source code of the application to obtain the final adaptive binary. Not all of these components are present together in any of the analyzed scenarios, however, there has always been a global view that has driven the work of this thesis.

We can clusterize the components into three different macro categories:

- Code-related tunable parameters.
- Data features.
- Heterogeneous components.

The first category comprehends all of those characteristics such as compiler flags and library-related parameters, whose optimality may be dependent on the environment of the execution of the code. The autotuning of these parameter has been investigated in Chapter 4 and Chapter 9.

In Chapter 4 we propose a methodology to seamlessly and automatically select, at function level, the optimal configuration of compiler flags and OpenMP parameters. In Chapter 9 instead, we study a particular application to find autotuning opportunities, and we propose a technique to set and enforce a time to solution on the application runtime.

The second category is related to the application data. Indeed, some applications may require different behavior according to different inputs, and the optimality of the application can change (there is no one-fits-all solution). We investigated this perspective mainly in Chapter 6 and Chapter 7.

In Chapter 6 we studied a Probabilistic Time-Dependent Routing algorithm for a navigation application and noticed that some characteristics of the input, which we called Data Features, could be used to tune the MonteCarlo simulation to improve the computation efficiency. In Chapter 7 we studied different object detection networks and demonstrated that no solution is optimal for all the images. Even if we were unable to build a predictor, we believe that this work shows the potential of the autotuning paradigm in this field.

The third category is related to the exploitation of heterogeneity since we want our application to be able to use all the computing resources available on the machine where it is running. This approach has been studied on the GPU with the evolution of the geometric docking application (Chapter 10, Chapter 11 and Chapter 12) and on FPGA with the development of a parametric library for unsigned long multiplication (Chapter 5).

In particular, in Chapter 10 we followed a traditional approach for GPU kernel development using the OpenACC language followed by a parameter space exploration and autotuning. In Chapter 11 we further optimized the previous work by considering hardware characteristics when selecting where to run the different kernels that compose the application. Finally, in Chapter 12, we optimized the application for a novel GPU and we applied the Data Feature approach in a heterogeneous context, showing that it can introduce benefits.

In Chapter 5 we developed a library to create, exploiting High Level Synthesis, hardware accelerators for the multiplication of large unsigned integers. The library is heavily parametrized and can create accelerators that cover several orders of magnitude in terms of performance and resource utilization, according to the constraints of the programmers.

The content of most of the chapters of this thesis has already been published in international conferences or journals. The reuse of the images from those sources has been highlighted under each caption.

CHAPTER 2

Previous work

The main focus of this thesis is to develop a general approach for application autotuning in a heterogeneous system. The approach will be proposed with several case studies, where it has been tailored to specific applications.

This chapter provides at first an introduction to the related research field, and it defines a common terminology in the state-of-the-art. Then it describes the mARGOt framework, an autotuning framework that I contributed to develop to support the analysis carried out in this thesis.

2.1 Background and definitions

The methodology proposed by this thesis belongs to the autonomic computing research topic. In this field, computing systems can be called autonomic if they are able, thanks to a set of elements, to manage themselves without a human in the loop. In the original work [9], four aspects of self-management have been identified.

- *Self-Configuration*: the system shall be able to configure automatically, according to high-level policies. When a new component is integrated, it has to seamlessly integrate into the system. An example can be found in [14].

- *Self-Optimization*: the system will have to seek to improve their performances. They have to continually monitor, experiment and tune their parameters.
- *Self-Healing*: the system has to detect and repair localized software and hardware problems, as proposed in [15].
- *Self-Protection*: the system must defend itself from attacks, as proposed in [16].

In this work, we will focus on *Self-Optimization* property. Previous surveys [17–19] can give a more detailed view on the autonomic computing field for the other *self-** properties.

In the autonomic computing field, a system is composed of both hardware and software components. Therefore, in literature, several approaches have been proposed to optimize the efficiency of both. In particular, we can divide these approaches into two orthogonal categories:

- **Resource Managers:** in this category fall all the approaches where the *Self-Optimization* property is obtained through resource management or allocation. Usually, there is a task at system level that is in charge of distributing the resources across the different applications. This approach is quite wide-spread and can be found in data centers [20, 21], in grid computing [22], in multicores [23–25], in embedded contexts [26, 27] or in heterogeneous systems [28].
- **Application Autotuners**: all the approaches where the *Self-Optimization* property is obtained at software level by the application itself fall in this category. Here the application can manage some configuration parameters to reach the end-user requirement. In this thesis, we will focus on this approach.

Before going in-depth with the literature in the autotuning field, we need to clarify the definition of some key concepts. The first important keyword is the term application. In this work, with application we consider a subset of all the possible software that may run on the system. In particular, we will consider only the applications that perform an elaboration without any human interaction, such as a molecular docking application or a navigation system. Another key concept is the definition of *metrics*. We call metric any measurable property of the application that can be targeted by an optimization problem, i.e. what in literature is called Extra Functional Property (EFP) or Non Functional Property, such as energy consumption, time to solution, or quality of the result that the end-user can be interested in minimizing or maximizing. Examples of metrics that we will see in this thesis are the minimization of energy consumption, in the navigation system application, while respecting user service level agreement. Or the time to solution of a batch molecular docking application while maximizing the output quality. As we already mentioned, many applications expose some tunable parameters, called *Software Knobs*, that can be modified to change the EFPs of the application according to the end-user requirements. Finally, there is the concept of *Input Features*. It is possible in some circumstances that some characteristics of the input (such as its size) can help the process of self-optimization. This happens if some correlation between the input and the metric can be found.

2.2 Autotuning

In this section, we will introduce and classify all those techniques proposed in the literature that aims at giving the *Self-Optimization* property.

A first classification of the autotuning techniques can be done by clustering them according to when autotuning happens, which means at deploy time or runtime. We define this classification as "Autotuning Time Classification" A second classification can be done according to the invasiveness of the autotuning technique in the original application. We define this classification as "Autotuning Integration Classification". In this section, we will see the difference between the two taxonomies and we will explore the state of the art following the second classification strategy

2.2.1 Autotuning Time Strategies

We can distinguish different autotuning approaches according to "when" the autotuning happens during the lifetime of the application. On one hand, the autotuning is performed during the software installation on the platform: variants are generated statically, they are tested (all or through a model-driven approach) and the best one is selected. This approach is also called static autotuning. On the other hand, the variations are generated at runtime giving more flexibility (at the cost of greater overheads). This approach is also known as dynamic autotuning. Between these two extreme solutions, there is a compromise in the middle: the variants can be created at deploy or design time, but selected at runtime.

Figure 2.2 shows the difference between these three approaches. the left part is common to all of them and shows how the original code can be enriched before applying autotuning strategies. The tuning parameters can be manually inserted into or exposed from the application, or they can be

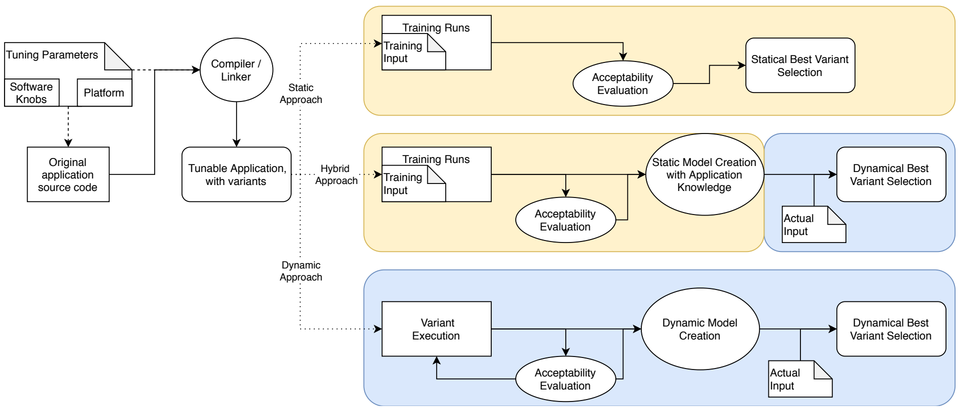


Figure 2.1: *Overview of the Autotuning Time strategies.*

deduced automatically by the compiler/linker. More details on this are provided by the second classification in Subsection 2.2.2. Once this operation is done, the tunable application with variants is ready for the autotuning process. In this image, the yellow background represents operations done at deploy time, while the blue background represents operations performed at runtime. The upper rectangle represents the static process: the variants are tested with some training input, their validity is checked and the most performant variant is selected to be deployed. The second rectangle represents the hybrid process: here the training phase is used to build a model, that will be inserted in the application to drive the variant selection at runtime, according to some input that can be features of the data or the status of the platform. We can notice that the rectangle is split in two: the first section is indeed done during the application deployment, while the final choice happens at runtime. In the last rectangle, all the autotuning happens at runtime. It is still required to create a model, and it is done by running the variants and measuring them. This is done by dividing the runtime of the application into two phases: the *training phase* and the *exploitation phase*. In the *training phase* the application tests the possible variants, with an exhaustive search or a model-driven approach. In the *exploitation phase* it uses the model created during the previous phase to select the best variant according to some input, which can be related to the application input data or the system status, or both of them.

2.2.2 Autotuning Integration Strategies

The other way to classify autotuning considers the invasiveness of the autotuning integration inside the original application. Figure 2.2 shows the

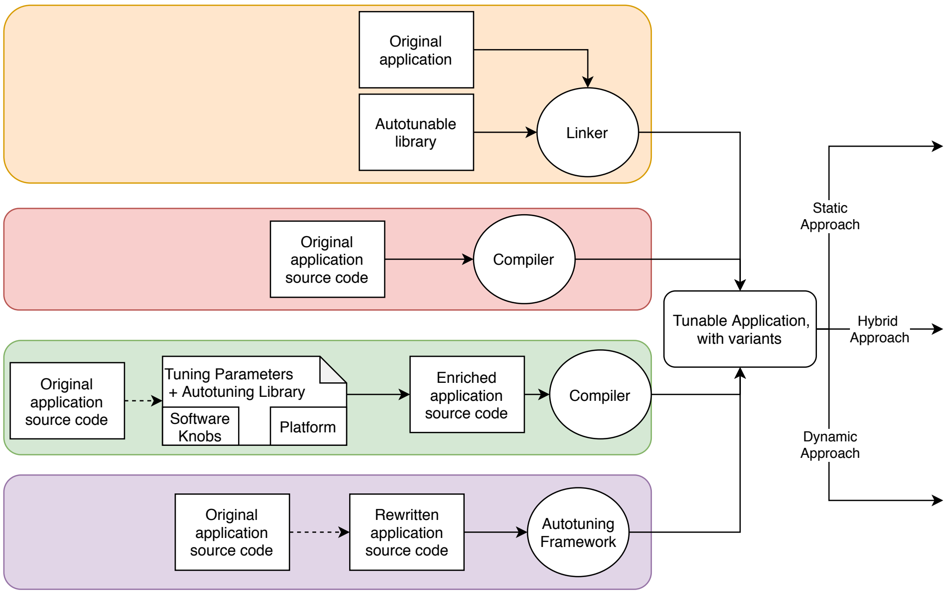

Original application
Autotunable library
Linker
Original application source code
Compiler
Original application source code
Tuning Parameters + Autotuning Library
Software Knobs
Platform
Enriched application source code
Compiler
Original application source code
Rewritten application source code
Autotuning Framework
Tunable Application, with variants
Static Approach
Hybrid Approach
Dynamic Approach

Figure 2.2: *Overview of the Autotuning Integration strategies.*

four levels of integration, from the less invasive to the most invasive one. Dotted lines represent human interventions required.

1. Library-level (orange box). These techniques consist in the creation of an autotuning library that has to be integrated into the application through API [29]. In this way, the application results agnostic to autotuning, which is performed only in the library [30, 31].

2. Compiler-level (red box). In these techniques, the compiler is in charge of autotuning the application. The programmer is not directly involved in the autotuning process [32, 33].

3. Application-level (green box). Here the programmer is directly involved. He is in charge of providing the software knobs or some variant to the autotuner (which is usually integrated as a library) [34].

4. Framework-level (purple box). Here the autotuning is performed by the framework, which requires a strong interaction of the programmer. It may be because it has to use a particular language [35, 36] , or wrap the application [37], or provide a function to check the Quality of Service (QoS) [38]. We decided to insert in this group also Domain Specific (DS) approaches since they usually require (re)writing the application in a specific language, or when they are easily integrated,

that is done under human supervision (e.g. [39]) the check for result accuracy acceptability.

Library level techniques

The first technique for autotuning an application that has been proposed in literature is the library approach: The idea is to isolate compute-intensive kernels behind a library Application Programming Interface (API) and to optimize the implementations for the underlying architectures. In this way, the application developer is released from the optimization task. This approach is domain-specific since the libraries exploit domain knowledge to perform aggressive optimizations. An example of this approach is the BLAS API (Basic Linear Algebra Subprograms, [29]). The API defines a set of primitives for linear algebra, that has become a de-facto standard for dense linear algebra applications. For example, the ATLAS [30] and SPIRAL [31] libraries exploit the BLAS interface.

The implementation of these two libraries employs two different approaches. ATLAS employs the concept of "automated empirical optimization of software" (AEOS). It consists of a collection of parametric or optimized routines to perform the same operation. At compile time, ATLAS tests and measures their performance, then it selects the fastest one to use at runtime. SPIRAL uses a domain-specific language (DSL) to write the routines. The framework then uses this language to generate optimized code for the library according to the underlying architecture. In both cases, targeting a restricted number of functionalities enables the autotuning libraries to explore a vast design space. They exploit heuristics to prune the space and find the optimal implementation.

Other examples of this approach can be found in different domains, such as sparse matrix [40], Fast Fourier transforms [41] or Stencil computation [42,43].

Finally, an interesting solution has been proposed in [44], where the library is in charge of optimizing memory, communication, and parallelization layout of an application targeting an HPC cluster. This allows the domain expert developer to focus only on creating the optimal algorithm without having to consider how the actual computations are organized.

Compiler level techniques

A second category comprehends all those tools that can insert autotuning technique at compile-time, without requiring user intervention. These techniques are more general and are not constrained by the domain of applicability. Usually, compiler optimizations have been designed with a do-not-harm philosophy. This means that they are not performed in some cases where they can slow down common workloads. This approach however eliminates a lot of possibilities. Indeed, some optimizations are architecture-dependent, and thanks to autotuning, the compiler can explore more aggressive optimization techniques that tailor the application to the underlying platform.

Examples of these techniques are insertion of parallelization with SIMD or GPU-kernel creations or loop tiling, unrolling, permutation, and so on. These techniques can be found in tools like ADAPT [32]. Here code is enriched with automatic parallelization, a monitoring system, and a runtime selection of the most performing variant of the code. This result can be obtained thanks to the online compilation of the different variants, where the parameters are selected and tuned. Others rely on polyhedral transformation techniques to obtain the variants. For example in [33] the compiler generates multiple candidates through a model based on polyhedral techniques, then tests the variants and selects the best one. A similar approach has been proposed in [45], where the authors propose a compiler that can generate, thanks to polyhedral models, parallel code for heterogeneous platforms. The compiler manages not only the kernel generation but also all the required data movement and the load balancing between the heterogeneous compute units.

Another interesting approach has been proposed in [46], where a source to source compiler introduces some approximation by modifying the generated assembly code: it removes or duplicates or moves some instructions to reduce the energy consumption of the application. It validates the generated code with statistical tests and accepts the variants only if the test gives an error lower than a given constraint.

In [47] the authors propose an autotuning framework for the Insieme Compiler, which is able to automatically analyze the source code, identify regions of interests and create several variants that can be selected at runtime. In [48] they further optimize this approach by enabling multi-region autotuning for parallel applications. In this follow-up work, the compiler can detect different regions of the application and autotune several parameters (such as number of OpenMP threads, loop tiling, and so on), and it evaluates the interferences of changing these parameters across different regions. This enables optimizations that are not possible when considering each kernel individually.

In [49] the authors suggest the use of a Deep Neural Network to autotune the code. They create a framework that is able to rewrite OpenCL source code in a way that produces a meaningful feature vector for a DNN, which is trained to select the optimal resource to use to run that code (CPU or GPU) or other feature such as thread coarsening.

Finally, [50] propose DDOT, an autotuner that introduces a data-driven approach for compiler and runtime parameters. It exploits existing knowledge collected through experiments on different application to suggest the optimal value for these parameters for every requesting application. DDOT is able to provide the optimal values for the parameters quickly and with high accuracy thanks to its utilization of collaborative filtering. Indeed, it is able to find a utilize similarities to previously optimized applications.

Application level techniques

The third category of autotuning techniques is the one strictly related to an application. As we already mentioned, some applications expose some software knobs that can be used to change their behavior. However, this is not always true. Usually, human intervention is needed to expose them from the original source code. The advantage of this approach is that allows to explore possibilities that for a general approach are not available (i.e. algorithm selection or application parameter tuning).

In [34, 51] authors suggest using control theory to create dynamic autotuner. The application requires software knobs that enable performance-accuracy trade-offs, and the developer is in charge of providing (or identifying) them. after that the autotuner is connected (or created [51]) and trained. During the runtime, the correct variant is selected according to platform condition and knobs value. Here the programmer is required only in the identification of the available knobs and in the evaluation of QoS of the application, so the human intervention is light.

An alternative approach suggests training a Bayesian network for automatic algorithm selection [52]. In this work, the autotuner consists of the bayesian network, which has to be trained at deploy time with training inputs to drive the choice of the correct variation at runtime. The intervention of the programmer here happens in two of the steps: the knob exposition and the training of the network. The knob required by this approach is to have different algorithmic implementations of an operation (such as different sorting algorithms). The training set must be representative of real-world instances, and this too must be provided by the user.

Finally, several approaches can select the optimal version between different implementations of a function [53, 54]. These approaches can target heterogeneous platforms, where the different implementations run on different hardware [55, 56]. They can manage workload splitting across the different compute units, or autotune kernel launch parameters typical of GPUs (such as grid configurations). These approaches are interesting because the autotuner is agnostic to the application. It sees the different versions of the function as software knobs, and the modeling algorithm can select the best variant according to the status and input. This allows adding this new perspective, function autotuning, to the classical software knobs.

An interesting approach to expose software knobs is provided by ATune-IL [57]. Here an instrumentation language is proposed that can be used to annotate through pragmas the original source code. After that, a source to source compiler generates the different variants, and autotuning is performed statically.

In this class we insert mARGOt [10], an autotuner that we developed and whose features will be explained more in detail in Section 2.3.

Frameworks and Domain-Specific techniques

We insert in the last category two different techniques, that have a strong impact on the original application. The application often needs to be completely rewritten to cope with the constraints imposed by this final category of techniques. Indeed, often the programming language is the key component of these techniques [35]. We also inserted some Domain Specific techniques because they can be applied, maybe in a seamless way, but only if an expert programmer evaluates their validity in the context of the application. Examples of this last case are [39, 58]. We can cluster these techniques into different groups, according to their application context.

Many approaches are related to the approximate computing field. Here, the quality of the result becomes a knob, that can be tuned: usually, by lowering the quality the application can save some energy. An example of this approach can be found in [59]. Here the programmer has to rewrite the application to use anytime computing techniques. The advantage is that a quick (and not precise) result is obtained in a lower time, and iterative refinements allow to increase the accuracy of the result itself. The execution of the application can be stopped at any moment, and this is another advantage of this approach. Other examples are [38,39,60]. In these approaches, there is no iterative refinement but a proactive prediction. Models are created that can select at compile time or run time the value for the knobs. In particular [60] uses Bayesian network to build the model, and selects at runtime the values of the knobs. In [38] statistical QoS are tested at compile-time and the selected version is the fastest among those that do not violate them. Finally, in [39] a subsampled image is used as a canary to select which approximations can be applied. Moreover, among the approximate computing techniques, some do not require human intervention in rewriting code. However, they are strictly domain-specific and there is still human in the loop since the decision of whether to apply or not must be taken from the human. Those techniques are [58, 61]. The first one applies approximation to CUDA kernels (such as removing atomic accesses or reducing the thread granularity), tests for QoS acceptability and selects at deploy time the best performing implementation. The second one targets six particular patterns in parallel kernels and applies approximations to them at compile time. The autotuning is performed statically by testing the QoS of the different solutions and selecting the best one that does not violate the constraints.

Another domain-specific approach, related to GPU kernel autotuning, is [62]. In this paper, the autotuner is in charge of managing at runtime the CUDA kernel parameters such as grid size, loop unrolling, ...

Other approaches, no more restricted to a domain, are complete frameworks that are used to wrap the original application or decompose it into tunable kernels that can also be exposed to other languages. An example of the first can be found in [37]. Here the focus is on the autotuning framework, that is agnostic from the application. It wraps the application itself and, once the knobs are given to the tool, it defines a Design Space Exploration (DSE) and uses models to optimally perform it. The human in this approach has to expose the knobs as application parameters, at compile-time, and to integrate it inside the framework. An example of the second approach is SEJITS [63]. Here the kernels are written using "efficiency language", such as C or CUDA, to obtain the best performance, while the global application is written using "consumer language" such as python. the framework is in charge to use just-in-time compilation to exploit the efficient kernels when available. The idea behind this approach is to have a library of kernels that can be used by multiple applications, hiding the complexity of efficient programming to high-level users.

Finally, some frameworks require a complete re-writing of the application in their own language. This approach is for sure the most invasive one, however allows more in-depth autotuning than the other approaches, since the language is designed for this purpose. The most important example is the Petabricks language [35, 36, 64, 65]. The language comes with the support of all the compiling infrastructure. It offers the possibility of selecting algorithm implementation [35]. A further refinement allows the choice to be driven at runtime by input features [64]. It is possible to manage approximate applications [36] and to search for the optimal configuration at runtime [65].

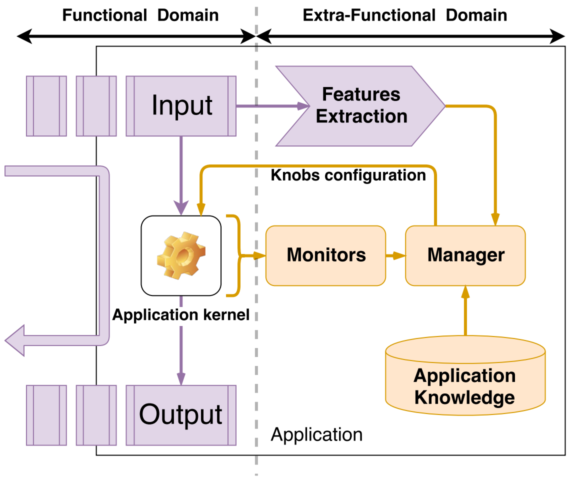


Figure 2.3: *Global architecture of the proposed framework. Purple elements represent application code, while orange elements represent mARGOt high-level components. The black box represents the executable boundary.*

Source: *[10]*

Slightly less invasive are OrCL [66] and Active Harmony [67]. Here the custom language is in the first case an annotation language and in the second case a scripting language used outside the application. OrCL requires the user to add annotations in Orio language to the original application. Such annotations are comments in the code that the Orio source to source compiler can parse and translate into C/OpenCL code. The autotuning of the parallel code happens in deploy time, where several parameters are tested and measured and the best configuration is found. Active Harmony is a framework where the original code has to be enriched with API calls to the framework. The developer needs to expose parameters and to notify their existence to the framework, that is in charge of autotuning them measuring the performances. This is done at deploy time and requires adding a Resource Specification Language (RSL) script to notify the framework which are the available resources for the autotuning.

2.3 mARGOt

mARGOt is a reactive and proactive autotuning framework that has been used in several of the techniques presented in this thesis. For this reason it is important to introduce it and some of its key concepts and definitions. Figure 2.3 shows an overview of the mARGOt framework and how it interacts with an application. In this chapter, we will assume that the target application has a single kernel g that elaborates an input i to generate the desired output o. However, mARGOt has been designed to be capable of managing different kernels (each one defined *block*) of an application in a completely independent way. We also assume that the kernel already exposes the software-knobs needed to alter its behavior. Let $\overline{x} = [x_1, \ldots, x_n]$ the vector of software-knobs, then we define a kernel as $o = g(\overline{x}, i)$.

Given this abstraction of the target application, we define the end-user requirements. The metrics of interest (the EFPs) are defined as the vector $\overline{m} = [m_1, m_2, \ldots, m_n]$. If the application programmer is able to find some properties of the input, we will define such properties as the vector $\overline{f} = [f_1, f_2, \ldots, f_n]$. Given these definitions, the requirements of the application can be formalized as in Equation 2.1:

$$\begin{aligned}
\max(\min) \quad & r(\overline{x}; \overline{m} \mid \overline{f}) \\
\text{s.t.} \quad & C_1 : \omega_1(\overline{x}; \overline{m} \mid \overline{f}) \propto k_1 \quad \textit{with } \alpha_1 \textit{ confidence} \\
& C_2 : \omega_2(\overline{x}; \overline{m} \mid \overline{f}) \propto k_2 \\
& \ldots \\
& C_n : \omega_n(\overline{x}; \overline{m} \mid \overline{f}) \propto k_n
\end{aligned} \tag{2.1}$$

where r is the objective function defined as a composition of any variable defined either in $\overline{m}$ or in $\overline{x}$ by using their mean values. C represents the set of constraints. Each C_i is a constraint expressed as the function ω_i, defined over the software-knobs or the EFPs and it must satisfy the relationship $\propto \in \{<, \leq, >, \geq\}$ with a threshold value k_i. If ω_i targets a statistical variable it also has to have a confidence α_i. Since mARGOt is agnostic to the distribution of the parameter, the confidence is expressed as an integer coefficient of its standard deviation (i.e. two times the standard deviation). If there are input features, then the value of the rank function r and the constraint functions ω_i may also depend from $\overline{f}$.

The main goal of mARGOt is to solve the following optimization problem: finding the configuration $\hat{\overline{x}}$ that satisfies all the constraints C and maximizes (minimizes) the objective function r, given the current input i. The application must have a configuration. If it is not possible to satisfy all the constraints, mARGOt will relax some of them, until it finds a feasible solution. For this reason, the constraints have a priority. mARGOt starts relaxing the lowest priority constraints first. Therefore, the end-user is required to give a priority to all the constraints. As shown in Figure 2.3, the mARGOt framework is composed of the application manager, the monitors, and the application knowledge. In the next subsections, we will see each component in detail, and we will conclude this section with some considerations on the integration effort required to insert mARGOt in the

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <points version="1.3" block="example">
 3    <point>
 4     <parameters>
 5      <parameter name="knob1" value="3.4"/>
 6      <parameter name="knob2" value="100"/>
 7     </parameters>
 8     <system_metrics>
 9      <system_metric name="metric1" value="212.862" standard_dev="6.49"
        />
10      <system_metric name="metric2" value="27.6" standard_dev="0.9"/>
11     </system_metrics>
12     <features>
13      <feature name="feature1" value="100"/>
14      <feature name="feature2" value="10" />
15     </features>
16    </point>
17  </points>
```

Figure 2.4: *XML configuration file to define the application-knowledge for an application.*

Source: *[10]*

application.

2.3.1 Application-knowledge

For generic applications, the relation between software-knobs, EFPs and input features is complex and unknown a priori. Therefore, we need to model the extra-functional behavior of the application to solve the optimization problem stated in Equation 2.1. In mARGOt a list of *Operating Points* (OPs) is used to model the application-knowledge. Each Operating Point θ expresses the target software-knob configuration and the achieved EFPs with the given input features; i.e. $\theta = \{x_1, \ldots, x_n, f_1, \ldots, f_n, m_1, \ldots, m_n\}$. Every *Operating Point* represent a working configuration of the application, with the expected values of the metrics given a configuration (i.e. a set of software knobs) and the input. In the mARGOt xml configuration files, all the software knobs are listed with the keyword *parameter*, the metrics with *system_metric* and the data features with *feature*. We choose this solution mainly for three reasons: firstly, to solve by inspection the optimization problem (which is the most efficient solution). Then, to guarantee that the chosen configuration is not illegal for the application. Finally, because it provides great flexibility in terms of management.

Figure 2.4 shows an example of an XML configuration file, containing a single Operating Point (lines 3-16). In this example, the target application exposes two software-knobs (*knob1* and *knob2*), has two metrics (*metric1* and *metric2*) and it is possible to extract two features from the input (*feature1* and *feature2*). For this reason, the OP is composed of three sections: the software-knobs configuration (lines 4-7), the metric section with the expected performance distribution (lines 8-11), and the related feature cluster (lines 12-15).

The OP list is a required input and mARGOt is agnostic to the methodology used to obtain it. Typically this is a design-time task, known as Design Space Exploration (DSE) in literature. It is a well-known problem, aimed at finding the Pareto Set. There are several previous approaches to find it efficiently [68–70]. The chosen methodology is out of scope from the mARGOt perspective.

Moreover, mARGOt has the capability of changing the application knowledge at runtime.

2.3.2 Monitors

It is important to observe the behavior of the application and the platform during the execution. For this reason, mARGOt has monitors. They are of critical importance because they provide feedback information. As we have seen, application knowledge defines the expected behavior which may change because of factors that are external from the application. For example, a power capper reduces the frequency of the processor. We expect the application to notice the performance degradation, and to react by changing its configuration to compensate. This is only possible thanks to feedback information.

If it is not possible to monitor an EFP at runtime, mARGOt can still work. It will operate in an open-loop, basing its decision only on the expected behavior.

2.3.3 Application Manager

This component is the core of the mARGOt dynamic autotuner since it provides the self-optimization capability. It is implemented using a hierarchical structure, as shown in Figure 2.5, where each level of the hierarchy targets a specific problem. The *Data-Aware Application-Specific Run-Time Manager* (DA AS-RTM) provides a unified interface to application developers to set or change the application requirements, to set or change the application-knowledge and to retrieve the most suitable configuration $\hat{\bar{x}}$. Internally, the DA AS_RTM clusters the application-knowledge according to the input features $\overline{f}$ by creating an *Application-Specific Run-Time Manager* (AS-RTM) for each cluster of Operating Points with the same in-

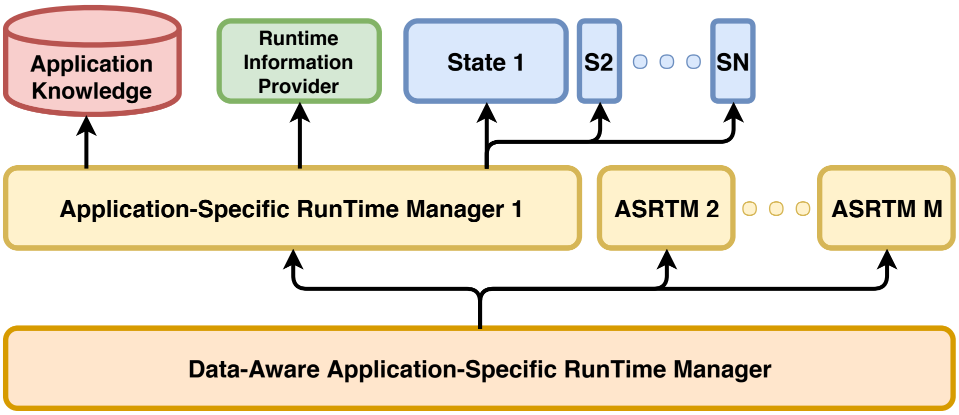

Application Knowledge
Runtime Information Provider
State 1
S2
SN
Application-Specific RunTime Manager 1
ASRTM 2
ASRTM M
Data-Aware Application-Specific RunTime Manager

Figure 2.5: *Overview of the Application Manager implemented in mARGOt, based on a hierarchical approach.*

Source*: [10]*

put features. Therefore, the clusters of OPs are implicitly defined in the application-knowledge. Given the input features of the current input, the DA AS-RTM selects the cluster with the features closer to the ones of the current input. Once the cluster for the current input is selected, the corresponding (AS-RTM) is in charge of solving the optimization problem stated in Equation 2.1. It has to select the configuration of the software-knobs $\hat{\bar{x}}$, according to changes in the execution environment and to the input features. However, it is possible that the objective function changes during the runtime. We define *state* a set of constraint and an objective function. It is possible to have different states in the AS_RTM of a *block*, however only one of them is the active one since only one optimization function can be active.

To solve the optimization problem, the RunTime Manager has to perform a single algorithm. At first, it assumes that the application-knowledge satisfies all the constraints, therefore all the OPs are valid thus contained in the valid OPs list L_{valid}. Then, for each constraint c_i, it iterates over the set of OPs and it performs two operations:

- It creates the list L_{c_i} of all Operating Points invalidated by the current constraint, and moves them from the list of the valid OPs to this list.
- It sorts all the OPs in L_{c_i} according to their distance from satisfying the constraint c_i.

After iterating over all the constraints, it sorts the list of valid OPs L_{valid} according to the objective function r. If the list of the valid Operative Points is not empty, it returns the one that maximizes the objective function. Otherwise, *mARGOt* iterates over the constraints according to their priority, in reverse order, until it finds a constraint c_i with a non-empty L_{c_i}. Then the best OP is the closest to satisfy the constraint c_i, i.e. $L_{c_i}[0]$. This algorithm must always return a single OP.

2.3.4 Integration Effort

While designing the framework, we focused on three points to ease the integration effort:

- separation of concerns between functional and extra-functional properties.
- limit the intrusiveness as much as possible.
- ease of use of the instrumentation code.

However, it is still required to the end-user or to the application developer to identify constraints, requirements, software knobs, and input features.

To ease the integration process, we provide a utility tool that generates a high-level interface for the target application. This tool takes as input two XML files that describe the extra-functional properties of interest. In particular, the main configuration file describes the adaptation layer, and the second configuration file describes the list of known operating points, as seen in Figure 2.4. The main configuration file defines:

1. The monitors of interest for the application;
2. The optimization parameters, i.e. the EFPs of interest, software-knobs, and input features;
3. The optimization problem stated in Equation 2.1.

Starting from these configuration files, the utility tool generates a library containing all the required glue code to hide, as much as possible, the implementation details. In particular, this library exposes five functions to the developers:

- **init**. A global function that initializes the data structures.
- **update**. A function that updates the software-knobs of a block with the optimal configuration found.
- **start_monitor**. A function that starts all the monitors of a block.

- **stop_monitor** A function that stops all the monitors of a block.
- **log** A function that logs the monitors of a block.

This library hides the details of the basic usage of the framework. However, if application developers require more advanced adaptation strategies, for example changing the application requirements at runtime, they will need to use the real mARGOt interface, since the high-level interface provided by the generated library will no more be enough.

2.4 Summary

In this chapter, we have seen the background and the state of the art in the autotuning field. We explored it through two perspectives, the time of autotuning and the intrusiveness. In a second moment we have introduced mARGOt, an autotuning framework that we developed, and we will see how we used it to enhance applications in Chapter 4, Chapter 9 and Chapter 6.

CHAPTER 3

Methodology

This chapter is focused on explaining the conceptual framework that is behind the work done in this thesis. As already mentioned in Chapter 2, this framework has not been implemented but it is an important reference model that has guided me through all my work. For this reason, we could call it a meta-framework, since it is an ideal entity. It is fundamental to understand the whole work done in this thesis since it has always driven the research.

We consider future applications as a sequence of self-tuning modules, that can adapt at runtime to the changing condition of the platform they are executed on. Moreover, they shall be able to exploit the heterogeneity, whenever available, and organize themselves to run each section of the application on the hardware that is most suitable to the computations that are being executed (i.e. if the application has a strong control-flow bound section, should be running on the CPU, while if there is a section with a lot of data-parallel computation it should run on the GPU). However, to reach this goal the original application needs to be changed and integrated by the developer. As we can see in Figure 3.1, there are two different areas. The first one is the user required code, and the second is the actual self-tuning module.

The user required code can be divided into two macro-areas, the first being the mandatory code (i.e. the application code and the constraint configuration) and the second the application-specific code, which is useful to have in a module but not mandatory. Application-specific code consists of pieces of code that have to be manually or semi-manually integrated into the original application to create more opportunities for the autotuning of the application during the runtime. Examples of application-specific code are the manual exposition of software knobs, such as searching accuracy-time to solution tradeoffs, or when inserting heterogeneity in a homogeneous application. The addition of the application-specific code to the application has to be done at design time since it needs to be performed by a programmer. More in detail, the operation that we envision in this category are:

- Manual Knob Exposition: the application is analyzed to find and expose some software knobs that were no present in the original application formulation. These knobs can be related to performance-accuracy tradeoffs, or other parameters that were in the original application decided once (such as command line parameters) and never changed during the run of the application itself. An example of this analysis is done in Chapter 9.

- Data Features: an analysis of the input data is performed, to take advantage during the runtime of some features of the input. This usually means that we want to cluster the set of inputs and manage the clusters in different ways during the runtime since we can take advantage of the features of the punctual input that we found with this analysis. An example of this is done in Chapter 6

- Heterogeneous Kernels: the application is analyzed and a hotspot kernel is ported to a more suitable architecture, that can be the GPU or an FPGA. The kernel is integrated into the original application flow, following the traditional approach of heterogeneous computing. An example of this can be found in Chapter 10.

The important section of Figure 3.1 is the right part, what is called the self-tuning module. This module is the key component of future applications. The original application is enriched with several components, thus becoming able to perform self-management during its runtime. We can notice from the picture that there are three main phases to obtain the ultimate goal of having an adaptive binary (which is the self-tuning module runtime implementation):

- create the enhanced application source code

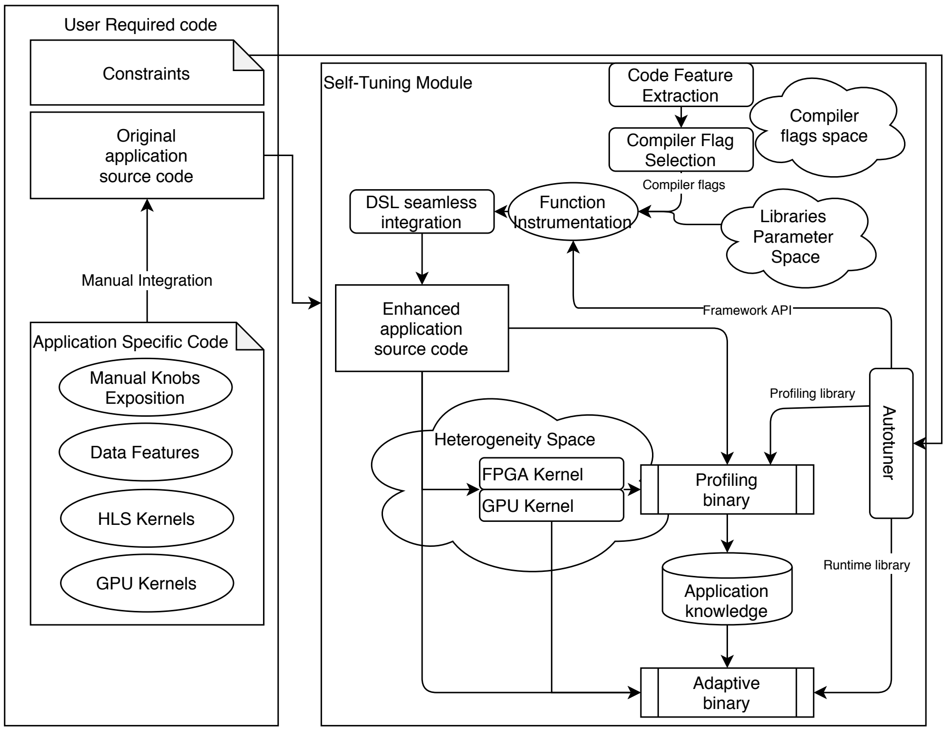


Figure 3.1: *The complete meta-framework.*

- profiling
- runtime adaptive application

The first step is probably the most complex one. This step can be partially automatized but in most cases, it requires interaction with the programmer.

In this step, we need to integrate into the original source code the possibility of self-tuning. This means that we need to insert an adaptivity layer, i.e. an autotuner, into the application and teach it which are the tunable parameters and how. In particular, we can notice from the picture that there are two different libraries for the autotuner, a profiling library, and a runtime library. These are needed since the profiling library is used at profile time to learn the behavior of the application (more in detail, of every profiled kernel) on the target platform, changing the input and the values of the software knobs. However, this library contains profiling functionalities that are not needed at runtime and lacks the knowledge needed to autotune. The runtime library is indeed pruned from the useless profiling functionalities and enriched with the Application Knowledge, and it consists in the adaptation layer that manages the application and gives the ability of self-tuning.

In this thesis, we used the mARGOt autotuner, described in detail in Section 2.3.

However, the autotuner alone is unable to do anything. To enable the self-tuning, we also need to provide the autotuner the software knobs. This work can be done manually, as we already mentioned, or in a semi-automatical way. Indeed, the top right part of the self-tuning module picture focuses on this use case. Some features are common to all the applications, such as compiler flags. Other possible knobs derive from libraries, that may be used in the program. In both these cases, it is possible to semi-automatically insert these knobs in the application code. We need to instrument the application at function level. In this way, we can learn the behavior of the different sets of parameters and compiler flags on these functions. We will see in Chapter 4 a study in this direction. The autotuner API can also be inserted during the function instrumentation, and they are needed to profile the behavior of the function. The last way to enrich an application is, as we already have seen, to insert some heterogeneous kernels. Sadly there is no way to do this automatically since as we will see in Chapter 10 even directive-based approaches require heavy modification of the original application source code.

Once all of these operations are done, and we have obtained the enriched code, a training phase occurs to extract knowledge from the application. A design space exploration has to be performed, to find the Pareto optimal frontier in the available parameter space. Previous research [68–70] have proposed methodologies to obtain the Pareto set. In this thesis, however, the methodology used to search the Pareto set is not important, and will not be investigated. This operation allows building the Application Knowledge, where the interaction of the software knobs with the evaluation metrics on the target machine is stored.

Once the Application Knowledge is obtained, it is possible to build the adaptive binary. This binary is the objective of the work of this thesis, and consists in the revised version of the original application as a sequence of self-tuning modules, able to adapt to the changing condition of the platform where they have been trained, or to changes in the requirements and the input data.

Part I

General Autotuning Techniques

CHAPTER 4

A Seamless Online Compiler and System Runtime Autotuning Framework

In this chapter, we address the problem of fine-grain autotuning, enabling the change of compiler flags across different functions or the number of involved OpenMP threads, with the final purpose of having the target application always working in the most efficient configuration. In particular, we propose SOCRATES, an approach where several tools are joined together to reach the self-tuning capability of the application. Moreover, we focus on reaching this goal with as little intrusiveness as possible, to ease the adoption of this solution by the programmers and to avoid introducing substantial changes in the original codebase. We demonstrate that thanks to SOCRATES we are able to maintain the running application in its optimal configuration (in terms of efficiency) while the objective function or the underlying platform change.

4.1 Introduction

Thanks to the continuous evolution of computing platforms, achieving performance portability of applications is a difficult task for developers. Performances are strongly dependent on the underlying platform and some characteristics of the input data. Moreover, they are also influenced by the system runtime. The autotuning approach has been proposed as the solution to this problem. Indeed, having code able to adapt to different platforms and conditions could enable performance portability. However, this approach has several unresolved questions. Among them, we can mention that writing such code needs a flexible and high-level language capable to express functional aspects, without constraining the implementation. In this way, it could be customized later, when the platform is decided, thus generating a program optimized for that platform.

As we have seen in Chapter 2, several approaches have been proposed, from the less intrusive but more restricted ones to rewriting completely the application to obtain adaptation. The target of these approaches is to give the autotuning capabilities to the application, thus finding the best configuration for the target platform. Usually, the less intrusive solution aims at finding one best-fit-all solution, without considering that the environment can change. Indeed, the workload may change, or the resource manager may allocate new cores to the application during the runtime. The solutions able to target these opportunities are the dynamic autotuners. However, their drawback is that they require a high level of intrusiveness in the original application.

In this chapter, we aim at obtaining a dynamic solution able to adapt at runtime changes of the configuration with an approach that has as little intrusiveness as possible. Indeed, configuring some extra-functional properties such as compiler flags and/or number of OpenMP threads can be not trivial if we want to always have the optimal configuration whenever the external conditions of the application are changing. This chapter introduces the SOCRATES approach. With this approach, we aim at offering the runtime autotuning of these Extra-Functional parameters at function level, with a framework that does not require any modification to the original application.

Figure 4.1 shows the components of the global autotuning vision targeted in this chapter. As we can see, most of the components are on the right side, the automatic one, while on the left side there are only the constraints and the original application source code. The main contribution indeed is the separation of concern: when writing the application, the developer does not need to be concerned with autotuning. After that, in a separate step, the autotuning is inserted into the application. We use an aspect-oriented language, LARA [12], to achieve the separation of concerns. Indeed, in this work, the extra-functional parts of the application

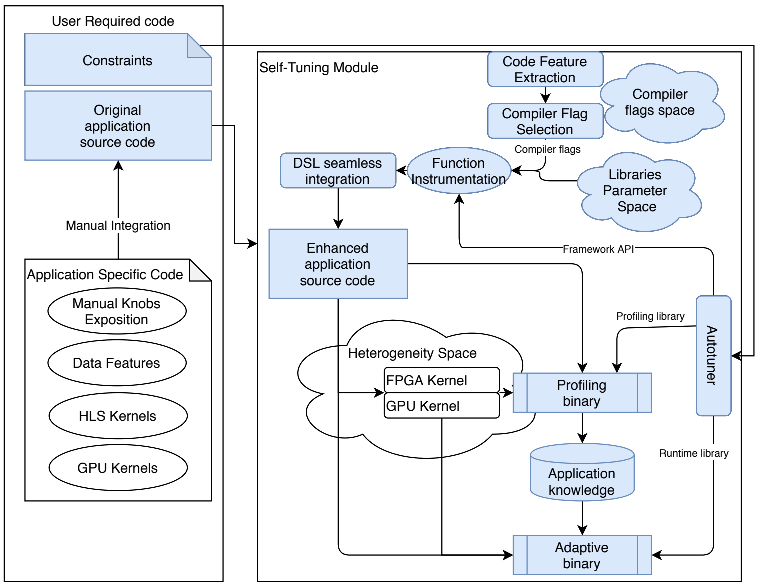


Figure 4.1: *Highlight of thesis approach targeted in this chapter.*

(included the mARGOt autotuner) are inserted in the application through LARA. Finally, SOCRATES introduces energy consumption as a key variable to be considered at runtime, thus introducing energy-efficient execution.

4.2 Background

The Aspect-Oriented Programming (AOP) approach [71] addresses several challenges in this context, by providing mechanisms that increase modularity and avoid code pollution. Not surprisingly, AOP has been intensively researched over the last decade (see, e.g., applications of AspectJ [72] and AspectC++ [73]). In this work, we adopted LARA DSL for its powerful selection and composition mechanisms provided, not only regarding explicitly weaving constructs (e.g., by inserting code) but also regarding hardware/software compiler and synthesis transformations.

There are approaches for the specification of code transformation and optimization strategies, such as CHiLL, PATUS, and Loopy. With CHiLL [33], we write code transformation recipes. These are scripts, separate from the main source files, with sequences of loop transformations to be applied to the program. On the other hand, PATUS [74] offers a DSL intended to be used for stencil codes. With this DSL, it is possible to control the application of several loop transformations, as well as the usage of architecture extensions (i.e., SSE). Loopy [75] allows the programmer to specify a series of loop transformations which are then automatically applied and guaranteed to be correct by formal verification. These are specified in a script (as in CHiLL) and are applied to the internal polyhedral representation.

Tuning the OpenMP parameters is not a novelty, since it has already been proposed in [76–78]. There the focus is on automatic parallelization of code with automatic selection of parameters done in a second phase. The approach of these works focuses on finding the one-fit-all solution for the given platform, without considering dynamic autotuning.

Overall, the proposed approach improves the state of the art thanks to the flexibility of its components: it allows to decouple the autotuning problem from writing the application and inserts dynamical autotuning that consider the evolution of the system in taking the optimal decision.

4.3 Proposed Methodology

SOCRATES aims at providing, in a seamless way, a framework able to enhance, at kernel level, an application with an energy-aware autotuning module. Figure 4.2 shows in detail the flow of the framework and shows all the tools involved.

The starting point of the proposed approach is a standard C/C++ source code describing the functional behavior of an application, i.e. $o = f(i)$ where a function f computes the output o from the given input i.

To reduce the compiler flag space, we used GCC-Milepost [13] and COBAYN [11]. The first tool is needed to analyze every kernel of the original code and to extract code features. These features are needed by COBAYN to select the most promising compiler flags for every kernel.

Once the compiler flag space is defined, we use the LARA toolbox to perform two actions, needed to obtain the adaptive kernels: 1) Multiversioning and 2) Autotuner insertion. These operations are shown in Figure 4.3. In particular, the first action transforms the original application into a tunable version. It inserts a set of *dynamic knobs* that can change its behavior. Thanks to this, the program model becomes $o = f(i, k_1, k_2, \ldots, k_n)$, where $k_1, k_2, \ldots, k_n$ are the set of knobs related to EFPs of the application (such as its execution time) or the result (such as its accuracy). In this way, several versions of the kernel are created and a wrapper is generated that will be used to call the selected version of the kernel. The second action is the introduction of the autotuner functionalities, i.e. the initialization function, with all the information to configure the dynamic knobs according to the requirements and the environmental con-

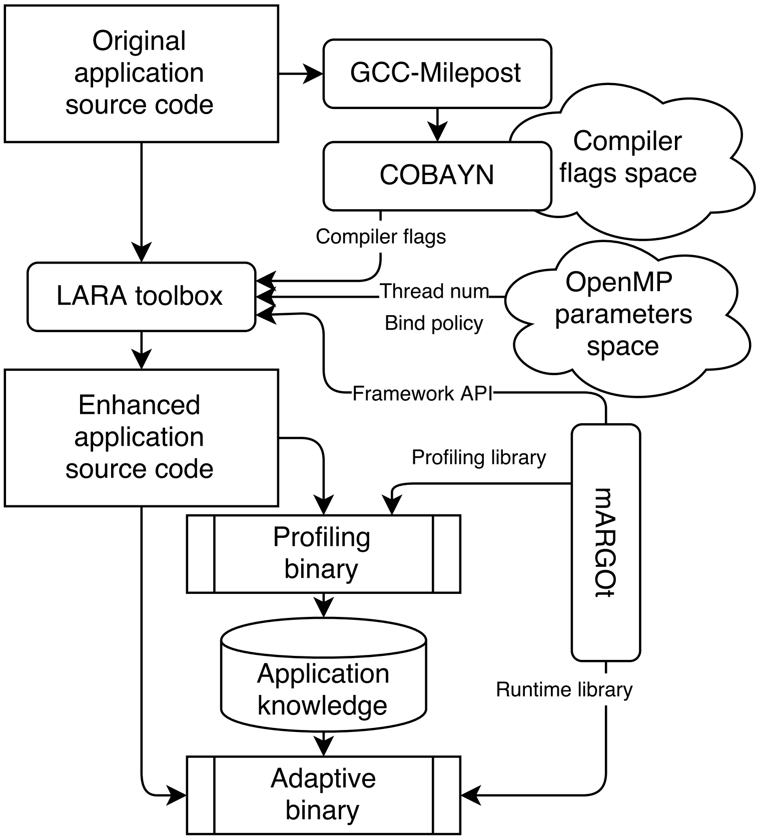


Figure 4.2: *Tool flow of the SOCRATES approach from the original application source code to the generation of the application adaptive binary.*

Source: *[79]*

dition. Moreover, it inserts the autotuner functions that wrap the kernel call and monitor its behavior. In particular, three of these functions are required: start, update and stop. The first function will start the monitoring system, the second will select the version of the kernel according to the monitor values and the third will stop the monitoring system. In the example, we only show an application with a single kernel, but the strong point of this methodology is that it can be applied on more than one kernel.

After the LARA step, we have obtained the Enhanced application code, which needs to be profiled in order to create the *application knowledge* required by the final adaptive application binary. This is done by running a profiling campaign on all the available alternative kernels, collecting their behavior on the underlying architecture. Finally, using the *application* *knowledge* we can create the final adaptive application.

Even if the overall approach is suitable for different contexts, we designed SOCRATES to address the following autotuning space:

Compiler Options (CO) : This knob represents a combination of compiler flags. We used the four standard optimization levels from gcc: `Os`, `O1`, `O2`, `O3`, plus some specific transformations that were deemed as the more interesting in [80]: *-funsafe-math-optimizations*, *-fno-guess-branch-probability*, *-fno-ivopts*, *-fno-tree-loop-optimize*, *-fno-inline-functions*, *-funroll-all-loops* ;

Number of threads (TN) : This knob sets the number of OpenMP threads between 1 and the number of logical cores;

Binding Policy (BP) :This knob sets the OpenMP binding policy: `spread` or `close`. We set the environmental variable OMP_PLACES to `cores`.

4.3.1 Step 1: Reduce the compiler flag space

Even if we did not consider all the possible compiler flags for the exploration, the Design Space is still too large. For this reason, we decided to use the COBAYN framework to prune the space and select, without having to measure every possible combination, the most promising compiler flags. COBAYN is an autotuning framework that exploits Bayesian Networks (BN) to extract the most suitable compiler optimizations from the source code. A required step before the prediction is the application characterization. Indeed, COBAYN uses application features to speed up the iterative compilation methodology, thus predicting which are the most suitable compiler optimization to enable. We perform the kernel characterization using GCC-Milepost, and we adapted COBAYN to work at function granularity. Thanks to this step, the 128 possible combinations of flags are reduced to 4 alternatives, before the application knowledge building step. These 4 alternatives are different for every kernel analyzed, since this step is performed in complete authonomy for every analyzed kernel.

4.3.2 Step 2: Integration

As we already mentioned, SOCRATES seamlessly integrates the autotuning into the original application thanks to the LARA DSL. Thw two strategies, *Multiversioning* and *Autotuner insertion*, are in charge of this job. They are used to enhance automatically the original code, obtaining a tunable application with an adaptive layer (the mARGOt layer). In particular, we use code transformation and code insertion strategies specified in

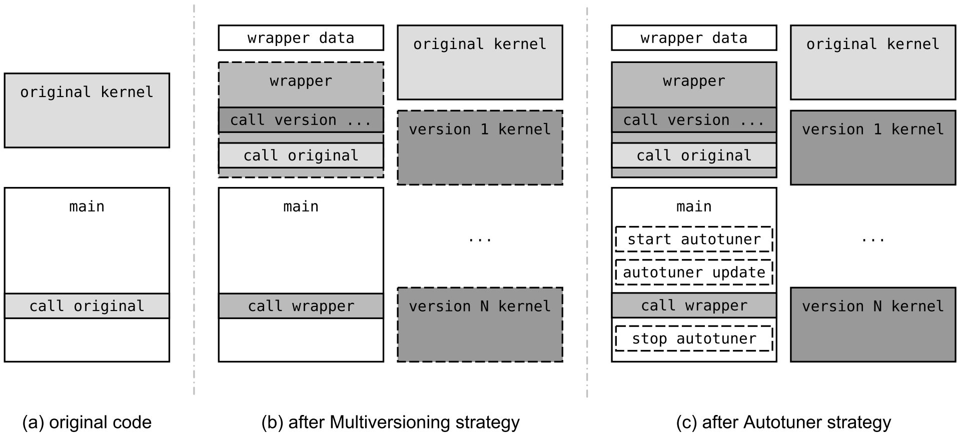


Figure 4.3: *Example of the automatic application code transformation from the original code (a) to the final adaptive code (c).*

***Source**: [79]*

LARA aspects to interact with the original code. MANET [81] is used as a source-to-source compiler to insert the required code as described in the aspects.

As we already mentioned, the *Multiversioning* strategy clones the kernel several times. The autotuning space has to be represented here. In particular, as we said, we have three dimensions: compiler flags, OpenMP threads, and OpenMP binding. Out of these three, two are statical parameters, while only the number of threads can be managed at runtime. For this reason, we need to clone the kernel several times. Each function clone represents a different version of the kernel in terms of compiler options and binding strategy. No cloned versions have been generated to manage the number of threads variable because it can already be changed at runtime. For each function clone, the strategy inserts GCC pragmas to set compilation flags (e.g., `#pragma GCC optimize ("O3,fno-ivopts")`) and OpenMP pragmas (e.g., `#pragma omp for proc_bind(close)`) to configure the parallelization of the kernels. Then it generates the wrapper, which allows switching the target version according to control variables. Finally, the strategy replaces each call of the kernel from the original source code, with a call to the wrapper (see Figure 4.3b). The entire process is fully automated.

The second strategy, *Autotuner insertion*, is responsible for the integration of mARGOt. As we already said, this strategy is in charge of two tasks. The first is to insert all mARGOt headers and the setup call needed by the

Table 4.1: *Metrics collected from the application of LARA strategies.*

Benchmark	**Att**	**Act**	**O-LOC**	**W-LOC**	**D-LOC**	**Bloat**
2mm	698	378	136	2068	1932	7.29
3mm	708	378	125	1801	1676	6.32
atax	684	250	81	1071	990	3.74
correlation	1347	410	138	2366	2228	8.41
doitgen	561	218	72	1018	946	3.57
gemver	631	218	94	1008	914	3.45
jacobi-2d	4429	154	145	2918	2773	10.46
mvt	339	154	64	571	507	1.91
nussinov	551	154	78	1356	1278	4.82
seidel-2d	445	154	47	565	518	1.95
syr2k	376	186	66	749	683	2.58
syrk	370	186	62	743	681	2.57
Average	928	237	92	1353	1261	4.10

autotuner. The second task is to wrap the kernel call with the mARGOt API that monitors the EFP and makes the variant selection.

4.4 Experimental Results

The platform used for the experiment is a NUMA machine with two Intel Xeon E5-2630 V3 CPUs for a total of 16 cores with hyperthreading enabled and 128 GB of DDR4 memory (@1866 MHz). We tested our methodology on 12 benchmarks from the Polybench/C benchmark suite [82]. We used SOCRATES to automatically generate the self-tunable applications, without adding a single line of code into the target applications. The Design Space considered for this campaign is the one described in Section 4.3. mARGOt is in charge of performing two tasks. The first one is to profile the application to build the application knowledge. This is done by performing a Design Space Exploration (DSE). The second task is to manage the application at runtime according to the application requirements given by the experiment. To evaluate this approach, we used a full-factorial analysis over the design space.

Table 4.1 presents some metrics regarding the application of LARA to each benchmark source. *Att* is the number of attributes of the source code that are checked by LARA. This number includes function signatures and pragmas. *Act* is the number of actions performed, including cloning and code insertions.

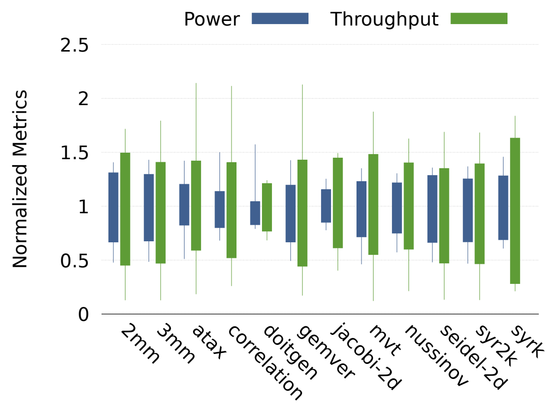


Figure 4.4: *Power/Throughput distribution of the Pareto-optimal software-knobs configuration.*

Source: *[79]*

The *LOC* columns represent, the number of lines of code of the original (*O*-) benchmark, the weaved (*W*-) benchmark and their difference (*D*-). The number of logical lines of source code in the complete LARA strategy is 265, which is used to calculate the *Bloat* metric [83]. This metric estimates how much code is weaved in the original application for each line of code in the LARA aspect.

These data present an overview of the complexity of the task, which should be done manually. Let's examine *2mm* as example. MANET automatically inspects multiple points in the source code, checking 698 attributes. Then it performs transformations (or insertions) on 378 of the inspected points. From the *Bloat* value we can see that we insert an average of 7.29 lines of C code per line of LARA aspect code. The large difference between benchmarks is explained because their kernels may be very different in size, with a different number of loops.

Figure 4.4 shows the analysis of the trade-off space between power consumption and throughput of the target kernels obtained with a full-factorial DSE. In particular, the boxplot shows the distribution between the throughput and the average power consumption. The y-axis represents the distribution of the target metrics for each evaluated application. In the construction of the graph, we considered only the Pareto-optimal configurations. As we can see, there is no one-fits-all configuration. This proofs the importance of the proposed approach.

Figure 4.5 shows the changes in the configuration for *2mm* if the tuning

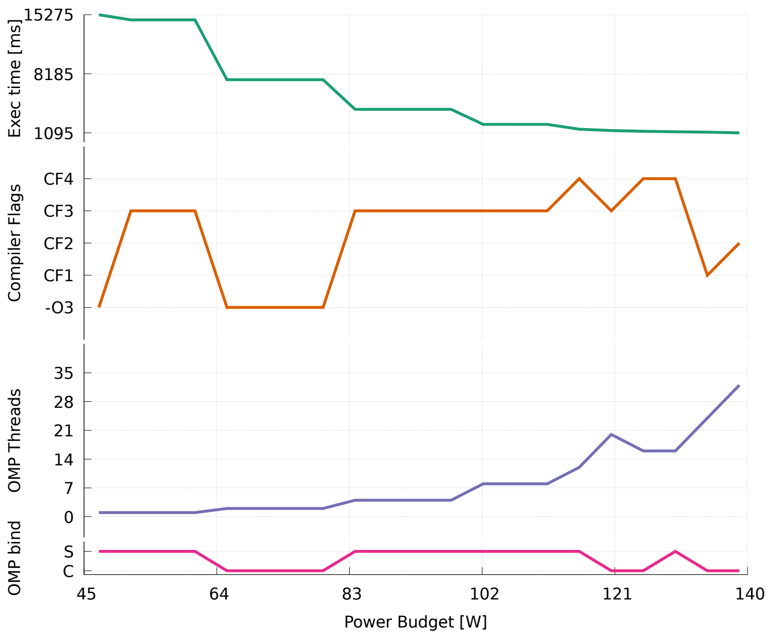


Figure 4.5: *Static analysis of the proposed approach. On the x-axis we can see the power budget and on the y-axis the values ot the knobs that minimize the execution time.*

***Source**: [79]*

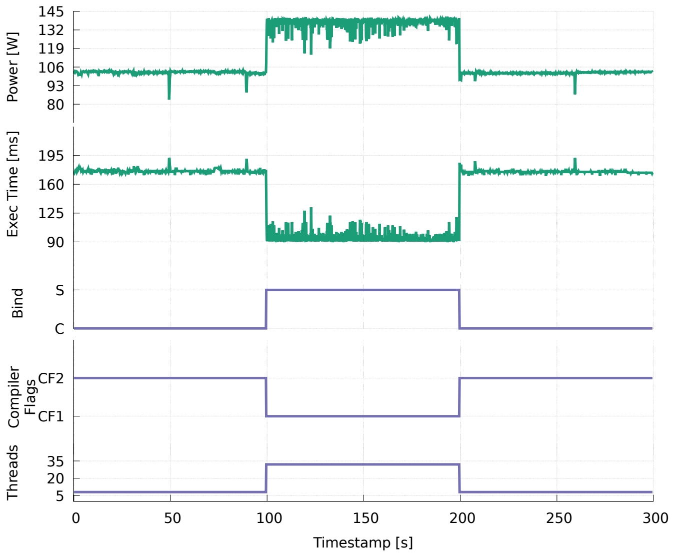


Figure 4.6: *Execution trace of the 2mm application. We changed the requirement each* 100 *seconds at runtime.*

Source*: [79]*

is done statically with a target power budget. We can see on the x-axis the target power budget, while on the y-axis there are the values of the software knobs in that configuration. The plot shows the available power-performance trade-off found in the Pareto curve. It can be seen that there are no knobs that are always the best. For this benchmark, COBAYN suggested the following flag combinations:

1. CF1): *O3, no-guess-branch-probability, no-ivopts, no-tree-loop-optimize, no-inline*;

2. CF2): *O2, no-inline,unroll-all-loops*;

3. CF3): *O2, unsafe-math-optimizations, no-ivopts, no-tree-loop-optimize, unroll-all-loops*;

4. CF4): *O2, no-inline*.

With the last experiment, we wanted to show the full potential of SOCRATES, with a runtime experiment. We simulate an experiment where the power budget changes twice, at first relaxing the initial constraint, then we recreate it. The idea behind this is that the application has two states in its execution, the energy save configuration and the throughput configuration. The first one optimizes the Throughput per Watt^2 (Thr/W^2), while the second is interested only in the throughput. Figure 4.6 reports the execution trace of the target application (*2mm*). We can notice that the parameter set changes whenever (at 100 and 200 seconds) we change the target configuration. mARGOt adapts the knobs to meet the configuration requirement.

4.5 Summary

This chapter contributes to the thesis by presenting SOCRATES, an autotuning framework designed to enable performance portability without requiring user contribution. We have shown by applying SOCRATES to the OpenMP polybench suite that it enables application adaptivity at runtime. It allows to always select the better configuration according to runtime constraints that can change during the application lifetime.

The outcome of this work has been published in the Design, Automation and Test in Europe Conference, 2018 [79].

CHAPTER 5

A library for tunable Multipliers Hardware Accelerators

In this chapter, we will consider the first heterogenous autotuning technique, targeting Field Programmable Gate Arrays (FPGA). In this context, we will create a tunable library that follows the idea of others in literature that can generate different accelerators for an important primitive operation in computer science, the large unsigned multiplication. This operation represents a significant computation effort in some cryptographic techniques. Thus, the use of dedicated hardware is an appealing solution to improve performance or efficiency.

The library can generate several large integer multipliers with different characteristics in terms of throughput and area occupancy through High Level Synthesis. In this way, even programmers that are not expert in using or programming FPGA can easily and quickly create accelerators for their application. The goal of this chapter is to provide a library that enables the end-user to explore a wide range of possibilities, in terms of performance and resource utilization, without requiring them to know implementation and synthesis details. Experimental results show the large flexibility of the generated architectures and that the generated Pareto-set of multipliers can outperform some state-of-the-art RTL designs.

5.1 Introduction

The increasing concern for security and safety [84] led to the more widespread usage of cryptographic techniques and an increase of their complexity. The multiplication between large integers is a common operation in this contest. As an example, the work in [85] uses the Pailler cryptosystem to perform data aggregation without decryption at intermediate hops in a network. However, this operation requires a significant computation effort, promoting the usage of efficient and optimized hardware component implementations to improve its performance, even if they have limited customization opportunities. The latter aspect is of paramount importance, when the component usage may have different requirements in terms of energy efficiency and performance. Therefore, researchers have spent a significant effort to investigate hardware accelerators for the multiplication of large numbers [86–90].

In this chapter, we propose a methodology to generate a throughput oriented hardware accelerator for large integers multiplication. By customizing high-level parameters, the users can search for the best compromise between resource utilization and performance, according to their requirements. We target Field Programmable Gate Arrays (FPGA), to exploit their flexibility. FPGA are a particular category of programmable hardware with reconfigurability capabilities. In the context of High Performance or High Throughput Computing, it is possible to use them to implement accelerators. A research path on tightly coupled FPGA-processor [91] aims at reducing the data transfer overhead significantly. Therefore, having specialized hardware units to multiply large integers near the CPU might be appealing. However, RTL programming is a costly and long procedure that requires a high level of specialization. For this reason, we target High Level Synthesis (HLS). HLS is a collection of methodologies that generate the hardware description starting from a high-level language. The benefits of this approach are two-fold. On one hand, it enables non-specialized developers to create hardware accelerators. On the other hand, by changing parameters in the high-level language, it is easier to generate a wide range of hardware descriptions. To the best of our knowledge, no HLS tool can support natively and in an efficient way large unsigned multiplication.

The main idea behind this chapter is to follow the approach of autotuning libraries such as ATLAS,MKL,SPIRAL adopted in High Performance Computing (HPC) [30,31,41,92]. These libraries expose a single interface,

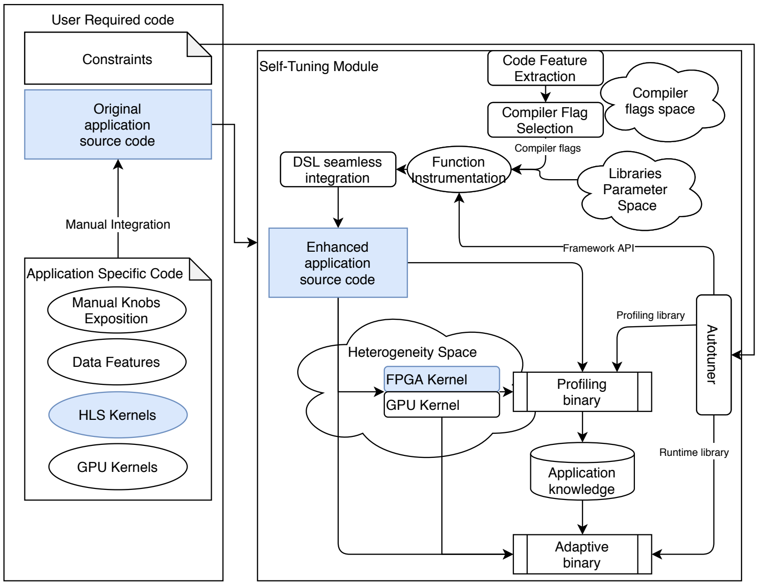


Figure 5.1: *Highlight of thesis approach targeted in this chapter.*

while they tune the actual implementation for the underlying architecture and input data. The proposed methodology is a parametric design for large integer multipliers, to enable trade-offs exploration between performances and area utilization.

To validate the methodology, we compare with state-of-the-art multipliers [89, 93]. In particular, we perform an initial Design Space Exploration (DSE) to identify multiplier configurations with Pareto-optimal trade-offs.

We may summarize the contributions of this chapter as follows:

- We propose a parametric approach to generate, using HLS, large integer multipliers.
- We provide the user the possibility to explore different levels of resource utilization.
- We propose a novel strategy to combine well-known multiplication algorithms.
- We perform a Design Space Exploration to analyze the performance-area trade-off.
- We publicly release the implementation code at https://gitlab.com/me2x/hls_multipliers.

In the context of the global framework, this chapter focuses on enabling the heterogeneous computing approach with a different architecture. In this case, the focus moves on creating an accelerator on an FPGA. This is only a first step toward a heterogeneous self-tuning module that relies on this platform. It is however important since it creates a starting point for future applications, that can exploit the proposed library and can couple it with the autotuning framework (for example to change the deployed multiplier according to constraints on the FPGA area available) and realize a real self-tunable module.

5.2 Background

This section describes the works in the literature related to the proposed approach. In particular, we focus on two main aspects. At first, we describe different implementations of large integer multiplication. Then we analyze autotuning libraries and how they approach alternative implementations.

Large integer multiplications If we focus on software, the GMP library [94] contains a collection of optimized algorithms. The optimal one is chosen at runtime according to the underlying architecture and operands size. This library is empirically considered the reference implementation for software multiplication. For example, both the FLINT library [95] and the NTL library [96] exploit GMP in their internal implementation. While software multiplications focus on the execution time, a hardware implementation must consider multiple extra-functional properties, such as area consumption and energy efficiency. Therefore, it is not possible to identify a single optimal solution for a given data size. A large fraction of literature investigates multiplication between small numbers, for example up to 128 bits [90]. An interesting approach has been proposed in a previous work [93], which employs an Open Source generator to optimize the usage of the Digital Signal Processor (DSP) available on the FPGA. However, it has been optimized for latency and not throughput. Rafferty et al. [89] provide an extensive comparison of different techniques to perform integer multiplications with arbitrary size and inspired us for this work. In particular, it is possible to use four base algorithms: the direct multiplication (Schoolbook), the Comba multiplication, the Karatsuba multiplication, and the Number Theoretic Transforms (NTT). Direct multiplication exploits the greatest number of DSPs, limiting its applicability for large numbers. On the opposite side, the Comba multiplier uses the least amount of resources, but it is the slowest. In the middle range lies the Karatsuba multiplier. Finally, NTT relies on Fourier transform and it requires a significant operand size (greater than 16384 bits [89]) to outweigh the initialization cost. Moreover, it is possible to combine different base algorithms, to leverage their strength. We consider the combined approach [89] as the baseline for the multipliers generated by the proposed approach. A significant fraction of the related works reported in this section targets FPGA. As previously mentioned, properly programming these devices with RTL requires experienced programmers. Since our objective is to provide the possibility to perform large unsigned multiplications to a large audience, we will target HLS. The proposed library will have to be integrated into the high-level code, and from there the HLS tool translates the functional behavior to logic level and it creates the hardware description [97–100].

Autotuning Libraries The typical workload of HPC includes scientific applications that belong to a wide range of domains. However, these applications typically share several computing-intensive patterns in their hotspots, usually algebraic operations. To implement an efficient code that performs such tasks requires the knowledge of the underlying architecture, introducing the problem of performance portability. Moreover, the application developer must perform this engineering task for every application. Autotuning libraries solve this problem by isolating a small set of performance-critical functionalities with a standard application programming interface (API). The function implementation uses different architecture-specific optimizations that are automatically applied according to the target platform. In this way, they release the application developer from the optimization task. An example of this approach is the BLAS API (Basic Linear Algebra Subprograms, [29]). The API defines a set of primitives for linear algebra, which has become standard for dense linear algebra applications. For example, the ATLAS [30] and SPIRAL [31] autotuning libraries exploit the BLAS interface. We have already analyzed this approach in detail in Subsection 2.2.2.

5.3 Target Class of Multiplication Algorithms

The basic multiplication method, named schoolbook, replicates the paper and pen method that is commonly used. Its drawback is that it requires a very large amount of load and store to update the result when we consider large integers. For this reason, literature investigates alternative multiplication methods. Among all of them, this work targets the intermediate size of operands, where the non-optimality of the direct approach starts to be a

