POLITECNICO DI MILANO

Scuola di Ingegneria Insdustriale e Dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

POLITECNICO
MILANO 1863

Efficient OpenCL Application Autotuning for Heterogeneous Platforms

Advisor: Prof. Cristina Silvano

Co-Advisor: Gadioli Davide

Master Thesis of:

Ahmet Erdem

814168

Academic Year 2015-2016

Abstract in English

Recently, OpenCL standard reached much wider audiences due to the increasing number of devices supporting it. At the same time, we have observed increase of differences among devices that support OpenCL. This situation offers to developers, who want to get high performance, a large spectrum of platforms. Given the additional OpenCL platform parameters along side application specific parameters, design space for exploration is seriously large. Furthermore, availability of more than one kind of device allows distribution of computation on the heterogeneous platform. Automatic design space exploration frameworks are one of the recent approaches to address these problems and to reduce the burden of programmers. In this Thesis, we propose more automatic and efficient techniques to prune the design space before moving on to the exploration phase. In addition, the Thesis proposes new methods for allocating the computational tasks on the available resources. To assess the proposed methodology, experiments have been carried out for two application case studies mapped on two heterogeneous computing platforms. Experimental results report that the proposed pruning technique reduces the huge exploration space to a feasible size while achieving 57% speed up utilizing heterogeneity of the the platform for the two selected use cases.

Abstract in Italian

Recentemente, lo standard OpenCL ha raggiunto una sempre più ampia diffusione grazie all'incremento del numero di dispositivi che lo supportano. Allo stesso tempo, abbiamo osservato un incremento delle differenze tra i dispositivi che supportano OpenCL. Questa situazione offre agli sviluppatori, che desiderano ottenere alte prestazioni, un ampio spettro di piattaforme. Dati i parametri addizionali relativi alla piattaforma offerti da OpenCL e i parametri specifici applicativi, la dimensione dello spazio di progettazione è enorme. Inoltre, la disponibilità di più di un tipo di dispositivo consente la distribuzione della computazione su piattaforma eterogenea. Framework automatici per l'esplorazione dello spazio di progetto rappresentano uno dei recenti approcci per affrontare questi problemi e per ridurre l'onere dei programmatori. In questa tesi, proponiamo tecniche automatiche ed efficienti per ridurre lo spazio di progettazione prima di passare alla fase di esplorazione. Inoltre, questa Tesi propone nuovi metodi di allocazione dei task computazionali sulle risorse disponibili. Per valutare la metodologia proposta, sono stati effettuati degli esperimenti realtivi a due applicazioni usate come casi di studio mappati su due piattaforme eterogenee. I risultati sperimentali mostrano che le tecniche proposte riducono l'enorme dimensione dello spazio progettuale ad una dimensione fattibile raggiungendo una velocizzazione del 57% sfruttando l'eterogeneità della la piattaforma per i due casi di studio selezionati.

Contents

List of Figures

Chapter 1

Introduction

The development of languages and tools have always been the keystone for adoption of the new technologies and approaches in the field of software development. With the presence of tools that are easy to use, well established methods are developed and as a result of this, more and more developers are attracted to the new ways of computing. One of the biggest innovative approach to the computing has occurred by multicore CPU becoming widespread even in the consumer devices. Few years prevalent multicore processors, fixed-function video adapters grew into programmable graphical processing units (GPUs).

These hardware developments helped processors to hit the power wall in silicon technology, which forced hardware vendors to abondon the tendency to increase in clock speeds and pursue innovative techniques such as placing multiple cores in the same package. While this technically allowed to multiply the processing power, it is necessary for developers to exploit the potential of parallelism lies in the processor.

In the first years of multicore era, the lack of tools was the big problem since explicit parallel programming was both new and hard to get it correct. Nevertheless, tools like OpenMP and Intel Threading Building Blocks (TBB) have come into play and adoption of handling the underlying power of multicore CPUs started to become a norm. With the understanding of computation power of future lies within parallelism of software, multiple frameworks now classified as MapReduce programming model emerged to handle this power in a more systematic way, especially at server side enterprise development.

While these advancements were on the CPU frontier, more and more demanding graphical applications like video games, real-time training simulations and CAD were pushing the edges of graphical power of GPUs. These applications usually developed using standards that industry-leader vendors agreed upon like OpenGL. While graphical applications dominate the domain of GPU computing, a paradigm was approaching. Due to rising computing power of GPUs' vertex and fragment shader processors (later unified) they managed to get attraction from performance demanding computational science and medical imaging workloads.

OpenGL is a framework for graphical applications. However developer in those fields managed to exploit the functionality provided by the framework in order to run their programs parallel on GPUs. These trend gave birth to what is known as General Purpose GPU computing (or GPGPU), meaning that using GPUs for workloads other than graphical purpose. Even though this paradigm seems emerged out of nowhere, the motivation behind it is pretty clearly the longing for more computational power. However as advanced as computing machines become, the need for completing tasks in a shorter span of time shall never cease.

Given this new way of utilizing computational power discovered, it is not surprising that hardware vendors started to come up with new tools and frameworks which lets developers to take advantage of underlying parallel architecture GPUs. CUDA and OpenCL are among those frameworks. While CUDA took the path of being proprietary and is only supported on NVIDIA GPUs officially, OpenCL has taken open-standard approach under Khronos Group which made possible any vendor to provide OpenCL implementation of their own. The earliest adoption of OpenCL standard outside of GPUs came from CPUs since accelerating programs using parallelism on CPUs has been widespread practice.

What OpenCL brought to the table is extremely important since throughout many experiments have shown that not all type of computational loads are suitable for all computing hardware. In order to address this issue, many manufacturers like AMD and Intel have introduced new generation of CPU dies that include what is called integrated graphical processors (iGPU although AMD specifically called the whole package APU). Having this kind of hardware available opened new opportunities to offload some of the workload of application which are especially computational heavy parts.

1.1 Motivation

Given the availability of hardware and computer systems which are consistent of CPUs and GPUs become prevalent every segment of the industry not only from mobile computing devices to high-end desktops where commercial multimedia applications dominate but also enterprise server-side computing to high performance computing (HPC). Leading to an era of heterogeneous computing, along with new burden on developers how to manually tune their programs and challenges of new programming paradigms. In order to deal with these rough nature of heterogeneous programming, various tools and techniques both compile-time and run-time have been being developed by compiler engineers. It is crucial to lower the barrier of development on this kind of platforms where heterogeneity persists. Because silicon industry is facing problems like power wall and dark silicon, the performance increase of each generation of CPUs due to clock speeds is diminishing. Thus, having multiple computing units with different degree of parallelism is a critically important aspect of today's computing systems.

Although embedded computing and mobile hand-hold devices are not as powerful as conventional computers, they include heterogeneous computing units likewise. Furthermore, their widespread availability due to popularity among consumers and low power consumption makes them attractive options for even computational heavy operations.

With all these diverse possibilities of hardware and importance of staying relevant in terms of application performance is vital for the industry. This creates important problems to cope with like how to utilize hardware and preserve possible performance benefits while migrating to another hardware. Especially, in the case of frameworks like OpenCL which supports different types of hardware (i.e. CPU, GPU), number of exposed parameters and diversity of optimal parameters on various architectures can easily be overwhelming for developers. Usually, developers hand-tune the application for the most prominent target architecture and baked the parameters inside the application, however this may create undesired sub-optimal solutions for potential future target hardware.

1.2 Proposed Approach

To address these problems, autotuning frameworks are recently been developed. There are two major kinds of application tuning:

- Static tuning, where autotuning is done at the design phase of the application.
- Dynamic tuning, done at run-time while application is running.

Dynamic tuning may have additional information compared to statically autotuned application, however dynamic autotuning implies there will be some run-time overhead and benefits of tuning must surpass in order to be meaningful tuning. In this work we will focus on statically autotuning.