```
$a, b \leftarrow$integers of $n$ bit
**for** $i \leftarrow 0$ **to** 2n-2 **do**
    **if** i < n **then**
        $pp_i \leftarrow \sum_{k=0}^{i} a_k * b_{(i-k)}$
    **else**
        $pp_i \leftarrow \sum_{k=i-n+1}^{n-1} a_k * b_{(i-k)}$
    **end if**
**end for**
$z \leftarrow \sum_{k=0}^{2n-2}(pp_k << k)$
```

Figure 5.2: *Algorithm of the Comba Multiplication*

```
$a, b \leftarrow$ integers of n bit
$a_h \leftarrow a >> n/2$
$a_l \leftarrow a \,\&\, n/2 - 1$
$b_h \leftarrow b >> n/2$
$b_l \leftarrow b \,\&\, n/2 - 1$
$z_h \leftarrow a_h * b_h$
$z_l \leftarrow a_l * b_l$
$z_{mid} \leftarrow (a_l + a_h) * (b_l + b_h) - z_h - z_l$
$z \leftarrow z_h << n + z_{mid} << (n/2) + z_l$
```

Figure 5.3: *Algorithm of the Karatsuba Multiplication*

problem, and before the size of the operands is too big to justify the NTT approach.

Comba Multiplication This multiplication algorithm, proposed by Comba [101], aims at reducing the number of load-store required to compute the results, without altering the complexity. Algorithm 5.2 shows the algorithm of this multiplication technique. It computes partial products between the digits of the operands, and the order of the partial products allows to build directly the final result by shifting and summing the computed partial product with the result. The complexity of the operation does not change.This algorithm is appealing when there are limited hardware resources [87].

Karatsuba Multiplication This multiplication algorithm, proposed by Karatsuba [102], aims at reducing the complexity of the operation. It splits the operands into two smaller terms and computes the final result as a polynomial multiplication and it can be applied recursively. Therefore, if the complexity of the operation using the previous algorithms is O(n^2), the complexity of the Karatsuba algorithm is O($n^{log_2 3}$). It replaces one n bit multiplication with three $n/2$ bit multiplications and 4 additions. The algorithm is reported in Algorithm 5.3 However, recursively applying the algorithm requires a high number of hardware resources [88].

Karatsuba-Comba Multiplication The Karatsuba algorithm is an appealing approach to implement a large integer multiplication, due to the lower complexity of the operation. Therefore, to mitigate the resource requirements, researchers investigated the possibility to combine the Karatsuba and the Comba algorithms [89, 103]. Previous works investigate both software implementation [103] and hardware implementation [89]. In particular, the latter focuses on a hardware implementation that performs a single iteration of the Karatsuba algorithm, while it uses the Comba algorithm to compute the three internal multiplications. Experimental results demonstrate that this solution has the lowest latency for operands between 512 and 16384 bits [89].

5.4 The Proposed Approach

This section describes the proposed methodology and its implementation. In particular, we propose to use a parametric high-level architecture description and to rely on a High Level Synthesis framework for the actual hardware implementation of the multiplier. At first, we describe the generated architecture template and extract the high-level parameters. Then, we describe how these high-level parameters change the generation of the multiplier. Since we are targeting HLS, we aim at exploiting the DSPs available on the FPGA board. For this reason, we will not decompose the operation to the bit level.

Architecture Template The proposed methodology provides an architectural template that combines the multiplication algorithms described in Section 5.3, allowing the exploration of a wide range of performance-resources trade-offs. The idea is to use a *divide et impera* approach. We use recursion to reduce the operands bit-width to increase multiplication efficiency, while we use sharing techniques to limit the required amount of logic in the computation. Figure 5.4 shows the architecture template and highlights the sizes of the operands across the different layers. We split the multiplication algorithm into three phases: (1) Karatsuba Operands Decomposition: we recursively apply the Karatsuba algorithm to reduce the complexity of the multiplication. Every recursion step is named "layer". (2) Products Evaluation using the Comba Algorithm. (3) Karatsuba Result Composition: we

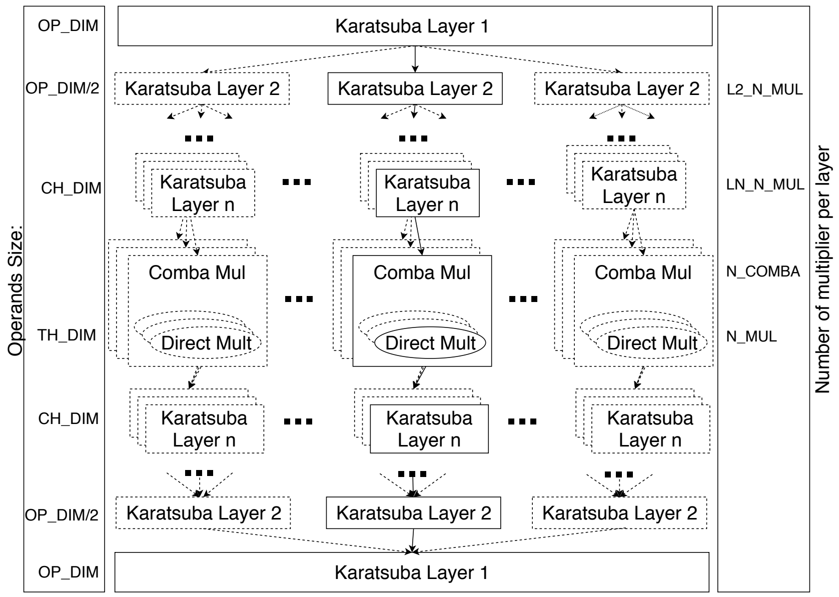


Figure 5.4: *Architecture Template of the proposed methodology. The components with dotted lines are optional and their instantiation depends from the high-level parameters configuration.*

***Source**: [104]*

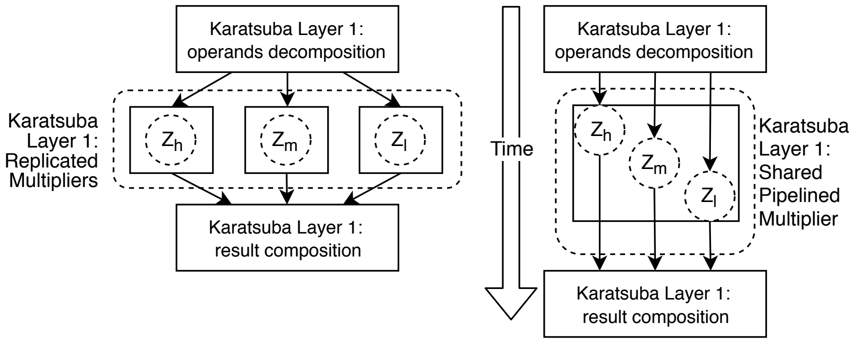

Karatsuba Layer 1: operands decomposition
Karatsuba Layer 1: Replicated Multipliers
Z_h
Z_m
Z_l
Karatsuba Layer 1: result composition
Time
Karatsuba Layer 1: operands decomposition
Karatsuba Layer 1: Shared Pipelined Multiplier
Z_h
Z_m
Z_l
Karatsuba Layer 1: result composition

Figure 5.5: *Karatsuba multiplication layer with replicated (left) and shared (right) multipliers for the inner multiplications.*

***Source**: [104]*

recursively apply the sum and shifts required by the Karatsuba algorithm to obtain the result, from the inner layer to the outer.

In the “operand decomposition” phase, at every layer, the dimension of the data is halved and the three multiplication are instantiated, following the traditional approach for the Karatsuba algorithm. We add, at every layer, a choice that allows the programmer to force the re-use of the inner multiplier to perform all the three multiplications needed by the Karatsuba algorithm in a pipelined fashion. This introduces the possibility to replicate or share resources, for each Karatsuba layer, as depicted in Figure 5.5. In particular, the dotted circles represent the required multiplications (z_h, z_m and z_l), while the squares represent the instantiated multipliers. On the left side of Figure 5.5, there is the traditional approach of the Karatsuba multiplication, where all the three multipliers are instantiated. On the right side of Figure 5.5, we represent the pipelined architecture for a Karatsuba layer, where a single multiplier is created and the three multiplications are pipelined. The idea of forcing the reuse of inner resources enables the trade-off between performance and resource usage. Therefore, we can use this mechanism to balance the initiation intervals of the different layers, to prevent idle inner layers.

When the bit size of the intermediate multiplication terms (i.e. z_h, z_m, z_l) reaches a given threshold, a last layer of Karatsuba instantiates its three multiplications using the Comba algorithm. This is the second phase of the multiplication. Even at this layer, the user can specify the number of instantiated Comba multipliers (1 or 3). We implement the Comba algorithm in a slightly different way with respect to the standard way with one single multiplier that evaluates all the partial products serially. This choice is done to enable parametrization inside this component. In particular, we divide the multiplication into four steps: (1) We split the two operands in digits, according to the size of the Direct multiplication. Each digit is stored in a different variable to enable parallel access. (2) We compute the partial products independently. According to the number of Direct multipliers, we can modulate the latency of this phase. If this parameter is equal to the number of multiplications, they are computed in parallel. If this parameter is equal to one, it computes the products serially. Otherwise, there is some degree of parallelism in the computation of these products. We use Direct Multipliers because they are mapped directly on DSP. (3) We compute the sums needed to generate the partial products (the sum of all the columns in Algorithm 5.2) into separate variables. (4) We reconstruct the final result. We use only sums, shifts, and masks to reduce the hardware complexity. This approach allows all the internal variables to be written and read-only once, thus leaving all the scheduling decisions to the HLS tool and enabling pipelined and/or parallel approaches, if enough resources are allowed for allocation. Finally, in the last phase, we combine the partial results of each Karatsuba layer.

Using this approach, we can expose to the designer, two categories of high-level parameters. The first one is related to the dimensions of the operands, while the second category controls the resource reuse. In particular, the parameters that fall in the first category are: *OP_DIM*(the bit size of the multiplier operands), *CH_DIM*(the bit size threshold of the last Karatsuba layer) *TH_DIM*(the bit size of the direct multiplication, i.e. the digit bit-width in the Comba algorithm). To generate a balanced multiplier, these parameters must be a power of two. The second category of parameters addresses resource utilization. In particular, the high-level parameters that fall in this category are: *L2_N_MUL*(the number of OP_DIM/2 bits Karatsuba Multiplier to instantiate, 1 or 3). *more LN_N_MUL*(the number of Karatsuba Multiplier to instantiate inside the *x-th* layer, 1 or 3. The number of these parameters is tied to the number of Karatsuba layers). *N_COMBA*(the number of Comba multipliers in each innermost Karatsuba layer component, 1 or 3) and *N_MUL*(the maximum number of Direct multipliers allowed in the implementation of each Comba). We designed the approach to be agnostic to the HLS engine. However, in the current implementation, we use Vivado HLS pragmas to enforce resource-related parameters. It is possible to port the methodology in a different HLS engine by changing this implementation detail.

Methodology Implementation This work aims at generating throughput oriented large integer multipliers, with a wide range of extra-functional properties, in terms of Initiation Time (i.e. the inverse of the throughput) and resource usage. In this way, the end-users can generate the most suitable multiplier according to their requirements. The recursive nature of Karatsuba is a good match for recursive functions. However, they are not supported by most HLS tools. Therefore, instead of using complex code generators, we propose to use C++ templates to solve the recursion at compile-time, generating the actual code in the multiplier declaration. While its usage in the application code is consistent, by changing the template arguments, it is possible to drastically change its hardware implementation, leading to different extra-functional behaviors. Therefore, the high-level parameters described before with the architecture template definition are implemented using variadic C++ templates. To the best of our knowledge, [89] proposes the first implementation of the Karatsuba-Comba multiplier. In particular,

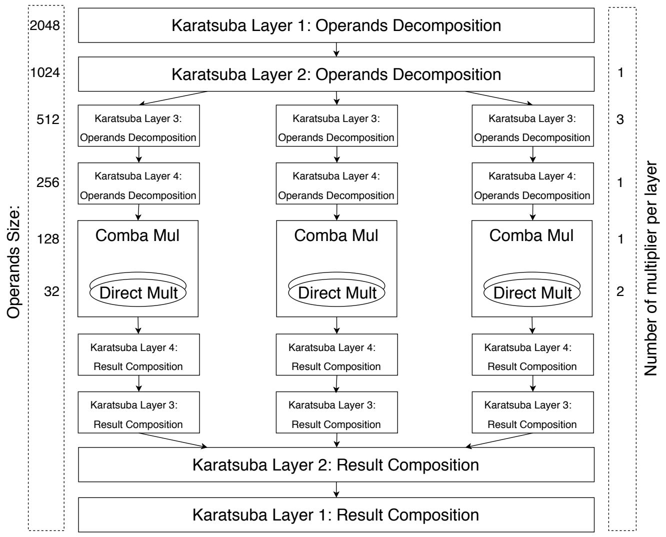

Operands Size:
2048
1024
512
256
128
32
Karatsuba Layer 1: Operands Decomposition
Karatsuba Layer 2: Operands Decomposition
Karatsuba Layer 3: Operands Decomposition
Karatsuba Layer 4: Operands Decomposition
Comba Mul
Direct Mult
Karatsuba Layer 4: Result Composition
Karatsuba Layer 3: Result Composition
Karatsuba Layer 2: Result Composition
Karatsuba Layer 1: Result Composition
Number of multiplier per layer
1
3
1
1
2

Figure 5.6: *An example of multiplier that can be generated by the proposed approach, according the high-level parameters configuration.*

it investigates a hardware design that uses a single layer of Karatsuba, to reduce the operands bit-width, and three Comba multipliers to perform the inner products. However, its implementation does not follow a pipeline approach and it exposes limited flexibility. On the contrary, our proposed approach is throughput oriented, and it enables a greater level of flexibility since it enables the reuse of pipelined components. Indeed, by changing high-level parameters, it is possible to set the number of Karatsuba layers. Moreover, it is possible to define for each layer the reuse policies to limit the area of the multiplier, leading to unexplored multiplier architectures.

Figure 5.6 shows an example of a generated multiplier architecture. We report the operands bit-size on the left side of the image, while on the right side we report the number of multipliers used in each layer. In particular, it aims at multiplying two 2048 bit integers (*OP_DIM*$=$ 2048). This instance exploits 4 layers of Karatsuba to reduce the size of the operands (*CH_DIM*$=$ 256). Moreover, we enforce the reuse of all the multipliers, except the ones of the third Karatsuba layer (*L2_N_MUL*$=$ 1, *L3_N_MUL*$=$ 3, *L4_N_MUL*$=$ 1, *N_COMBA*$=$ 1). Finally, we use two (*N_MUL*$=$ 2) 32 bits (*TH_DIM*$=$ 32) direct multipliers in each Comba multiplier. This multiplier is declared with a single line of cpp code, where we set the parameters in this simple way: multiplier <2048,256,32,1,2,1,3,1>(A,B,OUT), where A and B are two ap_uint<2048>and out is an ap_uint<4096>variables. The actual parametrized interface of the whole library is:

Table 5.1: *High-level parameters values explored in the DSEs. OP is the operand dimension, LM is the number of sub-multiplier, NC is the number of Comba multipliers, NM is the number of Direct Multipliers. Frequency is in MHz.*

OP	CH_DIM	TH_DIM	LM	NC	NM	Freq
2048	1024 512 256 128	128 64 32	1 3	1 3	1 2 4	100
1024	1024 512 256 128 64	64 32 16	1 3	1 3	1 2 4	150
512	512 256 128 64	64 32 16	1 3	1 3	1 2 4	250
256	256 128 64 32	32 16	1 3	1 3	1 2 4 8	300
128	128 64 32	32 16	1 3	1 3	1 2 4 8	350