In the work of [9] an external development tool-chain which statically searches optimal parameters for OpenCL applications to help developers. To be able to search meaningfully, there is phase before design space exploration where design space is pruned based on rules from OpenCL standard and application domain specific principles if they are provided. In its current condition, all of those parameters are expected to be explicitly stated by application developer using MiniZinc constraint modeling language.

The objective of this Thesis is to extend the functionalities of pruning phase of this tool[9] and also improve its usability by automatizing some parts of the tool. The first improvement to the existing DSE flow is automatic gathering of OpenCL parameters as much as possible so the developer needs to deal with minimum about of input to the DSE tool. Therefore with the removal of parameter acquisition burden, not only possibility of human error reduces but also the time to learn and use the exploration flow is significantly shortened.

Second, a kernel splitting method will be introduced to exploit computational resource on the occasion of heterogeneous platforms existence. Currently, the workflow can handle heterogeneity given that there are independent tasks that can be mapped to different hardware platforms. With this technique, it will be possible to map portions of the data to be processed on to available hardware resource and then, the results will be merged. Thus, the need for multiple independent tasks in order to leverage the power of heterogeneous architectures will not be necessary. Furthermore, types of algorithms that are suitable for this kind of splitting operation will be discussed.

1.3 Organization of the Thesis

The remainder of the Thesis is organized as follows:

- Chapter 2 introduces background information on OpenCL useful to better understand the methodology described in the following chapters along with case studies and target platforms which the techniques are applied.
- Chapter 3 describes in detail the methodology and the development approach proposed in this Thesis.
- Chapter 4 presents the experimental results in this Thesis with the

discussions as well as how the experiments have been carried out.

- Chapter 5 summarizes some concluding remarks on how the future works can continue to improve upon this work.

Chapter 2

Background

Some theoretical and technical topics in connection with the argument of this Thesis dissertation will be investigated in order to increase the understanding of the work and the methodology. In Section 2.1, the technical tools like OpenCL standard and constraint programming language MiniZinc will be discussed. Following that, in Section 2.2, what kind of computing architectures are relevant to the work that has been done and their impactful properties will be examined. Later, state of art technologies and some previous research works regarding the topic of this dissertation will be reviewed in Section 2.3.

2.1 External Tools and Frameworks

2.1.1 OpenCL Programming Model

Open Computing Language (OpenCL) which is maintained by Khronos consortium [5] is an open standard for developing parallel applications on heterogeneous systems by abstracting the underlying compute machine. Being a parallel framework, applications developed using OpenCL can take advantage of power of the underlying hardware. Whether the architecture of machine is GPU or CPU is not relevant as long as the manufacturer of the computing processor provides the necessary run-time environment and driver support. Since all conformant products can be targeted using the standard, platform portability of applications between different hardware products even if the products belong to different architectures and vendors. Therefore switching between different types of accelerator platforms like from multicore CPUs to GPGPUs becomes less of a problem for application developers. This results in reducing the design cost of applications with compared to platform-specific solutions in case of porting the application to another platform. Furthermore, by allowing accelerators to be programmable using methods already established and well-known by the software industry, OpenCL not only lowers the entry barrier for inexperience individuals but also grants re-usability to the platforms on wide range of fields, from multimedia to scientific computing and medical imaging.

Even though OpenCL has been inspired heavily by CUDA, it didn't constraint itself to a specific vendor or product group rather come up with an abstraction across different architectures. It adopted a computation offloading programming paradigm such that while application running regularly on a conventional CPU, the computationally intensive parts of the application are offloaded to the available accelerators. In this paradigm of computing, CPU that is running the application code is called host and accelerators are called OpenCL devices. Host is responsible for managing execution of OpenCL devices by taking care of data transfers between host and device, especially in the case of host and device not sharing the memory address space.

This separation of host and device requires two different abstractions:

- An OpenCL host API where the management of device resources such as buffers and kernel launching is abstracted.

- An OpenCL language derived from C that defines the parallel computation that will take place inside computation intensive kernels.

In addition to these abstractions, there are extensions that can enable to utilize certain characteristics of certain groups of hardware. These extensions often can be vendor-specific and deviate from portability which OpenCL brings to the table. Hence, they will mainly be ignored throughout this Thesis in order to maintain generalization of hardware platforms.

Using OpenCL, it is possible to develop applications that exploit either *data parallelism* or *task parallelism*. In order to obtain task parallelism using OpenCL, host API provides *clEnqueueTask* function. Using clEnqueueTask, it is possible to launch single works that will be scheduled by the OpenCL run-time environment. On the other hand, to achieve data parallelism the developer has to use *clEnqueueNDRange* from the host API of OpenCL standard.

OpenCL adopts data parallelism approach by describing the parallel computations as a group of work-items, called work-groups. This hierarchical parallelism has been realized by launching kernel functions with a number of work-groups including a set of work-items. Kernel function describes how each work-item defines the operations that is to be carried out on a single data. Therefore the collection of work-items under all work-groups together expresses the data parallelism for an application. Due to the nature of many algorithms (i.e. image processing) that adopted parallelism, OpenCL provides a way to define multiple dimensions on how the work-items and work-groups can form. This multi dimensional composition results in the iteration space with two nested grids: a global grid where work-groups are arranged and a local grid which describes how the work-items are structured.

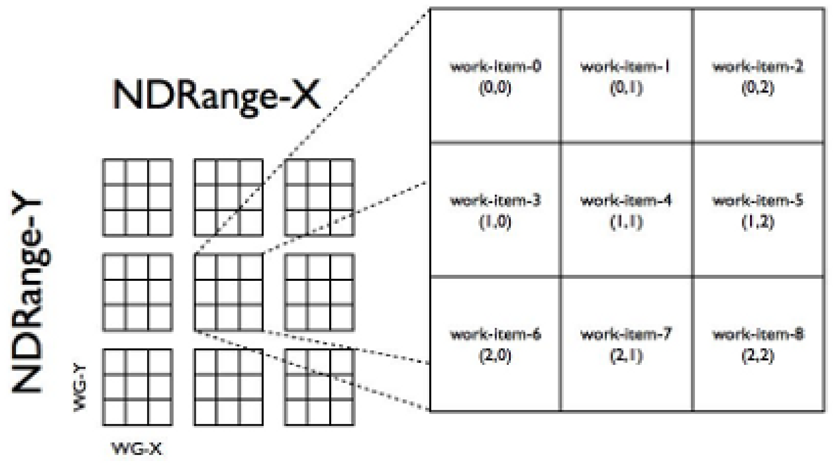


Figure 2.1: Global and Local Grid

The Figure 2.1 illustrates how the iteration space of an two dimensional work-items and work-groups is organized and in the context of OpenCL standard this space is called NDRange.

OpenCL Memory Model

In contrast to the modern CPUs, OpenCL uses hierarchical memory model where the control of the intermediate levels of the memory is in the hands of the application programmer. At each level of the hierarchy, different types of memory with potentially different characteristic reside. In order to leverage the benefits, application programmer has to explicitly specify the usage. Although this gives important control over the memory hierarchy leading to likely better utilization of memory, it is not as straightforward as conventional CPUs where the control mechanisms of memory (cache) hierarchy is hidden and controlled by dedicated hardware solutions like cache memory subsystem and memory management unit (MMU).

In OpenCL, the host program has conventional RAM for the application logic running on the host side, while each OpenCL device has its own global memory address space. The location and the type of the global memory is abstracted away by OpenCL hence if OpenCL device and host side have shared memory configuration. But that is not the necessarily the case for all hardware configurations, for instance, systems with discrete GPUs have their own memory, sometimes called VRAM (Video RAM), which is located closer to the graphical processor and accessible through connection bus between host device and the GPU (e.g. PCIe bus).

When we go down in the memory hierarchy, there is local memory address space which supposedly maps to memory implementation located inside the same die, thus may have much higher bandwidth compared to global memory. Local memory is a special programmable cache that provides a standard way for work-items within the same work-group to communicate. Although it allows work-items to write and read, local memories are allocated per work-group. Therefore, data exchange between different work-groups are not possible. Moreover, since amount of the local memory is limited and device dependent, it can limit the practically possible size of a work-group. This phenomena and why it is crucial will be discussed in Chapter 3.