```
multiplier<OP_DIM,CH_DIM,TH_DIM,N_COMBA,N_MUL,...>(ap_uint<
    OP_DIM>A, ap_uint<OP_DIM>B, ap_uint<2*OP_DIM>OUT)
```

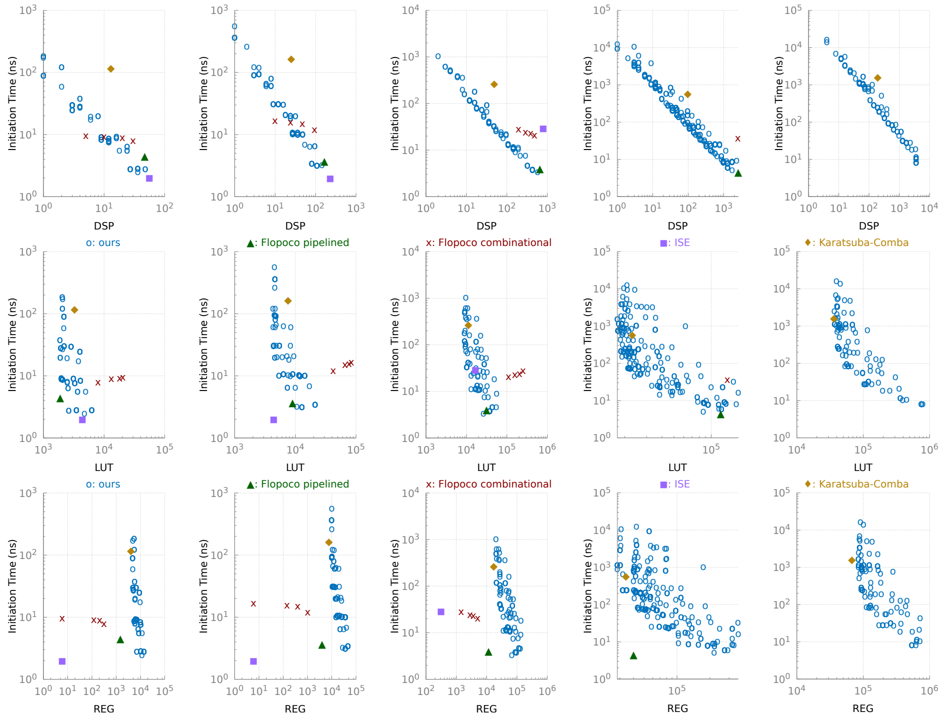
The variadic template is needed to manage the variable number of LX_N_MUL parameters. The code of the library component is made available[1].

5.5 Experimental Results

This section evaluates the benefits of the proposed approach. At first, we describe the Design of Experiment that we used to generate the multiplier implementations. Then, we analyze their extra-functional properties in terms of Initiation Time and resource usage at the post-place stage. Finally, we compare them with state-of-the-art multipliers. In particular, we consider RTL implementations [89] and several instances generated by FloPoCo [93]. To perform these tasks, we use the tool Vivado HLS 2018.2 and Vivado 2018.2, on a Virtex7 xc7vx980t.

Design of Experiments The methodology aims at generating multipliers with a wide trade-off space between performance and resource usage, given the target operands size. Table 5.1 reports for all the considered operands dimensions (from 128 bits to 2048) the values of the explored parameters that influence the architecture. We performed a DSE for all of these bit-widths, combining in a full factorial DoE these parameters. The idea is to explore up to five Karatsuba layers with different sharing policies and characteristics of direct multipliers. The Design Space dimensions for the different operands size, from 128 to 2048, are respectively 112, 240, 270, 558 and 540 configurations. We select the target frequency to compare fairly with the existing solutions [89]. Once the DSE is completed, we performed Pareto filtering to remove all the points that are dominated, also by state-of-the-art multipliers.

[1]https://gitlab.com/me2x/hls_multipliers



(a) *DSE 128 bit* **(b)** *DSE 256 bit* **(c)** *DSE 512 bit* **(d)** *DSE 1024 bit* **(e)** *DSE 2048 bit*

Figure 5.7: *Initiation time of different multipliers, by varying the number of used resources and the operand size. Each row represents a different resource, while each column represents a different operand size. The blue dot represents the multipliers created with the proposed methodology, the green triangles represent the multipliers created with the pipelined version of FloPoCo, the red cross are the multipliers created with the combinational version of FloPoCo, the lilac squares are the Vivado ISE multipliers (as reported in [89]) and the orange diamonds are the Karatsuba-Comba multipliers proposed by [89].*

Design Space Analysis Figure 5.7 shows the result of the Design Space Exploration of multipliers from 128 (Figure 5.7a) to 2048 (Figure 5.7e) bits. For each operand size, we report the trade-off between the Initiation Time and the target resource. The Initiation Time is the time elapsed between the start of two pipelined operations, i.e. the inverse value of the Throughput. In particular, the first row shows the DSP consumption, the second row the LUT usage, while the third row shows the Registers usage. All the reported plots use a logarithmic scale. If we focus only on the multipliers generated by the proposed approach (named "ours"), we can notice how the Initiation Time spans over 3 orders of magnitude, for all the operands size. Moreover, there is a strong correlation between Initiation Time and the number of DSP, where the lowest Spearman correlation coefficient among the different operand sizes is -0.96, with a p-value smaller than 0.0001. The Initiation Time is also correlated with the multiplier area: the Spearman coefficient is between -0.7 and -0.8, with a p-value smaller than 0.0001. Even if we expected these results, it shows how the methodology can provide to the end-user the multiplier that best fits its requirements, in terms of performance and area utilization. For example, in all the operand sizes that we analyzed, the end-user can always choose between a fast multiplier (e.g. with Initiation Time lower than $10ns$) and a small one (e.g. with less than 10 DSP).

Comparison with multipliers from Flopoco [93] FloPoCo is a VHDL generator of arithmetic cores. Even if it mainly targets small operands, it can generate large integer multipliers. Moreover, the library supports the generation of throughput-oriented pipelined components. Therefore, we compare the multipliers generated by the proposed methodology, with the ones generated by FloPoCo version 5.0. For each operand size, we generate different multipliers by changing the amount of available DSP, including one without restrictions. When the tool fails to generate a pipelined component, for example when we limit the amount of DSP, it will fall back to a combinational component. Figure 5.7 reports the generated pipelined multipliers (named "Flopoco pipelined") whenever it is possible to generate them. While it reports the generated combinational component (named "Flopoco combinational") when the pipelined version is unavailable. A deep exploration of resource parameters is difficult using FloPoCo. Indeed, when we consider the time requested by FloPoCo to generate the multiplier RTL description, we notice a drastic increasing trend due to the operands bit size. In particular, the tool can generate a 128 bit within an hour time frame. However, the time increases to days for 512 bit multipliers and it was unable to produce any 2048 bit multiplier after two weeks of computation. This is visible in Figure 5.7e, where no FloPoCo solution are plotted, but also in Figure 5.7d where only a combinational configuration is available. These times are several orders of magnitude higher by the minutes required by the proposed methodology to obtain the RTL. If we consider the combinational multipliers, we need to differentiate between larger than 512 bit integer multipliers and smaller than 256 bit integer multipliers. While for larger numbers (512-1024) FloPoCo has a worse DSP and LUT utilization compared to the ones generated by the proposed methodology, this is not always true for the small ones (128-256). Figure 5.7a and Figure 5.7b show a solution where FloPoCo has better DSP and Register utilization, at the same throughput level. However, that solution requires a much larger number of LUT. When we change the resource constraining parameters, we noticed that FloPoCo is not capable of generating trade-offs between resources and throughput. Indeed, it uses LUT to replace DSP to generate architectures with a similar level of throughput. If we consider the pipelined multipliers, the proposed methodology generates a module with a better DSP utilization, at the same LUT and throughput level. However, FloPoCo generates multipliers with better Register utilization. The only exception we found is depicted in Figure 5.7d where, by using a very large number of DSP, the FloPoCO multiplier outperforms ours in terms of throughput. Analyzing the solution generated by our methodology, we found that the carry propagation in the adder required by the final Karatsuba recomposition is the critical path. This highlights that there is still room for further improvement in the next future.

Comparison with Rafferty et al. [89] This work investigates the performance and resource usage of several multipliers. In particular, it shows how the ISE instantiated multiplier and the Karatsuba-Comba design have the best throughput according to the size of the operands (ISE up to 512 bit, Karatsuba-Comba above). Both of them are hand-optimized. The former belongs to the Xilinx ISE library and the latter is proposed in the paper [89]. Therefore, this section considers both of them (named “ISE [89]” and “Karatsuba-Comba [89]” in Figure 5.7). From the results, we can notice how the ISE multipliers are the ones with the best Initiation Time and the highest DSP consumption for small integers, up to 256 bit. However, they are Pareto-dominated by the multipliers generated by the proposed methodology, if we consider 512 bits operands. Moreover, when we target larger integers they become unavailable. The known Karatsuba-Comba multipliers (named “Karatsuba-Comba [89]”) are unable to reach the throughput of the multipliers generated by the proposed methodology, with the same amount of DSP, for all the analyzed bit sizes. However, we can notice that the hand-designed hardware can efficiently use LUT and Registers, compared to other multipliers that target a similar throughput. This behavior is due to the HLS procedure that is unable to reach the same optimization level when it translates high-level languages into hardware description. Both the solutions presented in [89] are not flexible and do not allow any performance-resource trade-off exploration.

5.6 Summary

Working with FPGA increases the complexity of evaluating the performance-power tradeoffs that the developer needs to consider when writing applications. In this chapter, we have seen a methodology that allows generating a tunable library of large integer multipliers using HLS. It uses a parametric Karatsuba-Comba multiplication template to instantiate throughput oriented multipliers. This flexibility enables the end-user to select the most suitable multiplier, according to the application requirements, after having evaluated the possible trade-offs. Indeed, from experimental results, we can notice how the proposed methodology can generate multipliers with a range of performances and resource usage that is greater than one order of magnitude.

The outcome of this work has been published in the Design, Automation and Test in Europe Conference, 2021 [104].

CHAPTER 6

Autotuning a Server-Side Car Navigation System

In this chapter, we study a HPC application and we try to improve the computation efficiency of this application by searching for a proactive way to limit the amount of computation required. In particular, we analyze a Probabilistic Time-Dependent Routing application (PTDR), a component in a traffic navigator application. We propose a novel approach for dynamically selecting the number of samples used for the Monte Carlo simulation that is used to solve the PTDR, thus improving the computation efficiency. The focal point of this approach is the study of the input and the research of a function that can extract characteristics of the input that can be used to drive the application. Once this function is found, we integrate it into the original application with the LARA aspect-oriented language, already used in Chapter 4 for a different purpose. We manage the runtime process of selection with mARGOt autotuner, which has been enriched with the proactive functionality thanks to this case study.

6.1 Introduction

The idea of smart city is a place where common tasks are automatized to ease the life of citizens. One of these tasks is traffic prediction: this can be used to avoid congestions, thus easing the life of the people that can predict the travel time when moving in the city, but also reducing car emissions. Moreover, if we consider the self-driving car vision, the routing requests are going to increase by a large amount, along with the necessity of real-time updates of the traffic situation. This is going to increase the computational resources dedicated to this task since the main computation required are operations on large graphs. Consequently, the trend is to move these tasks to more powerful infrastructures, such as HPC.

From the algorithmic point of view, the routing problem is well known in the literature. The optimal path between two points in a graph is a well-known problem and Dijkstra's shortest path algorithm has been proposed to solve it. However, this is not the only problem that a navigation system has to target. The system has to be able also to manage larger optimization problems, such ah route planning for a fleet of delivery vehicles or waste collection vehicles. Another targetable problem could be traffic management in the smart city context [105]. However, the definition of the optimal path can be not unique. It depends on the weights used in the graph that represent the road system. The shortest path is only based on geometrical distance, while the fastest path only considers the time elapsed in the trip. There might be even more complex criteria; however, their description is out of the scope of this work. The time needed to travel a road is affected by various elements, such as accidents, traffic congestion, road work, and so on. A simple starting point is to use the upper legal limit of speed, based on the assumption that each vehicle travels at the same speed. However, this approach is inaccurate because of the natural behavior of traffic.

Research efforts have been spent in the latest time in predicting the average speed on the road network using statistical analysis and various models. This has been made possible thanks to the collection of historical traffic monitoring data. However, a single-speed value prediction is not very useful since it is not visible the stochastic behavior of the traffic. The probability distribution of the speed at different times of the day allows incorporating real-world events that can cause major delays and affect traffic over vast areas. In this way, it is possible to compute the probability of arriving within a certain time. This change in the approach can be useful in creating a more accurate route planning system. This problem is called *Probabilistic Time-Dependent Routing* (PTDR).

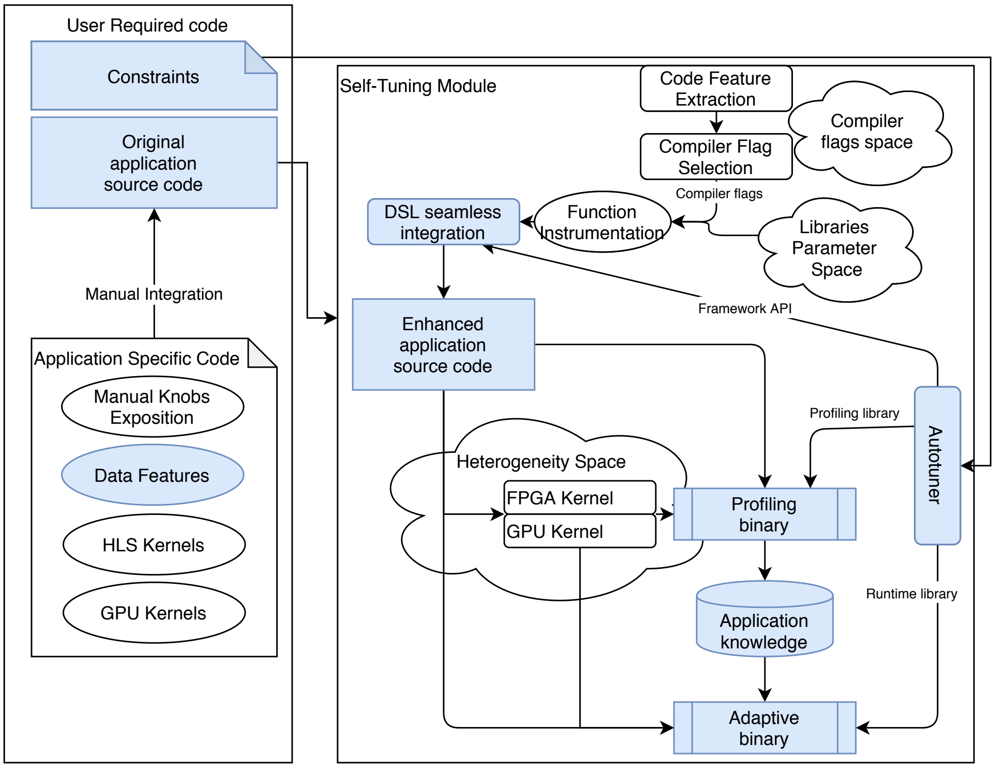


Figure 6.1: *Framework techniques used in this chapter.*

This problem has already been addressed in literature, and a solution with a scalable algorithm has been proposed in [106, 107]. We consider this work as the starting point for this chapter. In detail, the scalable algorithm uses the probability distribution of the travel times for the individual edges to estimate the distribution of the total travel times using Monte Carlo simulations. This algorithm is already integrated into an experimental server-side routing service, which is deployed into an HPC center, to offer optimal performance for a large number of requests, such as in the smart city context. There the PTDR algorithm used simulates a large number of vehicles traveling on a determined path at a definite hour of departure. The speed of the vehicles is sampled from a speed profile, which is a speed probability distribution associated with the edge of the graph. With this approach, the number of samples used in the Monte Carlo simulation is a parameter that strongly affects the accuracy of the prediction, as well as the computational effort. Since the system is designed to serve a large number of requests, small changes in the workload required to serve a single request can affect the overall efficiency. The original version [106] is based on a worst-case tuning of the required number of samples, without any capability of adapting to the road or the starting time. A reactive approach like the one suggested in [108] is not viable, thanks to its high overhead, we propose a methodology to proactively select the number of samples to use in the Monte Carlo simulation required for the PTDR algorithm.

In the context of the global framework, this chapter introduces the concept of data features. We define as data feature some characteristics of the input that can be captured with a quick function before the run of the kernel and that can drive the run to obtain benefits from the extra-functional point of view. Figure 6.1 shows the involved components, which in this circumstance are only the autotuning framework and some custom written functionalities to extract the data features. These functions are then integrated, together with the autotuner, with the DSL seamless integration process to enforce the separation of concern between the functional and the extra-functional code.

In particular, the contributions of this chapter can be summarised as follows:

- A methodology has been proposed to proactively self-adapt the PTDR algorithm presented in [106, 107] to the input data;
- A probabilistic error model has been studied to correlate the characteristics of the input data to the number of samples used in the Monte Carlo simulation;
- An aspect-oriented programming language has been used to separate the functional version of the application from the adaptivity layer.

6.2 Background

Many formulations of the problem of determining the optimal path in a stochastic time-dependent graph have been proposed [109]. The starting point of this work is closest to the *Shortest-path problem with on-time arrival reliability* (SPOTAR) formulation. This is a variant of the *Stochastic on-time arrival* (SOTA) problem, which has a practical solution as shown in [110]. The objective of these algorithms is to maximize the probability of arriving within a time budget and can handle optimal routing in a stochastic network. However, they are not able to manage time-dependent solutions. In [109] there is a practical result for a time-dependent SOTA approach. [111] approach the SPOTAR problem from the theoretical point of view, and suggests a way to extend it with time dependency. Other works show different theoretical approaches and practical application of the SOTA problem [110, 112–114]. In particular, the last work [113] presents a solution for the SPOTAR problem as a heuristic based on a policy-based SOTA approach. However, the authors assume the network as time-invariant, which is not true if we consider long travels. Moreover, this solution is not usable in on-line systems since the scalability to real-world graph is a concerning issue.

This chapter proposes an approach that is built on top of [106, 107], where an approximate solution of the time-dependent variant of the SPOTAR problem based on Monte Carlo simulations is proposed. Our approach is based on the k-shortest paths algorithm [115–117] to determine the paths to use for the PTDR estimation. This separation allows us to implement the whole approach in an online HPC system that can provide adaptive routing in real-time. Since the PTDR is based on Monte Carlo techniques, we can find in literature some methodologies to improve its efficiency. As reported in [118], there are two main ways to improve it. The first one is to improve the sampling efficiency, the second is to target the sampling convergence. However, in both cases, the optimal solution is reached by exploiting the iterative nature of the Monte Carlo simulation. Indeed, several techniques have been proposed to determine which is the best next sample to maximize the gathered knowledge [118, 119] thus improving the sampling efficiency. However, in the implementation under analysis, this approach is not valid since it requires analyzing the result at each sampling iteration, while we want to exploit the parallelism of the underlying HPC architecture [107] that excludes any iterative approach to the Monte Carlo. For the same reason, we have to discard also the approaches that require a statistical property evaluation after every iteration. For example, [108] check the error acceptability after every sample, which is not acceptable for our approach. Both approaches would introduce too much overhead. As already analyzed in [106], the specific problem requires that the number of samples is chosen a priori in a proactive rather than in a reactive manner.

A solution closer to the proposed one has been envisioned in [120], with a two-step approach. The authors suggest having a first shot with a reduced number of samples to provide an initial approximate solution as fast as possible, and then to refine it to the required accuracy in successive iterations. In our context, this idea suffers from two big weaknesses. First, the approximate result is suitable for scientific workflows where it can be used to trigger the next stage of computations, which is not our case. Second, in the iterative phase, it uses a reactive approach that we already discussed is not ideal in our case, since we require a proactive solution in this specific problem to exploit HPC capabilities.

6.3 Monte Carlo Approach for Probabilistic Time-Dependent Routing

Many theoretical formulations and several algorithms have been developed to solve the problem of computing the distribution of the travel times of a road [109]. In this chapter, we consider a *path-based* approach (*SPOTAR*) where the paths are known *a-priori* (since they are evaluated in the previous K alternative paths step) and the goal is to evaluate the travel-time distributions for each one of those paths [121].

The complete traffic navigator application pipeline is shown in Figure 6.2. The focus of this chapter is on the efficient estimation of the arrival time distribution (*PTDR - Probabilistic Time-Dependent Routing* phase). To provide a global view of the traffic navigator pipeline, the three main steps of the application can be described as follows:

1. Determinate the K-alternative paths. In this scenario getting the shortest path is not sufficient when no traffic information is inserted in the road computation. For this reason, we need to evaluate more than one road using a K-shortest path with limited overlap algorithm [115–117]. This step is out of the scope of this thesis.

2. For every path, evaluate the travel time using the PTDR module. The exact solution of the travel time estimation has exponential complexity, which is approximated with the Monte Carlo approach proposed in [106]. This module is the focus of this chapter, which aims at optimizing the Monte Carlo simulation used in the PTDR algorithm.

3. This last step gathers all the predicted times, and reorders all the measured paths, according to the timing distributions and user preference [122].

The need for performance pushes us to implement this pipeline in an HPC system. Moreover, we focus on improving the performances by not doing useless computation, following the approximate computing paradigm. This approach is designed to be used in an online system to serve a large volume of routing requests.

The definition of a probabilistic road network is inspired by the definition of the stochastic time-dependent network described by Miller-Hooks [121]. The only difference is in the conception of the segment travel times,

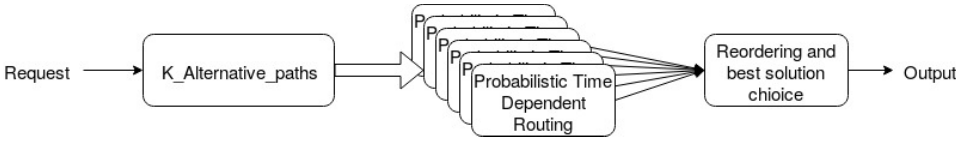

Request
K_Alternative_paths
Probabilistic Time Dependent Routing
Reordering and best solution chioice
Output

Figure 6.2: *The complete navigation pipeline.*

Source: *[123]*

which we substituted with the distribution of the speed probability (*speed profile*) at a given time of departure within a week. Formally, we define the network as follows.

We define $G = (V, E)$ as a well connected, directed, and weighted graph, where V is the set of vertices and E is the set of edges. Each vertex represents a crossroad or some point of interest with a geospatial property of the road. Each edge represents the individual road segments between the points of interest. Every path selected by the K-Alternative paths algorithm can be formally represented as a vector of edges $S = (s_1, s_2, \ldots, s_n)$, where $S_p \subseteq E$ and n is the number of segments in the path.

We are interested in finding a realistic estimation of the travel time θ as $\hat{\theta}_{S,t,P_S}$ where S is the given path, t is the time of departure, and P_S are the probabilistic speed profiles of all the segments of the given path S. More in detail, we divide the possible departure time into time windows $T = \{t : t = n \cdot \phi, n \in \mathbb{N}\}$ [124], where the length of the interval ϕ is determined by input data. $t \in T$ is a departure time within this set. P is the set of probabilistic speed profiles, given for each edge $e \in E$, where $P_S \subseteq P$. Each speed profile $p \in P$ is composed of a set of discrete speed values each of them with an assigned probability. These speed values are derived from historical traffic data and their quantity depends on how they were extracted from the data. The minimum speed value represents the speed when the road is congested, while the highest speed is the free-flow speed.

In this work, we consider a total time frame of *one week* where every interval lasts 15 minutes ($\phi = 900s$). This allows reflecting traffic variations at different hours across the day, managing differently all the days of the week. Extending the time frame, it is possible to consider other different factors, such as the seasons (that could influence the travel time having different wheater) or holidays. Every segment has 4 possible speed values, which are created according to the characteristics of the historical profiled data collected on the considered road.

Moving to the SPOTAR problem, we are not interested in finding one

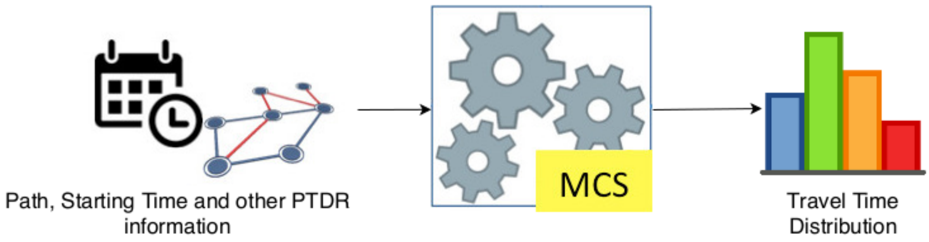


Figure 6.3: *The original approach for PTDR routing based on Monte Carlo simulations to derive the travel time distribution.*

Source: *[123]*

single precise travel time value θ, but we need to evaluate the probability distribution of the arrival time. Given the problem formalized above, we can estimate the travel time distribution traversing all the segments that constitute a path considering the distribution of their speed profiles. In particular, we can build a tree where every layer is a segment in the selected path, as done in [106]. The root of the tree is the starting segment of the trip, while the leaves are the end segment. Each node in the tree has l children, one for each value of the speed probability for that segment. The depth of the tree corresponds to the number of segments that build the path $|S|$. Each edge in the tree has a discrete speed value, the probability of that speed being the correct one and the length of the considered segment. A depth-first-search (DFS) can be used to evaluate the travel time, where at every level we select an arbitrary node in the tree. Then, after the traversal, we can build the travel time as the sum of the time spent in all segments, evaluated as length/speed, where the probability of that travel time is the product of all the visited edges probabilities. A single traversal represents a single trip, done by a car along the entire path. To build the exact probability distribution we need to perform an exhaustive search over all the possible nodes in the tree, from the root to the edges. This cannot be done, since it scales exponentially with the number of segments in the path.

However, it is possible to use a Monte Carlo approach to perform this task. If we generate a large number of random tree traversals, we can build a distribution of the travel times using the results of these traversals, thus building an approximate distribution that estimates the real one. We define the final distribution obtained with Monte Carlo simulation as $MCS(x, i)$. This distribution is a collection of θ values $(\theta_1 ... \theta_x)$. In the definition x is the number of random tree traversals, and i is the input set of the $\hat{\theta}$ function (i.e. S, t, P_S).

These distributions usually have a long-tailed shape, because of properties of the traffic such as accidents, which are rare events but have a strong impact on the travel time. For this reason, to estimate correctly the distribution of the travel times, large numbers of samples (x) are needed. As we already mentioned, we cannot rely on run-time stability analysis of the Monte Carlo simulation, since we target parallel architectures we need that every sample θ of the Monte Carlo simulation, which means each tree traversal, has to be independent of the others. However, we need to know beforehand the number of samples θ required to build the probability distribution, to efficiently exploit the parallelism.

To summarise, Figure 6.3 represent the original PTDR algorithm. All the required data (such as starting and ending point and starting time) are given to the Monte Carlo simulation, which can provide the probability distribution for the given route thanks to a fixed a-priori number of traversals.

6.4 The Proposed Approach

The considered Monte Carlo simulation uses a fixed number of samples x in all the performed run. This is the conventional approach, where this number is selected with worst-case analysis. Indeed, this is the lowest number of samples required to guarantee a target accuracy [107]. Here we present a technique that we propose, which allows selecting at runtime the number of samples needed for the Monte Carlo simulation to guarantee the accuracy according to characteristics of the input data.

However, before moving to the proposed methodology, we want to properly contextualize the problem. Even if we are trying to build the travel time distribution, what we are really interested in is to know a value, τ_i, that guarantees that the travel time will be within that value: $P(\theta < \tau_i) \geq y$ where i is the input set of the travel-time function and y is the probability value. This value, τ_i, is the output of the PTDR. To enable our approximation, we need to characterize this value with an additional property, y, where y is the probability that the travel time will be lower than τ. The resulting formulation of this property is $\tau_{i,y}$.

With the Monte Carlo simulation, we can evaluate the value of $\tau_{i,y}$ using x samples as follows $\hat{\tau}^x_{i,y} = MCS(x, i, y)$. In particular, the value $\hat{\tau}^x_{i,y}$ is obtained selecting the y-th percentile of the distribution obtained from the Monte Carlo simulation on a finite number of samples (i.e. if $y = 95\%$ then $\hat{\tau}^x_{i,y}$ is the 95^{th} percentile of the distribution).

With the proposed approach, we want to minimize the execution time of the MCS, while minimizing the prediction error defined as $error^x_{i,y} = \frac{|\tau_{i,y} - \hat{\tau}^x_{i,y}|}{\tau_{i,y}}$. The target problem can be expressed as follows:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & cpu\_time^x_i \\ \text{subject to} \quad & error^x_{i,y} \leq \epsilon \end{aligned} \tag{6.1}$$

where ϵ is the maximum tolerated error that can be done in the computation. We want to link this error to the output of the MCS, in particular to the desired percentile of the predicted travel time. This allows us to abstract from the actual path. Since there is a strong correlation between the execution time and the used number of samples x, it is possible to simplify the problem as a minimization of the number of samples x instead of cpu_time Considering the properties of the Monte Carlo approach, we can derive that $\tau_{i,y} \equiv \hat{\tau}^\infty_{i,y}$, where $\hat{\tau}^\infty_{i,y}$ is the output of the MCS function when evaluated with an infinite number of samples. Thus, we can rewrite the error as

$$error^x_{i,y} = \frac{|\hat{\tau}^\infty_{i,y} - \hat{\tau}^x_{i,y}|}{\hat{\tau}^\infty_{i,y}} \tag{6.2}$$

Another property of Monte Carlo is that the value $\hat{\tau}^x_{i,y}$ is a random variable, asymptotically normally distributed with mean $\mu_{\hat{\tau}^x_{i,y}}$ and standard deviation $\sigma_{\hat{\tau}^x_{i,y}}$ [125]. Then, thanks to the central limit theorem [126], if we consider a number of samples high enough, the average value does not depend on the number of Monte Carlo simulations. Moreover, the standard deviation decreases constantly while we increase the number of samples used in the Monte Carlo simulation. Given all of these properties, it is possible to define the error as a random variable characterized by a normal distribution with mean 0 and a standard deviation $\sigma_{\hat{\tau}^x_{i,y}} / \mu_{\hat{\tau}^x_{i,y}}$. In the following, we refer to the standard deviation of the error as $\nu_{\hat{\tau}^x_{i,y}} = \frac{\sigma_{\hat{\tau}^x_{i,y}}}{\mu_{\hat{\tau}^x_{i,y}}}$. This is defined as the coefficient of variation (relative standard deviation) of the result of the Monte Carlo simulation.

However, given the probabilistic nature of the problem, it is impossible to guarantee that the error will always be below ϵ. On the other hand, it is possible to relax the problem by introducing the concept of confidence interval (CI) on the error. In particular, thanks to the normal distribution of the error, we can correlate the selected confidence interval with the expected error:

$$P(error^x_{i,y} \leq \epsilon) \geq CI \Longrightarrow \hat{error}^x_{i,y} \leq n(CI) \times \nu_{\hat{\tau}^x_{i,y}} \leq \epsilon \tag{6.3}$$

where $n(CI)$ is a value that express the confidence level. We derived the confidence level from the 1-3 σ-intervals of the normal distribution, so n(68%)=1, n(95%)=2 and n(99.7%)=3. Thus, by decreasing the number of Monte Carlo simulations used to derive $\hat{\tau}^x_{i,y}$, on the one hand, we reduce the application execution time, but on the other hand we are also affecting the accuracy, which can be seen from the increase of the coefficient of variation $\nu_{\hat{\tau}^x_{i,y}}$.

An additional problem comes from the input dependency of $\hat{\tau}^x_{i,y}$. This means that, if the input is unknown (such as an unknown path) it is not possible to predict the possible Monte Carlo error, using only the number of samples as prediction variable. To deal with this, we need to find a feature u_i of the inputs i that can be used to quickly evaluate the input and find the number of samples necessary to contain the error below the threshold ϵ. In this way, we can evaluate the error using u_i instead of the real value i thus transforming the original problem into

$$error^x_{i,y} \leq n(CI) \times \nu_{\hat{\tau}^x_{u_i,y}}. \tag{6.4}$$

We found this feature u_i , which we called *unpredictability*, since it represents a set of characteristics of the inputs i (road, starting time,...) that provides information about how complex is the prediction of $\tau_{i,y}$. Therefore this unpredictability is also related to the number of samples required to ensure a confidence level on the error The details on the unpredictability feature and the function used to evaluate it are presented in Section 6.4.1.

With this new formulation of the problem, the error is no more directly related to the specific input set i but only to the identified feature u_i. Thus, the number of samples needed to satisfy the constraint on the confidence can be extracted by $\nu_{\hat{\tau}^x_{u_i,y}} \leq \frac{\epsilon}{n(CI)}$. The missing step to finalize the methodology is to determine the correlation between the unpredictability function and the output error. This has been done running a profiling phase on a representative set of inputs, to extract the values of $\hat{\nu}_{\hat{\tau}^x_{u_i,y}}$ that we then used to evaluate the correlation. The details on this profiling campaign are presented in Section 6.4.2

To summarise, Figure 6.4 shows the proposed methodology. We add an adaptivity layer on top of the Monte Carlo simulation to quickly determine, at runtime, from the characteristics of the actual input, the number of samples required to satisfy the accuracy requirement. This is possible thanks to the feature-extraction procedure that estimates the unpredictability given the input data. The dynamic autotuner combines the data feature with the profiled knowledge plus the extra-functional requirements to correctly configure the Monte Carlo simulation before its run.

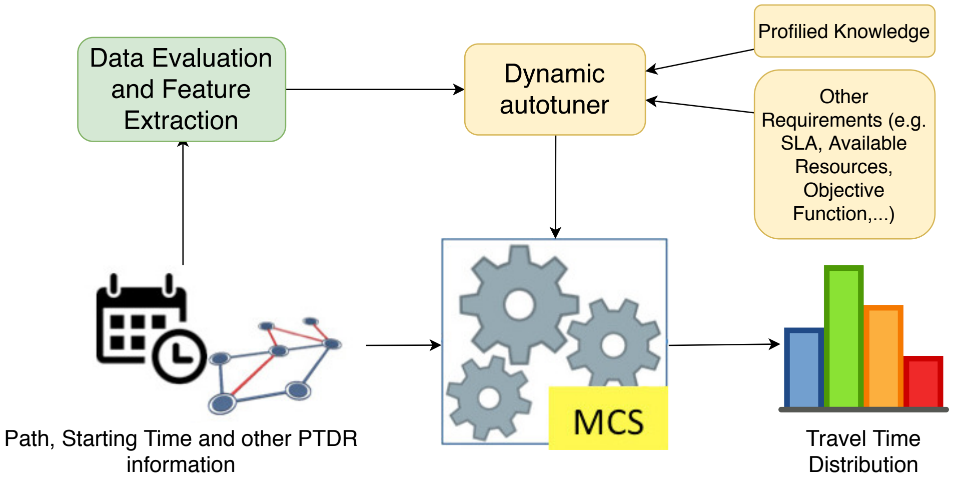

Data Evaluation and Feature Extraction
Dynamic autotuner
Profilied Knowledge
Other Requirements (e.g. SLA, Available Resources, Objective Function,...)
MCS
Path, Starting Time and other PTDR information
Travel Time Distribution

Figure 6.4: *The proposed adaptive approach for PTDR routing. We can notice that the Monte Carlo simulation is now driven not only by the input but also from the Dynamic Autotuner.*

***Source**: [123]*

6.4.1 Unpredictability Feature

The evaluation of the input, i.e. the extraction of the data feature, is an operation that has to be done at runtime, before the call to the Monte Carlo function. For this reason, this operation has to be very quick and lightweight, otherwise, the speedup of the computation in the Monte Carlo phase will be overshadowed by the cost of the extraction overhead, and the whole approach would be meaningless

From our experiments, we discovered that a measure of the unpredictability of the considered path can be extracted from a simple statistical property of a small set of travel times θ evaluated with a quick Monte Carlo simulation on a small number of fragments. The property we are referring to is the coefficient of variation. Intuitively, if the results of this quick Monte Carlo are very spread out, then the path is going to be difficult to predict, thus we need a high number of samples to build an accurate estimation of the distribution. On the other hand, if this coefficient of variation is very small, the path is probably very easy to predict.

The unpredictability function is defined as $u_i = \sigma^x_{\theta_i}/\mu^x_{\theta_i}$ where σ_{θ_i} and μ_{θ_i} are the standard deviation and the average of a set of travel times evaluated with a MCS done with the minimum number x of samples allowed at runtime. We decided to evaluate the unpredictability function with the first set of Monte Carlo samples to reduce the overhead due to the data feature extraction. The data feature extraction function will use the result of the first run to evaluate if more samples are needed, and how many.

To validate the unpredictability function as a proxy of the input, we use the Spearman correlation test [127] between the unpredictability function and the value of $\nu_{\hat{\tau}^x_{i,y}}$ used in the calculation of the expected error for different values of x and y over a wide range of inputs sets i. In all cases, the correlation values were larger than 0.918 showing a p-value equal to 0. These results corroborate our hypothesis, with the p-values showing that the results are statistically significant.

6.4.2 Error Prediction Function

The last missing step is to build the knowledge on the expected error given the unpredictability. To achieve this result we need to extract $\hat{\nu}_{\hat{\tau}^x_{u_i,y}}$ from profiled data. Our profiling campaign is done by running, several times for each configuration, the Monte Carlo simulation. We decided to use, for the number of samples, values ranging from 100 to 3000. 100 is the minimum number of samples required to have an estimation of the percentiles of the distribution, while 3000 is the number of samples that are good enough to satisfy the worst-case condition for the current Monte Carlo simulation, as shown in the previous work [128]. 2 more intermediate steps have been inserted between the minimum and the maximum value. These two levels, 300 and 1000, have been derived considering that the error of the Monte Carlo decreases as $1/\sqrt{n}$ [129]. In this way, at each sampling level, the error is almost halved.

Every set of the Monte Carlo simulation is run with the same configuration (in terms of number of samples) on a large set of inputs i (i.e. roads, starting time,...). From every run we extract $\nu_{\hat{\tau}^x_{u_i,y}}$ and u_i. We decide to use the *quantile regression* on these collected data [130] as predictor $\hat{\nu}_{\hat{\tau}^x_{u_i,y}}$. Since we are not interested in predicting the average of the final result, but we need the predictor in the inequality formula $\hat{\nu}_{\tau^x_{u_i,y}} \leq \frac{\epsilon}{n(CI)}$, the quantile regression enhance the robustness of the model. Any quantile value higher than the 50^{th} (the purely linear regression) guarantees stronger robustness for the considered inequality. The selection of the quantile is an additional parameter that can be tuned in the context of selecting the desired trade-off between robustness and performance.

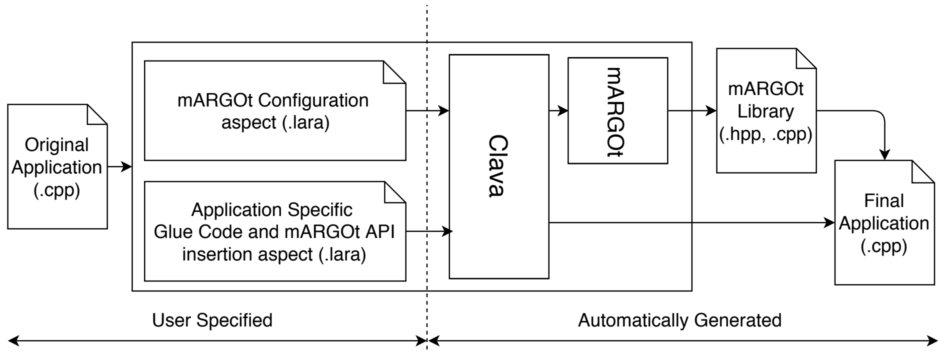

Original Application (.cpp)
mARGOt Configuration aspect (.lara)
Application Specific Glue Code and mARGOt API insertion aspect (.lara)
Clava
mARGOt
mARGOt Library (.hpp, .cpp)
Final Application (.cpp)
User Specified
Automatically Generated

Figure 6.5: *Integration flow outlining the two main LARA aspects and related actions: original code enrichment and autotuner configuration.*

***Source**: [123]*

6.5 Integration Flow

The previous section presented the proposed methodology from the theoretical perspective, highlighting the trade-offs between execution time and elaboration error. In this section, we focus on the application developer perspective by proposing an integration flow that is capable of enhancing the target application with limited effort. This integration flow enforces the separation between the functional and extra-functional concerns using an Aspect-Oriented Programming Language to inject the code needed to introduce the adaptivity layer in the target source code.

The adaptivity layer consists of the insertion of the mARGOt autotuner, which can manage the adaptivity concepts presented in Section 12.3. The target application is then transformed as shown in Figure 6.4. As we can see, the autotuner is in charge of selecting the number of samples that minimize the execution time, with a constraint on the error. This selection is done evaluating the unpredictability (data evaluation and feature extraction step) and using profiled knowledge, obtained through the procedure described in Section 6.6.1.

We want to hide all this code manipulation from the end-user, thus we adopt LARA [12] as language to describe the strategies and its Clava [1] compiler for the source to source code transformation. LARA is a Domain Specific Language inspired by Aspect-Oriented Programming. It allows the user to capture specific points in the code and then analyze and act on those points. This approach creates a new version of the application without

[1]Project repository: `https://github.com/specs-feup/clava`

```
// Load data
Routing::MCSimulation mc(edgesPath, profilePath);
auto run_result = mc.RunMonteCarloSimulation(samples, startTime);
ResultStats stats(run_result);
Routing::Data::WriteResultSingle(run_result, outputFile);
return 0;
```

Listing 6.1: *Original source code before integrating the adaptivity layer.*

changing the original source code, thus separating the functional concerns from the optimizations specified in the LARA aspect. Clava is a C/C++ source-to-source compiler of the LARA framework. This compiler can execute the LARA aspects and perform the code transformations described in the aspect on the original code, thus creating a new version.

In this work, we use Clava to perform two tasks:

1. insert the autotuner into the original source code.

2. configure the autotuner according to the requirements.

Figure 6.5 shows this process, from the original source code to the final application. It also highlights the two main LARA aspects used. We report in Listings 5.1–4 the original code, the two aspects, and the final code.

In particular, the code in Listing 6.2 is the LARA aspect needed to configure mARGOt. This code produces the adaptivity layer tailored to the application requirements. In lines 9–16, we define the software knob (*num_samples*), the data feature (*unpredictability*), the metrics (*error*) and the goal (i.e. the Service Level Agreement, $error < 3\%$). The goal in mARGOt is a condition that is needed when defining the optimization problem. Once all of this has been defined, it is possible to define the multi-objective constrained optimization problem, which has to be managed by the autotuner (lines 18–23). In mARGOt optimization problems are called states (line 19). The constraints are generated in line 23, where the number represents the priority of the constraint. In the case of more than one constraint, if the runtime is unable to satisfy both of them, it will relax the low priority one. Lines 21–22 define the objective function. In this case, the objective is the minimization of the number of samples so the aspect describes it as a linear combination (line 21) of the *num_samples* knob (line 22). The number in line 22 is the linear coefficient that has to be used in the linear combination for the considered knob. Finally, line 26 builds the LARA internal structure *margoCodeGen_ptdrMonteCarlo* which is needed by Clava to create the mARGOt configuration file and code generator.

```
aspectdef McConfig
  /* Generated Code Structure*/
  output codegen end

  /* mARGOt configuration */
  var config = new MargotConfig();
  var travel = config.newBlock('ptdrMonteCarlo');

  /* knobs */
  ptdrMonteCarlo.addKnob('num_samples', 'samples', 'int');
  /* data features */
  ptdrMonteCarlo.addDataFeature('unpredictability', 'float',
      MargotValidity.GE);
  /* metrics */
  ptdrMonteCarlo.addMetric('error', 'float');
  /* goals */
  ptdrMonteCarlo.addMetricGoal('my_error_goal', MargotCFun.LE, 0.03,
      'error');

  /* optimization problem */
  var problem = ptdrMonteCarlo.newState('problem');
  problem.setStarting(true);
  problem.setMinimizeCombination(MargotCombination.LINEAR);
  problem.minimizeKnob('num_samples', 1.0);
  problem.subjectTo('my_error_goal', 1);

  /* creation of the mARGOT code generator for the following code
      enhancement (McCodegen aspect) */
  margoCodeGen_ptdrMonteCarlo = MargotCodeGen.fromConfig(config, '
      ptdrMonteCarlo');
end
```

Listing 6.2: *LARA aspect for configuring the mARGOt autotuner.*

```
aspectdef McCodegen
  /* Target function, mARGOt code generator from McConfig aspect, # samples for feature extraction */
  input targetName, margoCodeGen_ptdrMonteCarlo, unpredictabilitySamples end

  /* Target function call identification */
  select stmt.call{targetName} end
  apply
    /* Target Code Manipulation */
    /* Add mARGOt Init*/
    margoCodeGen_ptdrMonteCarlo.init($stmt);
    /* add unpredictability code */
    $stmt.insert before UnpredictabilityCode(unpredictabilitySamples);
    /* Add mARGOt Update */
    margoCodeGen_ptdrMonteCarlo.update($stmt);
    /* Add Optimized Call Code */
    $stmt.insert replace OptimizedCall(unpredictabilitySamples);
  end
end

/* Unpredictability extraction code */
codedef UnpredictabilityCode(unpredictabilitySamples) %{
  auto travel_times_feat_new = mc.RunMonteCarloSimulation([[unpredictabilitySamples]], startTime);
  ResultStats feat_stats(travel_times_feat_new, {});
  float unpredictability = feat_stats.variationCoeff;
}% end

/* Optimized MonteCarlo call */
codedef OptimizedCall(unpredictabilitySamples) %{
  auto run_result = mc.RunMonteCarloSimulation(samples - [[unpredictabilitySamples]], startTime);
  run_result.insert(run_result.end(), travel_times_feat_new.begin(), travel_times_feat_new.end());
}% end
```

Listing 6.3: *LARA aspect for inserting the application-specific glue code (unpredictability extraction) and the required mARGOt calls.*

```cpp
// Load data
Routing::MCSimulation mc(edgesPath, profilePath);
auto travel_times_feat_new = mc.RunMonteCarloSimulation(100,
    startTime);
ResultStats feat_stats(travel_times_feat_new, {});
float unpredictability = feat_stats.variationCoeff;
if(margot::travel::update(samples, unpredictability)) {
  margot::travel::manager.configuration_applied();
}
auto run_result = mc.RunMonteCarloSimulation(samples - 100, startTime
    );
run_result.insert(run_result.end(), travel_times_feat_new.begin(),
    travel_times_feat_new.end());
ResultStats stats(run_result);
Routing::Data::WriteResultSingle(travel_times_new, outputFile);
return 0;
```

Listing 6.4: *Target source code after the integration of the adaptivity layer.*

The second aspect (shown in Listing 6.3) is the custom aspect that inserts the proposed methodology in the application. It takes as input (line 3) the target function call that we want to tune, the mARGOt code generator produced by the previous aspect (Listing 6.2), and the number of samples needed to evaluate the unpredictability feature. In line 6, we search in the code to find the statement (*stmt*) where there is the Monte Carlo function *call*. This is the target join point that will be manipulated. Lines 7–17 contain the manipulation actions done on the selected join point $stmt$ of the target code. There are two different types of operations: the first one is to integrate the mARGOt calls for the library initialization and to update the software knob (Lines 10 and 14). The second operation is to *insert* the glue code (LARA *codedef*) to evaluate the unpredictability (line 12 and lines 21–25), and to *replace* the original Monte Carlo call with the optimized one that does not repeat the unpredictability samples (line 16 and lines 28–31).

Overall, from the numerical point of view, the usage of LARA allows us to insert the methodology using only 53 lines of code to generate 221 lines of C++ code. However, this is not the main advantage. This approach is valid even with such a small amount of inserted code for three reasons.

1. The user will not have to care about the details of the mARGOt configuration files and its low-level C++ API.

2. The user will not need to provide the same information in different places. There is some mARGOt-specific information that has to be provided in several steps during the integration of the autotuner (such as in the configuration file, when inserting the API, ...). With this

automated insertion, the information has to be provided only once, thus reducing the error probability.

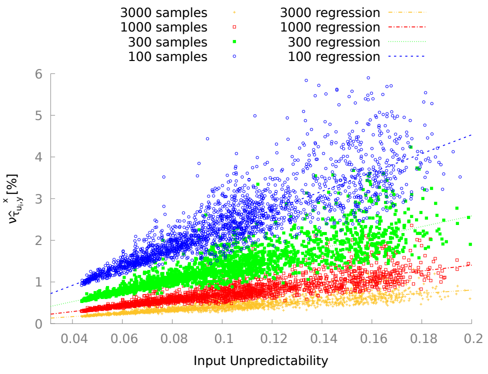
3. This approach allows separation of concerns. All the extra-functional optimization, including the proposed approximation methodology, are kept separate from the original application. In this way, the developer of the original application does not need to be involved in the optimization process.

6.6 Experimental Results

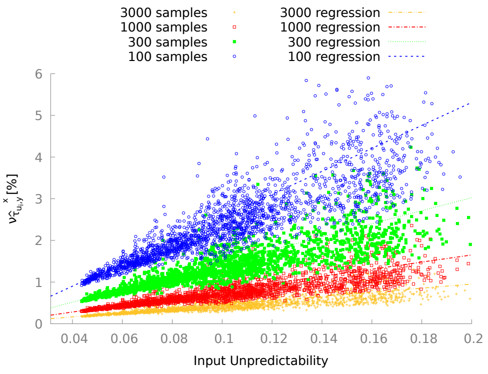
In this section, we will show the results of the proposed methodology applied to the PTDR algorithm. The platform that we used for the experiments consists of several nodes equipped with Intel Xeon E5-2630 V3 CPUs (@2.8 GHz) with 128 GB of DDR4 memory (@1866 MHz) on a dual-channel memory configuration. At first, we analyze the results of the model training, needed to build the model that estimates the expected error (see Section 6.6.1). Then, in Section 6.6.2 we validate the approach by verifying that it satisfies the constraint imposed on the error ϵ. We compare the proposed approach to the original version, which decides the number of samples a priori to satisfy the worst case, in Section 6.6.3. Finally, in Section 6.6.4, we evaluate the overhead introduced, and in Section 6.6.5 we evaluate the optimization impact on the whole process, at system level.

6.6.1 Training the Model

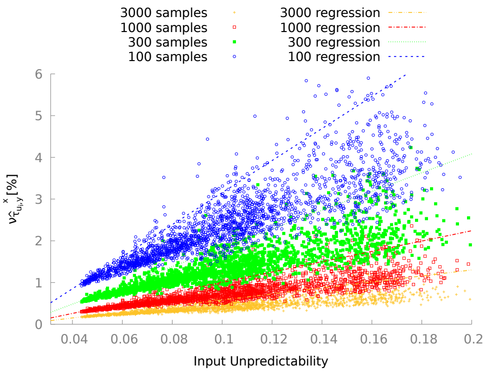
As we already said, before using the proposed approach we need to train the error prediction model. So, the first phase is done off-line, and it consists in training the error model ($e\hat{rror}^{x}_{i,y}$). This is done, as presented in Section 6.4.2, by using a different number of samples. To train the quantile regression model, we profiled data extracted by running random request to the PTDR algorithm on a training set. The training set has been built using 300 different paths across the Czech Republic, all characterized with time-slots. In this way, we are able to consider different speed-profiles for each segment of the paths. All the requests have been made for all the four levels of sampling used by the proposed methodology (i.e. 100, 300, 1000 and 3000, as described in Section 6.4.2). The output of the model training is represented in Figure 6.6. The points in the three plots represent the output of the profiling runs, and are the same across the different images. The lines are the quantile regression lines, which is the model that will be used at runtime. The three sub-figures are different for the value of the quantile



(a) *Quantile regression using the 50th perc.*



(b) *Quantile regression using the 75th perc.*



(c) *Quantile regression using the 95th perc.*

Figure 6.6: *Training of the error model by using different number of samples and quantile regressions. We can notice the different regression lines across the figures.*

Source: *[123]*

used for the linear regression. Figure 6.6a represents the 50th percentile, Figure 6.6b represents the 75th percentile, while Figure 6.6c represents the 95th percentile.

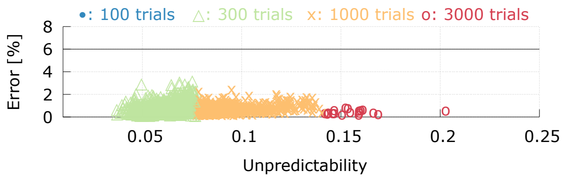
We can see that the three regressions are different. We pass from a more permissive one, on the 50th percentile in Figure 6.6a, where almost half of the points are below the corresponding regression line, to the most conservative one in Figure 6.6c where only a few points are above. We can also notice that the coefficients of the lines of the quantile regression are almost doubled passing from 75th to 95th percentile (e.g. for 100 samples, the coefficients pass from 0.27 to 0.38, while for 3000 samples they pass from 0.049 to 0.071).

With this final step, we are now ready to test the proposed methodology. The models are ready and can be integrated into the dynamic autotuner, which will be in charge of selecting at run time the most appropriate configuration that will satisfy the error constraint. In Section 6.6.2 we will demonstrate the effectiveness of this methodology.

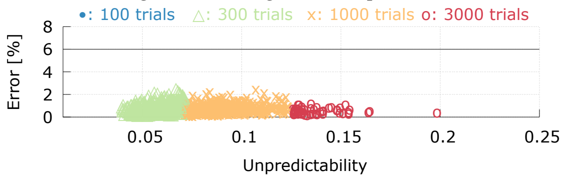
6.6.2 Validation Results

The set of results presented in this section aims at demonstrating the validity of the proposed approach, more in detail that the proposed way to dynamically select the number of samples is still able to satisfy the constraint on the target error. To do this, we randomly generated 1500 requests to the enhanced PTDR module. The routes used in this phase are different from the ones used in the training phase of the model, even if all of them come from a single dataset of routes in the Czech Republic. The approach is validated using three different quantile regressions (on 50^{th}, 75^{th} and 95^{th} quantile), two different target errors ϵ (3% and 6%) and a confidence interval (CI) for the error constraint equal to 99% (i.e. $n(99\%)$ = 3). The ground truth of the run is built running the Monte Carlo simulation with 1 million samples, which are enough to be considered a real estimation of the travel time distribution. Then we build the error as the maximum between different key percentiles of the difference of the run with one million samples against the autotuned run. The selected percentiles are: 5^{th}, 10^{th}, 25^{th}, 50^{th}, 75^{th}, 90^{th} and 95^{th} percentile. The error is then built as $max_{percentiles}(|MCS(x,i,y) - MCS(1000000,i,y)|)$.

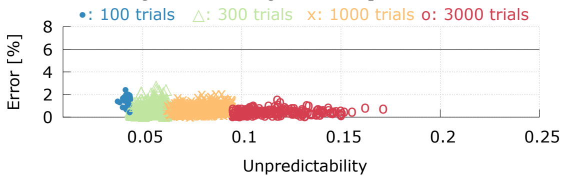
The output of this experiment is reported in Figure 6.7 and Figure 6.8. The first image targets an error constraint of 3%, while the second targets ϵ equal to 6%. On the x-axis, we have plotted the unpredictability feature of the path, on the y-axis the error. Every dot in the plot reports the result of



(a) *Quantile regression using the 50th perc.*



(b) *Quantile regression using the 75th perc.*

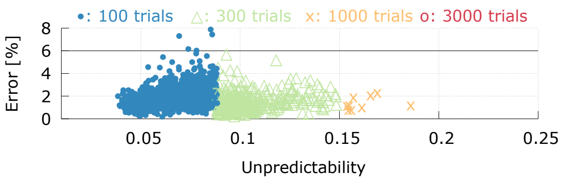


(c) *Quantile regression using the 95th perc.*

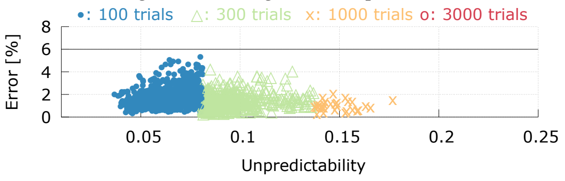
Figure 6.7: *Validation of the proposed approach by using 3% as target error and different percentiles for the quantile regression.*

Source: *[123]*

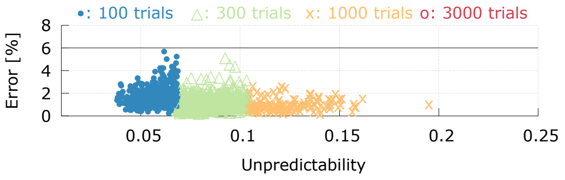
a PTDR request. The shape depends on the number of samples used in the actual Monte Carlo simulation. As we can see, usually the error is smaller than the target error. This was an expected result since the more conservative is the quantile regression, the less are the points that are above the line produced by it. Moreover, we also introduced a Confidence Interval of 99%, and we can notice that this confidence is respected in the results of this run. Moreover, we can notice comparing the different quantile regressions that the unpredictability threshold values shift from right to left. For example, if we look at Figure 6.7 ($\epsilon = 3\%$) we can notice that the maximum unpredictability value for having 300 samples moves from 0.075 to less than 0.06 respectively when considering the quantile regression from the 50th percentile, up to the 95th quantile. The same thing happens in Figure 6.8 ($\epsilon = 6\%$) where the same threshold moves from an unpredictability of 0.15 to 0.14 and 0.11 when using the 50th, the 75th and the 95th as quantile value for the regression. Finally, we can notice the difference in terms of the number of samples used when we change the target ϵ value. Indeed, while for ϵ equal to 3% (Figure 6.7) only a tiny fraction of the runs use



(a) *Quantile regression using the 50th perc.*



(b) *Quantile regression using the 75th perc.*



(c) *Quantile regression using the 95th perc.*

Figure 6.8: *Validation of the proposed approach by using 6% as target error and different percentiles for the quantile regression.*

Source: *[123]*

100 samples and there are a lot of cases where 3000 samples are needed to be used. If we relax the constraint on the error, setting ϵ equal to 6% (Figure 6.8) 3000 samples are never used and in a non-negligible fraction of the runs only 100 samples are required.

6.6.3 Comparative Results with Static Approach

In this section, we will show the advantages obtained by the proposed approach compared to the baseline version [106] where the number of samples is defined *a priori*. To provide a fair comparison, we selected the number of samples of the baseline with the data of the same training dataset.

To obtain the baseline, we need to analyze the cumulative distribution of the expected error distribution of the 4 levels of sampling used in this chapter (i.e. 100, 300, 1000, and 3000). The static value is the minimum sampling level able to pass a defined threshold of the cumulative value before reaching the error constraint value ϵ. This threshold value recalls the *robustness* meaning of the quantile regression used in our approach. For this reason, we will compare the proposed approach with this static tuning at the same percentile level. This means that the percentile used in the static tuned to set the threshold level is going to be the same percentile used in the dynamic approach to set the quantile regression. For example, if we use a quantile regression at 95%, we will compare it with the statically tuned version where the number of samples has been defined looking at the cumulative curve that reaches at least 95% before crossing the target error constraint. Figure 6.9 reports the cumulative distributions that are used in building the static model. If we consider the last vertical line, where the error constraint is set at $\epsilon = 6\%$, the static tuning says that for percentiles between 72^{th} and 98^{th} we need to use 1000 samples, if we want to have the certainty that the error is below $\epsilon = 6\%$ we will need to use 3000 samples, while if we want a percentile smaller than 72^{th} we can use 300 samples. On the other hand, if we look at the first vertical line ($\epsilon = 3\%$) we can select 3000 samples within the percentile interval 65^{th}-97^{th}, 1000 samples for percentile values smaller than 72^{th} (down to 5^{th}), while we need more than 3000 samples if the requested accuracy is very tight and we need a percentile larger than 97^{th}.

In Table 6.1 we report the comparison between the proposed adaptive technique and the original version (*baseline*). For the baseline, the number of samples is evaluated with the previously described autotuning approach. In particular, Table 6.1 presents the average number of samples (first number) and the gain respect to the baseline (second number) for different percentiles used in building the predictive models and different values of error constraint ϵ. Again, we used as dataset some randomly selected pairs of Czech Republic routes and starting times different from those used for the training. The route is chosen with absolute randomness, while the starting time is selected using a more realistic distribution of when people usually start driving, taken from [131, 132].

In all the considered cases, the proposed approach reduces the number

Table 6.1: *Average number of samples for the validation set using different quantile regression values (columns) and different error constraints. The results are reported for the* baseline *and proposed* adaptive *versions.*

		Average Number of Samples		
ϵ		50^{th} perc.	75^{th} perc.	95^{th} perc.
3%	baseline	1000	3000	3000
	adaptive	632 (-36%)	754 (-74%)	1131 (-62%)
6%	baseline	300	1000	1000
	adaptive	153 (-49%)	186 (-81%)	283 (-71%)

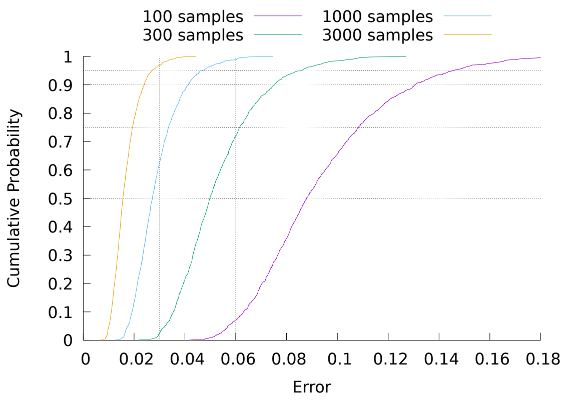


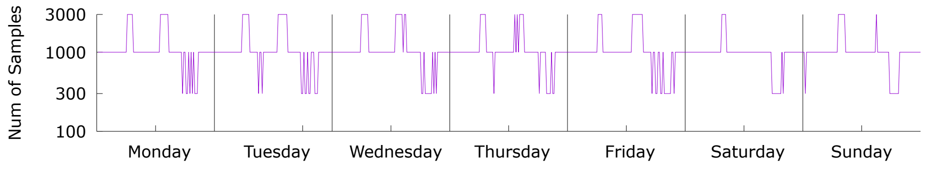
Figure 6.9: *Cumulative distribution of the error by using different numbers of samples over the training set.*

***Source**: [123]*

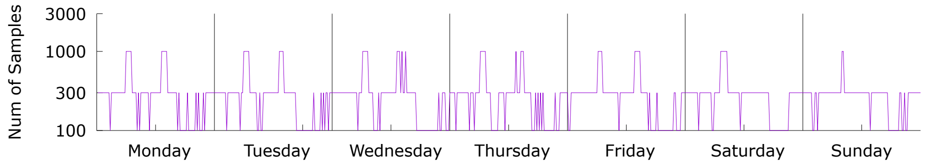
of samples with a minimum reduction of 36% and a maximum value of 81%. As expected, the average number of samples used by the dynamically tuned approach is lower when we relax the error constraint (i.e. 6%). The lower gain for the configurations using the 50th percentile is due to the fact that in this experiment the baseline requires a lower number of samples with respect to the latter cases (i.e. 1000 vs 3000 for $\epsilon = 3\%$ and 300 vs 1000 for $\epsilon = 6\%$). Focusing on the absolute numbers, it is evident that the least accurate percentile (50th) is the experiment with the lowest number of samples used. This reduction in samples is visible also in the execution time of the application since there is a linear dependency of the Monte Carlo simulation from the number of samples required to run it. There is but a small overhead introduced by the proposed methodology that is analyzed in Section 6.6.4. In particular, we observed an execution time speed-up between 1.5x and 5.1x.

We want to present the benefits of the proposed methodology from another perspective. Figure 6.10 shows the number of samples required by the proposed methodology when we perform the PTDR request on the same road, at different starting times. In particular, we perform this request every 15 minutes since that is the smaller time granularity (ϕ) we have for the database containing the speed profiles. The two plots (a) and (b) have been generated using targeting errors of respectively 3% and 6% while for both experiments we used a quantile regression on the 75th percentile.

Just by looking at the number of samples requested by the adaptive version of the Monte Carlo simulation, it is possible to recognize well-known



(a) *Error constraint ϵ=3%*



(b) *Error constraint ϵ=6%*

Figure 6.10: *Number of samples selected by the proposed adaptive method when the same request is performed every 15 minutes during the entire week.*

***Source**: [123]*

traffic behaviors in both plots. We can notice that during the weekdays the plots are characterized by two peaks (the first between 7 and 8 am, and the second between 4 and 5 pm) where the unpredictability forces the adaptive approach to use the highest value of samples. On the other hand, we can notice that in the evening the situation is more moderate and the path has a predictable behavior, that allows reducing the samples required for the requests that happen in that timeframe. This dynamic behavior, captured by the proposed methodology, cannot be exploited by using the original (baseline) version. Indeed, the original version must be tuned by considering the worst case, that is 3000 samples for the experiment in Figure 6.10(a) ($\epsilon = 3\%$) and 1000 samples for the experiment in Figure 6.10(b). In both cases, the static tuning requests a larger number of samples than the proposed technique, that instead can know when the worst case is happening and use the accurate solution at that moment (e.g. during the traffic peaks), while it can approximate the solution using fewer samples when the road is more predictable.

On the other hand, if we consider to statically tune to the *average* case (i.e. 1000 samples for the experiment in Figure 6.10(a) ($\epsilon = 3\%$) and 300 samples for the experiment in Figure 6.10(b)) we are losing the accuracy of the request at the moment where it is most needed, and wasting computational power in other moments of the day. Indeed, there are still many sampling reduction possibilities in predictable moments that will not be

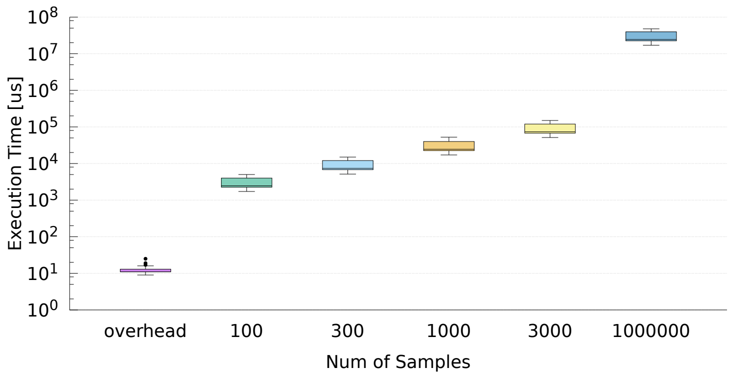


Figure 6.11: *Evaluation of the execution time overhead due to the additional code for the proposed method with respect to the target Monte Carlo simulation by varying the number of samples.*

***Source**: [123]*

captured, and as we already said, the prediction will not be able to satisfy the output quality constraint during the most critical periods. Finally, a static policy is also sub-optimal since the unpredictability does not capture only the starting time, but also other path characteristics such as if the road is an urban one or a countryside path, or the length of the road (which is important to know when it is expected to traverse a congested area).

6.6.4 Overhead Analysis

With the proposed methodology, we have inserted an overhead that is to calculate the unpredictability function. In Section 6.5 we described how we reduced the integration overhead from the application developer point of view. This section clarifies the time-overhead introduced to obtain the proposed adaptivity. In particular, as we already said, we insert additional computations to calculate the unpredictability function $\nu_{\hat{\tau}^{x}_{i,y}}$ and to call the autotuner to determine the right number of samples to be used. Note that the 100 Monte Carlo samples, required to extract the data feature, are not and will not be counted as part of the overhead, since they are reused (and thus discounted) in the second Monte Carlo call (see Listing 6.4).

Figure 6.11 shows in boxplot the execution time (y-axis) of the main points of the PTDR algorithm. In particular, the first box shows the overhead introduced by the proposed methodology. The following boxes represent the execution time of the Montecarlo from 100 to 1M samples. The dataset used to obtain this result is the usual validation dataset, built on several paths across the Czech Republic chosen randomly. We expected that the execution time of the Monte Carlo simulation was correlated to the number of samples, and this experiment confirms the hypothesis. The different paths used are between 300 and 800 segments long, and this is the main reason for the variability of the Monte Carlo computing time with a defined number of samples. Looking at the overhead, we can notice that it is almost negligible if we compare it to the time needed to perform the Monte Carlo simulation. Indeed, it is more than two orders of magnitude less than the smaller Monte Carlo simulation with 100 samples. To give a comparison, the overhead cost is comparable to the evaluation of a single sample of the Monte Carlo on a road composed of 200 segments.

6.6.5 System-Level Performance Evaluation

The final experiment that we present in this chapter aims at quantifying, at system level, the benefits of the proposed adaptive method. In this section, we present an analysis that evaluates the efficiency of the adaptive PTDR module when included in the full navigation pipeline presented in Figure 6.2 We used Java Modeling Tools (JMT) [133] to build a performance model of the whole pipeline. JMT is an environment built to perform performance evaluation and workload characterizations based on queuing models [134]. This tool can be used to perform capacity planning simulations and workload characterization to automatically identify possible bottlenecks. In particular, we build the model of the pipeline creating one station for each of the modules of the pipeline and we add a fork-join unit to model the parallel PTDR evaluations of the different paths found in the first station.

The model is shown in Figure 6.12. It has been annotated with the execution times of the different modules, derived by the profiling, considering a value for K (the number of alternative paths to evaluate) equal to 10. The system is modeled to be able to serve a load produced by 100K cars every 2 minutes. This number is due to the application scenario considered: on the one hand, we suppose that self-driving cars are always connected to the route planner, and on the other hand we estimate the number of requests following some studies on the Milan urban area [135, 136]. Here, the population of the considered area is around 4 million people, and every day there are more than 5 Million trips estimated. More than half of them are done with private cars.

Under these conditions and considering the configuration with $\epsilon = 6\%$ and $95^{th} percentile$, we found that the proposed technique allows obtaining

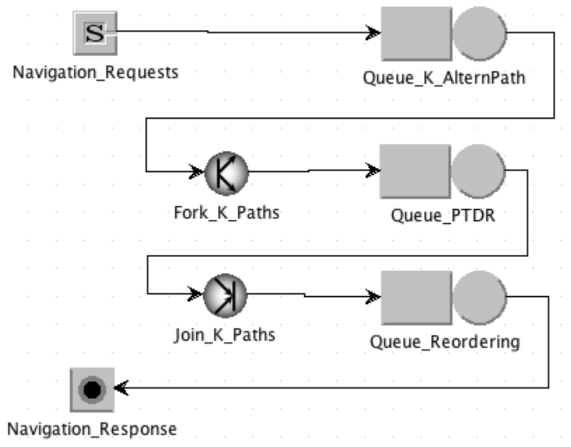


Figure 6.12: *Complete navigation pipeline modeled using JMT.*

Source: *[123]*

a 36% reduction in terms of the number of resources needed to satisfy the target workload. In particular, we have studied 2 cases. The first one considers the number of resources needed to satisfy the steady-state conditions, which means the throughput in terms of input requests satisfied by all the stages. In this case, with the static approach, we need at least 777 computing resources (cores). Among them, 400 cores (52% of the entire set) should be dedicated to PTDR. By applying the proposed technique, only 497 cores are needed, we can reduce to 120 (24% of the entire set) those required for the PTDR stage. The second case considers a more dynamic environment where following the rule of thumb proposed in [137] we keep the average utilization rate of each station below 70%. In this way, the system is not completely utilized and is more prepared to react to a burst of requests without introducing any delay. In this second case, with the static approach, we need 1010 cores to allocate the entire pipeline, and 572 of them (57% of the entire set) should be dedicated to the PTDR stage. By applying the proposed technique, we can reduce the number of cores to 646, and out of them, only 172 (26% of the entire set) are dedicated to the PTDR.

6.7 Summary

In this chapter, we focused on Probabilistic Time-Dependent Routing to show how it is possible to improve computation efficiency, introducing a methodology to reduce the computation when it is deemed unneeded. The methodology can quickly test the input data and extract some features that are used to drive the computation phase. The runtime decision is based on a probabilistic error model, learned offline. We have shown that the proposed approach, by focusing the computation effort where it is needed, can save a large fraction of simulations (between 36% and 81%) compared to static autotuning, since it can capture more information from the current input. We have also shown that considering the entire routing pipeline, the proposed approach can save actual resources, thus making the pipeline cheaper to deploy, without losing the capability of serving a large number of requests.

Finally, we have inserted the whole methodology in the original application with an aspect-oriented language (LARA) that allowed us to enforce the separation of concerns between the functional and the extra-functional requirements, and to ease the methodology integration for the programmer.

The outcome of this work has been published in the journal IEEE Transactions on Emerging Topics in Computing [123].

CHAPTER 7

Demonstrating the benefit of Autotuning in Object Detection context

In this chapter, we will show the potential benefit of autotuning the inference process in the Deep Neural Network (DNN) context, tackling the object detection challenge. We benchmarked different neural networks to find the optimal detector in the COCO 17 well-known database [138], and we demonstrate that there is not a one-fit-all solution. This is even more evident if we also consider the time to solution (i.e. the time required by a network to process an image) as a deciding factor. Indeed, we demonstrate that thanks to a reactive methodology it is possible to respect changing requirements that by using a single network would be violated. Moreover, we believe that a proactive approach could further improve the autotuning approach, allowing to select the best network among the available ones given some characteristics of every single image. However, we were not able to identify a predictor function that would allow this approach. Nonetheless, we believe that this work can be useful as a motivational work for further investigation in this direction.

7.1 Introduction

A lot of progress has been done in the last 10 years in the context of Neural Networks. They have recently been used to solve complex problems such as image classification [139], voice to text [140] or object detection [141]. Since their introduction, they have eclipsed the old methods that were used to perform these tasks. In particular, they have become the de-facto standard in the field of computer vision (image classification and detection) [142].

However, since there are a lot of different networks in literature, it is difficult to select the most suitable architecture (in terms of network deployed and hardware architecture used). DNNs are characterized by an accuracy metric. In the object detection field, this metric is called mAP (mean average precision). this metric tells the user how accurate is the network in finding an object and classifying it. This is not enough, since we may be interested in other characteristics of the network, such as the response time. In some contexts, such as in autonomous driving, an approximate detection in a quick time is better than an accurate one that comes too late.

An interesting job in classifying several networks by their accuracy and time to solution has been done in [143]. In this work, the authors classify some of the most important object detection networks and provide and compare their performances on a single GPU architecture.

Starting from that work, we benchmarked different networks on different CPU-GPU environments. From that experiment, we found out that there is no single one-fits-all network, even in terms of accuracy on a single image. For this reason, we decided to analyze the problem of autotuning in this field, searching for some characteristics of the application or of the network itself that may enable a runtime network selection, whenever is beneficial.

Thus the contributions of this chapter are:

- We performed a benchmarking campaign aimed at exploring accuracy-performance tradeoffs [1]. This result has been used in the MLPerf Inference v 0.5 benchmarking campaign, where this was the only academic contribution [2].

- We demonstrate through a simple automotive use case how the dynamic autotuning approach can satisfy changing constraints that a single network was unable to satisfy;

[1]This work has been performed while I was an intern in dividiti Ltd.

[2]https://mlperf.org/inference-results-0-5

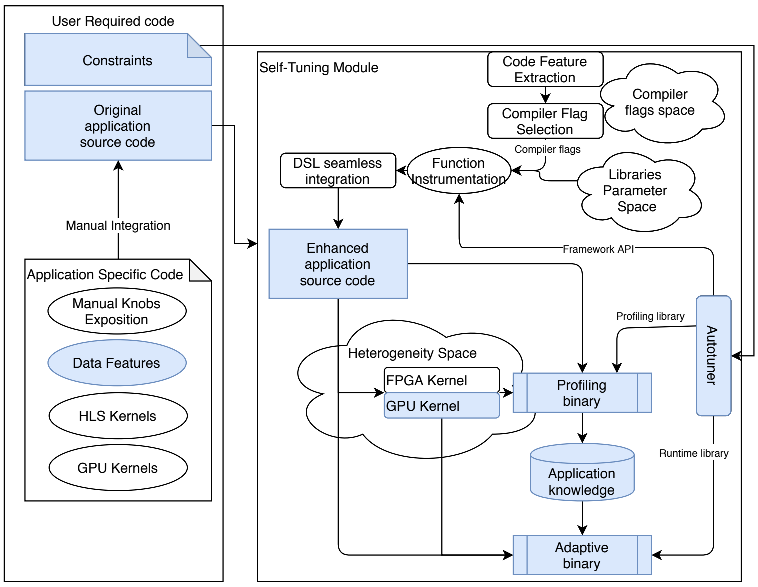


Figure 7.1: *Highlight of thesis approach targeted in this chapter.*

- We built an oracle function based on the benchmarking campaign, that can select the best network among the used ones for every image of the COCO17 dataset, thus evaluating the possible advantage in having a proactive (per image) autotuning approach;
- We highlighted the failed attempts done to employ the proactive method either by finding some image features and training a selector using common machine learning techniques, or adopting an image classification network.

In the context of the global framework, this chapter focuses on providing motivation and explaining the possible benefit of an autotuning approach in the context of DNN based computer vision, which is very important for current researches on autonomous driving cars.

The components of the holistic framework that are involved in this chapter can be seen in Figure 7.1. We can notice the GPU kernel presence since the networks were benchmarked using the TensorFlow backend on both CPU and GPUs. However, the focus point of this chapter is, as in the previous one, the data feature box, since we would like to have a proactive approach where the data feature drives the network selection done at runtime by the self-tuning module. Heterogeneity is available thanks to TensorFlow but is not considered in the autotuning approach studied in this chapter. For this reason, it has been inserted among the homogenous approaches.

7.2 Background

Thanks to recent advances in the deep learning field, a lot of different models have been proposed to tackle the object detection challenge. These networks have a very different accuracy and execution time, and selecting the most suitable one is very complex.

In the context of image classification, an interesting work [144] proposes an approach to select dynamically the network performing the inference, proving that is possible to improve both the accuracy and the inference time thanks to this autotuning. There the authors use a K-Nearest-Neighbor predictor to select, among 4 different models, which one is the best to use for every different image.

Another interesting approach is proposed by [145]. Here the authors suggest the usage of two networks, with a big/LITTLE approach: as for the big/LITTLE CPU, two architectures are used, one small and fast (the little architecture) and one that is more accurate and more time consuming (the big architecture). They perform the inference with the little network and they use the big as a fallback solution only if the little network prediction is deemed not accurate. However, even in this work, the solution is proposed for the image classification challenge.

Another dynamic methodology for the image classification has been proposed in [146]. Here the same network is trained several times, with different datasets, and an ensemble of networks is used to perform the inference. The networks are used sequentially and if a certain threshold metric is reached the result is returned without executing the remaining networks.

Several other design-time optimizations are proposed in literature to build the networks [147], to compress them [148] or to switch from image processing to more expensive and accurate input (eg. LIDAR) [149].

All of these work targeting the network selection are done in the context of image classification. Indeed, to the best of our knowledge, there is no work targeting the dynamic selection of the network in the object detection challenge.

7.3 Motivations

To show the potential benefit of having a self-tuning network, we run an extensive benchmarking campaign on different models and platforms. The objective of this campaign is to explore the behavior of different DNN on different platforms and with different configurations. In particular, we tested on CPU (with and without the AVX2 instructions) and GPU (with

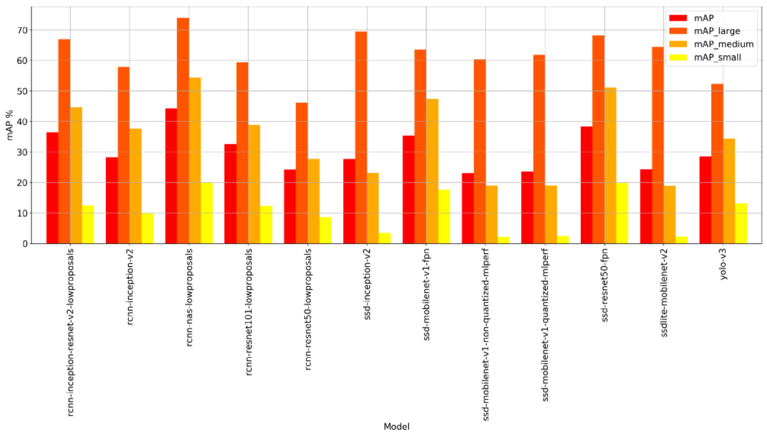


Figure 7.2: *Results of the benchmarking accuracy campaign.*

***Source**: [150]*

and without the TensorRT library support). We selected 12 different models. Most of them were coming from the Tensorflow Zoo [3], trying to balance the SSD-based and the Faster-RCNN based models. To those models, we added a reimplementation of the YOLO-v3 network.

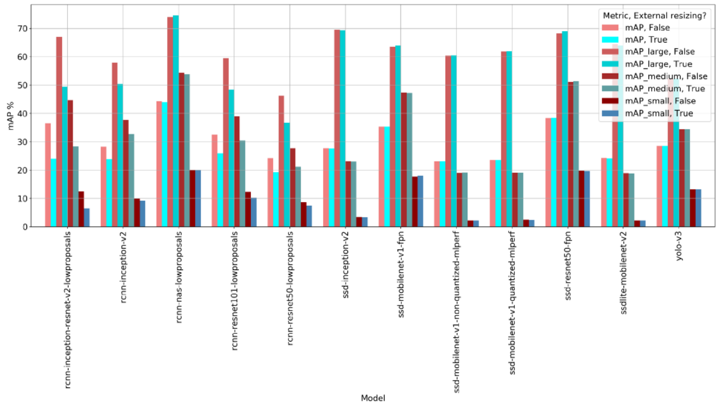
From the accuracy point of view, the campaign consists of 24 different experiments (12 models and with or without batch resizing). From the performance point of view, the number of experiments is increased to 360 (12 models, 5 TensorFlow configurations, and 6 different batch sizes). The experiments have been done on the whole validation set of the COCO 2017 dataset.

As a motivation for the proposed idea, we will analyze the results of this benchmarking campaign, firstly from the accuracy point of view, then from the performance perspective and finally, we will analyze the Pareto frontier of the whole space.

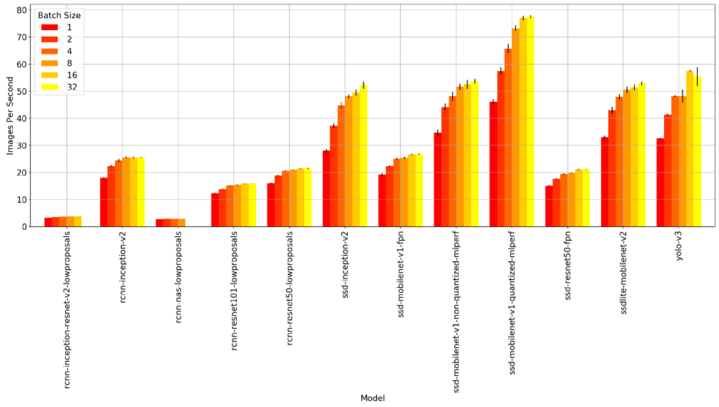
Figure 7.2 shows the results of the accuracy benchmarking done. The most accurate model is Faster-RCNN-NAS, which reaches the overall mAP of 44%. Usually, a model with a good overall mAP performs consistently well across all three object categories. There are, however, some exceptions: SSD-Inception-v2 has the 2nd best score on large objects, but performs rather poorly on medium and small objects; on the contrary, YOLO-v3 has the 2nd worst score on large objects, but is on the 4th place on small objects and performs OK on medium objects.

The bad accuracy obtained on small objects is a well-known problem

[3]https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md



(a) *Influence of batching on accuracy.*



(b) *Influence of batching on performances.*

Figure 7.3: *Influence of batching.*

Source: *[150]*

of SSD-based models. This is why the Feature Pyramid Network (FPN) feature have been introduced. Thanks to this feature, the SSD-ResNet50 and SSD-MobileNet-v1 models are able to reach 2nd and 3rd place on small objects (and on the 2nd and 4th place overall).

The images in the COCO dataset have different shapes and sizes. For the inference to be performed, they need to be resized to match the model input size. This is usually done inside the network, as a first layer. However, this is not possible when processing a batch of images. In this case, all the images of a batch need to be resized before the inference is performed. This procedure may damage the accuracy, as shown in Figure 7.3a

In particular, the Faster-RCNN are the problematic networks. Beside

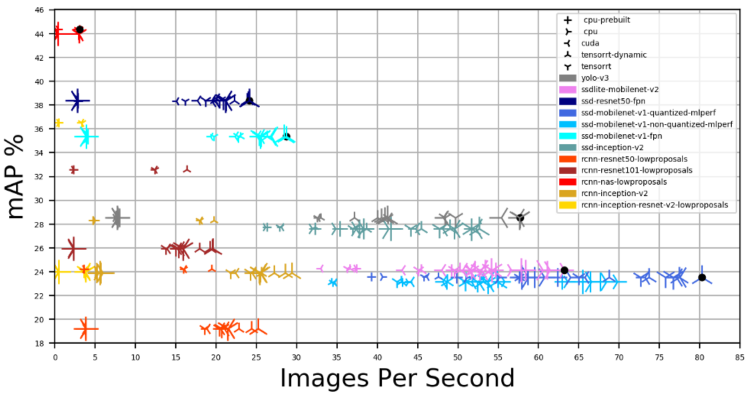


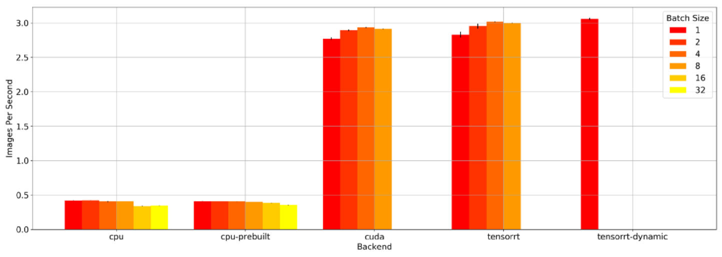
Figure 7.4: *Result of the benchmarking campaign with the measured accuracy. The batch accuracy loss can be seen in the Faster-RCNN models.*

***Source**: [150]*

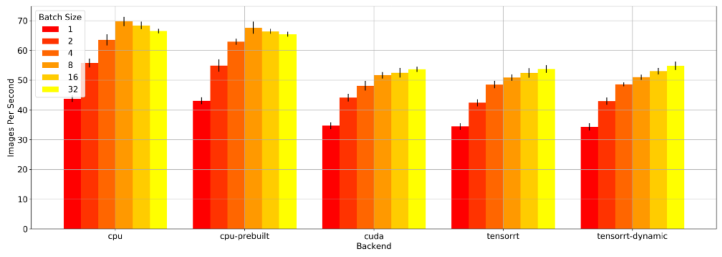
the Faster-RCNN-NAS, the other networks have a *keep-aspect-ratio* layer, which becomes problematic when resizing the images to a unique size.

However, batching images can be significant for the performances, so we need to consider this possibility and not just discard it a-priori. Indeed, as we can see from Figure 7.3b, with a fixed backend (CUDA without TensorRT in the image), increasing the batch size leads to an increase in the number of frames per second (FPS) processed. This growth can also be very significant, leading to almost double performances for some networks (YOLO v3 goes from 32 to almost 60 FPS). The complete result from the exploration can be seen in Figure 7.4. Here the color symbolizes the network used for the inference, while the shape is the backend and the size of the marker symbolizes the batch size. We can notice that the GPU backends are faster than the CPU ones and that the bigger points usually have the best performances. From the image, it is also possible to notice the accuracy drop due to batching of the Faster-RCNN networks.

However, the general behavior is not always true. Some networks show some unexpected results that show why autotuning could be very important in this field. The first is that batching can be detrimental to the performances. Figure 7.5a shows the effect of batching with different backend on the Faster-RCNN NAS. As we can see, the CPU backend is slowed down when batching is enabled. This is probably due to the big memory requirement. Another interesting result from the benchmarking campaign is shown in Figure 7.5b. The inference run on the CPU shows a higher FPS than the GPU one.



(a) *Faster-RCNN NAS performances with different backend*



(b) *SSD-Mobilenet v1 performances with different backend*

Figure 7.5: *Effects of different backend.*

Source: *[150]*

To conclude the motivational discussion, Figure 7.6 shows the best configuration (considering ideal accuracy for the Faster-RCNN networks that have problems with batching). We can notice that there is not a one-fit-all optimal solution, since both the optimal backend and the optimal batch size changes across the different models. Moreover, the networks on the Pareto set are also different if we consider different target accuracy. All these variations strongly suggest that, should a network selector function be found like in the methodology proposed in [144] for the image classification challenge, the object detection challenge could largely benefit from an adaptive autotuning approach.

7.4 The Proposed Approach

In this section, we will see the methodology followed while introducing autotuning in this context. At the first time, we will see how exploiting two networks in analyzing a stream of frames can allow adapting to different constraints (maximize the accuracy of the prediction or maximize the frame rate) reactively. With reactive, we mean that the autotuner reacts to the change of the constraints and allows the self-tuning module to respond to this change and still be able to respect the given constraints. This approach

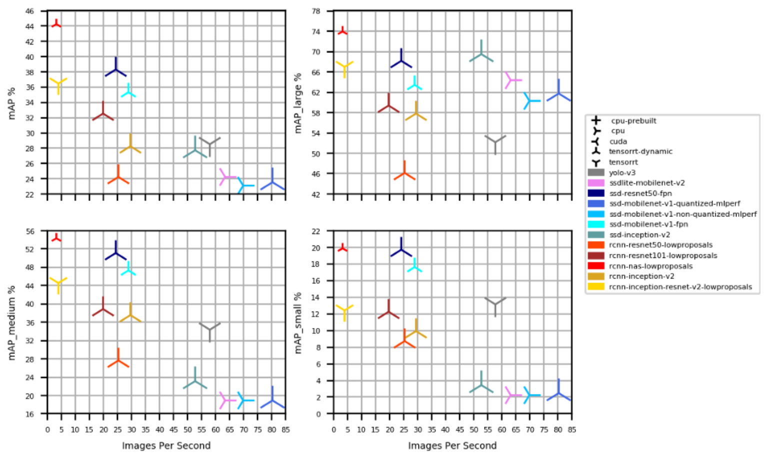


Figure 7.6: *Best performance for every network at the different accuracy metrics (small, medium and large).*

Source: *[150]*

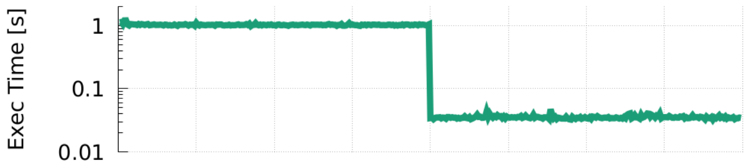
recalls the one from Chapter 4.

We then try to create a predictor function that can work as an oracle for unknown images. This is a proactive approach that relies on the concept of data feature that has been introduced in Chapter 6. To create the predictor, we will search for some data features and we use them to create a function that can predict which is the best network to use to perform the inference.

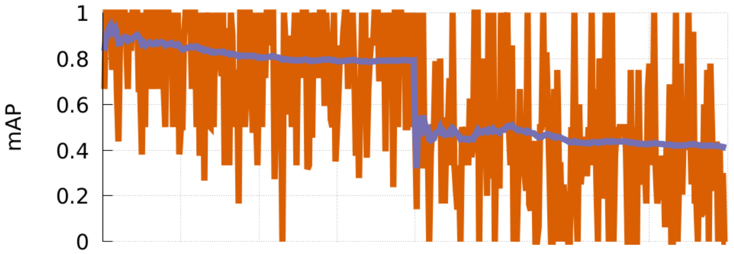
7.4.1 Reactive approach

In the reactive approach, the idea is of having the self-tuning module able to react to changes in the system or external conditions. Changes in external conditions are reflected in changes in the constraints. Figure 7.8 shows the approach from a high-level point of view: we must process a stream of images while respecting some constraints that may change during the runtime. We have a set of networks with different (and known) characteristics in terms of accuracy and time to solution. The autotuning module is in charge of selecting the most suitable among them according to the current constraints.

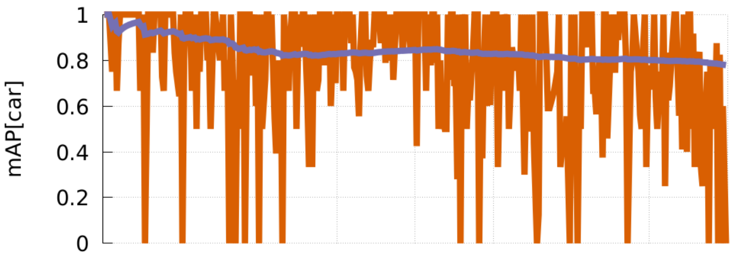
To show the validity of this approach, let us suppose a simplified scenario in the context of autonomous driving, which is one of the most important contexts in the object detection challenge. We need to find the possible obstacles on or near the road. To simplify the approach, let us suppose that we have 2 possible scenarios: highway and city driving. In the first case,



(a) *Execution time for every frame.*



(b) *mAP of all categories*



(c) *mAP of the car category*

Figure 7.7: *Execution log of the same stream with the two different networks, with a change of network in the middle.*

we need to have a quick response and we need to identify "big" objects such as cars, while in the second case we have a slower speed, which means that we can use a slower network but we require a greater accuracy since we need to identify the "small" pedestrians.

In this simplified example, the autotuner is in charge of switching from context 1 (city driving) to context 2 (highway) and back whenever a threshold speed is passed. For this experiment, we have used the KITTI dataset [151], which is a dataset created for the autonomous driving context. As the first network (the fast one for the highway context), we retrained the SSDLite-Mobilenet, while as the accurate network we retrained the Faster-RCNN NAS network.

We show in Figure 7.7 an example of a run where we hypothesize to start the trip inside the city, where the most accurate network is used, and

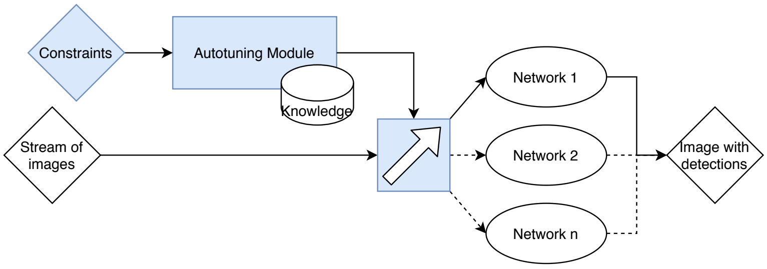


Figure 7.8: *Architecture of the Reactive module.*

after a certain number of processed frames we move to a highway context, where we need faster processing. In Figure 7.7a we report the processing time of a single frame. We can easily see that the first network (in the first half of the picture) has a slow execution time (1 sec per frame), while the second is way faster (less than 100ms). In Figure 7.7b we report the mAP of all the categories (car and pedestrian) that we are interested with. We can notice that the mAP of the second network is a lot worse than the first network. However, if we look at Figure 7.7c instead, we can notice that the accuracy loss of the second network when we are only interested in the mAP of the *car* category is slightly noticeable. In this way, we show that we can maintain the ability to find cars on the road within a constrained time to solution, which is smaller because of the higher navigation speed. This result confirms the benefit of dynamic autotuning in the context of the simplified scenario hypothesized before since we are able to respect the accuracy/response time request in both the contexts, while both the considered networks are not able to do it if taken individually.

7.4.2 Proactive approach

An orthogonal approach to the previous approach is the proactive one. The proactive approach to dynamically select the network aims at using characteristics of both the network and the image that is going to be processed to match the image with its best possible network. We believe that if there is not a one-fit-all best network while considering only the accuracy of the prediction, and there may be some features of the images that determine if a network behaves better than other networks in finding objects in that precise image. Thus, we are interested in finding those characteristics of the images, and building a predictor that may be able to select the optimal object detection network.

The first step is to verify that the best network to perform the inference

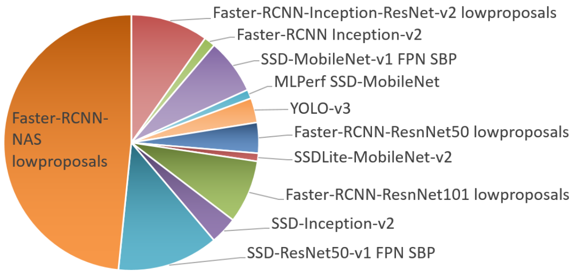


Figure 7.9: *Composition of the Oracle on the full COCO validation set. All the considered networks are present, which means that they are optimal in predicting some images.*

would change across the dataset. We create an oracle function, that selects the best network for all the images of the COCO validation set. In particular, the program selects the highest mAP after evaluating an image with all the networks, and as a tiebreaker, it uses the execution time (the fastest one among the network with the same accuracy wins). Figure 7.9 reports the pie chart of the oracle. We can notice that almost half of the chart is occupied by the Faster-RCNN NAS, the most accurate network. We expected this network to be dominating. However, this network does not have always the best accuracy. Moreover, the oracle shows that all the different networks are represented which means that they are optimal for at least some images. The second step is to search the data features and a prediction function to drive the network selection proactively given the target image. Figure 7.10 shows two different attempts that we performed in building the predictor. The first one, which we define "traditional Machine Learning (ML) approach", can be seen in Figure 7.10a. The second attempt, where we used neural network techniques, can be seen in Figure 7.10b. Figure 7.10a shows the pipeline that we designed to perform object detection with network selection done with the traditional ML approach. The first step is Feature Extraction, which is a module that is in charge of quickly analyzing the image and extract some features. Then the predictor module is a function in charge of driving the network selection. This function needs to be able to quickly select the network given the data features extracted from the previous step. Finally, the image is forwarded to the object detection network, which performs the detection task and returns the objects detected in the given image. To create the feature extraction module, we need to identify a small set of features that can be quickly extracted from the image.

We started the search of the data features from the ones used in [144] since the authors were already working in the DNN context. Other candidate features are taken from [39]. In this work, four easily obtained

Number of keypoints	Number of corners	Number of contours
Dissimilarity	Homogeneity	ASM
Energy	Correlation	Number of peaks
Contrast	Variance	Mean
Hues(4 bins)	Saturation (4 bins)	Brightness (4 bins)
Histogram of the three colors (3*8 bins)	Number of pixels that are edges in the image	Number of objects (connected components)
Aspect ratio	Histogram of gradients(8 bins)	

Table 7.1: *List of all the features collected to build the predictor.*

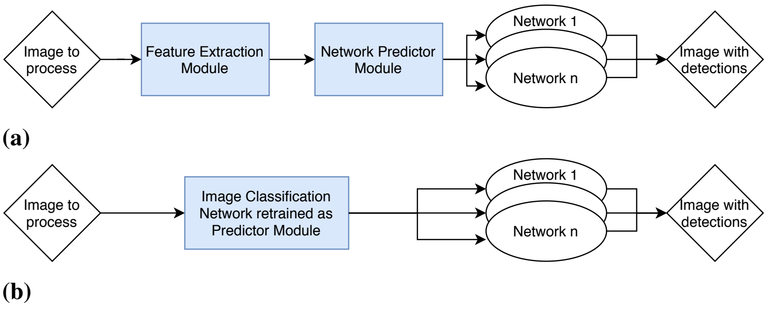
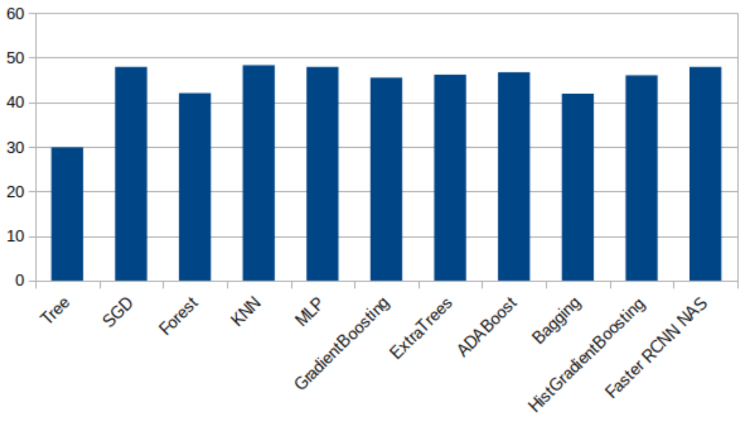


Figure 7.10: *Structure of the two attempts done in implementing the proactive approach to object detection, using a traditional Machine Learning approach (a) or using an Image Classification Neural Network (b).*

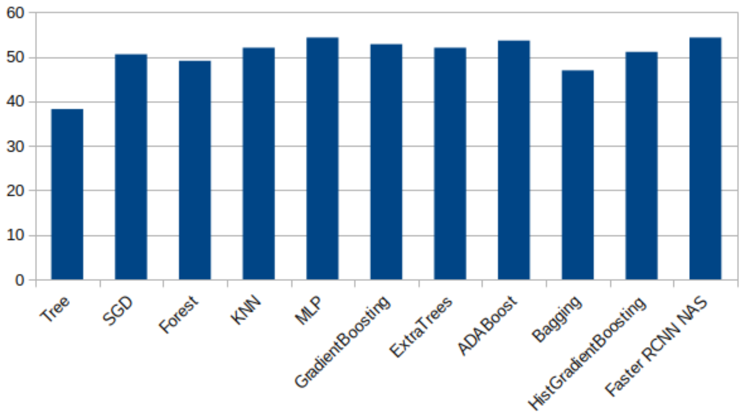
characteristics (*mean, variance, local homogeneity, and covariance*) are used to decide how to approximate an image. Moreover, we considered standard image processing features from literature [152]. We extracted all of these features and others using well-known Python packages, such as OpenCV [153] and Skimage [154], collecting in total over 50 image features. The complete list of the considered features is reported in Table 7.1. We did extract all of these features, however, we are aware that we need to reduce the number of features to use, since getting all of these would be too time-consuming. Moreover, some of them (for example the connected components) are too expensive in terms of extraction time and have been discarded a priori.

The following step is to build the classifier. To do this, we use both the output of the oracle and the extracted features of the images, since we need to learn the correlation between these features and the best network. We decided to use the scikit library [155] since it is a well-known and verified module for the most common ML algorithms. We used a Principal Component Analysis (PCA) to restrict the space of features, assigning to this methodology the duty of finding out which ones are the most important features that we have to consider. We then passed the output of the PCA to the following step, which is the model training. Before training the model, we have to create the training and the test set. From the available 5000 images (for which we have the array of features with the associated best network), we create a training set of 4500 images, while the other 500 are left as validation set. Since the goal is to implement a classification layer, we have tested most of the classifier engines available in the *scikit-learn* module. Among them, we tested *Decision Tree, Random Forest, Bagging, AdaBoost, Extra Trees, Gradient Boosting, SGD, MLP, KNeighbors*. However, no one of those algorithms was able to provide a robust classifier that could be used as the predictor, as we can notice from Figure 7.11. In particular, Figure 7.11a shows the result on the complete set of networks. In most cases, the accuracy of the validation set was around 40% which is also the number of occurrences of the most accurate model always (the last column in the figure). The tree predictor is the one that shows the worst result, around 30%. To reduce the noise in the data available for the learning phase, we restricted the number of models. We decided to use only the ones that were Pareto optimal in the benchmarking study. This reduced the number of available models to 6. Nonetheless, even with the reduced number of target networks, the traditional ML classifiers were unable to learn how to predict the best network to use to perform object detection given the image. The result of this final experiment is reported in Figure 7.11b. We can notice that even with this reduction in the possible networks there is no valid predictor: the last column (Faster-RCNN-NAS) is the predictor that always selects the Faster-RCNN-NAS network to perform the detection since it is the most accurate one. This predictor has an accuracy of 55%, which means that in more than half of the test images the RCNN-NAS has the optimal accuracy in the reduced validation set. All the predictors have a worse result, meaning that they can guess the optimal network with less accuracy than always selecting the same, and most used, network.

Since the traditional approach did not lead us to a working solution for our problem, we decided to attempt using a DNN classifier. In particular, we selected a MobilenetV2, trained on the ImageNet dataset. We decided to perform transfer learning, thus only modifying the last layers (the classifier layers) of the network, without changing the feature extraction layers. The network we used to perform the transfer learning has 154 frozen layers and the last layer has 1280 features coming out, to which we attach the dense layers used to perform the classification. The total number of parameters of this network is 5,149,771, and more than half of them are frozen, so they cannot be trained during the transfer learning. As we can see, we have much more features than with the previous approach. We used the keras [156] framework to perform the transfer learning. Since the oracle shows that



(a) *Accuracy of the predictors with all the networks*



(b) *Accuracy of the predictors with a restricted set of networks.*

Figure 7.11: *Results of the training of the different models.*

there is no a similar amount of images for all the network, we needed to rebalance the dataset to have a fair training phase. The training data have been preprocessed to obtain a balanced dataset where all the labels (in our case, the target networks) have the same amount of training images. This is a well-known technique used to avoid that the dataset unbalancing can influence the learning process. However, even this approach did not lead to a working predictor. The new predictor always learns to predict one or two networks.

We do not know the exact reason behind all of these failures. We believe that the main reason is that object detection is a much more complex operation if compared to image classification where a similar attempt was successful [144]. Indeed, the DNN used to tackle this challenge are more complex than the classification networks: [143] shows how most object detection networks are composed of two sections, a region proposal network that aims at creating the bounding boxes of the objects, and a feature extractor, which is an image classification network that provides the label to the object extracted with the first stage. We think that this failure may be due to the fact that the image features extracted with traditional image processing or with feature extraction layers trained for the classification problem are not enough. Indeed, these features may not be sufficient to model the region proposal problem. Thus, a different set of features may be needed.

7.5 Summary

In this chapter, we have studied the possibility of applying autotuning in the object detection context, where to the best of our knowledge has not been already attempted before. We have shown why the autotuning methodology could be very profitable for this context, with a large benchmarking campaign that demonstrates that there is not a one-fit-all optimal solution. We have seen that using a reactive approach leads to benefits: we were able to satisfy changing requirements by exploiting two networks that were unable to satisfy the given constraints if taken singularly. Even if our attempt in building a working predictor was not successful, we believe that this can be a motivational study that could inspire some researchers, more expert in the DNN field, that this approach could be meaningful.

The outcome of this work has been accepted for publication to the SAMOS 2021 international conference.

Part II

Geometric Docking case studies

CHAPTER 8

Background

In this secon half of the thesis we will describe some techiniques introduced on a molecular docking application called GeoDock. This application is a key component of this thesis, since half of the work that I did during my Ph.D. was focused on improving the performance of this particular use case.

GeoDock is a component, in charge of geometric docking, of the Ligen [157] tool, which is itself a component in the EXSCALATE (EXaSCale smArt pLatform Against paThogEns)[1] tool-flow. This tool-flow is the in-silico section of a real drug discovery pipeline.

The goal of a drug discovery process is to find new drugs starting from a huge exploration space of candidate molecules. Typically, this process involves several *in vivo*, *in vitro* and *in silico* tasks ranging from chemical design to toxicity analysis and *in vivo* experiments. Figure 8.1 shows the complete pipeline of a traditional drug discovery approach. We can notice that the virtual screening is a step in the exploratory research phase. Molecular docking represents but one stage of this step [158, 159]. It aims at estimating the three-dimensional pose of a given molecule, named *ligand* when it interacts with the target protein. The *ligand* is much smaller than the target protein. For this reason, we only consider a region of the protein,

[1]https://www.exscalate.eu/en/platform.html

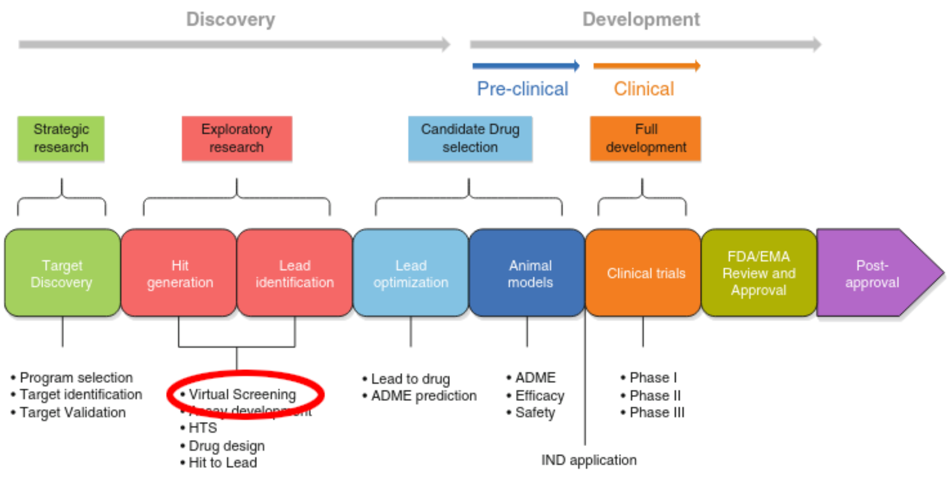


Figure 8.1: *Complete drug discovery pipeline. The virtual screening task has been highlighted.*

Source*: [160]*

called *pocket* (or binding site). The *pocket* is an active region of the protein where it is likely that an external small molecule can interact.

Molecular docking is a well-known research topic. There are two approaches to this task: the first one is deterministic and the second random-based.

Random-based approaches use well-known techniques to estimate the interaction between the *ligand* and the *pocket*. Among these techniques, there are genetic algorithms (like in [161, 162]) and Monte Carlo simulations ([163, 164]).

However, random-based techniques have a strong drawback, which is that they do not guarantee the reproducibility of the solution. Indeed, several companies require that the computational solution has to be repeatable. This requirement is given because, being the following steps of the drug discovery pipeline long and expensive, they don't want to commit basing their decision on a not repeatable solution. For this reason, it is often required to use a deterministic solution to the docking problem, to guarantee its repeatability.

Among deterministic approaches, an early work [165] considers only rigid movement of the ligand, without modification of its molecular structure. However, from a geometrical point of view, it is possible to identify a subset of bonds – named *rotamers* – which can split the ligand into exactly two disjoint non-empty fragments when they are removed. *Rotamers* can independently be rotated without changing the chemical connectivity of the ligand. Therefore, most approaches evaluate also the changes in the shape of the ligand that can be generated from the rotation of all its *rotamers*. For example, in [166] the molecular docking framework can deal with the flexibility of the ligand molecule by adopting a model of the electrostatic interactions to finalize the docking. Similar works such as DOCK [167], FlexX [168], FlexX-Scan [169] and sur-flex [170] provide deterministic molecular docking algorithms that are able to modify the shape of the ligands exploitintg the rotable bonds. These algorithms rely on both geometric and pharmacophoric properties in their docking procedure. However, all these works implement a different docking procedure with respect to GeoDock.

From the computational point of view, the evaluation of each ligand is independent of the evaluation of all the other candidates. This and the huge amount of candidates to process makes this problem an embarrassingly parallel one. Nonetheless, to find the optimal pose of the ligand when it interacts with the pocket, we still have to manage a large number of degrees of freedom, produced by the *rotamers*. To simplify the computation, the *pocket* is usually represented as a static structure, where the position of the atoms cannot change during the docking process. However, the ligand is a set of atoms connected by covalent bonds, i.e. atoms that share an electron pair. These atoms can move during the docking process and the shape of the ligand can change thanks to the rotation of the *rotamers*. Indeed, while the target pocket is represented as a rigid structure to simplify the computation, the ligand is represented as a flexible set of atoms bound together by chemical bonds, i.e. sharing electron pairs between atoms (covalent bond). This makes the evaluation of the interactions between a single ligand and the pocket from the chemical and physical perspective a computationally-intensive problem. For this reason, state of the art approaches [171–175] suggest separating the pose prediction task from the virtual screening task. These two tasks are very similar to each other in their organization: the first one (pose prediction) focuses on finding the best positioning of a *ligand* in a *pocket*. The second one (virtual screening) aims at selecting a small set of promising ligands across a large dataset of candidates that maximize the fit to the given binding site. However, several industrial applications [163,176] perform both tasks into a single software module.

A fundamental difference between the pose prediction and the virtual screening task lies in how the chemical and physical interactions between *ligand* and *pocket* are used. Indeed, in the pose prediction task, it is possible to estimate the position of the ligand without using them (geometrical approach) or not (pharmacophoric approach). However, chemical information is needed when performing the virtual screening. The pharmacophoric approach is the most computational-intensive, while the geometric approach is more lightweight, thus faster. However, the best solution according to a pharmacophoric score implies a good geometrical score and a more accurate prediction. Indeed, there are no guarantees on the chemical compatibility of the pocket-ligand pair when using the geometrical approach. The optimal solution obtained with a geometric approach may be either a non-valid solution or a poor solution from a pharmacophoric perspective. Therefore, it is mandatory to use pharmacophoric information when doing the final selection. The geometrical approach may be useful to filter among all the candidate *ligands* those that cannot geometrically fit the target pocket. This allows to perform the chemical scoring on an already reduced set, thus speeding up the virtual screening process as a whole.

Nowadays, the amount of candidates evaluated with a virtual screening pipeline is in recently reached 1 billion docked molecules on the entire Summit Supercomputer [177]. The objective of our research is to enable the exascale drug discovery paradigm, which can manage hundreds of billions or trillions of candidate drugs. This can be obtained thanks to the speed-up in the first phase of the drug discovery process, the one performed through computer simulations (in-silico). Indeed, having better docking performances allows testing more candidates. This can be obtained by applying autotuning techniques to the algorithms, or exploiting the heterogeneity of the HPC facilities as we will see in the following chapters.

CHAPTER 9

Introducing Autotuning in GeoDock in a Homogeneous Context

In this chapter, we studied the original monolithic GeoDock application to find optimization and autotuning opportunities. In particular, we analyzed the algorithm to identify some application related software knobs, that allow applying the approximate computing paradigm. Those knobs relax constraints on the correctness of the result, allowing higher throughput. Moreover, we study that is possible to find a relation between the exposed trade-offs and the input size. We exploit this relation to force a constraint on the time-to-solution, maximizing the output quality under this time constraint. The output of this work is an initial tunable version of the application, although it is still running on the CPU only.

9.1 Introduction

During the virtual screening process, the time budget is an important constraint that has to be considered when designing a screening campaign. It is common practice to have a domain-expert human, whose job is to tune the size of the database of molecules to dock according to the time budget.

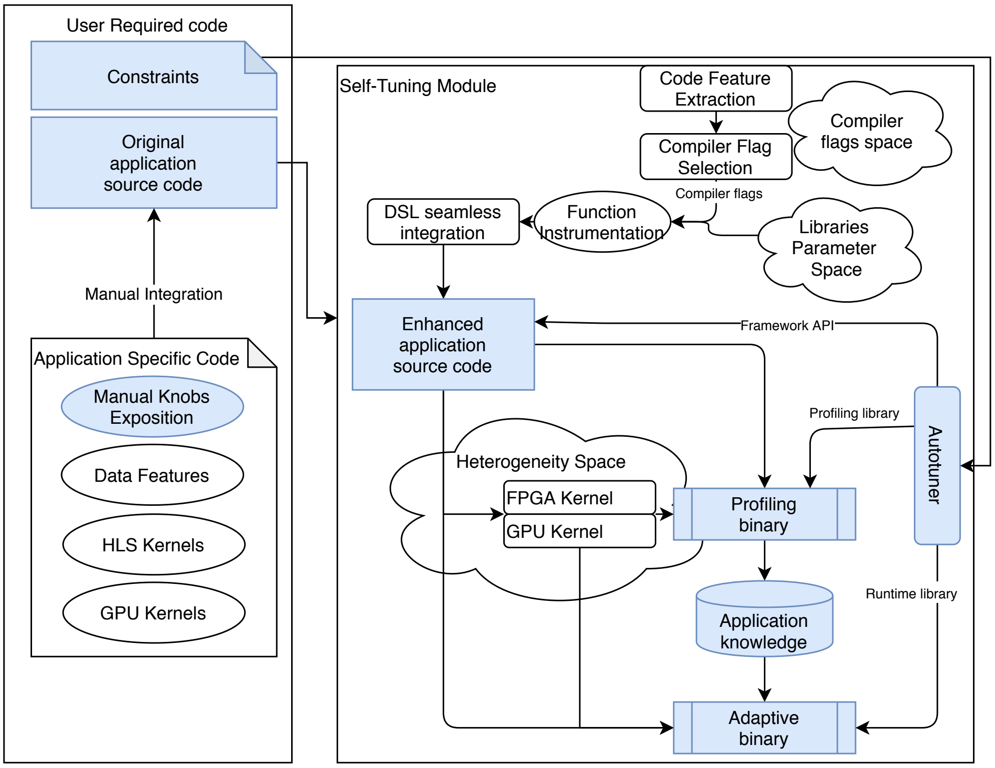


Figure 9.1: *Framework techniques used in this chapter.*

This approach constrains the space that can be explored. Indeed, it does not provide any guarantee neither to find a good local result nor that the global optimum will be part of the tested dataset. We have already described the application in Section 8. Since the algorithm is a greedy exploration of the input space, the best approach to increase the probability of finding a good match is to enlarge the number of tested candidates. This result can be obtained in two ways: the first is to adopt more powerful machines or introduce accelerated kernels, the second is to shorten the time elapsed in the evaluation of a single ligand-pocket pair. This can be done by introducing autotuning and approximation in the docking process.

In this chapter, we focus on *Geodock*. We show how, working with a homogeneous system without any help from accelerated kernels, we can increase the search space by studying the application and introducing some performance-accuracy trade-offs. In particular, we introduce some approximate computing techniques in the most compute expensive kernels, and we evaluate the trade-offs that these techniques can offer. Moreover, we create a performance model that, given enough data to process, can estimate the computation time, for every value of the set of software knobs. Thanks to this model, we enhance the application with an adaptive layer that is used to enforce a constraint on time-to-solution.

In the context of the general framework, we can see in Figure 9.1 the involved techniques. Manual intervention is required since the application has to be studied and the software knobs, that are domain-related, have to be exposed. Then the autotuner is inserted into the application. Profiling runs are performed to build the application knowledge and it is integrated into the adaptive binary. This resulting binary is able to enforce the time to solution.

To summarize, in this chapter we propose a methodology to enable tunable approximations to explore performance-accuracy trade-offs. We enhance *Geodock* with software knobs, and we use them to control time-to-solution in the virtual screening task. In particular,

1. *Geodock* has been analyzed to introduce approximate computing in the most significant kernels;
2. performance/accuracy trade-offs have been enabled by exposing software knobs that can drive approximations;
3. a performance model based on the software knobs and the input size has been created to estimate the time-to-solution;
4. *Geodock* has been enhanced with an autotuning layer that can satisfy a user-defined time budget.

9.2 Background

Approximate computing techniques are well-known methodologies used to generate trade-offs in accuracy-performance. We already summarized some of them that are used in autotuning in Chapter 2. Here we will focus on algorithm level techniques [178]. In this work, we exploited grid-based optimizations in the docking kernel. In computational physics, it is common to exploit multi-level grid models to obtain an accurate result in a restricted area of the full simulated environment.

In these works, the parameter that enables the trade-off is the size of the grid. Tweaking this parameter allows to increase or decrease the number of elements to process. Nested grids are a well-known paradigm, that has been used for a long time in modeling thermosphere [179–181] and ocean flows [182]. This paradigm enabled the adoption of variable-sized grids instead of the previously used regular grids. Variable-sized grids allow improving the performance of the models since they can focus the precision in the area of interest. They have been used in the climate forecast model. [183] demonstrated that variable resolution grids have in the long term the same accuracy in climate forecast as the nested grid model.

Tile optimization is another application of grid processing. In image rendering, an element is taken from each tile, and its value is used to select which computations are required for all the elements in that tile. this approach is called deep peeling [184–186].

9.3 Methodology

This section first introduces *Geodock*. In particular, we describe the algorithm, and we profile it to find the computationally heavy kernels. Then, with functional analysis, we search where the approximation is giving benefits while considering the correctness of the result. This process is what enables the accuracy-throughput trade-off. Finally, we describe how we used this knowledge to set up the auto-tuning and to enforce the time-to-solution

9.3.1 Application Description

The optimization of the pose of the ligand is the most computationally intensive kernel in the virtual screening task. *Geodock* takes as input a database of ligands and the target pocket. It modifies geometrically the ligand, moving the atoms according to the *rotamers*, searching the best pose among the possible ones. For each pose produced, it has to compute a score. It produces as output the score of each pocket-ligand pair.

Geodock performs the virtual screening task using only geometric features of the molecules. It estimates the pocket-ligand interactions, evaluating the similarity between the shape of the ligand and the three-dimensional shape of the pocket in PASS (Putative Active Sites with Spheres) format [187]. It used the *overlap score* function to score each ligand against the pocket. The overlap score, as defined in Equation 9.1, is the reciprocal of the minimum square distance between the ligand and the pocket:

$$o = \frac{l}{\sum_{i=0}^{l} \min_{j=0}^{p} d^2(L[i], P[j])} \tag{9.1}$$

where o is the overlap score, l is the number of atoms in the ligand L, p is the number of 3D points in the pocket P, and d^2 represents the squared

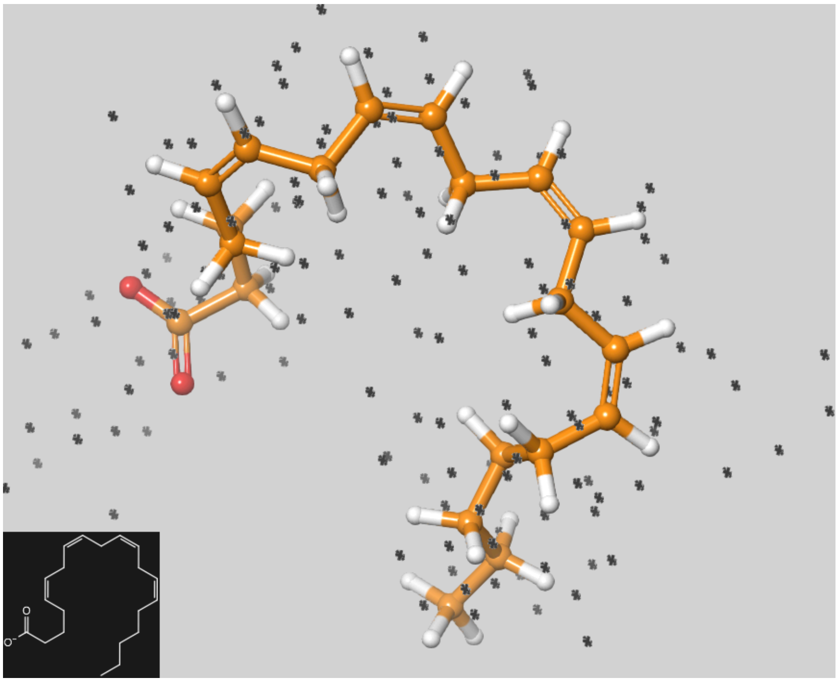


Figure 9.2: *visualization of a docked ligand (connected structure) inside a PASS version of the target pocket (dark spots).*

Source: *[189]*

distance between the i-th atom of the ligand and the j-th point of the pocket. The higher is the overlap, the better is the better geometric compatibility between pocket and ligand.

Figure 9.2 shows an example of docked a ligand inside a pocket (i.e. 1cvu [188]). The PASS representation of the pocket is reported in the image by the black points. They represent the center of the spheres used to model the binding site. The docked shape of the ligand is visible in the 3D image, and the bottom left corner highlights its planar representation. The larger points are the atoms L of the ligand while the connections between atoms are the bonds.

9.3.2 Analysis of *Geodock*

Geodock targets an HPC platform. It targets the multi-node parallelism thanks to the MPI master/slave paradigm. The master process reads the input database of ligands and dispatches them to the slaves, whenever they end the previous ligand evaluation. Each slave docks the ligand and com-

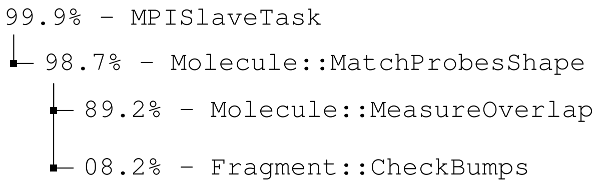

99.9% - MPISlaveTask
98.7% - Molecule::MatchProbesShape
89.2% - Molecule::MeasureOverlap
08.2% - Fragment::CheckBumps

Figure 9.3: *Application Call Graph profile. Functions taking less than 2% of the overall execution time are omitted.*

***Source**: [189]*

putes the overlap score with the target pocket. At the end of its job, each slave notifies the master of the overlap score found and waits for new data to be processed. *Geodock* does not consider other levels of parallelism. Indeed, being an embarrassingly parallel problem there is no need to parallelize inner loops since this would lead to an underutilization of some cores in the serial parts.

We started by profiling the application to locate the computational intensive hotspot. We used GPROF[1] to perform this task. Figure 9.3 shows the Call Graph report, grouping the individual functions by the caller.

As we can notice, the application spends most of its execution time in the `MatchProbesShape`. This is the kernel that performs the optimization of the ligand's pose. It uses a steepest descent algorithm to manage all the internal degrees of freedom of the ligand (i.e. the *rotamers* rotations). In this chapter, we focus on the introduction in this kernel of software knobs inspired by approximation techniques that allow us to manage the time-to-solution of this functionality.

Algorithm 9.1 shows the pseudo-code of the hotspot kernel. At first, the algorithm searches the *rotamers* (line 1) and finds all the possible ways to change the shape of the ligand. Then, it searches for the best shape by rotating the bonds one by one. (lines from 2 to 20). In particular, it grows left and right ligand fragments, starting from the two extremes of the bond (line 3) and it rotates them independently. Every fragment is rotated step by step up to a 360 degree angle (lines from 4 to 5); At each step, we have to check whether the ligand shape is valid since there is a non-null possibility of internal bumping of the molecule. (line 6), If a bump is found, it invalidates that shape and we need to continue with the following rotation step. If the ligand shape is valid, the algorithm computes the overlap score and checks if it is better than the previous score (lines from 7 to 9). At the

[1]GNU gprof `https://sourceware.org/binutils/docs/gprof/`