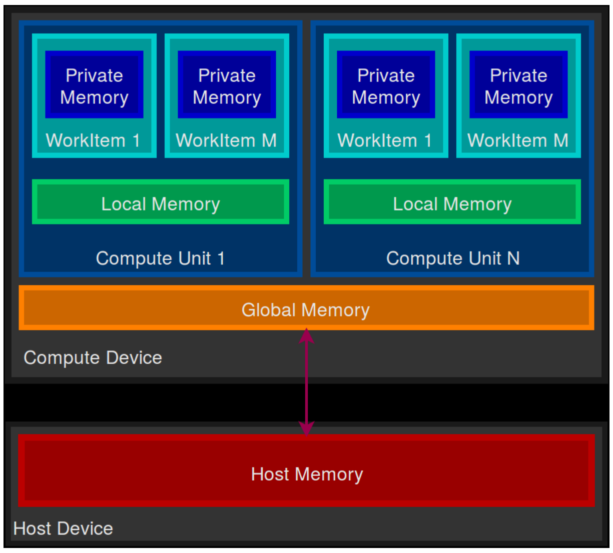


Figure 2.2: OpenCL Memory Model

Like shown in the Figure 2.2, there is one more type of memory which is called private memory. As the name suggests, private memory is exclusive to the individual work-items, consequently other work-items cannot access this address space. Inside the kernel function all the variables not declared with an address space will automatically defined in private memory address space [6].

Apart from these address spaces, there is constant memory address space which is allocated inside global memory as a special region and values inside constant memory are read-only from the perspective of program running on the OpenCL device. Hence, even though being located inside global memory, some architectures may take advantage of the read-only property of this address space [3].

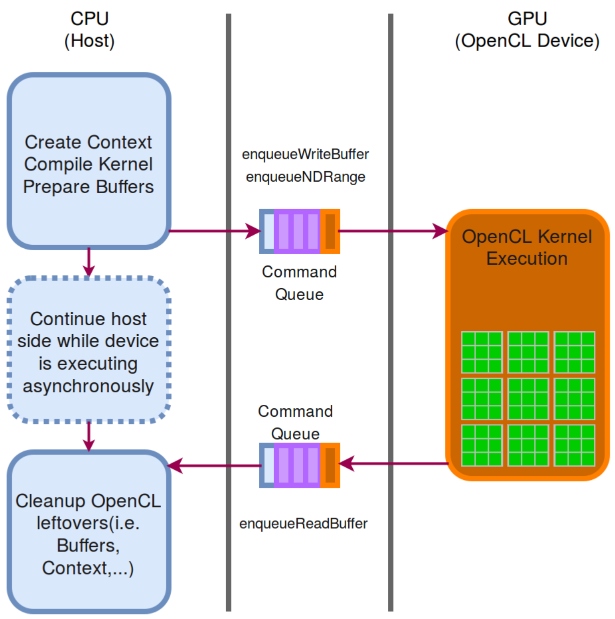


Figure 2.3: OpenCL Kernel Launching

As mentioned before, while OpenCL device runs the kernel code specified as iteration space NDRange, host side manages device and kernel related resource like buffers, context. Most importantly, host side initiates data transfer between host and device for buffers and launches the kernel using what is called command queue. By adopting this approach, OpenCL allows host to control device execution asynchronously. Thus, in the mean time of device execution, host side is free to continue its routine activities. Because of this ability of running programs on more than one type of architecture simultaneously, OpenCL applications are considered heterogeneous (Figure 2.3). Additionally, a hardware platform may contain more than one type of processor or accelerator with OpenCL run-time support. In this case, it is possible to make use of the computation power of the available OpenCL devices together to divide workload among themselves. Unless, it is necessary, this kind of approach often avoided due to complexity it brings to the application development.

2.1.2 MiniZinc

MiniZinc is a medium-level constraint modeling language. It is high-level enough to express most constraint problems easily [8]. It supports wide range of constraint solvers.

Constraint programming is a form of declarative programming where relationships between variables established with model described by the arrangement of a set of constraints which specifies the properties of the target solution. Contrary to imperative programming, constraints does not define sequences of execution steps, but rather the set of constraints is solved by giving value to each variable.MiniZinc, being a constraint programming language, allows to define parameters, decision variables and constraints in its own simple way.

Parameters and decision variables, similar to imperative languages, describe model variables , however the values of the parameters are single values, therefore there can only be one assignment per parameter and they determine fixed parameters for the whole solution. Differently than parameters, decision variables are associated with a set of possible that the variable can take. This set of possibilities are called variable's domain. Decision variables take value only when MiniZinc model is executed such that solving the system determines whether the variable can be assigned a value which is within the constraints, if so the decision variable takes that value. The next component of the MiniZinc model are the constraints. These statements that specify boolean expressions that the decision variables must satisfy within their domain to be a valid solution to the model.

There is more to the MiniZinc language and constraint programming, however it is not a topic of this Thesis rather it is a tool that has been taken advantage of.

2.2 Target Architectures

2.2.1 CPU Architectures

Central Processing Units (CPU) are general purpose processors and most of the time they are required to run operating system which handles preemptive scheduling of many processes. In order to handle the workloads of multiple processes executing, CPUs evolved into multicore machines where each core is big and complex with additional support for context switching and special memory management techniques to improve interprocess security like virtual memory address space mapping. While CPUs grew rich with features, they also kept the priority over the single-threaded performance. However, software development does not embrace the multithreaded application development immediately, therefore single-threaded performance per process are still relevant today. In order to achieve performance goals of the applications which mainly have sequential flow of execution, the individual cores get more and more complex with each iteration of generation especially on the commercial and high-end performance line of products by adopting advanced techniques like superscalar with execution units that are 4 or 8 issues wide, out of order execution and branch prediction. Additionally, almost all current generation CPU architectures provide a set of extensions to their ISA classified as SIMD instructions to exploit data level parallelism.

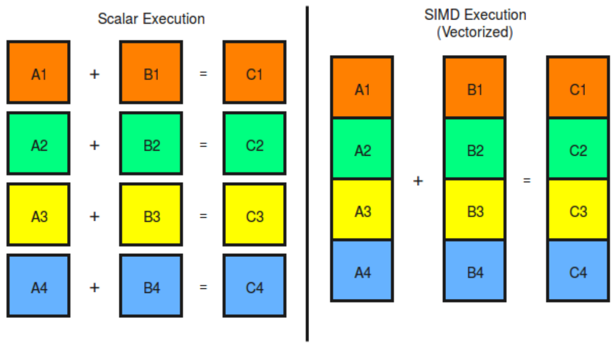


Figure 2.4: SIMD Instruction Execution

ISA extensions are quite abundant either in x86 or ARM architectures. For x86, there are SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) extensions, while for ARM, there is NEON instructions. Their main benefit is the increase the throughput by executing the same operation on multiple data in one clock cycle instead of issuing individual instruction per each data. This type of processing sometimes called vector processing. The number of data that is packed together depends on the width of the SIMD operations and the size of a single data. For instance, x86 SSE ISA extension is 256 bit wide hence, if the data is composed of 32 bit single precision IEEE floating point, each instruction can operate on 4 data points simultaneously (Figure 2.4). One down side is that, programmer has to explicitly use these instruction to fully benefit from them even though the compiler technologies provide a way to vectorize the code to utilize SIMD, it is usually limited by the structure of application code. These SIMD instructions became popular mainly due to heavy computational requirements of multimedia workloads due to the fact that multimedia algorithms can be inherently data parallel and set of data to be processed very often is large.