```
   Data: the pocket and the 3D structure of the ligand
   Result: the overlap score of the ligand
 1 get the list of rotamers;
 2 foreach rotamer do
 3     grow the right and left fragment;
 4     for angle in 0-360 degrees with step 1 degree do
 5         rotate left fragment to angle;
 6         if the ligand shape is feasible then
 7             measure the overlap of the ligand;
 8             check if the overlap is improved
 9         end
10     end
11     set the left fragment to best angle found;
12     for angle in 0-360 degrees with step 1 degree do
13         rotate right fragment to angle;
14         if the ligand shape is feasible then
15             measure the overlap of the ligand;
16             check if the overlap is improved
17         end
18     end
19     set the right fragment to best angle found;
20 end
21 return the overlap score of the ligand;
```

Algorithm 9.1: Pseudo-code of the `MatchProbesShape` kernel, which changes the shape of the ligand to maximize the overlap score.

end of the whole 360 degrees of exploration, we need to emplace the atoms in the best position found (line 11).

The kernel is agnostic in regard to whether the left or the right fragment is evaluated, and for this reason, we do not differentiate between the two fragments.

As we can notice from Figure 9.3, most of the time in the computation is spent into the scoring function (`Molecule::MeasureOverlap`). However, the implementation of this function has already been optimized. The contribution of this work is to reduce the number of calls to this function, thus avoiding useless computation when it is likely that they would not bring any improvement.

To achieve this result, we need to analyze the kernel and see if there is some possibility of skipping the scoring without losing information. We started by analyzing the rotation. Figure 9.4 reports the value of the score for a rotation of 360 degrees with a step of 1 degree. Thanks to this analysis

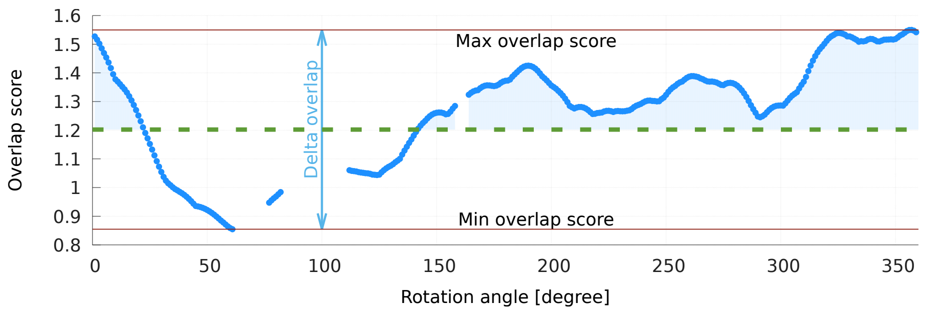


Figure 9.4: *Changes in the overlap score by rotating a fragment of the ligand. The x-axis represents the angle of the rotation, while the y-axis represents the overlap score of the ligand.*

***Source**: [189]*

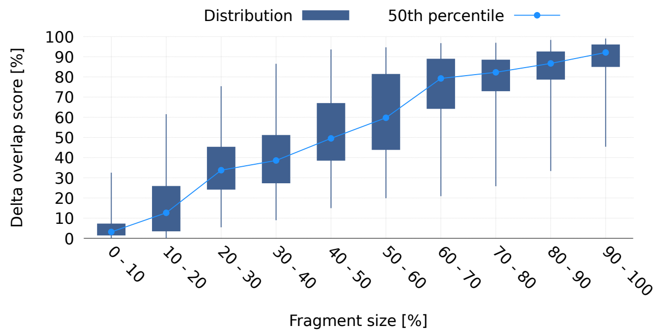
we can see how the rotation of a *rotamer* affects the score. The x-axis represents the rotation space, from 0 to 359 degrees, and the blue points represent the overlap score when the ligand is valid. The image reports the analysis done on a single ligand-pocket couple, however this behaviour is common to all the analyzed ligand-pocket couples.

We define *delta overlap* the difference between the maximum and the minimum overlap in a 360-degree rotation. We define as *peak* the set of contiguous and valid rotation angles that have an overlap score higher than 50% of the delta. We can immediately notice that it is not necessary to test all the angles, thus this is a first candidate approximation knob. However, to justify this, we need further analysis to find when and where we can skip computations.

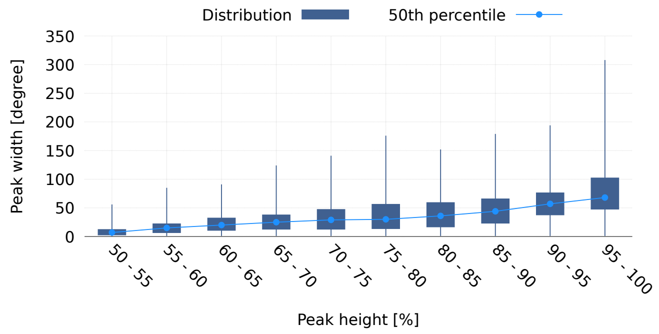
Figure 9.5a correlates the size of a *rotamer* with its impact on the overlap score. On the x-axis, we represent the relative size with respect to the size of the ligand. On the y-axis, we can see the normalized delta overlap.

It is easy to notice that small fragments have small deltas. This means that such they have a limited impact on the final score of the ligand.

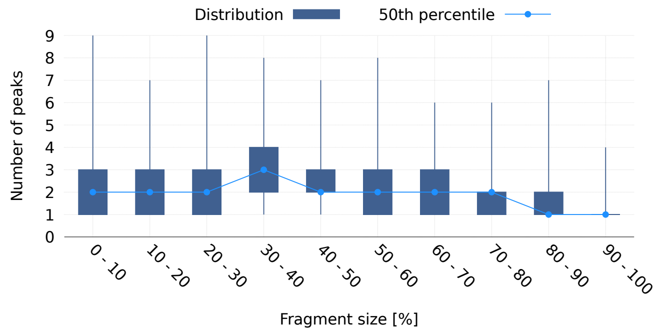
Figure 9.5b correlates the degree width of a *peak* with its height, which has been normalized with respect to the delta overlap. From this plot, we can notice that the wider the peaks are, the higher is the possibility that they contain the maximum overlap. Moreover, the narrow ones usually don't reach the maximum height. From these two analyses, we can deduce that the overlap is a smooth function, and angles that are neighbors have close values. Finally, we study the frequency of the peaks. Figure 9.5c shows the number of peaks generated by the rotation of a fragment (y-axis), clustering



(a) *Distribution of the delta overlap.*



(b) *Distribution of the peak geometry.*



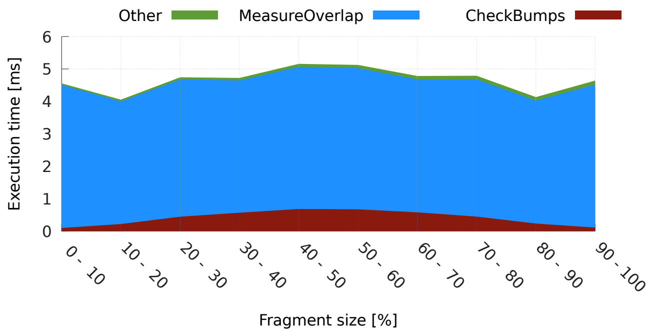
(c) *Distribution of the number of peaks.*

Figure 9.5: *Analysis on the peaks of overlap across different fragments. Each plot shows the minimum value, the 25th, 50th, 75th percentile and the maximum value.*

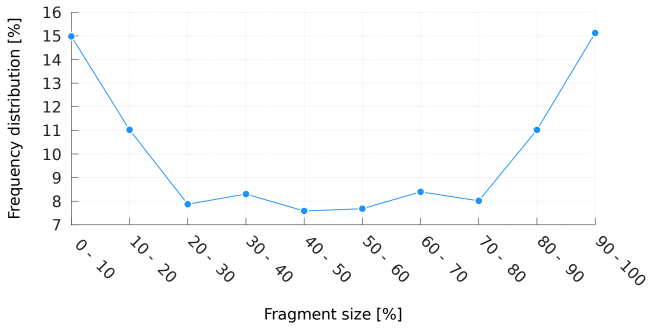
Source: *[189]*

them by fragment size (x-axis). We can notice that larger fragments usually have only one peak, while smaller ones tend to have more.

Besides the functional behavior of the most compute-intensive kernel, we study how the execution time is spent by the application to find the



(a) *`MatchProbesShape` execution time composition.*



(b) *Frequency distribution of the fragments.*

Figure 9.6: *Analysis of the execution time and the frequency of fragments, grouped by their relative size.*

***Source**: [189]*

best rotation angle. Figure 9.6a shows on the x-axis the fragment size and on the y-axis how the execution time is divided among the principal functions. We already knew that most of the execution time of the main kernel is spent in the scoring function, however from the image we can see that the execution time of that function does not depend on the fragment size. This was expected since the scoring involves the evaluation of the whole ligand. Figure 9.6b complete this analysis, showing the distribution of the size of a fragment compared to the whole ligand in the target database. We can notice that smaller and larger fragments are slightly more represented, however, all the dimensions are represented (the minimum appearance rate is around 8%).

9.3.3 Exposing Tunable Application Knobs

In the original version of the application, [176], several parameters are considered. They are able to change the behavior of the docking algorithm, from a chemical point of view. Most of them have no impact on the execution time of the application. The only exception is a parameter that allows skipping some rotations. It implements a traditional loop-perforation technique. In particular, instead of performing the rotation for all the degrees, it allows skipping some of them by rotating the fragment of 2 or more degrees in each rotation. This obviously improves the application performance but could degrade the accuracy.

Thanks to the analysis done in the previous section, we can introduce more aggressive software knobs to approximate the original application. We know that small fragments have a smaller impact on the overlap score value (see Figure 9.5a), so we can introduce a parametric loop perforation instead of the original flat loop perforation. This allows focusing only on the biggest and more important fragments of the ligand. The parametric loop perforation works in the following way: whenever the size of the current fragment is below a given THRESHOLD, we will evaluate it using a coarse-grained rotation, defined by an angle of LOW-PRECISION STEP degrees. Otherwise, we will use a fine-grained rotation, with an angle of HIGH-PRECISION STEP.

Moreover, since `MatchProbesShape` is clearly a greedy algorithm, it is possible that its accuracy is improved by repeating the whole procedure more times. For this reason, we define another knob REPETITIONS, which is the number of times that the whole procedure has to be repeated. This parameter can seem to increase the accuracy and the computation time of the application. However, combined with the approximations introduced in the kernel, it can be used to run multiple time an approximated version than once the original one. Given the greedy approach, this could also lead to better results.

Finally, we can use the peak analysis done previously to extract other information from the application. We noticed that the overlap score function is quite smooth with respect to the rotation space. Moreover, the most important peak of every fragment is usually a wide one (the median value is at 68 degrees). For this reason, we believe that we can use a peeling approach to divide the angle into tiles, and perform the high precision rotation only in the tile that contains the optimal result. In particular, for each fragment above the THRESHOLD parameter, we partition the 360-degree angle into tiles of a fixed size x. Then we perform a fast evaluation only on the

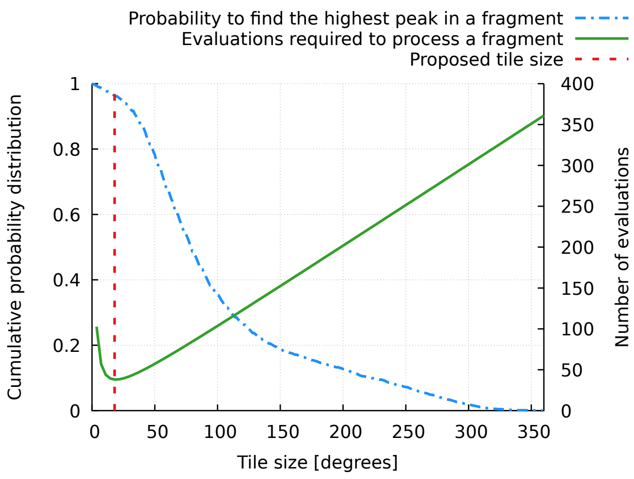


Figure 9.7: *For each tile size (x-axis), the relation between the number of evaluated rotations (y2-axis) and the probability that the width of the best peak is greater than the given size (y1-axis).*

Source: *[189]*

central element of each tile. In a second refinement, we examine the optimal tile with rotations of HIGH-PRECISION STEP degrees. With this tiling approach, the number of evaluated rotations (y) is a function of the tile size and of the HIGH-PRECISION STEP parameter, as described in Equation 9.2

$$y = \frac{360^\circ}{x} + \frac{x}{\text{HIGH-PRECISION STEP}} \tag{9.2}$$

Since we want to minimize the computation effort while preserving the probability of finding the best score, we minimize Equation 9.2. The solution of this is reported in Equation 9.3.

$$\hat{x} = 6 * \sqrt{10} * \sqrt{\text{HIGH-PRECISION STEP}} \tag{9.3}$$

For example, if we set HIGH-PRECISION STEP at the original accuracy (1 degree), the optimal tile size is at 18 degrees. As we can see from Figure 9.7, the probability of identifying the most important peak with this peeling element is still well above 90%, which means that this approximation is acceptable. Indeed, Figure 9.7 shows, for each tile size (x-axis), the probability that the most important peak is wider than the evaluated tile size (y1-axis, blue line) and the number of evaluated iterations (y2-axis, green line). The red line highlights the value obtained when minimizing the number of evaluation with the proposed technique. As a consequence Equation 9.3, we observe that a change in parameter HIGH-PRECISION STEP implies a change in the value of the optimal tile size and in the probability of finding the best peak.

To summarize, starting from the original algorithm described in Algorithm 9.1, we introduced five tunable software-knobs: HIGH-PRECISION STEP, LOW-PRECISION STEP, THRESHOLD, REPETITIONS and ENABLE REFINEMENT, that enable approximation on the application and reduce the number of evaluations of the score function. The driving idea is that we want to focus the elaboration only when it is required. And we try to avoid useless computation thanks to the functional behavior analyzed in Subsection 9.3.2.

To summarize, we report in Algorithm 9.2 the final parametric algorithm of `MatchProbesShape`. The outer loop (line 2) contains the original algorithm and consist of repeating the pose optimization according to REPETITIONS. The optimization of the pose is described for the left fragment between line 5 and line 16. We test the relative size of the fragment against the THRESHOLD (line 5), to decide if perform either a coarse-grained exploration or a fine-grained one. The coarse-grained exploration (line 6) uses the LOW-PRECISION STEP. The fine grained exploration (lines 9-15) is parametrized and has two possibilities. According to ENABLE REFINEMENT, we can perform a two-step optimization using iterative refinements, or a flat exploration using HIGH-PRECISION STEP. The two-step optimization is done with the peeling technique that we explained before. We evaluate the peeling elements of the rotation (line 10), then we refine the exploration of the most promising tile using HIGH-PRECISION STEP (line 11). Again, thanks to the symmetry of the problem, the procedure is applied to the right fragment (lines 17-28) in the same way.

9.3.4 Application Autotuning

The software knobs defined aim at decreasing the time-to-solution of the application. However, as a side-effect, they also reduce the accuracy of the results. From the end-user point of view, a manual selection of these parameters is not an easy task. Therefore we use the mARGOt autotuning framework to autotune the application and select the software-knobs configuration that maximizes the accuracy given a time budget.

The autotuner requires knowledge about the application behavior in order to select the most suitable configuration. There are two types of information needed by mARGOt to build the knowledge for this application:

1. Platform Independent knowledge, related to the error introduced with

```
Data: the pocket and the 3D structure of the ligand
Result: the overlap score of the ligand
get the list of rotamers;
for the number of REPETITIONS do
    foreach rotamer do
        grow the right and left fragment;
        if relative size of left fragment $\leq$ THRESHOLD then
            place the left fragment in the best angle found with step
              LOW-PRECISION STEP;
        end
        else
            if ENABLE REFINEMENT then
                evaluate the peeling element for each tile;
                place the left fragment in the best angle found in the best tile using
                  step HIGH-PRECISION STEP;
            end
            else
                place the left fragment in the best angle found with step
                  HIGH-PRECISION STEP;
            end
        end
        if relative size of right fragment $\leq$ THRESHOLD then
            place the right fragment in the best angle found with step
              LOW-PRECISION STEP;
        end
        else
            if ENABLE REFINEMENT then
                evaluate the peeling element for each tile;
                place the right fragment in the best angle found in the best tile
                  using step HIGH-PRECISION STEP;
            end
            else
                place the right fragment in the best angle found with step
                  HIGH-PRECISION STEP;
            end
        end
    end
end
return the overlap score of the ligand;
```

Algorithm 9.2: The tunable pseudo-code of the `MatchProbesShape` kernel.

the approximation.

2. Platform Dependent knowledge, related to the execution time of the

application on the actual machine that is used.

The first knowledge can be obtained by running only once, on a representative set of pocket and ligands, an experimental campaign. It is important that the set is large, in order to avoid bias. Once this knowledge is built, it can be used in different runs and on different platforms, since the error only depends on the knobs related to the approximation of the application.

On the other hand, to enforce a time-to-solution constraint, we need to know the execution time on the given platform and the input set beforehand. Given the architecture of the application, and since the problem is data-parallel, the overhead introduced by the MPI synchronization is considered negligible even when scaling on a supercomputer machine with a large set of nodes. Therefore, assuming that those nodes are homogeneous, we predict the time on the serial application and split it according to the number of available resources.

Considering a single software-knobs configuration, it is possible to use features of the input database to estimate the time to solution. For this reason, we model the entire database as a set of ligands with the same *average* features. In particular, the model is built with a multivariate linear regression with interaction to estimate the time-to-solution t_{la} for the average ligand. The vector of predictors $\overline{x}$ is composed by the number of points of the target pocket xp_p, the average number of atoms in a ligand xl_a, the average number of rotamers in a ligand xl_r, and all the possible interactions among them (i.e. $xp_p \cdot xl_a$, $xp_p \cdot xl_r$, $xl_a \cdot xl_r$, and $xp_p \cdot xl_a \cdot xl_r$). Thus, the target model is simply composed of $t_{la} = \overline{\alpha} \cdot \overline{x} + \beta$, where $\overline{\alpha}$ is the vector of predictor coefficient, while β is the intercept.

To generalize this approach, we consider the parameters of the regression as a function of the proposed software knobs. This is possible because the impact of the input on the execution time is strongly dependent on the software-knobs configuration. Considering this, we can build a model to estimate the time-to-solution as stated in Equation 9.4. Here, $\overline{k}$ is the vector of software knobs and ν is the number of ligands to dock, in the input database.

$$t = \nu \cdot (\overline{\alpha}(\overline{k}) \cdot \overline{x} + \beta(\overline{k})) \tag{9.4}$$

As we already mentioned, we should train the performance model every time we change the computing platform. However, the experiment described in Subsection 9.5.1 characterize the size of the database required to train the model.

To recap, we enhanced the original algorithm of the application finding and exposing software knobs, thus enabling performance-accuracy trade-offs. We used mARGOt to automatically configure the application requiring the user to provide only a couple of simple parameters, i.e. the number of available nodes and the available time-budget. The characteristics of the actual input can be either provided by the user or extracted by a preliminary input analysis.

9.4 Experimental Setup

Before showing the benefit of the proposed approach, we need to define the boundaries of the experiment, in terms of dataset, metrics of interest, and execution platform.

9.4.1 Data Sets

To evaluate the proposed methodology and its benefits, we used a database of 113K ligands.

The ligands were different both in terms of atoms (from 28 to 153) and in *rotamers* (from 2 to 53). Moreover, the order of the ligands inside the dataset is randomized, in order to remove bias due to their size.

We used 6 pockets protein pockets complexes derived from the RCSB Protein Data Bank (PDB) [188]: 1b9v, 1c1v, 1cvu, 1c2, 1dh3, 1fm9. In particular, the PASS [187] version of the pockets has also been used together with the database of ligands. The PASS version uses spheres to represent binding sites. This solution has been widely used in the context of fast docking [187].

9.4.2 Metrics of Interest

The most important way to measure the performance of *Geodock* is to consider two metrics, the throughput (i.e. how many atoms per second it can evaluate) and, as previously said, the time to solution.

Now we need to identify a metric to estimate the error introduced by the approximation. We call this metric *overlap degradation*. It is used to quantify the mean loss of accuracy introduced by approximation techniques with respect to the baseline. We consider as baseline the configuration that leads, on average, to the better overlap score: HIGH-PRECISION STEP $= 1°$, THRESHOLD $= 0$, REPETITIONS $= 3$ and ENABLE REFINEMENT $= false$. The *overlap degradation* is defined as described in Equation 9.5,

$$score_{degradation} = (1 - \frac{overlap_{approx}}{overlap_{original}}) \times 100 \qquad (9.5)$$

where $overlap_{approx}$ is the mean score of the top 1% of the ligands of the evaluated configuration, while $overlap_{original}$ is the mean score of the top 1% ligands of the baseline. Since this metrics evaluates the loss in accuracy of the approximated application, the lower it is its value, the closer is the approximated application to the original one.

9.4.3 Target Platform

The platform used to execute the experiments is composed of two dedicated nodes of the supercomputer GALILEO, at the CINECA supercomputing center. Each node is equipped with two Intel Xeon E5-2630 V3 CPUs (@2.8 GHz) and 128 GB of DDR4 NUMA memory (@1866 MHz) on a dual-channel memory configuration.

9.5 Experimental Results

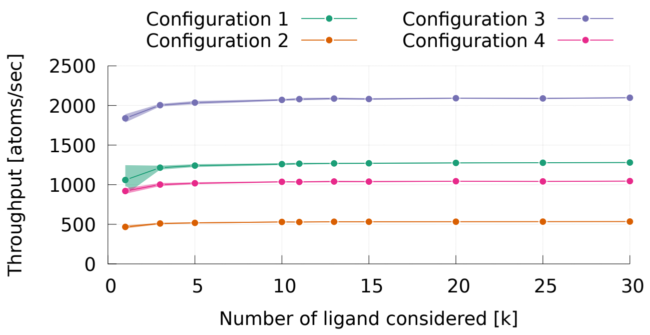
In this section, we evaluate the benefits of the proposed approach using four different experiments. The first experiment is needed to evaluate the data sensitivity. *Geodock* is a data-dependent application, so we want to find out how many ligands are needed to stabilize the input sensitivity. This means, how many ligands are needed to evaluate a configuration. The second experiment targets the approximation techniques. We want to show the enabled trade-off with respect to the baseline, evaluating the effect of the degradation of the overlap score on a single ligand.

The third experiment validates the accuracy of the time-to-solution model. Finally, the fourth experiment shows the benefits of the proposed approach, in two different scenarios.

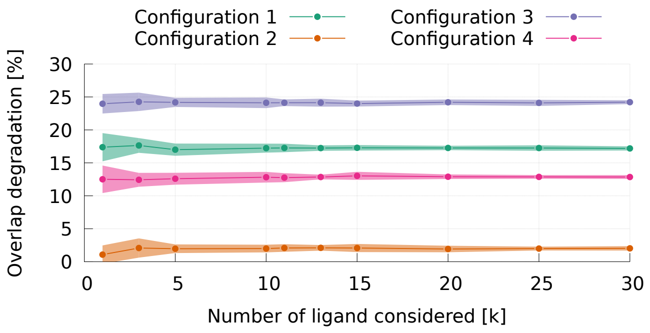
9.5.1 Data Dependency Evaluation

To evaluate the trade-off space, we need to find the set of Pareto optimal configurations, which are the configurations that are non-dominated considering both target metrics (throughput and accuracy). However, this application needs to work with a database of ligands that is heterogeneous in terms of the number of atoms and *rotamers*. Thus it is possible that the performance and the input dataset are correlated, and that the application performance depends on the input.

This experiment aims at finding the dependence of the performance of the configuration from the input dataset. If we manage to find its independence, we can avoid profiling the configuration behavior for every different



(a) *Throughput per process*



(b) *Overlap score degradation*

Figure 9.8: *Application analysis in terms of throughput per process and overlap score degradation, changing the number of ligands. For each configuration we show the average values (dot) and the standard deviation (colored area).*

Source: *[189]*

dataset. To this end, we evaluate the behavior of four different configurations of the enhanced version of *Geodock* in terms of tunable knobs. For each of them, we measure the throughput and the degradation of the overlap score while changing the number of considered ligands. The set of ligands used to evaluate all the configurations has been changed in every run, in order to simulate different datasets. We run this experiment 20 times, with 20 different input sets, for all the different configurations and we measure the two metrics (throughput and error) more times, after processing different numbers of ligands. Figure 9.8 shows the results of this experiment. In both plots, on the x-axis, we have the different amounts of evaluated ligands. On the y-axis, we have in Figure 9.8a the application throughput, while in Figure 9.8b we depict the overlap degradation. Each dot represents the average value of the configurations while changing the input database

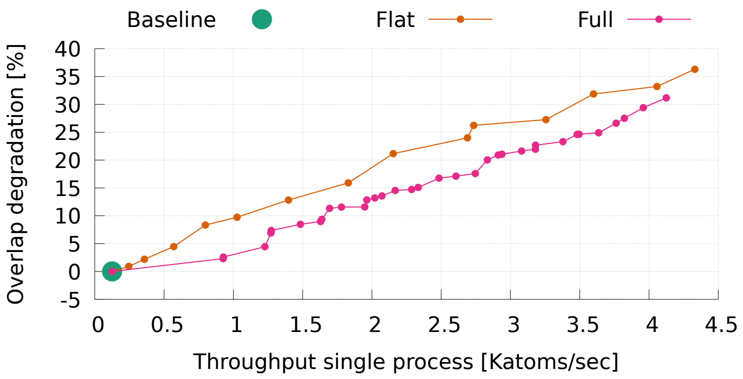


Figure 9.9: *Pareto front of* Geodock *in terms of overlap score degradation and throughput:* Flat *vs* Full.

***Source**: [189]*

of ligands. The transparent curve represents the standard deviation of the measures, which we consider an indicator of the uncertainty of the average value. We can notice (see Figure 9.8a) that the average of the throughput has a minimal dependency (as the small standard deviation demonstrate) on both on the number of ligands in the target database and on the input data. We can notice that after the 2K ligands value, all the plots are constant and their standard deviation is zero. This result was expected, given that we defined the throughput as the number of atoms evaluated, and not as the number of ligands. On the other hand, Figure 9.8b shows that the overlap degradation has a stronger dependency from the dataset than the throughput. Nonetheless, as can be evinced from the image, with a dataset larger than 5K ligands the value is steady and after 10K also the standard deviation is almost null. The overlap degradation depends on the top 1% ligands in the baseline configuration, so it is more related to the selected database. However, since it stabilizes we can say that no more than 10K ligands are needed to determine the expected throughput and error of any configuration.

9.5.2 Trade-off Analysis

With this experiment, we want to define the performance-accuracy trade-off created by the techniques proposed in Subsection 9.3.3. First, we run the different configurations on a single node of Galileo with a database of 20K ligands. Figure 9.9 reports the Pareto front of this design space exploration.

In particular, we want to compare the difference between the previous approach (flat loop perforation) proposed in the original paper [176] and the configurations obtained by the studies on the application done in this chapter. The design space exploration has been performed with a full factorial Design Of Experiments (DoE).

The *flat* design space is composed by only two parameters, that are: HIGH-PRECISION STEP $[1°, 2°, 3°, 5°, 10°, 15°, 45°, 60°]$, REPETITIONS [1, 2, 3].

On the other hand, the *Full* design space, which exploits the software knobs proposed in this chapter, is the following: HIGH-PRECISION STEP $[1°, 2°, 3°, 5°]$, LOW-PRECISION STEP $[45°, 90°]$, THRESHOLD [0, 0.3, 0.6, 0.8], REPETITIONS [1, 2, 3], ENABLE REFINEMENT $[true, false]$.

The *baseline* configuration is the most accurate, and can be obtained by both the *flat* and *full* approaches. This configuration is HIGH-PRECISION STEP=$1°$ and REPETITIONS=3 for the *flat* version, and HIGH-PRECISION STEP=$1°$, LOW-PRECISION STEP=$*$, THRESHOLD=0, REPETITIONS=3, and ENABLE REFINEMENT=$false$ for the *full* version.

As expected, as shown in Figure 9.9, the Pareto front of the *full* version strictly dominates the one built with the *flat* sampling. In particular, we want to highlight the first configuration on the *full* curve after the *Baseline*: here we enabled iterative refinement, and thanks to this knob we can significantly improve the throughput of the application (7.4X) with a limited overlap degradation (2.3%) compared to the *baseline*.

We used some pocket-ligand pairs from the Protein Data Bank (PDB) [188] to better see the effects of the degradation. The PDB is an online database that contains three-dimensional structural data of biological molecules, and the co-crystallized pose within the target pocket. This pose is the actual pose of the ligand for that pocket.

Figure 9.10 shows three scores for each pocket-ligand pair:

- the overlap score of the crystal as described in PDB, and scored without moving any atom with the overlap score function.
- the overlap score of the docked ligand with the *baseline* version.
- the overlap score of the first configuration of the approximated version (the first point in the *full* curve in Figure 9.9).

This experiment shows that the degradation of the overlap score is not only on average small but also if we consider a single ligand as target. The fact that the co-crystallized score is lower than the docked is not surprising: the real pose of the ligand also takes into account chemical issues that are not visible in the geometric approach.

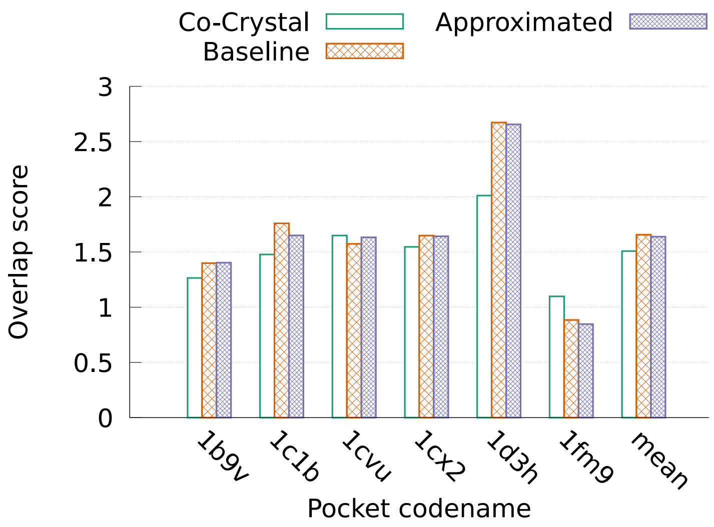


Figure 9.10: *Overlap score by varying the target pocket for the co-crystallized ligand, for the baseline and proposed approximated version.*

Source: *[189]*

9.5.3 Time-to-solution Model Validation

This experiment validates the time-to-solution model described in Subsection 9.3.4. In particular, the model is defined within the design space of the *full* version described in Subsection 9.5.2. The model consists in a linear regression (see Equation 9.4). In order to finalize the model, we need to evaluate the coefficients for each configuration. To do this, we run several times the application using 1K ligands for each available configuration.

The models obtained with this experiment have an average adjusted R^2 value equal to 0.977.

To validate the model we run an experimental campaign. We used a leave-one-out approach on the pocket and a different set of ligands with respect to the ones used to train this model. For every pocket and every configuration, we execute the entire application three times, with different datasets composed of 1K, 2K, 5K, or 10K ligands. Smaller datasets are not interesting since this model is needed to drive the virtual screening, an experimental campaign that targets a large number of ligands (millions or more). For each run we store the predictor value and the observed value, thus obtaining the prediction error. Figure 9.11 shows as boxplot the distribution of those errors. We can notice that the average error is below 1% of the total time to solution. This experimental campaign is run on more

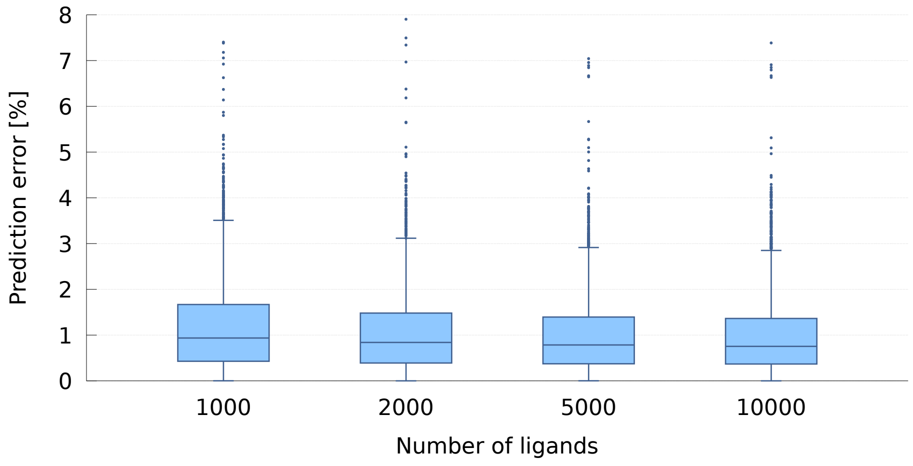


Figure 9.11: *Prediction error of the time to solution, with different sized dataset.*

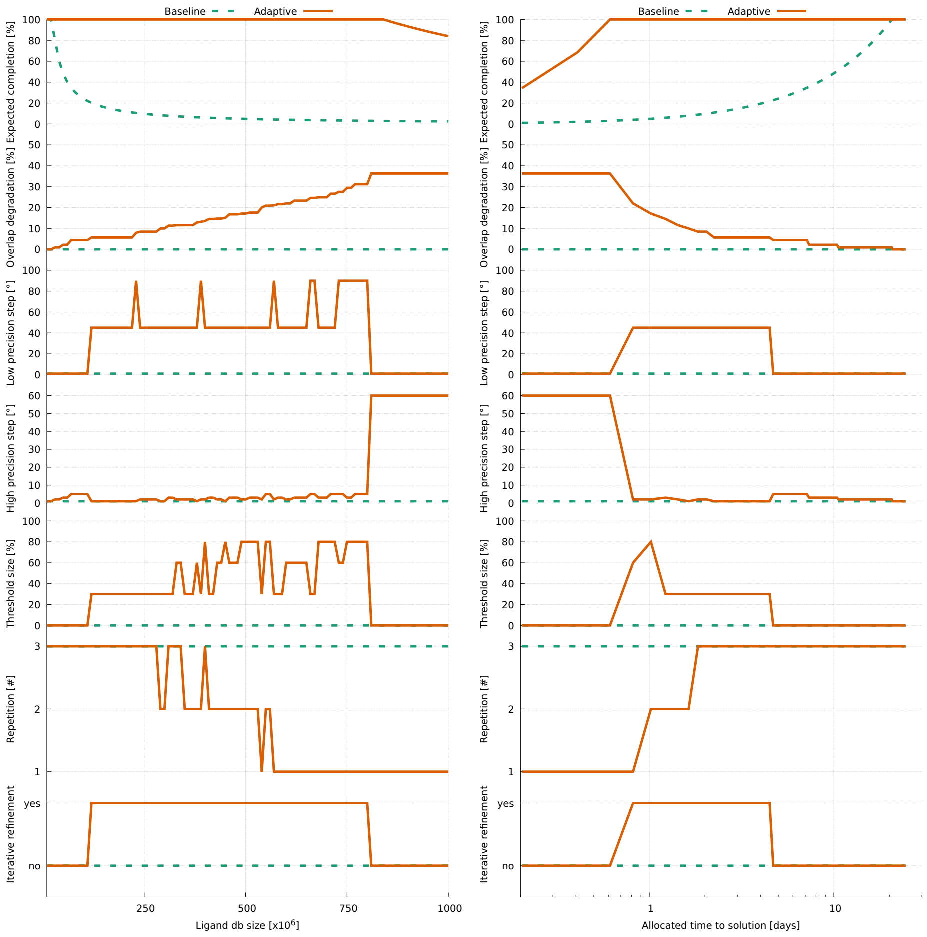
Source: *[189]*

than 15K different runs. The maximum outlier has an error prediction of 7.9%. Moreover, we can notice that the accuracy increases as the size of the dataset increases.

9.5.4 Use-case Scenarios

The final experiment evaluates the benefits of the proposed approach for the end-user, which is in this case a pharmaceutical company that wants to perform a virtual screening campaign within a specific time budget. We can see two exploitation scenarios of the performance-accuracy trade-off introduced with our methodology. In the first one, we allocate a time budget for the computation, and we investigate what is the effect of increasing the size of the database to be screened, to increase the probability of finding a drug. In the second one, we fix the size of the database, and we observe the effects of changing the time budget. We can think of these two scenarios as an attempt to provide the end-user with two high-level knobs: the first one being the number of ligands to be screened and the second one the time budget, thus the cost of the experiment. The time-to-solution model is used to set the right low-level application knobs included in *Geodock* to satisfy the constraints.

Figure 9.12 shows the result of this experiment, using eight nodes of the Galileo machine. In the top 2 plots, on the y-axis, we report the expected performance of the application. In the other plots, we report the selected



(a) *Scenario 1. Varying the size of the ligand database allocating one day to the time budget.*

(b) *Scenario 2. Varying the allocated time budget given the ligand database size equal to* 500×10^6.

Figure 9.12: Geodock *behavior in terms of expected percentage of ligand database completion, expected overlap degradation, and the selected configuration (a) by varying the size of the input, and (b) the time budget, when using 8 nodes of Galileo.*

Source*: [189]*

configuration of the software knobs.

For this experiment, we define the performance of the application as the expected completion percentage of the ligand database and the related overlap degradation of the result. The completion percentage reflects the percentage of ligands in the target database that have been docked. Each plot includes two lines, the dashed one represents the baseline version of

Geodock the other one reflects the adaptive version presented in this chapter.

The x-axis represents the proposed high-level parameter tuned by the end-user, according to the scenario (size of the dataset in Figure 9.12a or time to solution in Figure 9.12b). In particular Figure 9.12a the end-user is interested in imposing a constraint of one day to elaborate the whole dataset. The image shows how the knobs are changed according to the size of the dataset selected. On the other hand, Figure 9.12b shows an example of when the end-user would like to select the time budget to dock a database of 500×10^6 ligands, and the image show the changes in the configuration if the total time given to elaborate the dataset is changed.

In both cases, we can see that the proposed software knobs enable a lot of possibilities for the end-user to tune the problem size and time to solution. Moreover, by using mARGOt combined with the time to solution model, we free the end-user from the burden of having to select manually all of the software knobs and we can expose him to more high-level and straightforward parameters.

Indeed, even if the average trend of the application knobs values can be derived from their meaning, the actual values and the time at which is better to change the configuration according to the high-level constraints are very difficult to select without automatic support. A clear example of this is the parameter THRESHOLD. We can see in the middle range (300–700×10^6) of the problem size that it changes quite a lot of times in the experiment shown by Figure 9.12a.

In terms of application performance, we can notice in Figure 9.12a that the baseline dataset complexion rapidly decreases. Indeed, without adaptivity, it is difficult to process large datasets in a fixed time. On the other hand, thanks to the adaptive approach proposed in this chapter we can process 100% of the database up to 850×10^6 ligands, in the given time. This obviously provokes a degradation of the overlap score, however, this degradation is smooth, as can be seen from the second-row graph. The same result, from the opposite perspective, can be seen in Figure 9.12b, where we run the complementary experiment. Here we fix the size of the dataset and we ask the user to give a time to solution constraint to process that dataset. We can notice that both the baseline and the adaptive one are at the beginning not able to process the full dataset. However, by increasing the time budget, the adaptive solution quickly manages to exhaust the search, even if with a low-quality result. The baseline on the other hand is still processing less than 10% of the dataset when the adaptive solution finishes it. Further increase in the time budget leads to improving the quality, as can be seen in the second-row graph. It is interesting to note that the adaptive solution can exhaust the dataset in less than 1 day, with less than 40% of degradation, while the baseline is still stuck at less than 10%, and requires more than 20 days to finish the run.

Finally, this experiment shows how the *adaptive* approach can provide an output with a limited overlap degradation (less than 10%), while the baseline can process only the 10% of input data set (Figure 9.12b, more or less at 2 days of run time). This result demonstrates the effectiveness of the extracted low-level knobs in *Geodock*.

9.6 Summary

In this chapter, we have analyzed *Geodock* as a representative HPC application. From the analysis of the application and domain knowledge, we were able to identify five software-knobs that enable accuracy-performance trade-offs, by focusing the computation where it is really useful and has an impact on the output value. The adaptive version of *Geodock* is characterized by different levels of accuracy, that are automatically managed by mARGOt according to the needs of the end-user of the virtual screening experimental campaign. In particular, experimental results demonstrated how, by scaling the quality of the results, the application is able to complete a virtual screening campaign over a given ligand database, with different time budgets. These results are an important advantage for pharmaceutical companies where the usage of software and HPC systems have become an important asset in the search for novel drugs. Due to the large number of possible molecules to evaluate, the proposed approach can either lower the cost of the virtual screening process or allow to evaluate a larger number of ligands, thus increasing the chances of finding a good candidate drug.

The analysis derived from the work presented in this chapter has been used to optimize and tune a very large virtual screening run on the whole MARCONI machine from CINECA (>250Kcores, >10PetaFlops system). When this work has been done, this machine was at position number 17 on the top 500. In particular, this experiment has performed one of the largest virtual screening campaign for the ZIKA virus considering a database of 1.2B ligands. The resulting candidates are currently under in-vitro and in-vivo testing.

The outcome of this work has been published in the Journal of Supercomputing [189].

CHAPTER 10

Porting and Tuning Geodock kernels to GPU using OpenACC

In this chapter, we focus on porting Geodock to a heterogeneous node, showing how it is possible to significantly improve computation efficiency by using heterogeneous architecture. In this chapter, we will describe how we analyzed and rewrote the application to match the GPU parallel architecture, and we performed a minimal static autotuning to select, before the run, the optimal configuration for some GPU parameters. In this implementation of GeoDock, we used the OpenACC language to implement the parallel kernels on the GPU.

10.1 Introduction

In the last decade, energy consumption has become an important issue also in the HPC domain. For this reason, a switch from homogeneous systems to heterogeneous systems has begun. By using different hardware accelerators, such as GPUs or Xeon-phi, heterogeneous systems usually have better energy efficiency and can provide more FLOPs. Indeed, most of the top positions in the Green500 list (as of June 2020, [2]) are occupied by het-

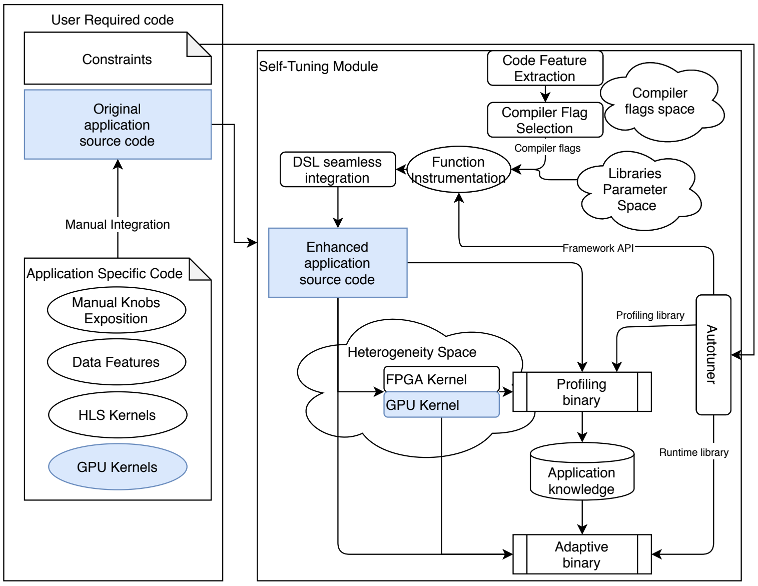


Figure 10.1: *Highlight of thesis approach targeted in this chapter.*

erogeneous machines. The improvement in efficiency often implies an increment in programming complexity, since application developers must use different paradigms to leverage features of these co-processors. In particular, GPUs have many computational cores that expose a much higher level of parallelism than CPUs. However, with respect to CPUs cores that have complex features, such as out-of-order and speculative execution, GPUs cores have a simpler architecture. Therefore, complex code and control flow operations lead to significant degradation of the performance of a GPU application.

GeoDock can leverage the parallelism on the *ligand* level using a classic MPI master/slave approach. However, in this paper we investigate the possibility to offload the geometrical docking kernel on GPU, to leverage its internal parallelism for decreasing the time to solution, leading to two main benefits for the end-user. On one hand, it decreases the monetary cost of the drug discovery process. On the other hand, it enables an increment of the number of *ligands* analyzed by *LiGen*, increasing the probability to find a good candidate.

In the context of the global framework, this chapter is focused on introducing heterogeneity in the application, and not on self-tuning the application at runtime. As can be seen in Figure 11.1, the self-tuning module is composed only by the enriched code with the GPU kernel, without any autotuning functionality inserted in. Nonetheless, this work is important because introduces some key concepts in working with heterogeneous platforms. Indeed, this is the starting point in creating a heterogeneous self-tuning module.

To summarize, the main contributions of this chapter are the following:

- We create a kernel for accelerating GeoDock, using OpenACC directives.

- We analyze the obtained performance and discuss the language and algorithm limits, comparing it with the CPU baseline.

10.2 Background

To harness GPU capabilities, application developers may choose between two main approaches. In the first approach, they use specific computing languages, such as CUDA [190] or OpenCL [191], for writing device code and for managing data transfers. Those languages provide application developers the finest control of the computation. However, even if the language is based on C/C++, they require to rewrite the algorithm according to the memory model and the parallelization scheme of the chosen language. Moreover, they introduce a maintainability problem since the device code is not usually suitable for running on the host device, which leads to code duplication.

A second approach is to decorate the original source code with compiler directives to highlight the region of code to offload and to describe data transfers between the host and device memory. The compiler generates automatically the device code and the required glue code for data transfer. The benefit of this approach is that the application is written in a single language, which may run on the device and the host as well. However, since the device code is automatically generated, it may suffer from performance penalties. Moreover, application developers are still in charge of exposing enough parallelism and of minimizing control flow operations, to have a performance improvement.

In this chapter we decided to use the directive language OpenACC [192] to exploit GPU capabilities. The starting point is an already optimized code for the CPU, that was already designed to expose parallelism to leverage the CPU vector units.

```
Input: Target Pocket and the initial pose of the ligand
Output: The geometric score of the evaluated poses
repeat
    $Generate\_Starting\_Pose(Pose\_id)$;
    for $angle\_x$ in range(0:360) do
        $Rotate(angle\_x, Pose\_id)$;
        for $angle\_y$ in range(0:360) do
            $Rotate(angle\_y, Pose\_id)$;
            $Evaluate\_Score(Pose\_id)$
        end
    end
    for $fragment$ in $ligand\_fragments$ do
        for $angle$ in range(0:360) do
            $Rotate(fragment, angle, pose\_id)$
              $Bump\ Check(fragment, pose\_id)\ Score(fragment, pose\_id)$
        end
    end
until $Pose\_id < N$;
```

Algorithm 10.1: Pseudo-code of the original algorithm that performs the geometrical docking for the CPU.

10.3 The Proposed Approach

This section describes the approach that we followed to accelerate the geometrical docking kernel of the docking application on GPUs. First, we analyze the application to identify opportunities to offload computation to the GPU. Then, we describe how we seized those opportunities to improve the application performances.

10.3.1 Application Description

LiGenDock application uses a mixed approach for docking a *ligand* in the target *pocket* . It starts considering geometric features, then it simulates the actual physical and chemical interaction for the most promising *ligand* poses. With GeoDock we focus only on the geometrical docking phase, used to filter out incompatible *ligands* .

Algorithm **??** shows the pseudo-code of the geometrical docking. Due to the high number of degrees of freedom, it is unfeasible to perform an exhaustive exploration of the possible pose of the *ligand*. For this reason, the application implements a greedy optimization heuristic with multiple restarts.

The outer loop generates *N* different initial poses for the target *ligand* ,

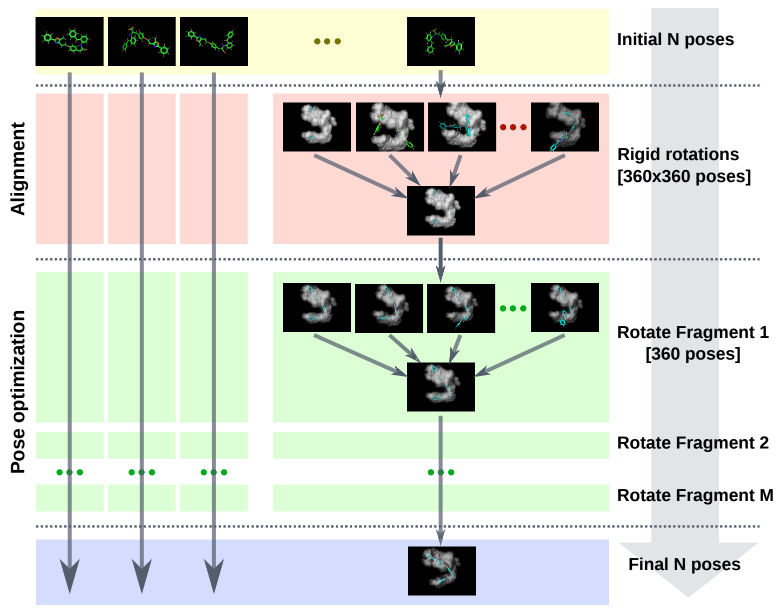


Figure 10.2: *Algorithm work-flow that estimates the ligand final poses, highlighting the independent computations. In this example, we use the 1fm9 pocket with the related co-crystallized ligand .*

Source: *[193]*

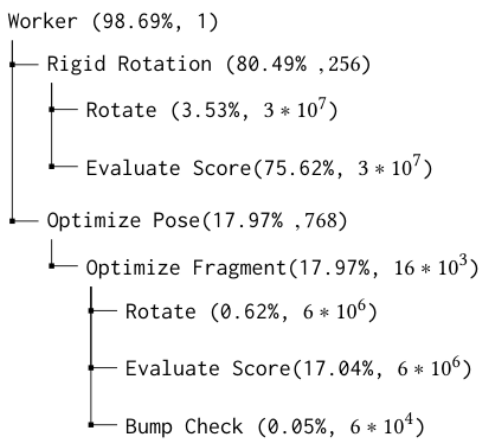
maximizing the probability to avoid local minimum. Each iteration of the outer loop aims at docking the i^{th} initial pose of the *ligand* .

Within the body of the outer loop, the docking algorithm is divided into two sections. The first one (lines 3-9) performs rigid rotations of the *ligand* , to find the best alignment with the target *pocket* , according to the scoring function. We will refer to this section of the algorithm as Rigid Rotation or Alignment. In the last section of the algorithm (lines 10-14), we optimize the shape of the *ligand* by evaluating each *fragment* in an independent fashion (line 10). In particular, we rotate each *fragment* to find the angle that maximizes the scoring function without overlapping with the other atoms of the ligand (lines 11-13). We will refer to this section of the algorithm as Optimize Pose. We need to evaluate each *fragment* sequentially, since a *fragment* may include another *fragment*. Therefore, if we parallelize the pose optimization over the *fragments*, we might change the *ligand* structure in an unpredictable way, invalidating the outcome of the application.

Finally, Figure 10.2 depicts the geometric docking workflow, highlighting data dependencies. In particular, the initial poses are independent, since every initial pose represents the actual starting point of the docking algorithm. For every starting pose, we perform rigid rotations to select the most suitable alignment of the initial pose of the *ligand* for the target *pocket* . After the Rigid Rotations, we proceed with the Pose Optimization phase, evaluating each fragment of the *ligand* sequentially. As output, we retrieve N poses, one for each starting pose.

10.3.2 Profiling

To identify bottlenecks of the application on CPU, we profiled the application using Score-P, a well-known profiling tool [194]. Figure 10.3 reports the result of this analysis for the most significant functions. In particular, for each function we report the percentage of time spent in that function, comprehending children, and the number of times that it is called in the algorithm. From the results, we noticed that the main bottleneck of the application is the scoring function. Even if the function itself is rather simple, we need to call it every time we modify the *ligand* structure, to drive the pose optimization process. In particular, the scoring function evaluates "how good" is the position of every atom of the *ligand* inside the *pocket* . The actual score of the *ligand* is the average score of the atoms. Due to the code optimization, this function leverages the CPU vector units to process the score of each atom, leading to an execution time of less than $100ns$. However, due to the high number of calls from the algorithm (10^7), this



```
Worker (98.69%, 1)
├── Rigid Rotation (80.49% ,256)
│   ├── Rotate (3.53%, 3 * 10^7)
│   └── Evaluate Score(75.62%, 3 * 10^7)
└── Optimize Pose(17.97% ,768)
    └── Optimize Fragment(17.97%, 16 * 10^3)
        ├── Rotate (0.62%, 6 * 10^6)
        ├── Evaluate Score(17.04%, 6 * 10^6)
        └── Bump Check (0.05%, 6 * 10^4)
```

Figure 10.3: *The application profiling result. For each significant function, we show two information: the percentage of time spent in that function (comprehend sub-functions) and the number of calls.*

Source*: [193]*

function becomes the bottleneck of the application. Moreover, the functions that rotate the *ligand* atoms or that test whether a pose is valid, have a negligible impact on the overall execution time, since they are also able to exploit the vector units of the CPU.

From the profiling analysis, the application complexity is not restricted to a single complex function, but it is due to the high number of alternative poses to evaluate for finding the best one. Moreover, since the algorithm is greedy, we need to perform multiple restarts to lower the probability of finding a local minimum. Therefore it seems to fit the parallel nature of the GPU paradigm. On the other hand, we don't have a single kernel to offload to the GPU, but we need to address the whole algorithm, or the data transfer cost would be higher than the benefit.

From the implementation point of view, we decided to use the OpenACC directive language to offload application code to the kernel. Moreover, OpenACC provides to application developers the possibility to explicitly control data transfers, minimizing the related overhead.

10.3.3 Implementation

From the CPU profiling of the application, we implemented a first version of the algorithm that aims at minimizing data transfer, while maintaining the application structure. We decided to introduce parallelism on the number of poses (they can all be managed in parallel and are completely independent). In this way, we transfer data only at the beginning and at the end of the docking algorithm (i.e. once in the lifetime of the *ligand*).

From the implementation point of view, the following changes are required to generate the binary of the offloaded kernel, i.e. the parallel region in OpenACC jargon. All the data structures interacting with the offloaded kernel have to be compliant with the OpenACC guidelines [195] for handling data. In particular, this means that the data structures that interact with the offloaded kernels must manage data transfers in the constructor and destructor. The constructor allocates memory on the device side and it copies the initialized data into device memory. The destructor must free both the device and the host memory. Moreover, it is mandatory to mark each function called inside the parallel region with the OpenACC “routine” directive.

Since we plan to parallelize the computation over the initial poses, we require a private data structure to represent the initial pose of the *ligand*. The OpenACC language provides the `private` keyword to express this concept. However, the system runtime available on our platform was not able to support this feature[1]. Since it is a class containing arrays, whose copy constructor has been redefined according to the OpenACC manual [196], it is not clear the source of the problem. Therefore, we decided to bypass the issue by replicating the initial pose and by using manual management of the data. Even if it required a fair amount of code refactoring, we still tried to maintain the original structure of the application.

With this modification, we fixed the illegal access issue, but the GPU application was slower than the CPU one. We analyzed the problem and noticed that this was not due to data movement since everything was resident on the GPU. We found out that we were not really exploiting the parallelism of the GPU because parallelizing the computation of all the poses was not giving enough work to the GPU. At this point the Rigid Rotations are still happening sequentially for every restart. Indeed, having to insist on the same data structure for all the Rigid Rotations of one pose was limiting the amount of exposed parallelism.

To obtain an advantage from the use of the accelerator, we had to rework the source code to find (and expose) more parallel computation, as shown in Algorithm 10.2. To achieve the desired result, we had to modify the rotation and scoring functions, unifying them to avoid storing all the tem-

[1]The GeoDock execution triggered an illegal access to the GPU memory when trying to transfer the private data structure.

```
**Input:** Target Pocket, initial pose of a ligand
**Output:** A set of scores, one for each pose
repeat
    $Generate\_Starting\_Pose(Pose\_id)$;
until $Pose\_id < N$;
for $angle\_x$ in range(0:360) do
    for $angle\_y$ in range(0:360) do
        repeat
            $Rotate\_and\_Score(angle\_x, angle\_y, Pose\_id)$;
        until $Pose\_id < N$;
    end
end
$Reductions\ Set\_Optimal\_Pose$ repeat
    for $fragment$ in $ligand\_fragments$ do
        for $angle$ in range(0:360) do
            $Rotate(fragment, angle, pose\_id)$
             $Checkbump(fragment, pose\_id)$ $Score(fragment, pose\_id)$
        end
        $Reductions\ Set\_Optimal\_Pose$
    end
until $Pose\_id < N$;
$ResultRetrievalfromGPU$
```

Algorithm 10.2: Pseudo-code of the final algorithm offloaded to the GPU, where the Rigid Rotations are parallelizable. Inside the Rotate_and_Score function the initial pose is only read from the kernel, which evaluates on the fly the score of the atom after applying the rotation. In this way, we eliminate the need to store all the rotation poses, and we can perform the two loops in a complete parallel way. Only the optimal pose will be stored after the reductions (outside of both for loops). The same principle applies to the Pose_Optimization phase, however only the inner loop can be parallelized since the fragments must be managed sequentially.

porary *ligand* poses. With this modification, we were able to expose more parallelism, since the rigid rotations are no more sequentially executed on a shared data structure. After the computation, we schedule a reduction to retrieve the best score, storing only the best pose of the *ligand* for the following step. Finally, we rotate the ligand data structure accordingly, to forward to the optimization phase.

This implementation improved the computation efficiency, moving the bottleneck from the Rigid Rotation section to the Optimize Pose function. We applied the same technique to expose parallel computation also in the pose optimization phase. However, the exposed parallelism is limited by two factors. On one hand, we rotate a fragment along with a 1-dimensional axis instead of a free rotation in a 3-dimensional space. On the other hand, we must optimize each fragment in sequence to save the ligand consistency. Therefore, we exploit again the pattern of parallel evaluation followed by a reduction, but the number of data is smaller: we need to perform a reduction at the end of every fragment evaluation. As previously stated, since we have to process the *fragments* sequentially, it is not possible to expose more parallelism with respect to the first approach.

10.4 Experimental Results

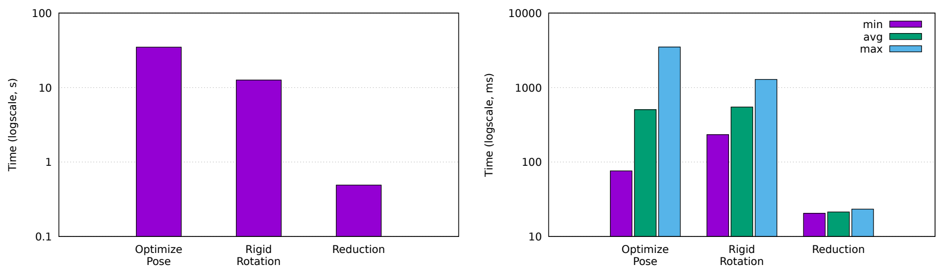
We performed the measurement on a target machine with an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz CPU and an Nvidia Tesla K40m GPU. The operating system was CentOS 7.0, and we compiled the program using PGI 17.10. We compiled the baseline using GCC 5.4, with the avx flag to enable vectorization on top of the O3 optimization level.

We analyzed the performance of the GPU kernel using nvprof [197], in terms of execution time, occupancy, and multiprocessor activity. The GPU occupancy is the number of used warps, in percentage. The multiprocessor activity is the percentage of time when the streaming multiprocessors have one or more warps issuable, i.e. not in a stalled state.

The input dataset for the experiments uses 23 different *ligand* and *pocket* pairs, taken from PDB database [198]. In particular, we used the following *pockets*: *1b9v, 1br6, 1c1b, 1ctr, 1cvu, 1cx2, 1d3h, 1ezq, 1fcx, 1fl3, 1fm6, 1fm9, 1fq5, 1gwx, 1hp0, 1hvy, 1lpz, 1mq6, 1oyt, 1pso, 1s19, 1uml* and *1ydt*. For each *pocket* , we docked the relative co-crystallized *ligand* . We used those molecules to have a correct estimation of the execution time of the application.

10.4.1 Performance evaluation on the GPU

To optimize the application performance, we tried different mapping of the computation on the GPU. OpenACC offers three levels of parallelism: vector, worker, and gang. Vector level parallelism is the SIMT (Single Instruction, Multiple Threads) level on GPU. Gang level is the outer-most parallelism level, where all the elements are independent and the communication between gangs is forbidden. Worker is an intermediate level used to organize the vectors inside a gang. We investigated how these levels of parallelism are mapped on Nvidia GPUs by the PGI compiler. The only related information was found in the PGI development forum, where one of the developers mentioned that "worker is a group of vectors which conceptionally maps to a CUDA warp. Our actual implementation maps a vector to threadIdx.x and worker to threadIdx.y." This means that the vector and worker



(a) *Total execution time of the accelerated kernel, for all the 23 pockets, divided into the different functions.*

(b) *minimum, maximum and average of the execution time of a single call of each function.*

Figure 10.4: *Execution time of the three ported kernels, cumulative and of a single call.*

Source: *[193]*

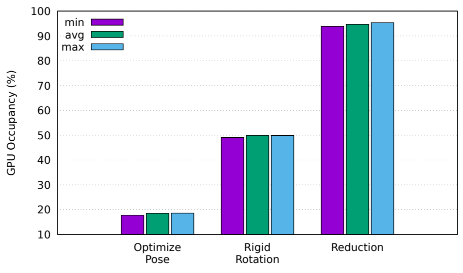
levels are the dimensions of a CUDA block, while the number of gangs is the CUDA grid. Therefore, we split the initial poses at gang level, since all of them are independent. We set all the functions that change the position of the atoms at vector level. The intermediate loops are set at worker level. From the CUDA specification, it is known that the block maximum size is 1024 [199], which can be divided into three dimensions. However, only 2 dimensions are addressable with OpenACC. Using this information, we performed a Design Space Exploration to tune the block size, taking into account Nvidia's recommended best practices. From experimental result, the best size configuration for each function is:

- Rigid Rotations: 8 workers and 128 vector length.
- Optimize Pose: 64 workers with 1 as vector length.

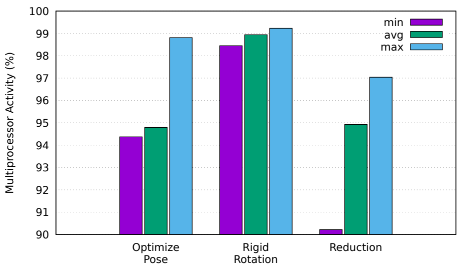
In particular, Figure 10.4a reports the total execution time of the GPU kernels. The total time is the sum of the execution time of a function across all the different datasets. We can notice that on GPU the bottleneck shifted from the Rigid Rotations to the Optimize Pose function.

Focusing on the execution time of single functions, we can notice from Figure 10.4b that the Optimize Pose has the greatest variance. This result is expected since this function depends on the number of *fragments* of each *ligand* , and on how likely they overlap with each other. We can also notice that the execution time for the Reductions is constant.

We also tried to let the compiler select the configuration. In this case, the automatically selected configuration led to a decrease in performance. For example, the compiler selected to organize Rigid Rotations in blocks of



(a) *Occupancy*



(b) *Activity*

Figure 10.5: *GPU utilization of the accelerated kernel, divided into the different functions, and multiprocessor activity in the kernel.*

Source: *[193]*

128 vectors, with no workers, and in 360 gangs, with all the intermediate loop serialized. This configuration achieved a low occupancy (24%) and 4.5 times the execution time (67 seconds). The best solution we found in terms of execution time for the Optimize Pose is to avoid vector parallelism, due to control flow issues in the inner loop.

Even if the selected configuration is the best for the execution time, none of these kernels was able to obtain full utilization of the GPU. It is possible to see the result of this experiment in Figure 10.5a: We were able to reach an almost full utilization only in the Reduction.The Rigid Rotations kernel was able to reach a 50% utilization. The Pose Optimization has low utilization, due to the inherent control flow, i.e. the sequential optimization of the *fragments*.However, as reported in Figure 10.5b, we can notice that all the involved processors, in all the considered functions, are heavily loaded: the lowest result is indeed 90%.

Finally, we analyze the cost of data transfer. From Figure 10.6 we can notice that it can be considered negligible: the total amount of data transferred, considering all the 23 different dataset execution, is less than 100MB as can be seen from the left y-axis. The total elapsed time in data transfers is around 15ms, across all the executions, and it can be seen on the right y-axis.

10.4.2 Performance comparison with baseline

Figure 10.7 shows the execution time of the original kernel on the CPU. As previously mentioned, the most expensive function is Rigid Rotation that takes 206 seconds. We can notice that for this kernel the GPU version has

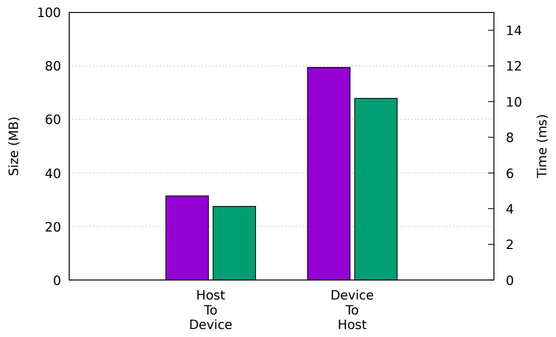
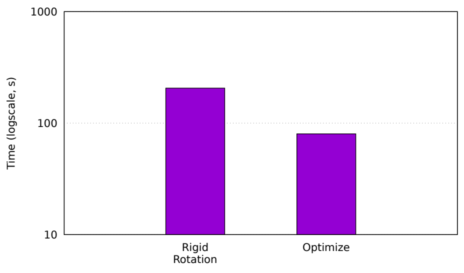
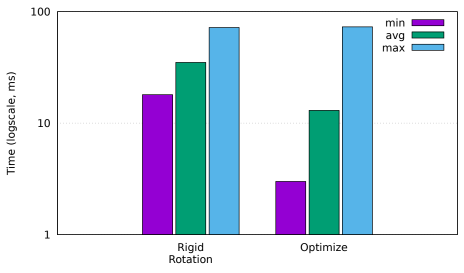


Figure 10.6: *Data Transfer between CPU and GPU*

Source: *[193]*



(a) *Cumulative execution time of the original CPU kernels*



(b) *Minumum, maximum and average execution time of a function call in the original CPU application*

Figure 10.7: *Execution time of the original kernels on the CPU.*

Source: *[193]*

Metric	Original	OpenACC Version
Execution Time	298s	831s
Number of Intructions	2,849,354,869,375	6,309,979,483,835
Cache Misses	380,672	783,114,693
IPC	3	2.4

Table 10.1: *Comparison of the original application and the execution of the OpenACC application on the CPU: OpenACC version shows worse performance overall due mostly to data management, as the huge increase in cache misses shows.*

a speedup of 16x (from 206s to 12s). On the other side, the Optimize Pose has only a 2x speedup (from 80 to 34), even if in the GPU version we are performing all the initial poses in parallel. This behavior is expected since we can exploit more parallelism in the Rigid Rotation function, while the sequential nature of Optimize Pose hinders the GPU performance.

If we observe the single function execution times for the CPU in Figure 10.7b, we can notice that even on CPU, the Optimize Pose has the largest variance.

10.4.3 Performance evaluation on the CPU

One of the reasons for choosing OpenACC over a CUDA implementation was to have a single source code for different architectures. Given the changes in the application that we made to optimize the performance on the GPU, this experiment aims at evaluating the performance of the new application on the CPU. From the execution time perspective, we noticed almost a 3x slowdown. To investigate the reasons behind this behavior, we used linux perf to analyze the performance counters. The results of the experiment are reported in Table 10.1.

Even if the IPC is slightly lower, the cache misses are 3 orders of magnitude higher. Moreover, the number of instructions is more than doubled. As expected, the replicated initial poses of the *ligand* and the inserted code to perform the reductions deteriorates the performance on the CPU. If the GPU programming paradigm requires independent data, to leverage the architecture parallelism, CPU architectures benefit from data locality. Moreover, on GPU the best practice is to perform the same operation on different data, while on CPU it is better to perform different operations on the same data.

10.5 Summary

In the drug discovery process, the virtual screening of a large chemical library is a crucial task. The benefits of an improvement in the time spent on evaluating the interaction from a *ligand* and the target *pocket* , are twofold. On one side it reduces the monetary cost of the process, on the other side it enables the end-user to increase the number of the evaluated *ligands* , increasing the probability of finding a better solution.

In this chapter, we focued on a newer version of GeoDock, again optimized only for the CPU, and we performed a porting of the most compute intensive kernel to the GPU using the OpenACC directive language. We performed an experimental campaign to evaluate the performance of the application in terms of execution time, occupancy, and multiprocessor activity. We also evaluated the OpenACC paradigm as "write once run everywhere", and noticed that the application has to be changed to obtain performances on different architectures.

We believe that it is possible to further improve the obtained results with a different approach. In the following chapter, we reorganized the application structure to exploit asynchronous queues and offload only the sections with heavy parallelism (i.e. the Rigid Rotation kernel), while using the CPU for the control flow bound sections (i.e. the Optimize Pose kernel).

The outcome of this work has been published in the 6th international workshop on Parallelism in Bioinformatics 2018 [193].

CHAPTER 11

Optimizing GeoDock Throughput in Heterogeneous Platforms

In this chapter, we further optimize the geometric docking application to take full advantage of the whole heterogeneous platform. In particular, in the previous chapter, we have seen that simply porting the kernel to the GPU gives an improvement in performances, however, it has drawbacks. The most obvious one is that it leaves the CPU idle. Moreover, even if we consider using the CPU to run the same kernels in parallel, distributing the dataset between the two compute units, we noticed that different sections of the application are more suitable for the different architectures.

With this chapter we investigate a different way to divide the computation across the available computing resources, trying to bind the computation to the most suitable architecture. From the implementation point of view, we used OpenMP on top of the OpenACC implementation to organize the computation.

We show with an experimental campaign that this approach is able to fully exploit the underlying node and obtain a better throughput up to 25% just with a re-organization of the computation without changing the computational resources available on the node.

11.1 Introduction

As we already mentioned, power consumption is becoming a key factor in the HPC context. For this reason, accelerators have begun to be used alongside the traditional CPUs in this context. Among them, the most commonly used accelerators are GPGPUs. Indeed, most of the top positions in both the Top500 and Green500 are occupied by heterogeneous platforms that exploit the GPGPU as an accelerator. Depending on the application algorithm, hardware accelerators might significantly improve the application throughput with respect to general purposes CPUs, considering the same power consumption.

However, to create a heterogeneous application the programmer must consider the characteristics of the application and of the available compute units. In the previous chapter (Chapter 10) we investigated the benefits and limitations of using the OpenACC [192] language extension in a molecular docking application, to accelerate the computation done in the most compute-intensive kernels of GeoDock. In this chapter, we implement a hybrid version by using OpenMP [200] and OpenACC to leverage optimally all the processing elements on a heterogeneous node. In particular, given the limitation analyzed in the previous chapter, we aim at mapping each phase of the application on the most suitable processing element. To summarize, the contributions of this chapter are the following:

- We propose a Hybrid CPU/GPU version of the geometric docking algorithm capable to fully exploit the node heterogeneity;
- We analyze the resource utilization of the different solutions to find the best configuration also in presence of multi-GPU nodes, searching for the best balance.
- We discuss the obtained results, comparing them with the sequential CPU application, the GPU implementation of the previous chapter and the traditional data-splitting across the different compute units of the dataset

In the context of the global framework, this chapter is focused on optimizing heterogeneity in the application, and not on the self-tuning at runtime. Indeed, as we can notice in Figure 11.1, the involved components are the same as the previous chapter: the enriched code with the GPU kernel, without any runtime autotuning functionality inserted in. Nonetheless, this work is still in the context of the global framework, because it is focused on tuning the application. The performed tuning in this case is not automatical

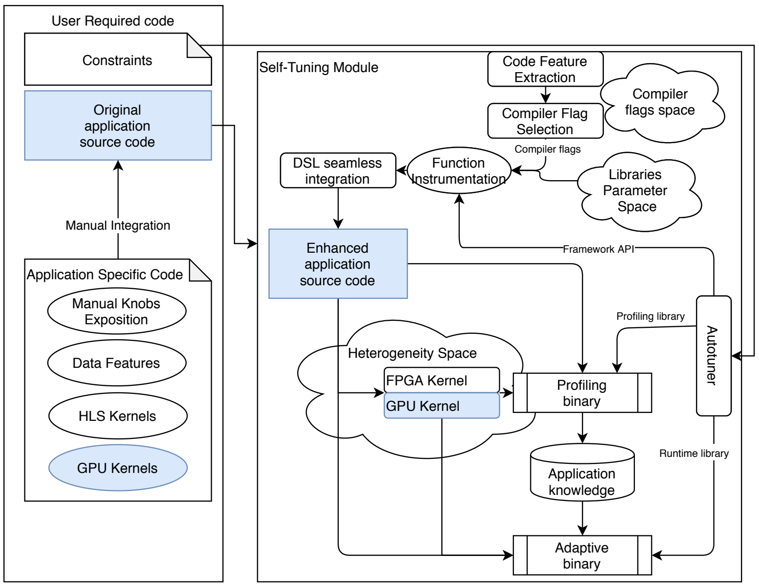


Figure 11.1: *Highlight of thesis approach targeted in this chapter.*

but manual and involves a re-writing of the source code to re-organize the parallelism and the exploitation of the GPU. The tuning performed in this chapter is interesting because introduces the concept of optimality of a kernel with the underlying architecture, and the idea of selecting the running place of the different pieces of the application according to their characteristics.

11.2 Background

In this chapter, we leverage directive-based languages, like OpenACC [192] and OpenMP [200] to improve the original application and manage the parallelism of the heterogeneous node. In these languages, the application developer uses compiler directives to annotate the source code. The toolchain transforms and compiles the offloaded kernels, and generates the code to transfer the data between host and device. Moreover, it automatically generates the initialization code. The benefit of this approach is the ease of use. The programmer writes the entire application with a single language independently from the actual target, i.e. CPU host or the accelerator. With this approach, a single source code can be executed on different hardware, thus enabling functional portability. However, the application developer is still in charge of writing an algorithm suitable for the device memory model and parallelization scheme. Despite the multi-platform approach of those languages, the kernels must be tuned according to the target platform, since performance portability is still an open problem [201]. Indeed, one of the conclusions of the previous chapter is that the code modified and optimized for the GPU was not efficient when compiled and run on the CPU as the original code.

To improve computation efficiency, we aim at optimizing the exploitation of all the computational resources available in an HPC node, while continuing to use MPI for inter-node communication. In modern systems, the HPC node typically includes several CPUs and GPUs. The multi-GPU problem has been investigated in literature. For example, the approaches proposed in [202, 203] suggest extending OpenMP to support multiple accelerators seamlessly. OpenACC has runtime functions to support the utilization of multiple GPU, however, lacks GPU to GPU data transfer, in single node [204] and multinode [205]. A previous work in literature [206] investigates a hybrid approach with OpenMP and OpenACC. It proposes the usage of OpenMP to support a multi-GPU OpenACC application, assigning each GPU to an OpenMP thread. In this context, each OpenMP thread performs the data transfer between the host and the target device, without performing any other computation.

In this chapter, we extend the previous one by suggesting a new approach that offloads to the GPU the most compute-intensive kernels, using a hybrid approach of OpenMP and OpenACC. In this way, we exploit multi-GPU nodes offloading to accelerators only the kernels that maximize the advantage of being run on the GPU, while keeping the utilization of this resource as high as possible. Unlikely previous approaches, we rely on the CPU to execute the kernels that are less suitable for the GPU. Moreover, to maximize the utilization of the resources, every OpenMP thread has a GPU associated. In this way, the computing thread has access to both the CPU and the GPU. Thanks to this, we can split the workload across the compute units, associating the kernels to the compute units according to the characteristics of the function to evaluate and of the compute unit itself.

11.3 The Proposed Approach

In this section, we describe the proposed approach to accelerate GeoDock using GPUs. First, we quickly recap in Section 11.3.1 the work done in the previous chapter where we accelerated the whole algorithm with GPUs, relying on OpenACC. Then, we describe in Section 11.3.2 the analysis that leads us to implement the hybrid OpenMP/OpenACC solution. As we have already explained, the idea is to allocate the workload on the heterogeneous

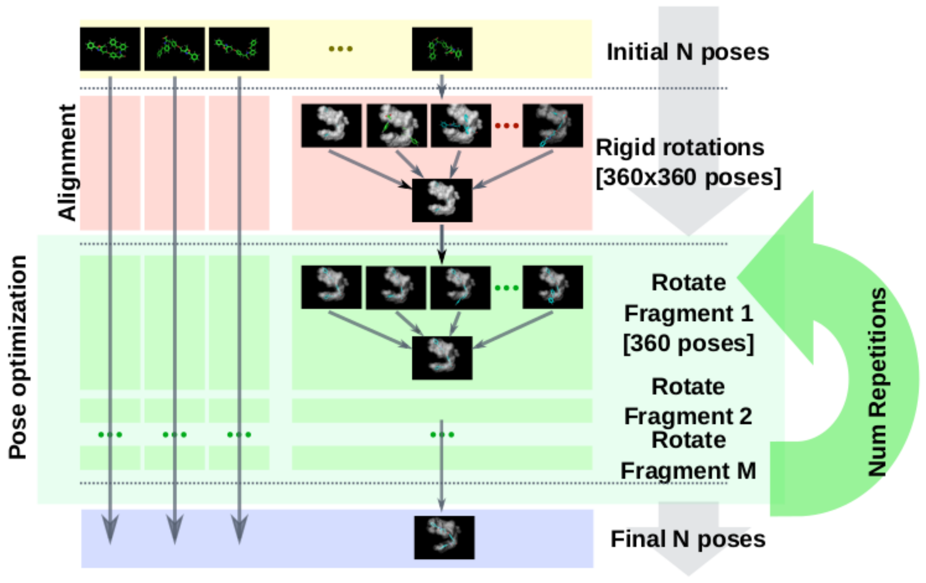


Figure 11.2: *Overview of the GPU implementation of the docking algorithm. Each box represents a computational part of the application that might be executed independently. The optimization phase must evaluate each fragment sequentially and the whole procedure might be repeated to refine the final result.*

***Source**: [207]*

resources according to different hardware capabilities, to improve the overall performance of the application.

11.3.1 OpenACC Implementation

In Chapter 10 we developed a pure GPU version of the algorithm, where both the main kernels (alignment and pose optimization) are offloaded to the GPU. In particular, starting from the profiling analysis, we implemented a first version of the algorithm that aims at minimizing the data transfer while maintaining the application structure.

Figure 11.2 shows a graphical representation of the algorithm described in the previous chapter, highlighting independent sections of the algorithm, by using different boxes. We might consider each restart of the docking algorithm as a different initial pose. Given that every initial pose might proceed independently, we have the first level of parallelism to map on the GPU. Given an input ligand, it is possible to generate and dock the ligand

Listing 11.1: *Pseudo-code of the GPU algorithm.*

```
 1  load(pocket);
 2  for (ligand : ligands)
 3  {
 4      ligand_t ligand_arr[N];
 5      #pragma acc parallel loop
 6      for (pose_id = 0; pose_id < N, pose_id++){
 7          ligand_arr[pose_id] = ligand;
 8      }
 9      #pragma acc parallel loop gang
10      for (pose_id = 0; pose_id < N, pose_id++){
11          generate_starting_pose(ligand_arr[pose_id]);
12          #pragma acc worker
13          align_ligand(ligand_arr[pose_id], pocket);
14          #pragma acc loop seq
15          for(rep = 0; rep <num_repetitions; rep++){
16              #pragma acc worker
17              optimize_pose(ligand_arr[pose_id], pocket);
18          }
19      }
20  }
```

initial poses on the device side. All the phases of the docking algorithm are performed in parallel, on different data, and we only extract the result at the end of the algorithm. In this way, we transfer data only at the beginning and at the end of the docking algorithm (i.e. once in the lifetime of the ligand).

From the implementation point of view, we use OpenACC to avoid rewriting the application source code in a different language. OpenACC can operate with data structures that are resident on the GPU and usable across different kernels. The implementation details are reported in Chapter 10.

Listing 11.1 describes the pseudocode of this GPU implementation. We can notice the replication of the data before the docking procedure, needed to increase the exposed parallelism. The original ligand is copied according to the number N of multiple restarts of the algorithm (lines 4–8). Once we initialize the memory on both the device and the host side, we evaluate each starting pose in the parallel region (lines 9–19) offloaded to the GPU. It is possible to notice how in the pseudocode there are no pragmas for transferring data between host and device. All the data transfers are managed with constructors and destructors of the data structures, according to the OpenACC standard. To leverage all the levels of parallelism available in the GPU, we inserted different levels of parallelism in the code as well. OpenACC offers three levels of parallelism: vector, worker, and gang. Vector level parallelism is the SIMT (Single Instruction, Multiple Threads) level on GPU. Gang level is the outer-most parallelism level, where all the elements are independent and the communication between gangs is forbidden. Worker is an intermediate level used to organize the vectors inside a gang.

In particular, the vector and worker levels are the dimensions of a CUDA block, while the number of gangs is the CUDA grid. Therefore, we split the initial poses at gang level, since all of them are independent (line 9–10). We set all the internal functions (not shown in Listing 11.1) that change the position of the atoms at vector level. The intermediate functions (i.e. `align_ligand` and `optimize_pose`) are set at worker level (lines 12–13 and 16–17). The pose optimization loop (lines 15–18) is marked with a `loop seq` pragma. This is mandatory to force the compiler to execute that loop sequentially.

To evaluate the performance of this GPU version, we profiled the application and compared the results with the CPU baseline. In the alignment phase, we obtained a good speedup (16x). We noticed that the pose optimization was less suitable for GPU acceleration since too few operations per kernel were possible. Indeed, the sequentiality of the fragments and the control flow operations, inserted by the correctness checks, limit the reached speedup over the baseline CPU version. The final speedup for this kernel was only 2x. Moreover, the profiling results show how the bottleneck of the application is changed. With the GPU version, approximately 70% of the time is spent in the Optimize Pose kernel, while the Alignment takes less than 30%. This is a different result with respect to the profiling done on the baseline application on CPU.

More in-depth analyses of this are described in Chapter 10.

11.3.2 Hybrid OpenMP/OpenACC Implementation

The GeoDock application implemented in GPU, and described in the previous section, has two main limits. On one hand, it is not able to use the available CPU cores to perform the computation. On the other hand, not all the phases of the application can fully exploit the architectural features of the GPU. As a consequence, the application is wasting or misusing a large fraction of the node computation capabilities. Given that our target is to optimize the performance of GeoDock on the full node, this section investigates the possibility to split the workload between CPU and GPU.

Starting from the profiling information of the GPU implementation, instead of simply partitioning the data among CPU processes and GPU processes, we modified the algorithm to bring the pose optimization phase back to the CPU. In particular, we would like to exploit the multicore architecture, enabling each CPU thread to evaluate one ligand, and offloading only the alignment kernel to the GPU. The basic idea, depicted in Figure 11.3, is to exploit the GPU for the kernel that benefits most of the massive-parallel

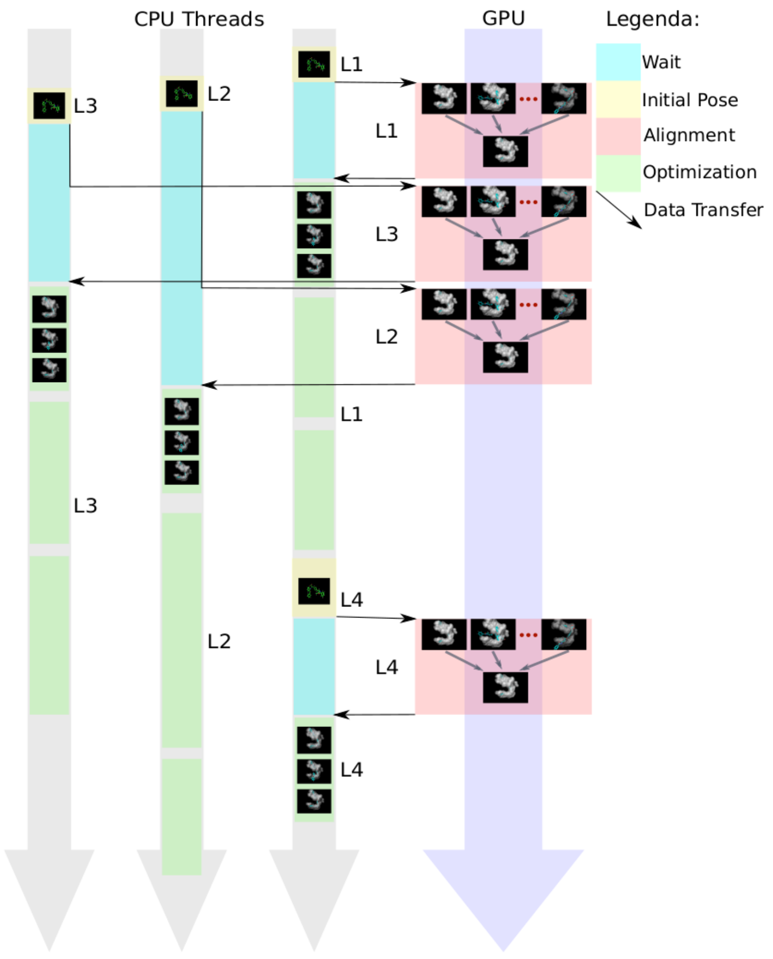


Figure 11.3: *Overview of the hybrid OpenMP/OpenACC implementation of the docking algorithm. The alignment phase of the ligand is offloaded to GPU, while the optimization phase is performed in the CPU. Each OpenMP thread interacts with a single GPU. The arrows identify data transfer between host and device.*

***Source**: [207]*

Listing 11.2: *Pseudo-code of the hybrid algorithm.*

```
1   load(pocket);
2   #pragma omp parallel
3   for (ligand : ligands)
4   {
5       #pragma omp single nowait
6       #pragma omp task
7       {
8           ligand_t ligand_arr[N];
9           #pragma omp critical
10          #pragma acc data
11          {
12              #pragma acc parallel loop
13              for (pose_id = 0; pose_id < N, pose_id++){
14                  ligand_arr[pose_id] = ligand;
15              }
16              #pragma acc parallel loop gang
17              for (pose_id = 0; pose_id < N, pose_id++){
18                  generate_starting_pose(ligand_arr[pose_id]);
19                  #pragma acc worker
20                  align_ligand(ligand_arr[pose_id], pocket);
21              }
22          }
23          for (pose_id = 0; pose_id < N, pose_id++){
24              for(rep = 0; rep <num_repetitions; rep++){
25                  optimize_pose(ligand_arr[pose_id], pocket);
26              }
27          }
28      }
29  }
```

architecture while mapping the other kernels to the CPU. Each CPU thread takes care of a different ligand, avoiding data movement among threads, therefore maximizing also the parallelism determined by the ligand library. The only data movements are between CPUs and GPUs.

In this GeoDock implementation, we use OpenACC for the GPU kernel programming, while we exploit OpenMP for the CPU-level parallelism. Listing 11.2 shows the pseudocode of the algorithm. The outermost loop that iterates over the ligand library is parallelized using an OpenMP parallel region (line 2), where for every different ligand we create a `single nowait` task (lines 5–6). *Task* is a construct that was introduced in OpenMP 3 and it is used to describe parallel jobs leaving the organization of the parallelism to the scheduler. They are particularly effective for parallelizing irregular algorithms. The `single` keyword specify that an OpenMP region (in this case the *task*) executes a single instance of the related region. It is required to enforce that each *task* is in charge of an iteration of the outermost loop. The `nowait` keyword is used to skip the implicit barrier at the end of the single region. In this way, the thread does not wait for the completion of the task but can execute the next iteration of the loop.

The execution time spent for docking a ligand depends on several factors, such as the number of atoms, the number of fragments, and geometrical properties of both the target pocket and ligand. Since these factors might drastically change between ligands of the same library, we might consider our docking algorithm as an irregular application. Therefore, the proposed implementation leverages the tasks construct to create a task for every ligand to be docked. As soon as an OpenMP thread becomes free, a pending task is assigned to it, until there is a task waiting to be executed. Moreover, we use the tied task implementation to limit migration, restraining a task to be executed on the same thread that generated it. Moreover, we bind each OpenMP thread to a physical core, by using the OpenMP environment variable `OMP_PLACES=cores`. In this way, we can associate a ligand to one physical core, avoiding the extra movement of the data.

Beside the *Task* construct, we tried to exploit other OpenMP strategies to make the application parallel. In particular, we evaluated the *taskloop* keyword, which is a construct used to create one task for each iteration of a loop, and the traditional *parallel for*. The first construct is not supported by PGI 17.10, so we were not able to exploit it. The *parallel for* has empirically shown slightly worse throughput in the considered case. We also tried all the OpenMP scheduling algorithms (static, dynamic, and guided) and none of them was able to improve the performance obtained with the task construct.

The GPU kernel is implemented inside an OpenMP `critical` region (line 9) to avoid race conditions. We also considered using OpenACC features, such as asynchronous queues, however, they performed worse than this implementation. We exploit the implicit barrier at the end of the parallel region to enforce thread synchronization at the end of the library of ligands to be docked.

The GeoDock algorithm implementation is similar to the one described in Section 11.3.1. In particular, the data replication (lines 12–15) and the alignment phase (lines 16–21) are almost the same. The only difference is in the data structure implementation, due to the limited support of C++ standard libraries from OpenACC. For this reason, we manually managed data copies before and after the critical section used to offload the alignment to the GPU. These changes are omitted in the application pseudocode. However, we encountered a key issue in the memory management of the hybrid solution. In the GPU version, we used CUDA unified memory to reduce the impact of data organization on the application developer. This feature enables addresses accessible from different types of architectures (normal CPU and CUDA GPU cores) hiding the complexity of the management from the programmer. If we use this implementation, the Unified Memory

Listing 11.3: *Pseudo-code of the hybrid multi-GPU algorithm.*

```
1   load(pocket);
2   omp_lock_t lock_array[N_GPUS];
3   #pragma omp parallel
4   for (ligand : ligands)
5   {
6       #pragma omp single nowait
7       #pragma omp task
8       {
9           ligand_t ligand_arr[N];
10          omp_set_lock(lock_array[tid%N_GPUS]);
11          #pragma acc set device_num(tid%N_GPUS)
12          #pragma acc data
13          {
14              #pragma acc parallel loop
15              for (pose_id = 0; pose_id < N, pose_id++){
16                  ligand_arr[pose_id] = ligand;
17              }
18              #pragma acc parallel loop gang
19              for (pose_id = 0; pose_id < N, pose_id++){
20                  generate_starting_pose(ligand_arr[pose_id]);
21                  align_ligand(ligand_arr[pose_id], pocket);
22              }
23          }
24          omp_unset_lock(lock_array[tid%N_GPUS]);
25          for (pose_id = 0; pose_id < N, pose_id++){
26              for(rep = 0; rep <num_repetitions; rep++){
27                  optimize_pose(ligand_arr[pose_id], pocket);
28              }
29          }
30      }
31  }
```

support for the Kepler architecture fails to properly allocate memory[1]. To solve this issue, we manually manage the memory allocation and transfers, by using OpenACC pragmas. For this reason, we created a data region around the offloaded kernels (line 10). The pose optimization kernel is no more decorated with pragmas (lines 23–26) because it is executed on the CPU, therefore we need to iterate over the aligned poses (line 23).

Tuning considerations.

The hybrid approach requires careful tuning to efficiently exploit the computing resources of the heterogeneous node. We can highlight two possible problems: GPU idle time and CPU thread waiting time. In the first case, the CPU threads are not able to provide enough data to fully exploit the GPU, leading to resource underutilization. It is possible to notice this effect in Figure 11.3 on the GPU side. After the execution of the alignment phase of the ligand L2, all the other CPU threads are still busy on the pose optimization phase. Therefore, The GPU is in idle state until the alignment

[1]When we enable the multi-threading with OpenMP, the CUDA managed memory fails. The manager tries to allocate the memory, from different threads, in the same area and returns a runtime error.

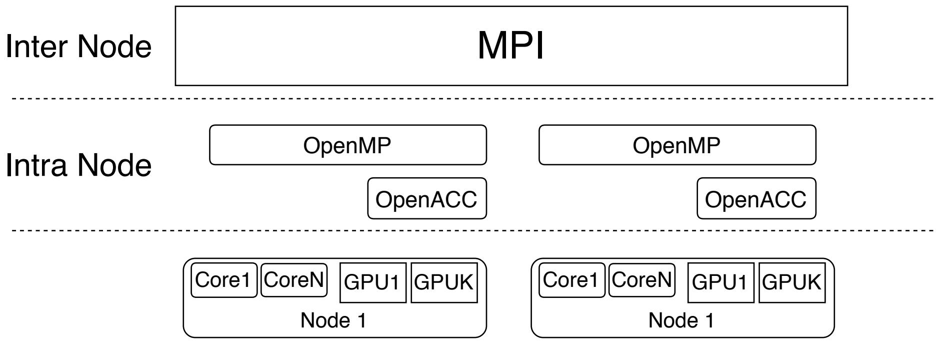


Figure 11.4: *Complete organization of the application.*

Source: *[207]*

of the ligand L4 is offloaded. The second problem happens when there are too many CPU threads and the GPU is overloaded. In this case, each CPU thread can have a long waiting time before accessing the GPU to offload the alignment kernel. In Figure 11.3 we can notice an example of this problem at the beginning of the execution, where the ligand L2 and the ligand L3 are waiting for alignment of the ligand L1 to end. For these two reasons, balancing the load between CPU and GPU is very important. In the experimental results, we show how we tuned the number of threads to optimize the full node performance.

Multi-GPU.

The considerations on the application tuning are even more important when we address multi-GPU nodes. From the implementation point of view, to distribute the workload across multiple devices, it is enough to provide different values to the `#pragma acc set device_num(...)`. We used the thread number to decide on which GPU the thread will offload the kernel.

Moreover, we substituted the original critical section with an OpenMP mutex. This gave us the possibility to exploit the parallelism in the kernel offloading having one kernel in each GPU. The algorithm is reported in Listing 11.3. In particular, we set the device using the related OpenACC pragma (line 11), after locking the mutex (line 10). In this way, a set of threads is associated with a single GPU. As already mentioned, tasks are associated with a thread only when they start executing, and not at their creation. This characteristic of tasks manages the load balancing.

Multi-Node.

GeoDock has been designed to run on large HPC machines, exploiting MPI for inter-node communication. The complete application organization is reported in Figure 11.4. OpenMP and OpenACC are used to manage CPU and GPU parallelism within the node as described in previous sections. MPI is used to handle data parallelism across the different nodes. We employ a pure Master-Slave paradigm, where the master process dispatches groups of ligands to the slave processes. However, the focus of this work is intra-node optimization, since the advantages obtained on a single node are replicated in all the involved nodes. For this reason, we will run and analyze all the experiments on a single node, neglecting the MPI overheads due to the communication with the master process.

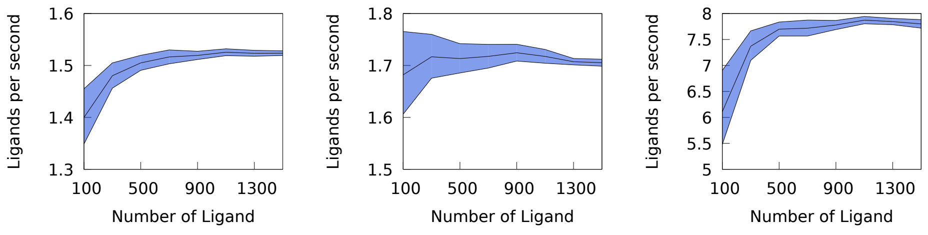
11.4 Experimental Results

We performed the experimental campaign using a single GPU node of the GALILEO2 machine at CINECA[2]. The target node is equipped with a 2x8-core Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz CPU and two NVIDIA Tesla K80 GPU cards. The operating system was CentOS 7.0, and we compiled the program using PGI Compiler 17.10 enabling the *fastsse* flag to activate the vectorization on top of the *O3* optimization level.

The data shown in this section are the results of several runs using different sets of ligands and a single target pocket to keep into consideration possible performance variability. The evaluation has been done considering a large set of ligands. The ligands complexity was different in terms of number of atoms and fragments. In particular, the number of atoms per ligand across the whole set is on average 39.6 with a standard deviation of 6.7. The maximum number of atoms is 73, and the minimum is 28. The average number of fragments is 13.3 with a standard deviation of 4.3. The maximum number of fragments is 34 while the minimum is 6.

Given that the application performance is dependent on the ligand complexity, our first experiment wants to define a reasonable number of ligands that is large enough to absorb those differences. Figure 11.5 shows the convergence analysis done by increasing the size of the set of ligands used for the experiment. In particular, we performed several runs with a different number (and different set) of ligands, and we measure the application throughput reporting the average and the standard deviation. We can see that all three different versions of the code can be considered stable with

[2]http://www.hpc.cineca.it/hardware/galileo-0



(a) *Pure CPU* **(b)** *Pure GPU* **(c)** *Hybrid*

Figure 11.5: *Analysis of the stability of the time required to dock a ligand.*

***Source**: [207]*

few ligands to dock (>500). In all the next experiments we used 1500 ligands, to further absorb the possible performance variance. This set of ligands can be seen as a workload of a slave MPI process running on a single node in the context of a larger master-slave MPI application.

To easily and fairly compare the different implementations we presented, we need to define the terminology that we use in the following sections. We use the term CPU process CPU^{proc} when we refer to an MPI process executing the baseline CPU version of the docking algorithm, described in Chapter 10. We use the term GPU process GPU^{proc} when we refer to an MPI process executing the OpenACC version presented in Chapter 10 and recalled in Section 11.3.1, which uses one CPU thread and one GPU. Finally, we use the term hybrid process $HY^{proc}_{\#ompTh,\#GPUs}$ when we refer to an MPI process executing the OpenMP/OpenACC version described in Section 11.3.2. In particular, $\#ompTh$ is the number of OpenMP threads, while $\#GPUs$ is the number of used GPUs. The required balancing among the threads is needed to optimally use the underlying resources. However, there is no need to divide the amount of data a-priori, since the MPI master will dispatch the ligands upon request of the slave process, thus avoiding unbalances among the processes. For this reason in the following analysis we will only consider how to divide the available resources into CPU^{proc},GPU^{proc} and $HY^{proc}_{\#ompTh,\#GPUs}$.

11.4.1 Single GPU

This section analyzes the performance of the proposed hybrid implementation of GeoDock, focusing on a single GPU case, to compare with previous implementations. The first experiment aims at defining a baseline throughput, in terms of ligands per second, using the reference dataset. Figure 11.6

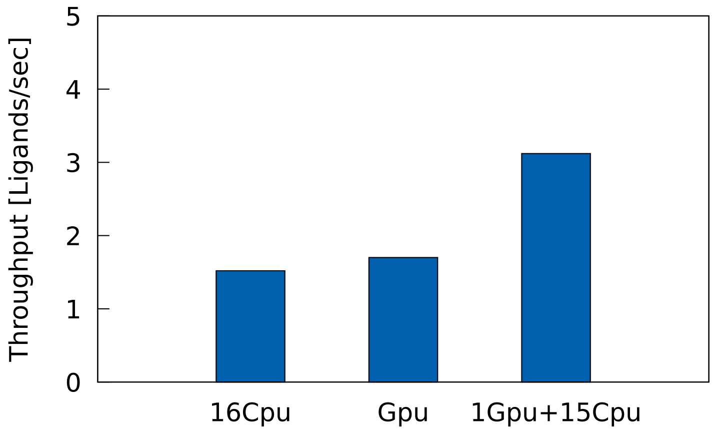


Figure 11.6: *Throughput of the GeoDock application considering a single GPU, by varying its configuration.*

Source*: [207]*

shows the throughput computed with different configurations of GeoDock implementations: (i) by using all the CPUs ($16\times CPU^{proc}$); (ii) by using the GPU ($1 \times GPU^{proc}$); (iii) by using the GPU and the remaining CPU cores ($15 \times CPU^{proc} + 1 \times GPU^{proc}$). The number of CPU^{proc} and the sum of CPU^{proc} and GPU^{proc} has been kept equal to the number of cores available in the node because having more OpenMP threads than CPU cores does not increase the performance. The results show how a single GPU^{proc} has a speedup of 1.1x with respect to the original application that exploits all the CPUs of the node ($16 \times CPU^{proc}$). Therefore, if consider the configuration that uses the GPU and the CPU cores ($15 \times CPU^{proc} + 1 \times GPU^{proc}$), we can achieve a speed-up of 2x. We use this third configuration as the baseline for comparing the proposed hybrid approach.

The second experiment aims at analyzing the performance of the hybrid solution, by varying the number of OpenMP threads from 1 to 16. Figure 11.7 shows the experimental results. In particular, the x-axis represents the number of the OpenMP threads used in the evaluated configuration ($HY^{proc}_{n,1}$, with n={1,...,16}). The y-axis represents the reached throughput. The solid line represents the throughput reached only by the $HY^{proc}_{x,1}$, while the dashed line represents the throughput when it is combined with a number of CPU processes sufficient to fill the node, i.e. $HY^{proc}_{n,1} + (16 - n) \times CPU^{proc}$.

The results can be split into two different regions. On the left side of

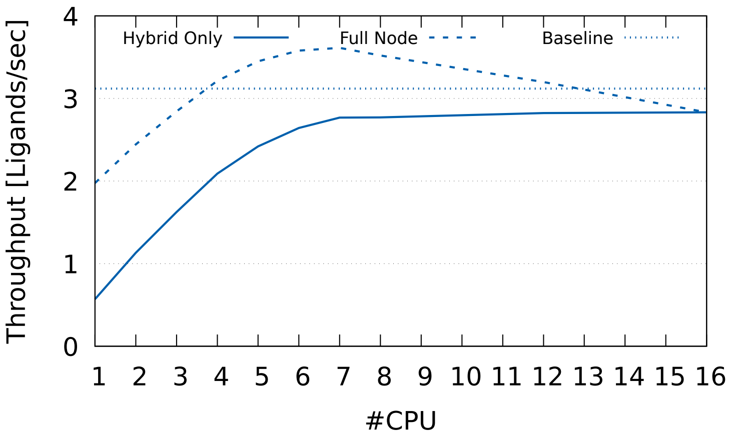


Figure 11.7: *Scaling analysis of the hybrid approach in terms of throughput, by changing the number of OpenMP threads. Baseline represents the* $15 \times CPU^{proc} + 1 \times GPU^{proc}$ *application.*

Source: *[207]*

the figure, we can see that by increasing the number of OpenMP threads up to 7 the performance of $HY^{proc}_{n,1}$ almost linearly increases. This is mainly due to the increment of GPU usage. Few CPU threads are not able to fully exploit the GPU. On the other hand, starting from 7 OpenMP threads the $Hy^{proc}_{n,1}$ performance reaches a saturation point since the GPU starts to be the bottleneck. Indeed, at 7 OpenMP threads, the GPU is already fully used, and with more CPU threads feeding it does not increment the throughput. Similarly, the performance of $HY^{proc}_{n,1} + (16 - n) \times CPU^{proc}$ reaches the maximum throughput when the saturation for the hybrid version happens (i.e. $HY^{proc}_{7,1} + 9 \times CPU^{proc}$). After this configuration, the performance of $HY^{proc}_{n,1} + (16 - n) \times CPU^{proc}$ reduces while increasing the number of the OpenMP threads for the hybrid version. If we use more CPU threads for the hybrid version, we are reducing the number of the CPU process we can exploit and thus their cumulative throughput contribution. The result for the optimal configuration reports a speedup of 1.16x with respect to the $15 \times CPU^{proc} + 1 \times GPU^{proc}$ configuration, which was set as our baseline.

11.4.2 Multi-GPUs

This experiment analyses the performance of the hybrid approach according to the number of available GPUs in the target node. The analysis is done by considering up to 4 GPUs which is the limit of our node (each

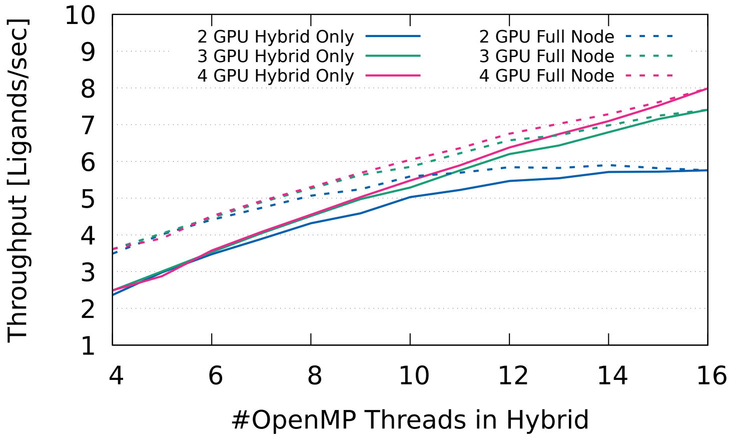


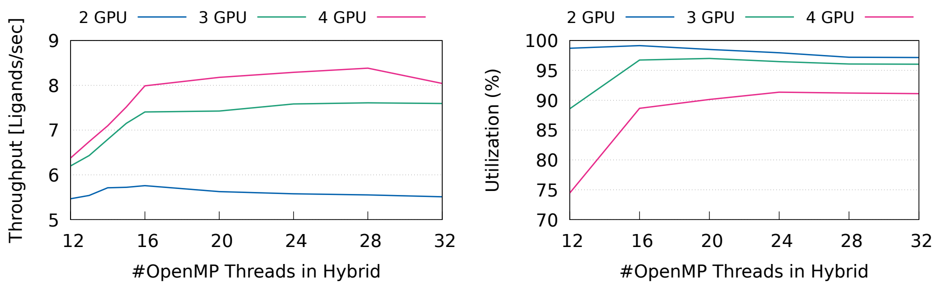
Figure 11.8: *Scaling analysis of the hybrid approach in terms of throughput, by changing the number of OpenMP threads and the number of GPUs.*

***Source**: [207]*

K80 card includes 2 GPUs). This analysis does not aim at showing which is the optimal amount of resource to use in the node to obtain the maximum throughput. Its objective is to show how to organize the resources to maximize the throughput. So it should not be surprising that using all the 4 available GPUs will provide the best overall throughput. The analysis carried on using only 2 or 3 GPUs are interesting since they allows us to validate the proposed methodology.

Figure 11.8 shows the application throughput of the hybrid process while varying the number of OpenMP threads (x-axis) and the number of GPUs. In particular, solid lines represent GeoDock configurations that uses only hybrid processes ($HY_{n,k}^{proc}$, with n={4, ..., 16} and k={2, 3, 4}). While dashed lines represent the full node behaviour, where it uses CPU^{proc} for the unused cores, i.e. the $HY_{n,k}^{proc} + (16 - n) \times CPU^{proc}$ configuration.

If we focus on GeoDock configurations that use only the hybrid approach with a node composed of two GPUs, experimental results show how the application has an almost linear growth up to 8 cores (from 2.5 ligands per second to 4.5 ligands per second). Then, the throughput gain slows down and it is almost negligible if we increase the number of OpenMP threads from 12 to 16 (the throughput ends at 5.8 ligands per second). On the other hand, if we focus on GeoDock configurations that use 3 and 4 GPUs, we have a steady growth in the application throughput over the entire range of OpenMP threads. We might conclude that with the number of



(a) *Throughput* **(b)** *GPU Utilization*

Figure 11.9: *Analysis of the application when using more OpenMP threads than CPU availables.*

Source: *[207]*

OpenMP threads we considered, the GPUs are under-utilized. In particular, the GeoDock configurations that use 3 GPUs are slowing down a bit only in the last part of the plot, while in the GeoDock configurations that use 4 GPUs the throughput grows almost linearly.

The dashed lines represent the usage of CPU^{proc} for the spare cores. The results show a similar trend. The main difference lies in the first part of the plot, where there are few OpenMP threads employed in the hybrid approach. The maximum throughput for the full node is the maximum point of these lines. We can notice from the picture that, according to the number of GPUs, the highest point of the functions is reached by the following configurations: $HY^{proc}_{14,2} + 2 \times CPU^{proc}$, $HY^{proc}_{16,3}$, and $HY^{proc}_{16,4}$.

To improve the GPU utilization when we have access to three and four GPUs, we empirically evaluate the benefits of using more OpenMP threads than available cores. Figure 11.9 depicts the application throughput (Figure 11.9a) and GPU utilization (Figure 11.9b) of the multi-GPU hybrid approach. In this experiment, we evaluated a number of OpenMP threads between 12 (less than the number of cores) and 32 (two times the number of cores). When we have access to two GPUs, the application reaches the peak throughput by using 16 OpenMP threads, which is equal to the number of available cores. This is an expected result since the application completely utilizes the GPUs without overloading the system with OpenMP threads. However, when we have access to more than two GPUs, the application reaches the peak throughput by using a number of OpenMP threads greater than the available cores. This benefit is due to the increased utilization of GPUs. Indeed, the operating system can serve an available worker

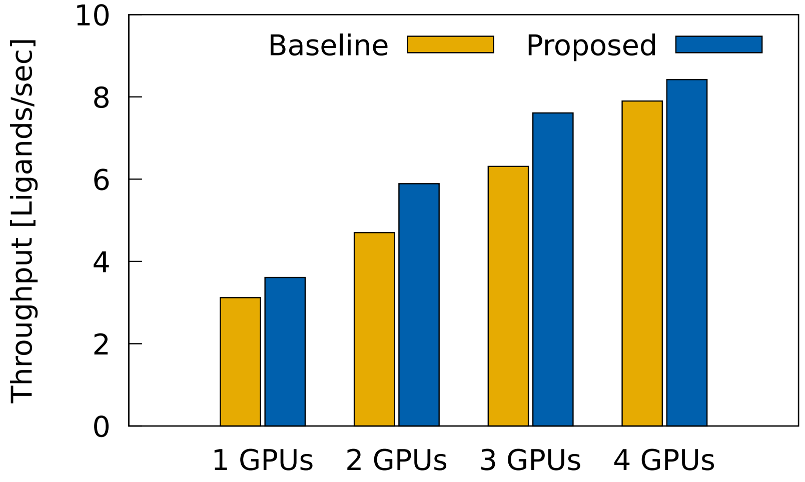


Figure 11.10: *Comparison of the baseline configuration of GeoDock with the proposed hybrid approach, by changing the number of GPUs. The baseline is the configuration with* $k \times GPU^{proc} + (16 - k) \times CPU^{proc}$.

Source: *[207]*

while a second thread waits for the completion of the kernel on the GPU. In particular, with three GPUs the best configuration is $HY_{24,4}^{proc}$, while with four GPUs the best configuration is $HY_{28,4}^{proc}$. If we increase the number of OpenMP threads above such configurations, the context switch overhead hinders the application performance.

To conclude our performance analysis, Figure 11.10 shows the comparison of the best configurations exploiting the full heterogeneous node. As *baseline* we used $k \times GPU^{proc} + (16 - k) \times CPU^{proc}$ (where k is the number of available GPUs) and as *proposed* we select the best configuration obtained with the hybrid approach according to the previous analyses. In all cases, by varying the number of available GPUs, the configurations including the hybrid version have a higher throughput. This is due to the best exploitation of the GPUs only for the kernels where there is a higher speedup. In particular, the performance improvement in the case of 1, 2, 3, and 4 GPUs is respectively 15%, 25%, 20%, and 6%. As mentioned before, in the case of 4 GPUs the performance speedup against the baseline is lower because the 16 cores are not enough to fully exploit all the GPUs.

11.5 Summary

Working with heterogeneous platforms introduces more complexity when writing applications since the programmer needs to consider the creation of kernels for the accelerators and how to distribute the computations that have to be performed across the different hardware. This is even more complex since the different hardware have different characteristics, and the developer needs to consider those characteristics when designing the applications and selecting which kernel will run where.

In this chapter, we have seen how it is possible to further optimize the GeoDock application that we have seen in the previous chapter. In particular, we have seen how, thanks to an improved organization of the computation and on the selection of the more suitable platform for the different kernel, we were able to improve the throughput of GeoDock (up to 25%) without changing the underlying architecture.

The outcome of this work has been published in the Journal of Supercomputing [207].

CHAPTER 12

Improving GeoDock GPU efficiency with Cuda and Dynamic Kernel Tuning

With the rise of the COVID19 pandemic, the GeoDock application (described in Section 8) needed to be further optimized to be usable in the search for a drug to find a therapeutic cure against the virus. In this chapter we will show a porting of GeoDock to a newer supercomputer, with a more powerful GPU component, and we will optimize the application with CUDA in order to obtain the most from the powerful V100 GPGPU. This lead to a drastic improvement in the performances (more than 3x). Moreover we introduce some data-driven autotuning in the application that allow us to avoid some not needed computation, further improving the speedup under certain circumstances.

The work described in this chapter has been carried out as part of the Exscalate4COVID European project, and since GeoDock is a key component in the EXSCALATE pipeline for drug discovery, it was fundamental for the largest virtual screening ever performed (November 2020) [208].

12.1 Introduction

The SARS-COV2 pandemic has created new challenges for the whole world, which had to put a lot of effort into researching a way to contrast the spread of the epidemy. In this context, we insert our effort spent in improving the performance of GeoDock. The application is needed to perform a large virtual screening campaign to find potential candidate drugs that can contrast the virus. However, to manage such a large number of candidates (in the order of billions) in a reasonable time we need to improve the capabilities of the application itself. Moreover, knowing beforehand the target platform capabilities (Marconi100 supercomputer from CINECA and HPC5 from ENI) allows us to make some decisions in the porting of the application, that has been tailored for those machine architectures. In particular, since both of them are heterogeneous machines, with the same GPU accelerator (the NVIDIA V100 GPGPU cards), we decided to rewrite the most compute-intensive kernels from OpenACC to CUDA, to harness the full power of the GPU.

From the autotuning point of view, we inserted in the main algorithm some knowledge that makes it able to select at runtime the number of iteration (and thus, the dimension of the CUDA grids). This mechanism exploits some features of the data (and of the docking algorithm) to optimize, without loss of precision, the execution time. For this reason, we classify it as proactive autotuning.

Thus the contributions of this chapter are:

- An in-depth analysis of the porting from OpenACC to CUDA of the important kernels of GeoDock.

- The introduction of the proactive autotuning in the application.

In the context of the global framework, this chapter provides a working example of a heterogenous autotuning module.

The components of the framework that are involved in this chapter can be seen in Figure 12.1. We can notice the GPU kernel presence since the key point of this work is the automatic tunability of those kernels. We use the data feature to select the kernel grid parameters, thus creating a real dynamic application that will change its configuration according to its data input.

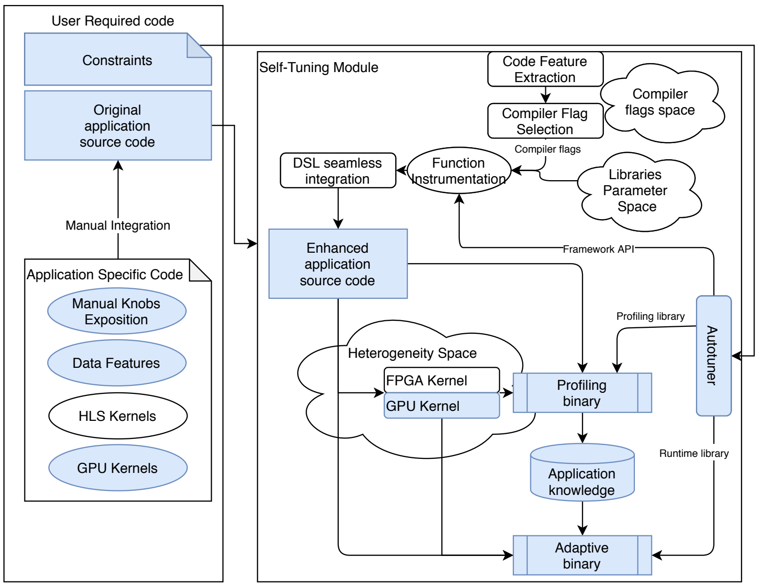


Figure 12.1: *Highlight of thesis approach targeted in this chapter.*

12.2 Background

CUDA (Compute Unified Device Architecture) is a programming model that has been created by NVIDIA to program their GPU architectures. Its language allows more fine-grained control over the GPU than other general approaches like OpenACC or OpenCL. This allows the programmer to obtain the maximum performance from these devices. Indeed, as several works in the state of the art show [209, 210], the CUDA implementation is always able to deliver the best speedup when using NVIDIA devices. This is not surprising at all, since this language is designed to work only with NVIDIA GPGPU.

In particular, this language gives access to the full control of the memory. Indeed, using OpenACC we were not able to fully control memory allocation and utilization, with only the copy pragmas being available. As stated in [209], OpenACC has some directive to work with cached memory. However, the programmer is not able to control them, since everything is managed by the compiler. This is true also for other memory functionalities, that are only used if the compiler decides that it is possible to use them.

On the other hand, with CUDA we can access all the memory functionalities. In particular, we are interested in textures, pinned memories, and shared memory. Another feature that becomes available is the warp level primitives. This set of instructions, that have been made easy to use with the introduction of cooperative groups in CUDA 9 [211], enables the co-operation of different threads in the same warp or sub-warp. In particular, they allow us to use the data resident in registers of different threads in the same warp, thus enabling the exchange of data between the threads without having to move data into memory.

Finally, from a preliminary version, we noticed that the CUDA performances were higher than the OpenACC ones.

12.3 Porting to CUDA

In this section, we will describe with a top-down approach the reorganization of the code done while porting the application to CUDA. In the first moment, we will see some general considerations and ideas that have driven the whole porting. Then we will see in detail the porting of the main kernels, and all the techniques adopted to optimize their execution time. Finally, we will compare this solution with the previous one to show the performance improvement.

12.3.1 General Considerations

Since more levels of parallelism are needed to fully exploit the GPU, to optimize the application we must aim at a SIMT (Single Instruction Multiple Threads) approach, where different threads are executing the same operation on different inputs. This must be done since threads are the inner parallelism level in the CUDA hierarchy. They are organized in blocks (of max 1024 threads), which are themselves organized in grids. A block is mapped on a single streaming multiprocessor (SM), while the grid is distributed across the SM on the GPU. This organization is visible in Figure 12.2

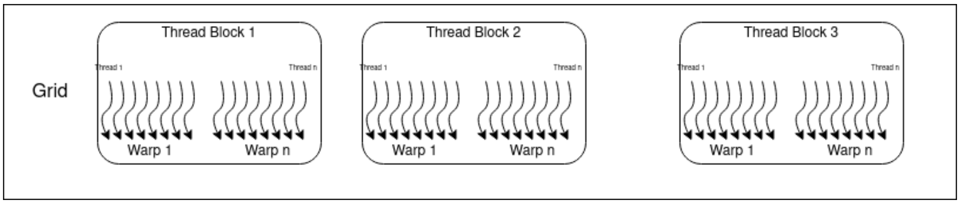


Figure 12.2: *GPU Threads Layout*

An important detail in organizing the code is the attention to warps: a warp is a group of 32 threads, that are executed at the same time, running the same instruction on different data. This is the finest grain available: different threads inside a warp cannot execute different instructions. Whenever that is needed, it creates the so-called "warp-divergence". Warp divergence is a particular condition where some of the instructions executed by the warp are needed by some threads only, while other threads need to do other operations. This situation creates overhead: all the threads must execute all the instructions and discard the unused ones. The most common instructions that create warp divergence are the conditional instructions, such as loops, if, and so on. This condition creates important slowdowns in the execution of the code so must be avoided whenever possible.

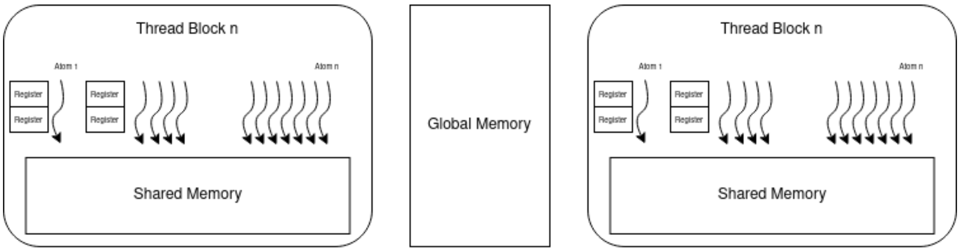


Figure 12.3: *Gpu Memory Layout*

Another important feature of the GPU that must be considered is the memory hierarchy, visible in Figure 12.3. It is really important that all the data are close to where they are needed, however, GPUs have small cache memory and a different organization with respect to the CPU. There are 3 different levels of memory on the board:

- Global Memory: the slowest and biggest memory available on the accelerator. data must be moved here from the host memory.
- Shared Memory: a small scratchpad (max 96KB on the V100) shared among the threads of a block, and accessible by all of them. It can be used to synchronize threads.
- Register set: a private space for each thread where to store the variables. is the fastest (and smallest) memory. There are between 60 and 80 registers per thread available, and it is important to not overload them since register spilling is one of the most frequent slowdown causes. Moreover heavy register pressure can limit the number of threads in a thread block (if not enough registers are available, we need to limit the number of threads in a threadblock to "free" the needed registers).

On the application side, the main challenge is the reorganization of the computation, to maximize the parallelism, while keeping the computation balanced. We have already found three different levels of parallelism in the application, as can be seen in Chapter 10 and Chapter 11. These parallelism levels allow us to target the multinode machine, and to distribute the computation across different nodes with limited impact of communication on the performances. The outer level is the number of candidate ligands to evaluate: since this number is in the order of billions, we can consider this problem embarrassingly parallel. This allows us to distribute the ligands across the different nodes and cores of the supercomputer, with a very small communication overhead (we do not need to synchronize the nodes during the runtime, but only at the beginning and at the end of the computation). The intermediate level is due to the adopted docking algorithm: multiple restarts where every restart has a different starting pose. For this reason, all the poses are independent and can be evaluated in parallel. Indeed, besides the initial situation, even at this level all the data can be considered independent and don't need synchronization before the final score evaluation phase, where the best-scored pose is selected. The inner level of parallelism is given by the atoms: every ligand is composed of several atoms that are moved according to some rotations and translation. This is the SIMT approach that we are searching for since the same instruction (rotate or translate) is applied to different data (the atoms). Moreover, we need to give strong attention to memory organization. We aim at reducing to the minimum the number of expensive operations (such as data transfer, memory allocation, and deallocation). Meanwhile, we need to keep the data as close as possible to the compute units.

To optimize the utilization of the GPU accelerators, they are shared among different threads, asynchronously. Every ligand will be tied to a thread, that is tied to an asynchronous queue and a reserved space in the GPU memory. The reservation of the space at thread level instead of at ligand level allows us to allocate and deallocate that memory only once in the lifetime of the thread. This is a first optimization that allows saving a lot of memory operations, since this memory space is not linked to the docking of a single ligand, but is linked to the lifetime of the application. The drawback of this approach is that we need to allocate the *worst case* space, and this must be known at compile-time. This introduces a limitation on the maximum size of the processed ligands. However, this is not a real issue in the application since it can be changed at compile time. Moreover, some data structures (such as the pocket space) can be shared among all the threads that are using the same GPU: this can be done since they are read-only data structures, not modified in the docking process. The access to the pocket does not follow a coalesced pattern but the access point is given by the x,y,z coordinates of the atom and for this reason, has a random pattern. Random accesses in memory are a costly operation in GPU since they disable the coalesced access mechanism that allows providing data to all the threads in a warp with a single read operation. However, there is a feature in CUDA that allows improving the performance in these situations, which is the texture cache. Texture caches allow organizing data in 2D or 3D spaces and are optimized for semantical data locality. This means that accessing points in the space that are close to the previous ones is usually faster since they should already be cached. We expect that rotations and translation in the 3D space will not place atoms "too far" across the different iterations. For this reason, we use the texture cache to store the protein pocket values.

On the other hand, when multi-dimensional arrays are needed and they have to be accessed from different thread-blocks, it is very important to organize the data in a way that allows the reads to be coalesced. For this reason, we extensively use CUDA pitched arrays in storing temporary values that are needed across kernels. Pitched arrays are an instrument provided by CUDA that inserts automatically padding at the end of every line of multi-dimensional arrays, to optimize memory accesses. In particular, it avoids bank-conflicts and allows coalesced accesses.

Finally, to reduce the data transfer between the GPU and the host all the kernels involved in the docking process were ported to the GPU. From the previous experience with OpenACC, we know that the most expensive operation in CPU is the alignment kernel, while in the GPU it becomes the pose optimization kernel, which occupies from 50 to 90% of the wall time, according to the molecule characteristics (size, number of rotatable bonds,...).

12.3.2 Kernels analysis and optimizations

Initial Poses The first kernel takes as input the original stretched position of the ligand and generates all the restarts. It is a quite small kernel since it has to perform few rotations to generate the different initial poses. For this reason, each atom in every pose has its own thread. The number of restarts is mapped across different thread blocks, as can be seen in Figure 12.4. Memory-wise, we can see from Figure 12.5 that we exploit the shared memory to synchronize the position of the bond identifiers atoms, while all the other atoms are stored in the registers. The only accesses to the global memory are done to read the initial pose at the beginning of the computation and to store the result at the end of the kernel.

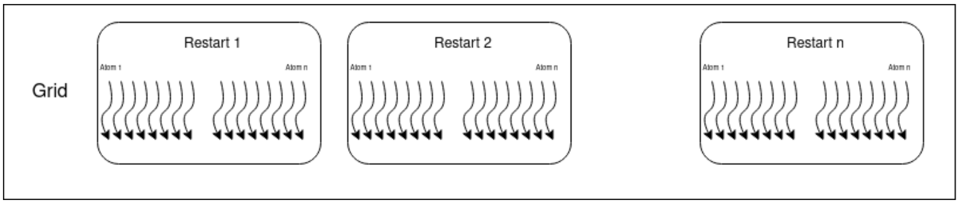


Figure 12.4: *Initial pose, Parallelism Layout*

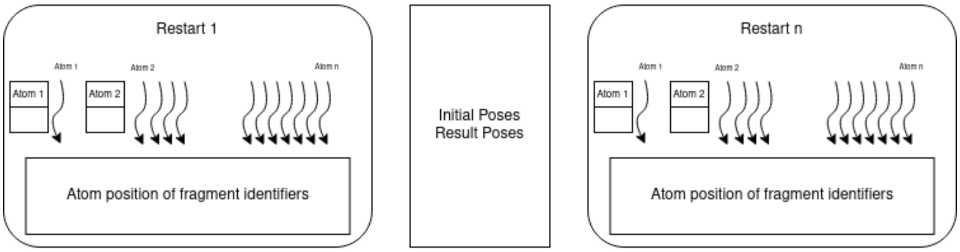


Figure 12.5: *Initial pose, Memory Layout*

Move to Center This kernel is needed to move the center of mass of the ligand towards the center of the space (the coordinates 0,0,0). The computation of this kernel is quite lightweight, however, it has a strong synchronization point. This happens when we need to evaluate the central point of the ligand, which requires evaluating the mean position of the atoms. For this reason we organize the computation in warps (1 warp = 1 restart) as can be seen in Figure 12.6. This allows us to use warp primitives to synchronize, without having to wait for all the threads. Moreover, the warp primitives allow the use of quick reduction operations and the broadcast of the result. Since all of these operations are done on a single warp, there is no stall introduced by this approach. The memory in this kernel is quite straightforward, atom coordinates are read from the global memory, stored in registers, and the output is written back at the end of the kernel. No shared memory is needed, as shown in Figure 12.7.

Alignment This kernel used to be the bottleneck in the CPU version of the application. However, it is a GPU-friendly algorithm, and we already know from the OpenACC porting (Chapter 10) that it is going to be accelerated significantly by using the GPU. This happens because the different rotations can be fully parallelized and there are almost no control flow operations in

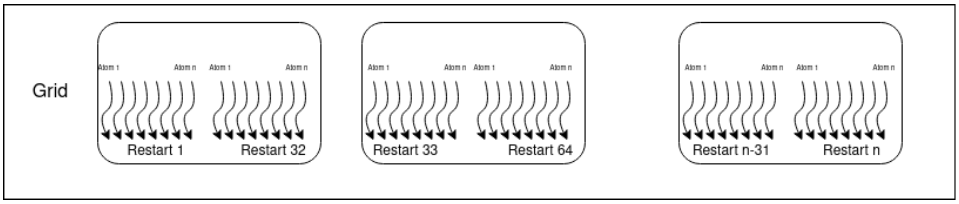


Figure 12.6: *Move to center, Parallelism Layout*

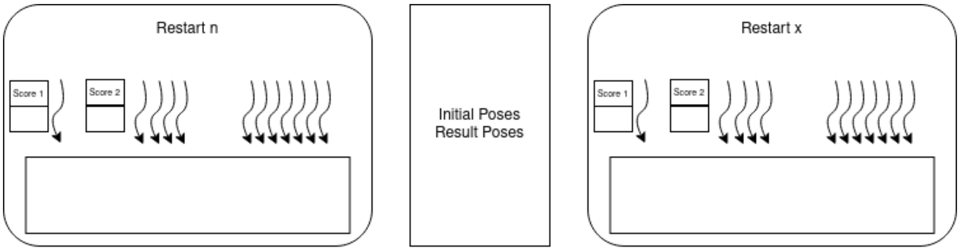


Figure 12.7: *Move to center, Memory Layout*

the algorithm. Moreover, to maximize the distribution of the computations, we divided the alignment phase into 3 different kernels:

1. Score evaluation and reduction to obtain the maximum score. The reduction can manage a maximum of 1024 angles. The limit to the number of angles is due to the fact that every thread evaluates an angle, and the max number of threads per block is 1024.
2. If more than 1024 angles are being tested in a single pose, a reduction kernel is in charge of selecting the best angle.
3. The final kernel is in charge of placing the molecule with the angle that was showing the best score.

Using this organization, we can have different parallelism and a better balancing among the threads.

Evaluate Score This is the main kernel of the alignment phase, where most of the computations are done. Since there are quite a lot of evaluations to be performed, we map a full rotation on a single thread, that evaluates the rotation matrix and applies it to all the atoms of the ligand. With this organization up to 1024 angles can be evaluated in a thread block, as can be seen in Figure 12.8. For this reason, it is possible that more than 1 thread block handles a restart. If this happens, we need kernel number 2 (the reduction kernel), otherwise, that kernel can be skipped.

Figure 12.9 shows the memory organization of the kernel: we use the shared memory to store the starting position of the atoms, in this way the load can be distributed across the different threads, and they are close to the compute units. They will be read-only data for the whole duration of the kernel, so there is no concurrency issue in having them shared across threads that are doing different operations. The rotation matrices are local to each thread, so they are saved in the registers. The global memory is used only at the beginning and at the end of the kernel.

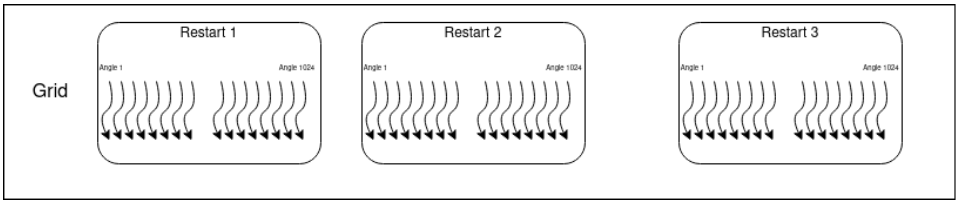


Figure 12.8: *Align Score Parallelism Layout*

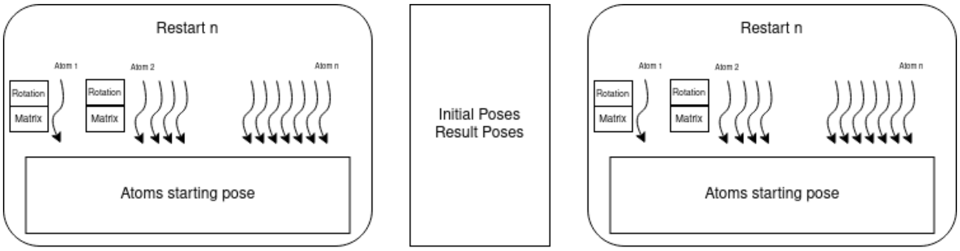


Figure 12.9: *Align Score Memory Layout*

Reduction This kernel performs a quick reduction between the remaining scores: every thread loads a score from the global memory with coalesced access, then performs a warp reduction and store the resulting best score (and the pose that generates it). Every warp manages a restart (as can be seen in Figure 12.10. The shared memory is not used here, since the scores can be directly stored in the registers (as per Figure 12.11).

Emplace This kernel is a very quick one that places the molecule in its best pose. Every thread manages an atom, every thread block is a restart. From the memory point of view, the global memory accesses are coalesced, and everything stays in the registers.

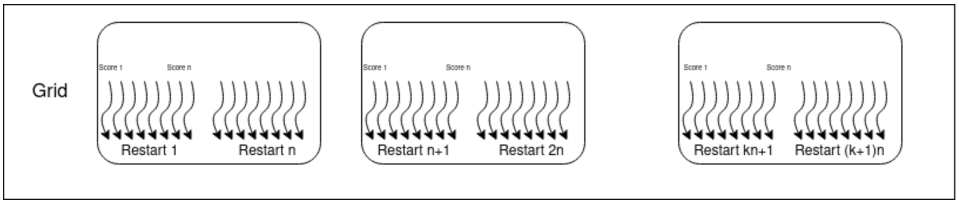


Figure 12.10: *Align Reduction Parallelism Layout*

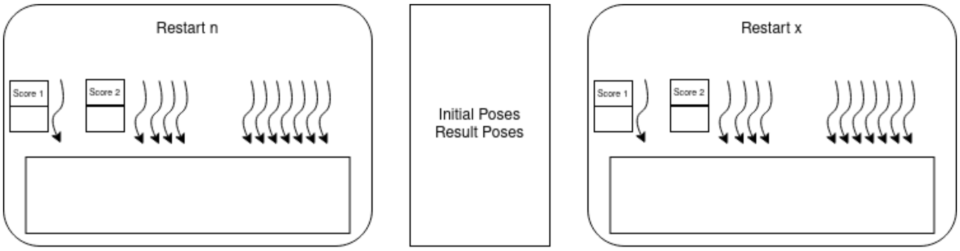


Figure 12.11: *Align Reduction, Memory Layout*

Optimize pose The algorithm that performs this operation is the most complex to parallelize, since it has to be performed sequentially fragment per fragment, and contains some patterns (early termination and triangular matrix) that are not friendly for a GPU environment.

The original code of the algorithm is reported in Listing 12.1. We want to maintain the early termination inside the check_bump while keeping the warps aligned and the data as close as possible to the processing units. Moreover, we want to avoid replicating the same operation across different threads when they are not needed.

The algorithm is reorganized to separate some sections into different kernels. The separation is important because it allows organizing the computation grids in a different way across the functions. The final pseudo code is reported in Listing 12.2.

Listing 12.1: *Pseudo-code of the original function*

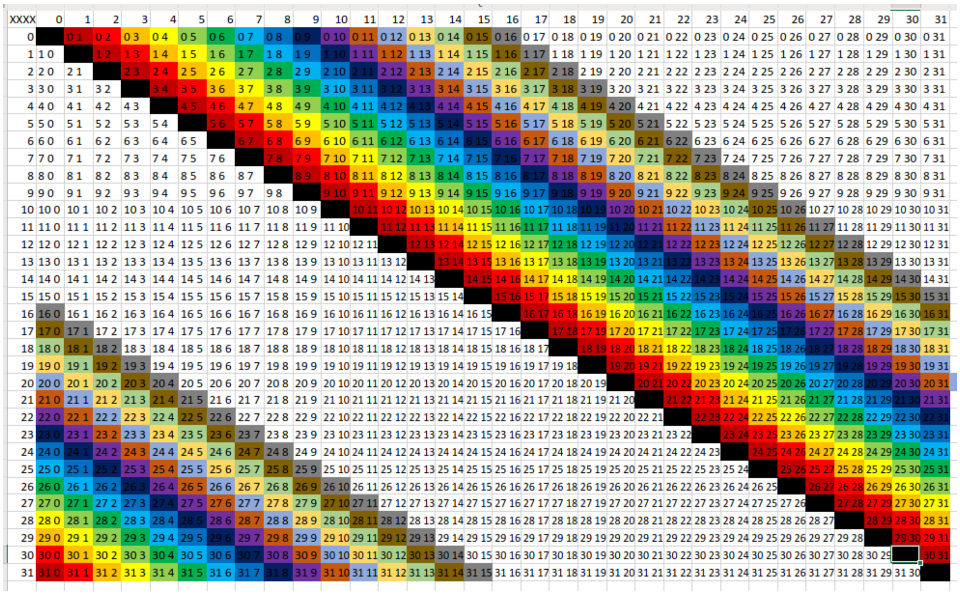
```
for (frag:fragments)
{
  for (angle: angles)
  {
    rotate_pose(ligand, angle, frag);
    check_bump(ligand);
  }
}
vibrate (ligand)
```

Listing 12.2: *Pseudo-code of the final function*

```
for (frag : fragments)
{
  evaluate_rotation_matrices<<<restarts,angles>>>(ligand,frag);
  Rotate_bump_pose <<<restarts, (32, angles)>>>(ligand, matrices,frag
      ,score_arr);
  Final_reduction<<<restarts, scores>>>(scores_arr, best_angles);
  Emplace ligand<<<restarts, atoms>>>(ligand,best_angles);
}
vibrate(ligand);
```

The matrix evaluation is extracted into a separate kernel. This is done because the matrices will be used across different threads, so they will be either re-evaluated a lot of times or they need to be shared using the shared memory. However, as we will see, we already use the shared memory so that is not a viable option. For this reason, we extracted this computation from the original function to not repeat it. The rotate kernel is warp-sized, and this is enforced to maintain the early exit in the check_bump function. Every warp will evaluate if there are internal collisions within the atoms of the ligand, and if they find one, they will mark the position as invalid. Doing this operation warp-sized allows using the voting primitives and enforce that there is no warp divergence when forcing the early termination. The final reduction has the same functionality as the reduction in the alignment phase and is used whenever there are more angles to evaluate than 16. The number is different from the previous reduction function for two main reasons. The first is that, since 32 threads are cooperating in doing the work, only 32 different poses can be evaluated in a thread-block. The second reason is that this kernel has a heavy register pressure, and if we run with 32 poses (1024 threads total) it fails because of the register pressure. For this reason, we limit the "angles" variable to 16. Finally, the emplace kernel puts the ligands in the optimal position.

Rotate_Bump_Pose This function is at the same time the most difficult function to parallelize and the most important kernel to work on, since it becomes the bottleneck of the whole application on the GPU, as seen from Chapter 10. It has been re-written multiple times, each time changing the whole organization of the kernel in the attempt of improving its performance. The final version is organized in warps, and the check_bumps are no more organized in a triangular pattern, but we exploited the fact that $bump(a, b) == bump(b, a)$ to evenly distribute the computation across the threads. Warp organization has a lot of benefits:



XXXX	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	■	0 1	0 2	0 3	0 4	0 5	0 6	0 7	0 8	0 9	0 10	0 11	0 12	0 13	0 14	0 15	0 16	0 17	0 18	0 19	0 20	0 21	0 22	0 23	0 24	0 25	0 26	0 27	0 28	0 29	0 30	0 31
1	1 0	■	1 2	1 3	1 4	1 5	1 6	1 7	1 8	1 9	1 10	1 11	1 12	1 13	1 14	1 15	1 16	1 17	1 18	1 19	1 20	1 21	1 22	1 23	1 24	1 25	1 26	1 27	1 28	1 29	1 30	1 31
2	2 0	2 1	■	2 3	2 4	2 5	2 6	2 7	2 8	2 9	2 10	2 11	2 12	2 13	2 14	2 15	2 16	2 17	2 18	2 19	2 20	2 21	2 22	2 23	2 24	2 25	2 26	2 27	2 28	2 29	2 30	2 31
3	3 0	3 1	3 2	■	3 4	3 5	3 6	3 7	3 8	3 9	3 10	3 11	3 12	3 13	3 14	3 15	3 16	3 17	3 18	3 19	3 20	3 21	3 22	3 23	3 24	3 25	3 26	3 27	3 28	3 29	3 30	3 31
4	4 0	4 1	4 2	4 3	■	4 5	4 6	4 7	4 8	4 9	4 10	4 11	4 12	4 13	4 14	4 15	4 16	4 17	4 18	4 19	4 20	4 21	4 22	4 23	4 24	4 25	4 26	4 27	4 28	4 29	4 30	4 31
5	5 0	5 1	5 2	5 3	5 4	■	5 6	5 7	5 8	5 9	5 10	5 11	5 12	5 13	5 14	5 15	5 16	5 17	5 18	5 19	5 20	5 21	5 22	5 23	5 24	5 25	5 26	5 27	5 28	5 29	5 30	5 31
6	6 0	6 1	6 2	6 3	6 4	6 5	■	6 7	6 8	6 9	6 10	6 11	6 12	6 13	6 14	6 15	6 16	6 17	6 18	6 19	6 20	6 21	6 22	6 23	6 24	6 25	6 26	6 27	6 28	6 29	6 30	6 31
7	7 0	7 1	7 2	7 3	7 4	7 5	7 6	■	7 8	7 9	7 10	7 11	7 12	7 13	7 14	7 15	7 16	7 17	7 18	7 19	7 20	7 21	7 22	7 23	7 24	7 25	7 26	7 27	7 28	7 29	7 30	7 31
8	8 0	8 1	8 2	8 3	8 4	8 5	8 6	8 7	■	8 9	8 10	8 11	8 12	8 13	8 14	8 15	8 16	8 17	8 18	8 19	8 20	8 21	8 22	8 23	8 24	8 25	8 26	8 27	8 28	8 29	8 30	8 31
9	9 0	9 1	9 2	9 3	9 4	9 5	9 6	9 7	9 8	■	9 10	9 11	9 12	9 13	9 14	9 15	9 16	9 17	9 18	9 19	9 20	9 21	9 22	9 23	9 24	9 25	9 26	9 27	9 28	9 29	9 30	9 31
10	10 0	10 1	10 2	10 3	10 4	10 5	10 6	10 7	10 8	10 9	■	10 11	10 12	10 13	10 14	10 15	10 16	10 17	10 18	10 19	10 20	10 21	10 22	10 23	10 24	10 25	10 26	10 27	10 28	10 29	10 30	10 31
11	11 0	11 1	11 2	11 3	11 4	11 5	11 6	11 7	11 8	11 9	11 10	■	11 12	11 13	11 14	11 15	11 16	11 17	11 18	11 19	11 20	11 21	11 22	11 23	11 24	11 25	11 26	11 27	11 28	11 29	11 30	11 31
12	12 0	12 1	12 2	12 3	12 4	12 5	12 6	12 7	12 8	12 9	12 10	12 11	■	12 13	12 14	12 15	12 16	12 17	12 18	12 19	12 20	12 21	12 22	12 23	12 24	12 25	12 26	12 27	12 28	12 29	12 30	12 31
13	13 0	13 1	13 2	13 3	13 4	13 5	13 6	13 7	13 8	13 9	13 10	13 11	13 12	■	13 14	13 15	13 16	13 17	13 18	13 19	13 20	13 21	13 22	13 23	13 24	13 25	13 26	13 27	13 28	13 29	13 30	13 31
14	14 0	14 1	14 2	14 3	14 4	14 5	14 6	14 7	14 8	14 9	14 10	14 11	14 12	14 13	■	14 15	14 16	14 17	14 18	14 19	14 20	14 21	14 22	14 23	14 24	14 25	14 26	14 27	14 28	14 29	14 30	14 31
15	15 0	15 1	15 2	15 3	15 4	15 5	15 6	15 7	15 8	15 9	15 10	15 11	15 12	15 13	15 14	■	15 16	15 17	15 18	15 19	15 20	15 21	15 22	15 23	15 24	15 25	15 26	15 27	15 28	15 29	15 30	15 31
16	16 0	16 1	16 2	16 3	16 4	16 5	16 6	16 7	16 8	16 9	16 10	16 11	16 12	16 13	16 14	16 15	■	16 17	16 18	16 19	16 20	16 21	16 22	16 23	16 24	16 25	16 26	16 27	16 28	16 29	16 30	16 31
17	17 0	17 1	17 2	17 3	17 4	17 5	17 6	17 7	17 8	17 9	17 10	17 11	17 12	17 13	17 14	17 15	17 16	■	17 18	17 19	17 20	17 21	17 22	17 23	17 24	17 25	17 26	17 27	17 28	17 29	17 30	17 31
18	18 0	18 1	18 2	18 3	18 4	18 5	18 6	18 7	18 8	18 9	18 10	18 11	18 12	18 13	18 14	18 15	18 16	18 17	■	18 19	18 20	18 21	18 22	18 23	18 24	18 25	18 26	18 27	18 28	18 29	18 30	18 31
19	19 0	19 1	19 2	19 3	19 4	19 5	19 6	19 7	19 8	19 9	19 10	19 11	19 12	19 13	19 14	19 15	19 16	19 17	19 18	■	19 20	19 21	19 22	19 23	19 24	19 25	19 26	19 27	19 28	19 29	19 30	19 31
20	20 0	20 1	20 2	20 3	20 4	20 5	20 6	20 7	20 8	20 9	20 10	20 11	20 12	20 13	20 14	20 15	20 16	20 17	20 18	20 19	■	20 21	20 22	20 23	20 24	20 25	20 26	20 27	20 28	20 29	20 30	20 31
21	21 0	21 1	21 2	21 3	21 4	21 5	21 6	21 7	21 8	21 9	21 10	21 11	21 12	21 13	21 14	21 15	21 16	21 17	21 18	21 19	21 20	■	21 22	21 23	21 24	21 25	21 26	21 27	21 28	21 29	21 30	21 31
22	22 0	22 1	22 2	22 3	22 4	22 5	22 6	22 7	22 8	22 9	22 10	22 11	22 12	22 13	22 14	22 15	22 16	22 17	22 18	22 19	22 20	22 21	■	22 23	22 24	22 25	22 26	22 27	22 28	22 29	22 30	22 31
23	23 0	23 1	23 2	23 3	23 4	23 5	23 6	23 7	23 8	23 9	23 10	23 11	23 12	23 13	23 14	23 15	23 16	23 17	23 18	23 19	23 20	23 21	23 22	■	23 24	23 25	23 26	23 27	23 28	23 29	23 30	23 31
24	24 0	24 1	24 2	24 3	24 4	24 5	24 6	24 7	24 8	24 9	24 10	24 11	24 12	24 13	24 14	24 15	24 16	24 17	24 18	24 19	24 20	24 21	24 22	24 23	■	24 25	24 26	24 27	24 28	24 29	24 30	24 31
25	25 0	25 1	25 2	25 3	25 4	25 5	25 6	25 7	25 8	25 9	25 10	25 11	25 12	25 13	25 14	25 15	25 16	25 17	25 18	25 19	25 20	25 21	25 22	25 23	25 24	■	25 26	25 27	25 28	25 29	25 30	25 31
26	26 0	26 1	26 2	26 3	26 4	26 5	26 6	26 7	26 8	26 9	26 10	26 11	26 12	26 13	26 14	26 15	26 16	26 17	26 18	26 19	26 20	26 21	26 22	26 23	26 24	26 25	■	26 27	26 28	26 29	26 30	26 31
27	27 0	27 1	27 2	27 3	27 4	27 5	27 6	27 7	27 8	27 9	27 10	27 11	27 12	27 13	27 14	27 15	27 16	27 17	27 18	27 19	27 20	27 21	27 22	27 23	27 24	27 25	27 26	■	27 28	27 29	27 30	27 31
28	28 0	28 1	28 2	28 3	28 4	28 5	28 6	28 7	28 8	28 9	28 10	28 11	28 12	28 13	28 14	28 15	28 16	28 17	28 18	28 19	28 20	28 21	28 22	28 23	28 24	28 25	28 26	28 27	■	28 29	28 30	28 31
29	29 0	29 1	29 2	29 3	29 4	29 5	29 6	29 7	29 8	29 9	29 10	29 11	29 12	29 13	29 14	29 15	29 16	29 17	29 18	29 19	29 20	29 21	29 22	29 23	29 24	29 25	29 26	29 27	29 28	■	29 30	29 31
30	30 0	30 1	30 2	30 3	30 4	30 5	30 6	30 7	30 8	30 9	30 10	30 11	30 12	30 13	30 14	30 15	30 16	30 17	30 18	30 19	30 20	30 21	30 22	30 23	30 24	30 25	30 26	30 27	30 28	30 29	■	30 31
31	31 0	31 1	31 2	31 3	31 4	31 5	31 6	31 7	31 8	31 9	31 10	31 11	31 12	31 13	31 14	31 15	31 16	31 17	31 18	31 19	31 20	31 21	31 22	31 23	31 24	31 25	31 26	31 27	31 28	31 29	31 30	■

Figure 12.12: *Warp reorganization of the triangular matrix. Different colors represent the iterations to evaluate the triangular matrix. The x and y values are the iteration number of the atoms whose bumping is checked.*

- Allows the use of warp voting primitives.
- Allows moving data between threads.
- Allows forcing the SIMD, with no thread misalignment within the warp.
- Early exit frees a full warp and not a thread, which is very important since as we already mentioned warps are the smallest issue unit in CUDA.

The reorganization of the triangular matrix follows the pattern shown in Figure 12.12. Every different color is an iteration of the loop, and we can notice that with 16 iterations we can check all the bumps between 32 atoms.

Moreover, this re-organization has two perks: the first one is that the atom positions can be exchanged between the threads, without the need of accessing the memory. For example, if we consider thread 0, at the first iteration it performs $bump(0, 1)$ and in the second $bump(0, 2)$. The coordinates of atom 2 however are already in the registers, since they were used by the first iteration of thread 1, which was $bump(1, 2)$. So, it is possible to use the shfl primitive to move those values from thread 1 to thread 0, without having to read them from the global memory. The second advantage is given by the balance of the threads: all are doing the same amount of operations. Indeed, after the check is done on the single thread, a warp-voting primitive is called and if there is a bump the whole warp performs the early exit from the function. And since an angle is evaluated by a warp, other useless evaluations are avoided. It is important to notice that the other warps (that are working on different angles, so are independent) are not influenced by this early exit operation.

More details on the kernel organization can be seen in Figure 12.13 and Figure 12.14. The first figure shows the organization of the parallelism: every warp manages a different angle, every thread block works on a different restart. It is possible that more than one thread block works on a single restart if more than 16 angles are required. This limit, as we already mentioned, is due to register pressure.

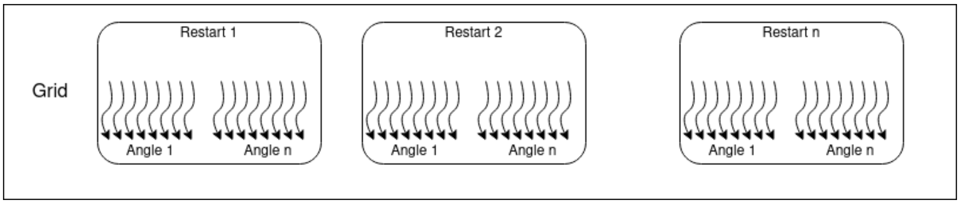


Figure 12.13: *Rotate Bump Kernel, Parallelism Layout*

The second figure shows some detail of the memory organization: the rotation matrices are read from the global memory, while the atoms (that are not read-only in this situation) are stored in the shared memory. Moreover, we can see that some atoms are in the registers: these are representing the phenomenon previously described when the check bump function moves them across the threads without reading the memory.

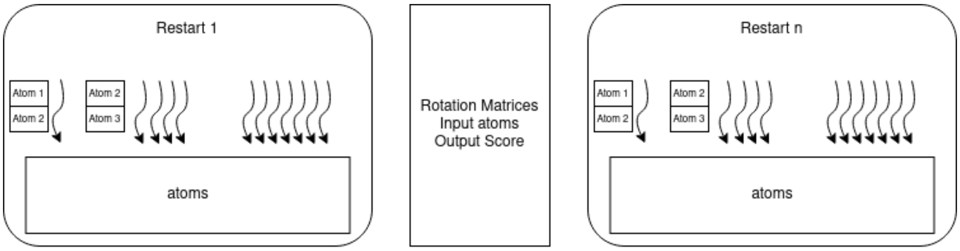


Figure 12.14: *Rotate Bump Kernel, Memory Layout*

Reduction and Emplace The final reduction and the emplace kernels are akin to the one of the alignment (see 12.3.2 and 12.3.2), so they won't be described a second time.

Geometric Scoring Functions To evaluate the generated poses we use three geometric scoring functions, which evaluate how the ligand is positioned in the pocket, if all its atoms are inside and if there is some bump in the structure of the ligand.

Pacman Score This function interacts with the pocket and evaluates both the pacman and the is_in_pocket values. To evaluate the Pacman Score, we need to count all the pocket spaces that are neighbors of the atoms. However, spaces that are neighboring more than one atom need to be counted only once. For this reason, we need to implement a sorting algorithm: we use a bitonic sort, which is a GPU-oriented algorithm that works well with small datasets. This is our case since the whole data can be stored in the shared memory. After the sorting algorithm, a function counts the number of different occupied spaces and obtains the Pacman score.

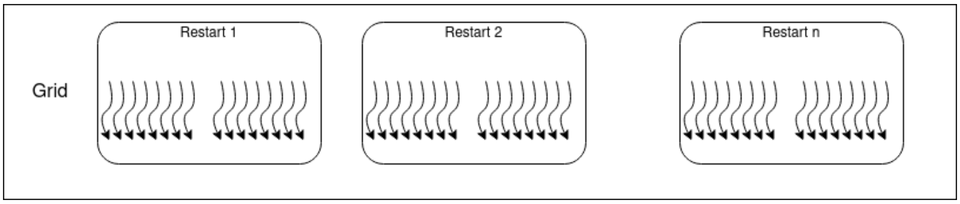


Figure 12.15: *Pacman Kernel, Parallelism Layout*

Figure 12.15 shows the parallelism organization. In this function, the number of active threads per restart is greater than the number of atoms, since this improves the sort, making it more distributed. Indeed, this function could use the whole (1024) thread space, however, it has been capped at 256 threads per thread-block because of register pressure. Every restart is mapped on a different thread block.

From the memory perspective, as shown in Figure 12.16, we use the shared memory to store the visited space in the pocket before the sorting algorithm. This space is then sorted and counted, so the function works only with the shared memory. Once again, we managed to limit the accesses to global memory only to get the initial data and to store the final results.

Ligand is bumping This function checks if the outcome of the docking process has some internal bumps. The kernel of the function works as the

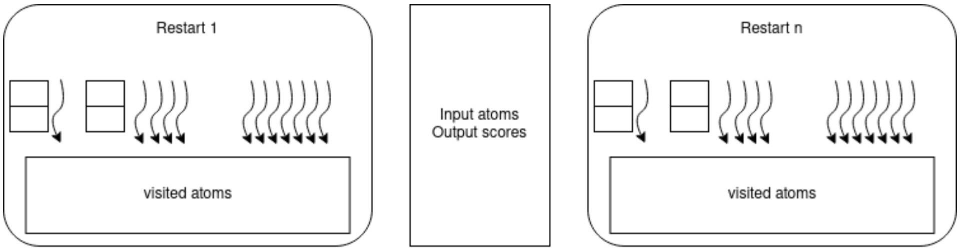


Figure 12.16: *Pacman Kernel, Memory Layout*

rotate_bump function, however, in this kernel, we need to check all the fragments. Nonetheless, it is a simplified version of the check_bump function since we don't have to rotate. We still want to exploit the early exit, so we organize the kernel in warps, as can be seen in Figure 12.17.

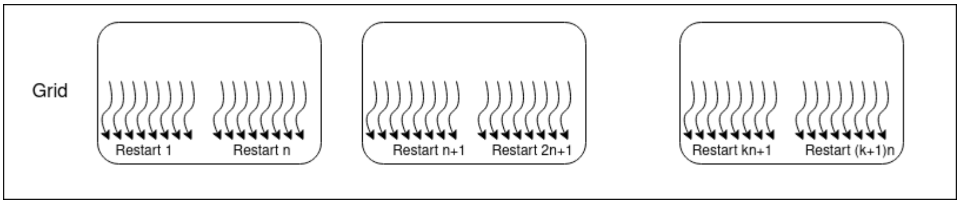


Figure 12.17: *Pacman Kernel, Parallelism Layout*

From the memory perspective, shown in Figure 12.18, we adopted the mechanism of passing the atom position between threads with shfl operations, so we don't need any shared memory, and global memory is read only at the beginning of the kernel.

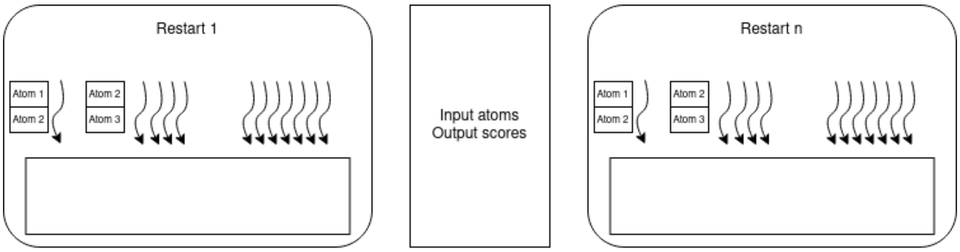
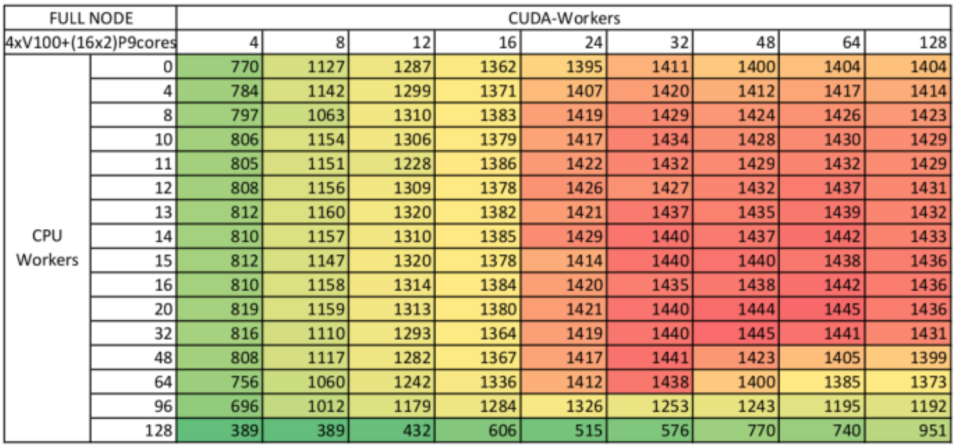


Figure 12.18: *Pacman Kernel, Memory Layout*

12.4 Experimental Results

To compare the final CUDA implementation with the previous OpenACC one, we have run several experiments and measurements on a single node of the Marconi100 Supercomputer in CINECA. The node is equipped with 2 IBM Power9 CPUs and 4 NVIDIA v100 GPU connected with NVLink. Thanks to this extensive rewriting of the kernels, we have obtained a strong speedup compared to the previous OpenACC version: the CUDA application can process more than 3 times the amount of ligands processed by the previous application. Indeed, using the full node the OpenACC version has a throughput of 461 Ligands per second. 336 Ligands per second is the throughput of the 4 V100 GPU, while the other 130 ligands are processed by the power9 CPUs. The OpenACC has been configured in a similar way to the CUDA application, with all the kernels processed on the GPU. This version is more similar to the version presented in Chapter 10 than to the one of Chapter 11 because these GPGPUs are way more powerful than the K40 used in the previous experiments, and performing only the alignment kernel on GPU would have hurt performances. We used the asynchronous threads mechanism to have more threads insisting on a GPU, oversubscribing the device, also in the OpenACC version of the application, to have a fair comparison.

Tuning the CUDA application To optimize the utilization of the Marconi100 node, we need to tune the available parameters. In particular, we are interested in knowing what is the optimal number of GPU threads and CPU threads. We define as GPU thread a thread that processes a ligand using GPU kernels (and the CPU is used only to move data and enqueue the kernels), while a CPU thread works only on the CPU.



FULL NODE		CUDA-Workers								
4xV100+(16x2)P9cores		4	8	12	16	24	32	48	64	128
CPU Workers	0	770	1127	1287	1362	1395	1411	1400	1404	1404
	4	784	1142	1299	1371	1407	1420	1412	1417	1414
	8	797	1063	1310	1383	1419	1429	1424	1426	1423
	10	806	1154	1306	1379	1417	1434	1428	1430	1429
	11	805	1151	1228	1386	1422	1432	1429	1432	1429
	12	808	1156	1309	1378	1426	1427	1432	1437	1431
	13	812	1160	1320	1382	1421	1437	1435	1439	1432
	14	810	1157	1310	1385	1429	1440	1437	1442	1433
	15	812	1147	1320	1378	1414	1440	1440	1438	1436
	16	810	1158	1314	1384	1420	1435	1438	1442	1436
	20	819	1159	1313	1380	1421	1440	1444	1445	1436
	32	816	1110	1293	1364	1419	1440	1445	1441	1431
	48	808	1117	1282	1367	1417	1441	1423	1405	1399
	64	756	1060	1242	1336	1412	1438	1400	1385	1373
	96	696	1012	1179	1284	1326	1253	1243	1195	1192
	128	389	389	432	606	515	576	770	740	951

Figure 12.19: *Performance Heatmap of GPU and CPU threads.*

To perform this experiment we use a dataset that is representative of all the possible ligands' size and fragment numbers, since we need to statically setup the amount of workers before running the experiments. In this way, we can avoid to repeat this effort when we change the dataset. We will see in the following section an orthogonal technique to optimize the application considering input characteristics. Figure 12.19 shows the outcome of the design space exploration done. On the columns, we report the number of GPU threads allocated for the job, while the rows have the CPU threads. We can notice that the total amount of threads can be more than the available threads on the Power9 processors since we are open to the possibility that oversubscription and hyperthreading could bring some benefit. However, from the results of the experiment, we can notice that this hypothesis is not true, since the optimal points on the heatmap are around 64GPU and 20CPU threads, which are less than the number of logical cores available on the node. Nonetheless, the hyperthreading approach is correct, since the optimal point has more thread than the number of physical cores of the node.

Finally, thanks to this CUDA porting, we were able to increase the throughput by 1000 ligands per second, on the same dataset. Indeed, the total throughput of the node was 1445 ligands per second: the porting provided a speedup of more than 3x compared to the OpenACC version of Chapter 10.

12.5 Autotuning

Even if autotuning is not the focal point of this chapter, we believe that is possible to further speed up the application by applying the techniques developed in the previous chapters. In particular, we analyzed the application and noticed that there is an important data dependency that, if addressed correctly, can allow the application to save some operations. In particular, it is a repetition of the evaluation of some poses that happens whenever there are not enough fragments in the ligand to generate the initial poses. These repetitions are not improving the accuracy of the result but they are only performing the same pose evaluation more times, so it is desirable to avoid them. We applied the technique developed in Chapter 6, to proactively select the number of poses to test. In particular, we used the number of fragments as data-feature and the number of poses as software knob. It is important to notice that this software knob strongly influences the application since for several kernels it is involved in the grid organization. Indeed, by changing this parameter the dimension of the CUDA grids are effectively modified, and the amount of calculation required to obtain the

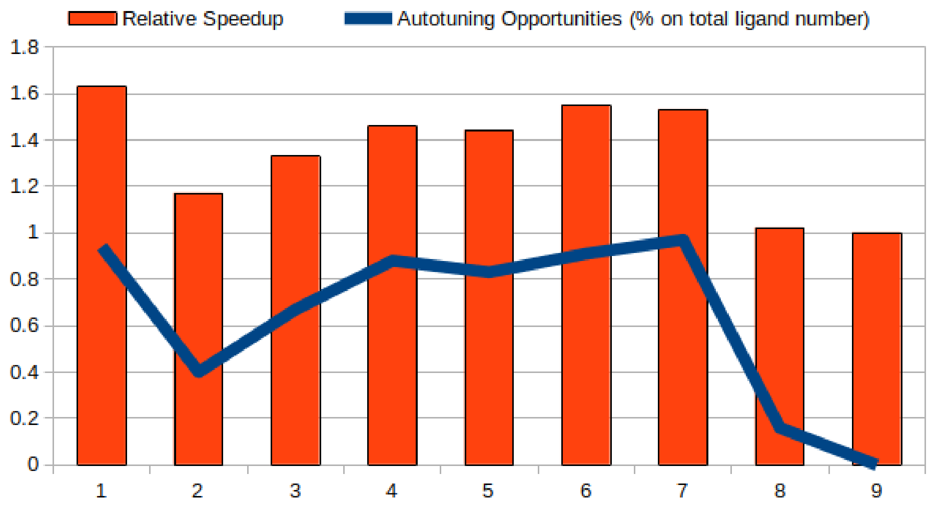


Figure 12.20: *Speedup compared to the CUDA version without autotuning and percentage of autotuning opportunity in some different datasets.*

solution changes. This has a heavy influence on the application whenever this proactive autotuning is triggered.

However, it rarely happens with the dataset that we used previously. The reason is that the dataset that we used to measure the performance is a dataset of large ligands, and this phenomenon is triggered only by small ligands. In particular, the dataset we used has an average of 20 fragments per ligand, and less than 10% of the ligands have less than 10 fragments. The autotuning opportunity is only available for ligands with less than 4 fragments. However, when we were preparing for a larger molecular docking experiment, we were given a more representative set of datasets. Among those datasets, we noticed that there were some of them where most ligands had a small number of fragments.

Figure 12.20 shows the result of this autotuning approach on several datasets, with different autotuning opportunities. We define as "autotuning opportunity" the percentage of ligands with less than 4 fragments with respect to the total amount of ligands in a dataset. We define as "relative speedup" the ratio of the throughput of the autotuned version of the application compared to the baseline one. As we can see, whenever the autotuning opportunity is over 90%, the autotunable version of the application has a strong speedup, of at least 1.4x, which comes completely for free. Indeed, this speedup does not introduce any degradation in the application output accuracy. Moreover, looking at Table 12.1, we can notice that there is no

Dataset ID	1	2	3	4	5	6	7	8	9
Atoms Number	36.34	40.87	52.28	66.17	75.82	91.22	95.68	69.67	73.53

Table 12.1: *Average size of the ligand of the datasets used to explore the autotuning approach introduced.*

correlation between the speedup obtained with autotuning and the dimension of the ligands. It is also interesting to analyze the last two datasets (8 and 9). The speedup obtained by the first dataset is very little (1.02x) while the second dataset reports no variation in execution time from the original application without autotuning. Dataset 8 is a collection of ligands with a large number of fragments (average of 18 and 12 of standard deviation), which gives a very limited opportunity for the autotuning approach to show its benefits. Indeed, the autotuning approach is strongly dependent on the input data. The last dataset shows a very important result: the autotuning inserted in the application does not generate any performance loss whenever the dataset is completely not suitable for the autotuning (0%). This result justifies the entire approach, since the insertion of autotuning, in the worst case, does not generate any performance loss. On the other hand, in the best case, it provides a speedup of 1.6x without any accuracy loss.

12.6 Summary

In this chapter, we have seen the optimization and re-organization of the code of GeoDock. In particular, this work has been fundamental in enabling the trillion docking experiment performed in the context of the EXSCALATE4COV project. In that experiment, which is the largest molecular docking experiment ever attempted up to date, a trillion of molecules have been docked in 60 hours on two supercomputers, Marconi100 and HPC5[1]. This experiment has been done to search for a ligand that could be a candidate drug to cure the infection of the COVID-19 virus. As we have seen, thanks to the porting to CUDA the speed of GeoDock has dramatically increased, and since this is the bottleneck of the whole application, this allowed a significant increment in the throughput. Indeed, without this porting the trillion docking experiment would not have been possible. Moreover, we were able to apply previously discovered autotuning techniques to this new context, thus validating them in a different context from their original one. This also proofs that those techniques not only valid in their original context. However, the introduced autotuning is at the moment very limited and we are currently trying to discover other opportunities.

[1]https://www.hpcwire.com/2020/11/25/exscalate4cov-runs-70-billion-molecule-coronavirus-simulation/

The outcome of this work has been published in an abstract at the EuroHPC 2021 summit and a paper is currently in preparation. This work has also won the HIPEAC Tech Transfer award in October 2020[2]

[2]https://www.hipeac.net/news/6940/winners-of-the-hipeac-tech-transfer-awards-2020/

CHAPTER 13

Conclusions

In this thesis, we addressed the problem of application autotuning, and how we can give the application an adaptation layer even in heterogeneous contexts. The main outcome is a meta-framework composed of a collection of techniques that allows the user to enhance the application with a proactive or adaptive behavior. Those techniques require the user to interact with the application since in most cases some knowledge is not available without human interaction. However, after the initial effort, the application shows effective benefits during its lifetime.

We have experimentally evaluated the proposed approach and its exploitation in different scenarios, in particular seeing how those techniques can benefit also real HPC applications. The remainder of this chapter summarises the finding and limitations of the proposed approach and provides recommendations for future works.

13.1 Main contributions

The main results of the work carried out in this thesis might be summarised as follows:

1. A methodology has been proposed to review applications in order to

reformulate them as a sequence of self-tuning modules. A self-tuning module is in this context a kernel that performs one of the functionalities of the original application that has been enhanced with an adaptivity layer. Thus it is able to adapt, in a reactive or a proactive way, to changes in the execution context of the application due to external factors (such as the condition of the machine) or input data.

2. An approach has been proposed to provide to application developers a way to automatically and seamlessly find the best possible configuration of an application according to the underlying architecture for different hotspots kernels. This approach provides self-optimization capabilities to the target application and targets compiler options and OpenMP parameters.

3. A Probabilistic Time Depending Routing application has been studied and a methodology to proactively reduce the computation load required with limited accuracy loss has been proposed. This methodology allows increasing the computation efficiency, thus leading to reduced energy and time consumption.

4. In the context of Deep neural networks, we have benchmarked several object detection networks to build a knowledge base. We then supposed a scenario in the automotive context where none of the benchmarked networks was able to respect all the constraints, however, if we use more than one network it becomes possible to satisfy the requirements. This is possible with the introduction of application autotuning, which reacts to the changing environment and select the most suitable network at runtime. Moreover, we believe that it could be possible to create a predictor that is able to select the most suitable network according to the characteristics of the image. However, in this case, we were only able to prove that this could be beneficial but we were unable to build the actual predictor.

5. A molecular docking application designed for High Performance Computing platforms has been studied several times, in a first moment to introduce a mechanism to enforce a time to solution on a homogeneous node, then its hotspot kernels have been ported to GPGPU to increase its performances. This work has been done using the OpenACC language, and the resulting application has been optimized to exploit in the best way the available resources of the node. Finally, the search for the maximum throughput forced us to re-organize the application one more time and rewrite its kernels using CUDA. On this

final version of the molecular docking application, we also applied autotuning techniques to avoid some useless computation whenever certain conditions on the input were met. This application has been used in the world largest molecular docking experiment

6. We created an open-source library that can be used in High Level Synthesis to create hardware accelerators for the multiplication of Large Unsigned Integers. The focus of this library is its flexibility, which allows creating large and high throughput multiplier and small and slow ones. This design choice is done to follow the paradigm of autotuning libraries that tailor the functionality to the underlying architectures (in this case, we tailor the architecture to the user's needs).

13.2 Recommendation for future works

Experimental evaluations of the proposed techniques have promising results; however, there are still possible improvements that can be investigated. In our opinion, the most challenging open questions are the following:

1. The search for knobs and metrics must be provided by end-users and application developers since they are application-specific. However, there are works [6] in literature where generic error metrics are found and applied to different applications. It could be interesting to investigate their effectiveness in order to create a methodology to automatically insert adaptivity in applications.

2. The extraction of the data features that enables the proactive autotuning is always delegated to the human programmer. Indeed, the programmer must have the intuition that some characteristics of the input can be used to drive the selection of the software knobs. It could be very interesting to study some methodologies to automatically find some input features and their impact on general applications.

3. Some of the proposed techniques support automatic integration. An interesting work would be to enlarge this support. This could become even more important if the previous point (a methodology for automatic search of input feature) becomes real.

We hope that the work discussed in this thesis will motivate researchers to further investigate the application autotuning topic, even in heterogeneous contexts. We believe that our view on future applications, seen as a sequence of self-tuning modules, is correct and we hope that the proposed meta-framework could help future programmers in organizing their applications.

Publications

Articles published or under review in international journals

1. Kim Grüttner, Ralph Görgen, Sören Schreiner, Fernando Herrera, Pablo Peñil, Julio Medina, Eugenio Villar, Gianluca Palermo, William Fornaciari, Carlo Brandolese, Davide Gadioli, Emanuele Vitali, Davide Zoni, Sara Bocchio, Luca Ceva, Paolo Azzoni, Massimo Poncino, Sara Vinco, Enrico Macii, Salvatore Cusenza, John Favaro, Raùl Valencia, Ingo Sander, Kathrin Rosvall, Nima Khalilzad, Davide Quaglia "CONTREX: Design of embedded mixed-criticality CONTRol systems under consideration of EXtra-functional properties." Microprocessors and Microsystems 51 (2017): 39-55.

2. Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, João M.P. Cardoso, Carlo Cavazzoni, Daniele Cesarini, Stefano Cherubin, Federico Ficarelli, Davide Gadioli, Martin Golasowski, Antonio Libri, Jan Martinovič, Gianluca Palermo, Pedro Pinto, Erven Rohou, Kateřina Slaninová, Emanuele Vitali "The ANTAREX domain specific language for high performance computing" Microprocessors and Microsystems 68 (2019): 58-73

3. Davide Gadioli, Emanuele Vitali, Gianluca Palermo, Cristina Silvano "mARGOt: a Dynamic Autotuning Framework for Self-aware Approximate Computing" IEEE Transactions on Computers, 68 (2019), 713-728

4. Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Martin Golasowski, João Bispo, Pedro Pinto, Jan Martinovič, Kateřina Slaninová, João M. P. Cardoso, Cristina Silvano "An Efficient Monte Carlo-based Probabilistic Time-Dependent Routing Calculation Targeting a Server-Side Car Navigation System" IEEE Transactions on Emerging Topics in Computing

5. Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Andrea Beccari, Carlo Cavazzoni, Cristina Silvano " Exploiting OpenMP and OpenACC to accelerate a geometric approach to molecular docking in heterogeneous HPC nodes". Journal of Supercomputing 75 (2019): 3374-3396

6. Davide Gadioli, Gianluca Palermo, Stefano Cherubin, Emanuele Vitali, Giovanni Agosta, Candida Manelfi, Andrea R. Beccari, Carlo Cavazzoni, Nico Sanna, Cristina Silvano "Tunable Approximations for Controlling Time-to-Solution in an HPC Molecular Docking Mini-App", The Journal of Supercomputing, 77: 841-869

Articles published in proceedings of international conferences

1. Davide Gadioli, Ricardo Nobre, Pedro Pinto, Emanuele Vitali, Amir H Ashouri, Gianluca Palermo, João M. P. Cardoso, Cristina Silvano "SOCRATES-A seamless online compiler and system runtime autotuning framework for energy-aware applications" Design, Automation and Test in Europe Conference and Exhibition (DATE), 2018

2. Cristina Silvano, Gianluca Palermo, Giovanni Agosta, Amir H Ashouri, Davide Gadioli, Stefano Cherubin, Emanuele Vitali, Luca Benini, Andrea Bartolini, Daniele Cesarini, João M. P. Cardoso, João Bispo, Pedro Pinto, Riccardo Nobre, Erven Rohou, Loïc Besnard, Imane Lasri, Nico Sanna, Carlo Cavazzoni, Radim Cmar, Jan Martinovič, Kateřina Slaninová, Martin Golasowski, Andrea R Beccari, Candida Manelfi "Autotuning and adaptivity in energy efficient HPC systems: the ANTAREX toolbox" Proceedings of the ACM International Conference on Computing Frontiers, 2018

3. Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, João M. P. Cardoso, Carlo Cavazzoni, Stefano Cherubin, Davide Gadioli, Martin Golasowski, Imane Lasri, Jan Martinovič, Gianluca Palermo, Pedro Pinto, Erven Rohou, Nico Sanna, Kateřina Slaninová, Emanuele

Vitali “ANTAREX: A DSL-based Approach to Adaptively Optimizing and Enforcing Extra-Functional Properties in High Performance Computing” Euromicro Conference on Digital System Design, 2018

4. Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, João M. P. Cardoso, Carlo Cavazzoni, , Daniele Cesarini, Stefano Cherubin, Federico Ficarelli, Davide Gadioli, Martin Golasowski, Imane Lasri, Antonio Libri, Candida Manelfi, Jan Martinovič, Gianluca Palermo, Pedro Pinto, Erven Rohou, Nico Sanna, Kateřina Slaninová, Emanuele Vitali “Supporting the Scale-Up of High Performance Application to Pre-Exascale Systems: The ANTAREX Approach,” 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)

5. Emanuele Vitali, Davide Gadioli, Fabrizio Ferrandi, Gianluca Palermo “Parametric Throughput Oriented Large Integer Multipliers for High Level Synthesis” Design, Automation and Test in Europe Conference and Exhibition (DATE), 2021

6. Emanuele Vitali, Anton Lokhmotov, Gianluca Palermo “Dynamic Network selection for the Object Detection task: why it matters and what we (didn’t) achieve” to appear in SAMOS, 2021

Articles published in proceedings of international workshops

1. Emanuele Vitali, Gianluca Palermo “ Early stage interference checking for automatic design space exploration of mixed critical systems” Proceedings of the 9th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, 2017

2. Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Andrea Beccari, Cristina Silvano “Accelerating a Geometric Approach to Molecular Docking with OpenACC” Proceedings of the 5th International Workshop on Parallelism in Bioinformatics, 2018

Posters published/presented in poster sessions co-located with international conference

1. Emanuele Vitali, Davide Gadioli, Andrea Beccari, Carlo Cavazzoni, Cristina Silvano, Gianluca Palermo “An hybrid approach to accelerate a molecular docking application for virtual screening in heteroge-

neous nodes" Proceedings of the 16th ACM International Conference on Computing Frontiers, 2019

2. Gianluca Palermo, Davide Gadioli, Emanuele Vitali, Cristina Silvano, Federico Ficarelli, Chiara Latini, Andrea Beccari, Candida Manelfi " EXSCALATE4COV: Towards an Exascale-Ready Docking Platform Targeting Urgent Computing" EuroHPC Summit Week 2021.

Bibliography

[1] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[2] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, December 2007.

[3] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan. Approximate computing and the quest for computing efficiency. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[4] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334. ACM, 2006.

[5] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. 2009.

[6] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 25–34. ACM, 2010.

[7] Amir Yazdanbakhsh, Divya Mahajan, Bradley Thwaites, Jongse Park, Anandhavel Nagendrakumar, Sindhuja Sethuraman, Kartik Ramkrishnan, Nishanthi Ravindran, Rudra Jariwala, Abbas Rahimi, et al. Axilog: Language support for approximate hardware design. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 812–817. IEEE, 2015.

[8] Sana Mazahir, Osman Hasan, Rehan Hafiz, Muhammad Shafique, and Jörg Henkel. An area-efficient consolidated configurable error correction for approximate hardware accelerators. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[9] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[10] Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Cristina Silvano. Margot: a dynamic autotuning framework for self-aware approximate computing. *IEEE transactions on computers*, 68(5):713–728, 2018.

[11] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim.*, 13(2):21:1–21:25, 2016.

[12] João M. P. Cardoso, José G. F. Coutinho, Tiago Carvalho, Pedro C. Diniz, Zlatko Petrov, Wayne Luk, and Fernando Gonçalves. Performance-driven Instrumentation and Mapping Strategies Using the LARA Aspect-oriented Programming Approach. *Softw. Pract. Exper.*, 2016.

[13] Grigori Fursin et al. Milepost-gcc: Machine learning enabled self-tuning compiler. *Intern, Journal of Parallel Programming*, 2011.

[14] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[15] L. Dailey Paulson. Computer system, heal thyself. *Computer*, 35(8):20–22, 2002.

[16] Rema Ananthanarayanan, Mukesh Mohania, and Ajay Gupta. Management of conflicting obligations in self-protecting policy-based systems. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 274–285. IEEE, 2005.

[17] Amina Khalid, Mouna Abdul Haye, Malik Jahan Khan, and Shafay Shamail. Survey of frameworks, architectures and techniques in autonomic computing. In *2009 Fifth International Conference on Autonomic and Autonomous Systems*, pages 220–225. IEEE, 2009.

[18] Markus C Huebscher and Julie A McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7, 2008.

[19] Sara Mahdavi-Hezavehi, Vinicius HS Durelli, Danny Weyns, and Paris Avgeriou. A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. *Information and Software Technology*, 90:1–26, 2017.

[20] William E Walsh, Gerald Tesauro, Jeffrey O Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 70–77. IEEE, 2004.

[21] Gerald Tesauro, Nicholas K Jong, Rajarshi Das, and Mohamed N Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*, pages 65–73. IEEE, 2006.

[22] David Abramson, Rajkumar Buyya, and Jonathan Giddy. A computational economy for grid computing and its implementation in the nimrod-g resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.

[23] Henry Hoffmann, Martina Maggio, Marco D Santambrogio, Alberto Leva, and Anant Agarwal. Seec: a general and extensible framework for self-aware computing. 2011.

[24] Juan A Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B Bartolini, Nitesh Mor, et al. Tessellation: refactoring the os around explicit resource containers with continuous adaptation. In *Proceedings of the 50th Annual Design Automation Conference*, page 76. ACM, 2013.

[25] Frank Hannig, Sascha Roloff, Gregor Snelting, Jürgen Teich, and Andreas Zwinkau. Resource-aware programming and simulation of mpsoc architectures through extension of x10. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, pages 48–55. ACM, 2011.

[26] Luigi Palopoli, Tommaso Cucinotta, Luca Marzario, and Giuseppe Lipari. Aquosa-adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31, 2009.

[27] Shuangde Fang, Zidong Du, Yuntan Fang, Yuanjie Huang, Yang Chen, Lieven Eeckhout, Olivier Temam, Huawei Li, Yunji Chen, and Chengyong Wu. Performance portability across heterogeneous socs using a generalized library-based approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(2):21, 2014.

[28] P. Bellasi, G. Massari, and W. Fornaciari. A rtrm proposal for multi/many-core platforms and reconfigurable applications. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8, 2012.

[29] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[30] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[31] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, and Manuela Veloso. SPIRAL: Automatic implementation of signal processing algorithms. In *High Performance Extreme Computing (HPEC)*, 2000.

[32] Michael J Voss and Rudolf Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, pages 163–170. IEEE, 2000.

[33] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *Workshop on Languages and Compilers for Parallel Computing*, 2010.

[34] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011.

[35] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.

[36] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 85–96. IEEE Computer Society, 2011.

[37] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May OŔeilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 303–315. IEEE, 2014.

[38] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[39] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input responsiveness: using canary inputs to dynamically steer approximation. *ACM SIGPLAN Notices*, 51(6):161–176, 2016.

[40] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[41] Matteo Frigo and Steven G. Johnson. Fftw: an adaptive software architecture for the fft. In *ICASSP*, 1998.

[42] Yongpeng Zhang and Frank Mueller. Autogeneration and autotuning of 3d stencil codes on homogeneous and heterogeneous gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 24(3):417–427, 2012.

[43] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[44] Philipp Gschwandtner, Herbert Jordan, Peter Thoman, and Thomas Fahringer. Allscale API. *Comput. Informatics*, 39(4):808–837, 2020.

[45] Philip Pfaffe, Tobias Grosser, and Martin Tillmann. Efficient hierarchical online-autotuning: a case study on polyhedral accelerator mapping. In *Proceedings of the ACM International Conference on Supercomputing*, pages 354–366, 2019.

[46] Jonathan Dorn, Jeremy Lacomis, Westley Weimer, and Stephanie Forrest. Automatically exploring tradeoffs between software output fidelity and energy costs. *IEEE Transactions on Software Engineering*, 45(3):219–236, 2017.

[47] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. A multi-objective auto-tuning framework for parallel codes. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2012.

[48] Juan José Durillo, Philipp Gschwandtner, Klaus Kofler, and Thomas Fahringer. Multi-objective region-aware optimization of parallel programs. *Parallel Comput.*, 83:3–21, 2019.

[49] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017.

[50] Sotirios Xydis, Eleftherios Christoforidis, and Dimitrios Soudris. Ddot: Data driven online tuning for energy efficient acceleration. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[51] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 13–24. ACM, 2015.

[52] Haipeng Guo. A bayesian approach for automatic algorithm selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI03), Workshop on AI and Autonomic Computing, Acapulco, Mexico*, pages 1–5, 2003.

[53] Ari Rasch, Michael Haidl, and Sergei Gorlatch. Atf: A generic auto-tuning framework. In *High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017 IEEE 19th International Conference on*, pages 64–71. IEEE, 2017.

[54] Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, et al. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. In *International Workshop on Applied Parallel Computing*, pages 328–342. Springer, 2012.

[55] Jesús Cámara, Javier Cuenca, and Domingo Giménez. Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters. *The Journal of Supercomputing*, pages 1–20, 2020.

[56] Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, pages 195–202. IEEE, 2015.

[57] Christoph A Schaefer, Victor Pankratius, and Walter F Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In *European Conference on Parallel Processing*, pages 9–20. Springer, 2009.

[58] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[59] Joshua San Miguel and Natalie Enright Jerger. The anytime automaton. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 545–557. IEEE Press, 2016.

[60] Xin Sui, Andrew Lenharth, Donald S Fussell, and Keshav Pingali. Proactive control of approximate programs. *ACM SIGOPS Operating Systems Review*, 50(2):607–621, 2016.

[61] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. *ACM SIGPLAN Notices*, 49(4):35–50, 2014.

[62] Martin Tillmann, Thomas Karcher, Carsten Dachsbacher, and Walter F Tichy. Application-independent autotuning for gpus. In *PARCO*, pages 626–635, 2013.

[63] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. *Programming Models for Emerging Architectures*, 1(1):1–9, 2009.

[64] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May OŔeilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices*, volume 50, pages 379–390. ACM, 2015.

[65] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May OŔeilly, and Saman Amarasinghe. Siblingrivalry: online autotuning through local competitions. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 91–100. ACM, 2012.

[66] Nick Chaimov, Boyana Norris, and Allen Malony. Toward multi-target autotuning for accelerators. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 534–541. IEEE, 2014.

[67] Cristian Ţăpuş, I-Hsin Chung, Jeffrey K Hollingsworth, et al. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.

[68] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Respir: a response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1816–1829, 2009.

[69] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, May 2012.

[70] Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland. Can search algorithms save large-scale automatic performance tuning? *Procedia Computer Science*, 4:2136 – 2145, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[71] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. 1997.

[72] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[73] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[74] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[75] Kedar S. Namjoshi and Nimit Singhania. Loopy: Programmable and Formally Verified Loop Transformations. In Xavier Rival, editor, *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 383–402. Springer Berlin Heidelberg, 2016.

[76] Dheya Mustafa and Rudolf Eigenmann. Portable Section-level Tuning of Compiler Parallelized Applications. In *High Performance Computing, Networking, Storage and Analysis*, 2012.

[77] Troy A. Johnson, Sang-Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P. Midkiff. Experiences in using cetus for source-to-source transformations. In *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing*, LCPC'04, pages 1–14, Berlin, Heidelberg, 2005. Springer-Verlag.

[78] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. *International Journal of Parallel Programming*, 41(6):753–767, 2013.

[79] D. Gadioli, R. Nobre, P. Pinto, E. Vitali, A. H. Ashouri, G. Palermo, J. Cardoso, and C. Silvano. Socrates - a seamless online compiler and system runtime autotuning framework for energy-aware applications. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1143–1146, 2018.

[80] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Trans. Archit. Code Optim.*, 9(3):21:1–21:30, October 2012.

[81] P. Pinto, R. Abreu, and J. M. P. Cardoso. Fault Detection in C Programs using Monitoring of Range Values: Preliminary Results. *ArXiv*, 2015.

[82] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench*, 2012.

[83] Cristina Videira Lopes and Gregor Kiczales. *D: A language framework for distributed programming*. PhD thesis, Northeastern University, 1997.

[84] Marc Duranton, Koen De Bosschere, Christian Gamrat, Jonas Maebe, Harm Munk, and Olivier Zendra. The hipeac vision 2017, 2017.

[85] R. Lu, X. Liang, X. Li, X. Lin, and X. Shen. Eppa: An efficient and privacy-preserving aggregation scheme for secure smart grid communications. *IEEE Transactions on Parallel and Distributed Systems*, 23(9):1621–1631, Sep. 2012.

[86] G. C. T. Chow, K. Eguro, W. Luk, and P. Leong. A Karatsuba-based Montgomery multiplier. In *FPL 2010*, pages 434–437, Aug 2010.

[87] Lukas Malina and Jan Hajny. Accelerated modular arithmetic for low-performance devices. In *TSP 2011*. IEEE, 2011.

[88] Nadia Nedjah and Luiza de Macedo Mourelle. A review of modular multiplication methods ands respective hardware implementation. *Informatica*, 30(1), 2006.

[89] C. Rafferty, M. O'Neill, and N. Hanley. Evaluation of large integer multiplication methods on hardware. *IEEE Transactions on Computers*, 66(8):1369–1382, Aug 2017.

[90] Martin Kumm, Oscar Gustafsson, Florent De Dinechin, Johannes Kappauf, and Peter Zipf. Karatsuba with rectangular multipliers for FPGAs. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 13–20. IEEE, 2018.

[91] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta. A reconfigurable computing system based on a cache-coherent fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 80–85, Nov 2011.

[92] Intel. Math kernel library.

[93] Florent de Dinechin and Bogdan Pasca. Large multipliers with fewer DSP blocks. In *FPL 2009*, pages 250–255. IEEE, 2009.

[94] Torbjrn Granlund and Gmp Development Team. *GNU MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, United Kingdom, 2015.

[95] William Hart, Fredrik Johansson, and Sebastian Pancratz. Flint – fast library for number theory, 2011.

[96] Victor Shoup. Ntl 11.3.2: A library for doing number theory, 2019.

[97] Xilinx. Vivado high level synthesis.

[98] Xilinx. SDAccel Development Environment.

[99] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *FPL 2013*, pages 1–4.

[100] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *FPGA conference*, pages 33–36. ACM, 2011.

[101] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Syst. J.*, 29(4):526–538, October 1990.

[102] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[103] Johann Großschädl, Roberto M. Avanzi, Erkay Savaş, and Stefan Tillich. Energy-efficient software implementation of long integer modular arithmetic. In *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 75–90.

[104] Emanuele Vitali, Davide Gadioli, Fabrizio Ferrandi, and Gianluca Palermo. Parametric throughput oriented large integer multipliers for high level synthesis.

[105] Paolo Toth and Daniele Vigo. *Vehicle Routing: Problems, Methods, and Applications*, volume 18. SIAM, 2014.

[106] Radek Tomis, Lukáš Rapant, Jan Martinovič, Kateřina Slaninová, and Ivo Vondrák. Probabilistic time-dependent travel time computation using monte carlo simulation. In *International Conference on High Performance Computing in Science and Engineering*, pages 161–170. Springer, 2015.

[107] Martin Golasowski, Radek Tomis, Jan Martinovič, Kateřina Slaninová, and Lukáš Rapant. Performance evaluation of probabilistic time-dependent travel time computation. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pages 377–388. Springer, 2016.

[108] Michael J. Gilman. A brief survey of stopping rules in monte carlo simulations. In *Proceedings of the Second Conference on Applications of Simulations*, pages 16–20. Winter Simulation Conference, 1968.

[109] Anton Agafonov and Vladislav Myasnikov. Reliable routing in stochastic time-dependent network with the use of actual and forecast information of the traffic flows. In *Intelligent Vehicles Symposium (IV), 2016 IEEE*, pages 1168–1172. IEEE, 2016.

[110] Samitha Samaranayake, Sebastien Blandin, and A Bayen. A tractable class of algorithms for reliable routing in stochastic networks. *Procedia-Social and Behavioral Sciences*, 17:341–363, 2011.

[111] Yu Marco Nie and Xing Wu. Shortest path problem considering on-time arrival probability. *Transportation Research Part B: Methodological*, 43(6):597–613, 2009.

[112] Maleen Abeydeera and Samitha Samaranayake. Gpu parallelization of the stochastic on-time arrival problem. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–8. IEEE, 2014.

[113] Mehrdad Niknami and Samitha Samaranayake. Tractable pathfinding for the stochastic on-time arrival problem. In *International Symposium on Experimental Algorithms*, pages 231–245. Springer, 2016.

[114] Evdokia Nikolova, Jonathan Kelner, Matthew Brand, and Michael Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. *Algorithms–ESA 2006*, pages 552–563, 2006.

[115] Andreas Paraskevopoulos and Christos Zaroliagis. Improved Alternative Route Planning. In Daniele Frigioni and Sebastian Stiller, editors, *ATMOS - 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems - 2013*, volume 33 of *OpenAccess Series in Informatics (OASIcs)*, pages 108–122, Sophia Antipolis, France, September 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[116] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. Alternative routing: k-shortest paths with limited overlap. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, pages 68:1–68:4, 2015.

[117] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. Exact and approximate algorithms for finding k-shortest paths with limited overlap. pages 414–425, 2017.

[118] Hans Janssen. Monte-carlo based uncertainty analysis: Sampling efficiency and sampling convergence. *Reliability Engineering & System Safety*, 109:123 – 132, 2013.

[119] Q. Xu, M. Sbert, M. Feixas, and J. Sun. A new adaptive sampling technique for monte carlo global illumination. In *2007 10th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pages 191–196, Oct 2007.

[120] J. S. Miguel and N. E. Jerger. The anytime automaton. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 545–557, June 2016.

[121] Elise Miller-Hooks and Hani Mahmassani. Path comparisons for a priori and time-adaptive decisions in stochastic, time-varying networks. *European Journal of Operational Research*, 146(1):67–82, 2003.

[122] Jan Martinovič, Václav Snášel, Jiří Dvorský, and Pavla Dráždilová. Search in documents based on topical development. In Vaclav Snášel, Piotr S. Szczepaniak, Ajith Abraham, and Janusz Kacprzyk, editors, *Advances in Intelligent Web Mastering - 2*, pages 155–166, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[123] Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Martin Golasowski, Joao Bispo, Pedro Pinto, Jan Martinovic, Katerina Slaninova, Joao Cardoso, and Cristina Silvano. An efficient monte carlo-based probabilistic time-dependent routing calculation targeting a server-side car navigation system. *IEEE transactions on emerging topics in computing*, 2019.

[124] Mohammad Asghari, Tobias Emrich, Ugur Demiryurek, and Cyrus Shahabi. Probabilistic estimation of link travel times in dynamic road networks. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 47. ACM, 2015.

[125] JM Juritz, JWF Juritz, and MA Stephens. On the accuracy of simulated percentage points. *Journal of the American Statistical Association*, 78(382):441–444, 1983.

[126] D. C. Montgomery and G. C. Runger. *Applied Statistics and Probability for Engineers*. John Wiley and Sons, 2003.

[127] D. Zwillinger and S. Kokoska. *CRC Standard Probability and Statistics Tables and Formulae*. Chapman & Hall, 2000.

[128] Radek Tomis, Jan Martinovič, Kateřina Slaninová, Lukáš Rapant, and Ivo Vondrák. Time-dependent route planning for the highways in the czech republic. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pages 145–153. Springer, 2015.

[129] Phelim P. Boyle. Options: A monte carlo approach. *Journal of Financial Economics*, 4(3):323–338, 1977.

[130] Roger Koenker. *Quantile Regression*. Econometric Society Monographs. Cambridge University Press, 2005.

[131] Federal Highway Administration US Department of Transportation. *US Department of Transportation, Federal Highway Administration – Traffic Report*. 2014.

[132] gov.uk UK Department for Transport. *Average annual daily flow and temporal traffic distributions*. 2017.

[133] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: Performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, March 2009.

[134] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[135] Milano Agenzia Mobilita' Ambiente e Territorio. *Annual Mobility Report*.

[136] Marco Bedogni, Milano Agenzia Mobilita' Ambiente e Territorio. *Road Traffic Measures in The City of Milan*.

[137] Marco Gribaudo, Pietro Piazzolla, and Giuseppe Serazzi. Consolidation and replication of vms matching performance objectives. In Khalid Al-Begain, Dieter Fiems, and Jean-Marc Vincent, editors, *Analytical and Stochastic Modeling Techniques and Applications*, pages 106–120, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[138] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr DollÃ¡r. Microsoft coco: Common objects in context, 2014. cite arxiv:1405.0312Comment: 1) updated annotation pipeline description and figures; 2) added new section describing datasets splits; 3) updated author list.

[139] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[140] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[141] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[142] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[143] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7310–7311, 2017.

[144] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. Adaptive deep learning model selection on embedded systems. *ACM SIGPLAN Notices*, 53(6):31–43, 2018.

[145] E. Park, D. Kim, S. Kim, Y. Kim, G. Kim, S. Yoon, and S. Yoo. Big/little deep neural network for ultra low power inference. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 124–132, 2015.

[146] Hokchhay Tann, Soheil Hashemi, and Sherief Reda. Flexible deep neural network processing. *arXiv preprint arXiv:1801.07353*, 2018.

[147] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[148] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys 2018, pages 389–400, New York, NY, USA, 2018. Association for Computing Machinery.

[149] A. Mazouz and C. P. Bridges. Multi-sensory cnn models for close proximity satellite operations. In *2019 IEEE Aerospace Conference*, pages 1–7, 2019.

[150] Emanuele Vitali and Anton Lokhmotov. omni benchmarking object detection. `https://towardsdatascience.com/omni-benchmarking-object-detection-b390cc4114cd`. Accessed: 2021-02-22.

[151] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[152] Mahmoud Hassaballah, Aly Amin Abdelmgeid, and Hammam A Alshazly. Image features detection, description and matching. In *Image Feature Detectors and Descriptors*, pages 11–45. Springer, 2016.

[153] OpenCV. Open source computer vision library, 2015.

[154] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[155] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[156] FranÃ§ois Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[157] Andrea R Beccari, Carlo Cavazzoni, Claudia Beato, and Gabriele Costantino. Ligen: a high performance workflow for chemistry driven de novo design. *Journal of Chemical Information and Modeling*, 53(6):1518–1527, 2013.

[158] Evanthia Lionta, George Spyrou, Demetrios K. Vassilatis, and Zoe Cournia. Structure-based virtual screening for drug discovery: Principles, applications and recent advances. *Current Topics in Medicinal Chemistry*, 14(16):1923–1938, 2014.

[159] Andrea R. Beccari, Marica Gemei, Matteo Lo Monte, Nazareno Menegatti, Marco Fanton, Alessandro Pedretti, Silvia Bovolenta, Cinzia Nucci, Angela Molteni, Andrea Rossignoli, Laura Brandolini, Alessandro Taddei, Lorena Za, Chiara Liberati, and Giulio Vistoli. Novel selective, potent naphthyl trpm8 antagonists identified through a combined ligand- and structure-based virtual screening approach. In *Scientific reports*, 2017.

[160] Robin Duelen, Marlies Corvelyn, Ilaria Tortorella, Leonardo Leonardi, Yoke Chai, and Maurilio Sampaolesi. *Medicinal Biotechnology for Disease Modeling, Clinical Therapy, and Drug Discovery and Development*, pages 89–128. 08 2019.

[161] René Thomsen and Mikael H Christensen. Moldock: a new technique for high-accuracy molecular docking. *Journal of medicinal chemistry*, 49(11):3315–3321, 2006.

[162] Gareth Jones, Peter Willett, Robert C Glen, Andrew R Leach, and Robin Taylor. Development and validation of a genetic algorithm for flexible docking. *Journal of molecular biology*, 267(3):727–748, 1997.

[163] Richard A. Friesner, Jay L. Banks, Robert B. Murphy, Thomas A. Halgren, Jasna J. Klicic, Daniel T. Mainz, Matthew P. Repasky, Eric H. Knoll, Mee Shelley, Jason K. Perry, David E. Shaw, Perry Francis, and Peter S. Shenkin. Glide: A new approach for rapid, accurate docking and scoring. 1. method and assessment of docking accuracy. *Journal of Medicinal Chemistry*, 47(7):1739–1749, 2004. PMID: 15027865.

[164] Ming Liu and Shaomeng Wang. Mcdock: a monte carlo simulation approach to the molecular docking problem. *Journal of computer-aided molecular design*, 13(5):435–451, 1999.

[165] Fan Jiang and Sung-Hou Kim. “soft docking”: matching of molecular surface cubes. *Journal of molecular biology*, 219(1):79–102, 1991.

[166] P Nuno Palma, Ludwig Krippahl, John E Wampler, and José JG Moura. Bigger: a new (soft) docking algorithm for predicting protein interactions. *Proteins: Structure, Function, and Bioinformatics*, 39(4):372–384, 2000.

[167] Todd JA Ewing, Shingo Makino, A Geoffrey Skillman, and Irwin D Kuntz. Dock 4.0: search strategies for automated molecular docking of flexible molecule databases. *Journal of computer-aided molecular design*, 15(5):411–428, 2001.

[168] Bernd Kramer, Matthias Rarey, and Thomas Lengauer. Evaluation of the flexx incremental construction algorithm for protein-ligand docking. *Proteins: Structure, Function, and Bioinformatics*, 37(2):228–241, 1999.

[169] Ingo Schellhammer and Matthias Rarey. Flexx-scan: Fast, structure-based virtual screening. *PROTEINS: Structure, Function, and Bioinformatics*, 57(3):504–517, 2004.

[170] Ajay N Jain. Surflex-dock 2.1: robust performance from ligand energetic modeling, ring flexibility, and knowledge-based search. *Journal of computer-aided molecular design*, 21(5):281–306, 2007.

[171] Douglas B. Kitchen, Hélène Decornez, John R. Furr, and Jürgen Bajorath. Docking and scoring in virtual screening for drug discovery: methods and applications. *Nature Reviews Drug Discovery*, 3, Nov 2004. Review Article.

[172] Paul D Lyne. Structure-based virtual screening: an overview. *Drug Discovery Today*, 7(20):1047 – 1055, 2002.

[173] Jayashree Srinivasan, Angelo Castellino, Erin K. Bradley, John E. Eksterowicz, Peter D. J. Grootenhuis, Santosh Putta, and Robert V. Stanton. Evaluation of a novel shape-based computational filter for lead evolution: Application to thrombin inhibitors. *Journal of Medicinal Chemistry*, 45(12):2494–2500, 2002. PMID: 12036357.

[174] Evanthia Lionta, George Spyrou, Demetrios Vassilatis, and Zoe Cournia. Structure-based virtual screening for drug discovery: Principles, applications and recent advances. *Current topics in medicinal chemistry*, 09 2014.

[175] Aleix Gimeno, MJosé Montes, Sarah Tomás-Hernández, Adrià Cereto-Massagué, Raúl Beltrán-Debón, Miquel Mulero, Gerard Pujadas, and Santi Garcia-Vallve. The light and dark sides of virtual screening: What is there to know? *International Journal of Molecular Sciences*, 20:1375, 03 2019.

[176] Claudia Beato, Andrea R Beccari, Carlo Cavazzoni, Simone Lorenzi, and Gabriele Costantino. Use of experimental design to optimize docking performance: The case of ligendock, the docking module of ligen, a new de novo design program, 2013.

[177] Atanu Acharya, Rupesh Agarwal, Matthew Baker, Jerome Baudry, Debsindhu Bhowmik, Swen Boehm, Kendall Byler, Leighton Coates, Sam Yen-Chi Chen, Connor J. Cooper, and et al. Supercomputer-based ensemble docking drug discovery pipeline with application to covid-19, Jul 2020.

[178] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–6, May 2013.

[179] TJ Fuller-Rowell. A two-dimensional, high-resolution, nested-grid model of the thermosphere: 1. neutral response to an electric field “spike”. *Journal of Geophysical Research: Space Physics*, 89(A5):2971–2990, 1984.

[180] TJ Fuller-Rowell. A two-dimensional, high-resolution, nested-grid model of the thermosphere: 2. response of the thermosphere to narrow and broad electrodynamic features. *Journal of Geophysical Research: Space Physics*, 90(A7):6567–6586, 1985.

[181] Wenbin Wang, Tim L Killeen, Alan G Burns, and Raymond G Roble. A high-resolution, three-dimensional, time dependent, nested grid model of the coupled thermosphere–ionosphere. *Journal of Atmospheric and Solar-Terrestrial Physics*, 61(5):385–397, 1999.

[182] Lie-Yauw Oey and Ping Chen. A nested-grid ocean model: With application to the simulation of meanders and eddies in the norwegian coastal current. *Journal of Geophysical Research: Oceans*, 97(C12):20063–20086, 1992.

[183] Michael S. Fox-Rabinovitz, Lawrence L. Takacs, Ravi C. Govindaraju, and Max J. Suarez. A variable-resolution stretched-grid general circulation model: Regional climate simulation. *Monthly Weather Review*, 129(3):453–469, 2001.

[184] FÃ¡bio F. Bernardon, Christian A. Pagot, JoÃ£o L. D. Comba, and ClÃ¡udio T. Silva. Gpu-based tiled ray casting using depth peeling. *Journal of Graphics Tools*, 11(4):1–16, 2006.

[185] Cass Everitt. Interactive order-independent transparency. *White paper, nVIDIA*, 2(6):7, 2001.

[186] Cheng-Kai Chen, Chris Ho, Carlos Correa, Kwan-Liu Ma, and Ahmed Elgamal. Visualizing 3d earthquake simulation data. *Computing in Science & Engineering*, 13(6):52–63, 2011.

[187] G Patrick Brady and Pieter FW Stouten. Fast prediction and visualization of protein binding pockets with pass. *Journal of computer-aided molecular design*, 14(4):383–401, 2000.

[188] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic Acids Res*, 28:235–242, 2000.

[189] Davide Gadioli, Gianluca Palermo, Stefano Cherubin, Emanuele Vitali, Giovanni Agosta, Candida Manelfi, Andrea R Beccari, Carlo Cavazzoni, Nico Sanna, and Cristina Silvano. Tunable approximations to control time-to-solution in an hpc molecular docking mini-app. *The Journal of Supercomputing*, 77(1):841–869, 2021.

[190] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[191] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[192] Rob Farber. *Parallel programming with OpenACC*. Newnes, 2016.

[193] Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Andrea Beccari, and Cristina Silvano. Accelerating a geometric approach to molecular docking with openacc. In *Proceedings of the 6th International Workshop on Parallelism in Bioinformatics*, PBio 2018, pages 45–51, New York, NY, USA, 2018. ACM.

[194] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[195] OpenACC.org. Openacc programming and best practices guide, June 2015. `https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf`.

[196] OpenACC-Standard.org. *The OpenACC Application Programming Interface*, November 2017. `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf`.

[197] NVIDIA. Profiler user guide, May 2018. `https://docs.nvidia.com/cuda/profiler-users-guide/`.

[198] Helen M Berman, John Westbrook, Zukang Feng, Gary Gilliland, Talapady N Bhat, Helge Weissig, Ilya N Shindyalov, and Philip E Bourne. The protein data bank, 1999–. In *International Tables for Crystallography Volume F: Crystallography of biological macromolecules*, pages 675–684. Springer, 2006.

[199] NVIDIA. Cuda toolkit documentation, June 2018.

[200] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[201] S. Sawadsitang, J. Lin, S. See, F. Bodin, and S. Matsuoka. Understanding performance portability of openacc for supercomputers. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 699–707, May 2015.

[202] Yonghong Yan, Jiawen Liu, Kirk W. Cameron, and Mariam Umar. Homp: Automated distribution of parallel loops and data in highly parallel accelerator-based systems. pages 788–798, 05 2017.

[203] Yonghong Yan, Pei-Hung Lin, Chunhua Liao, Bronis R. de Supinski, and Daniel J. Quinlan. Supporting multiple accelerators in high-level programming models. pages 170–180, 02 2015.

[204] Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, and Barbara Chapman. Multi-gpu support on single node using directive-based programming model. *Sci. Program.*, 2015:3:3–3:3, January 2016.

[205] Stefano Markidis, Jing Gong, Michael Schliephake, Erwin Laure, Alistair Hart, David Henty, Katherine Heisey, and Paul Fischer. Openacc acceleration of the nek5000 spectral element code. *The International Journal of High Performance Computing Applications*, 29(3):311–319, 2015.

[206] Rengan Xu, Sunita Chandrasekaran, and Barbara Chapman. Exploring programming multi-gpus using openmp and openacc-based hybrid model. 05 2013.

[207] Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Andrea Beccari, Carlo Cavazzoni, and Cristina Silvano. Exploiting openmp and openacc to accelerate a geometric approach to molecular docking in heterogeneous hpc nodes. *The Journal of Supercomputing*, 75(7):3374–3396, 2019.

[208] Gianluca Palermo, Davide Gadioli, Emanuele Vitali, Cristina Silvano, Federico Ficarelli, Chiara Latini, Andrea R Beccari, and Candida Manelfi. Exscalate4cov: Towards an exascale-ready docking platform targeting urgent computing. EuroHPC Week, 2021.

[209] Seyyed Reza Miri Rostami and Mohsen Ghaffari-Miab. Finite difference generated transient potentials of open-layered media by parallel computing using openmp, mpi, openacc, and cuda. *IEEE Transactions on Antennas and Propagation*, 67(10):6541–6550, 2019.

[210] Gabriell Alves de Araujo, Dalvan Griebler, Marco Danelutto, and Luiz Gustavo Fernandes. Efficient nas parallel benchmark kernels with cuda. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 9–16. IEEE, 2020.

[211] M Harris and K Perelygin. Cooperative groups: Flexible cuda thread programming, 2017.