Another very important aspect of computing architecture is cache memory subsystem since CPUs are a lot more faster compared to the main memory. In CPUs even the cache memory is hierarchical within itself and each level is usually referred with a number signifying how distant it is to the processor. For example, L1 cache represents the cache that is closest therefore the fastest cache in the memory subsystem. The main reason why cache exists is that there is an important trade of between access latency of a memory cell and total capacity of the memory. Cache system in CPUs are transparent meaning that software running on a core has no information what so ever about the cache system or how it is handling the memory request of the processor. This transparency results in processor to implement caching algorithm and mechanisms as hardware solution which consumes large areas on the processor die.

To summarize, CPUs typically contain a number of large and power processor cores that includes highly sophisticated memory systems. All these features requires processor die area, moreover these advance features requires additional power which is an important restriction for processors. Thus, CPU architectures trade power consumption and area for high single-threaded performance. Even thought, the emphasis on individual core is huge, modern CPUs includes reasonable amount of cores. For instance, 4th generation (codename Haswell) Intel CPUs come with core amount ranging from 4 to 18[7].

2.2.2 GPU Architectures

In contrast to CPU architectures, modern GPUs take largely the opposite approach where they accommodate large number of small and less capable cores. These cores issue instructions in order and have very simple branch prediction logic. Therefore, GPUs dedicate much more die area to arithmetic logic units compared to CPUs where there is also significant area reserved for control logic considering same area dies as illustrated in Figure 2.5. Furthermore, they include specialized hardware circuitry to implement graphic application logic like rasterization operations and texture fetching. Similar to CPU architectures, graphic processors adopts SIMD like approaches to benefit from data parallel workload. In fact, since graphical workloads are hugely data parallel and principals of GPU architectures based on these workloads, whole architecture design is exist to exploit data parallel workload characteristics. Because of these reasons, GPU architectures considered a form of manycore architecture.

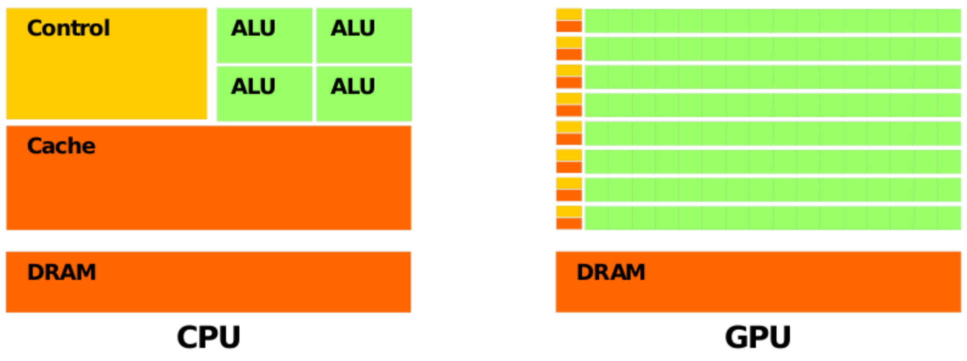


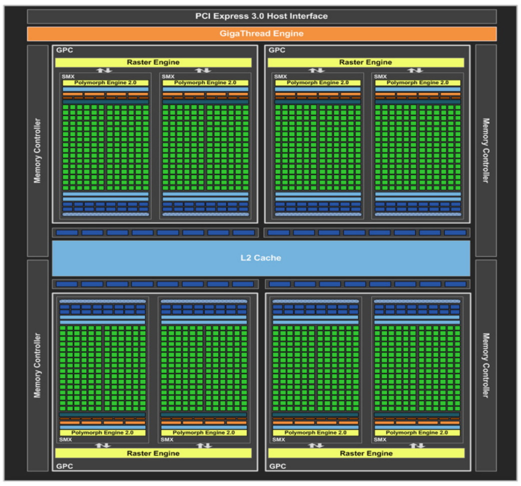
Figure 2.5: CPU vs GPU

Manycore architectures are multicore processors specifically designed for highly parallelized workloads. In case of GPUs, they contain many weak cores with a high latency to memory to handle the arithmetic operations. Despite the fact that CPU may also contain significant number of cores, performance of individual cores between CPU and GPU significantly differs. Furthermore, while in a CPU it can be convenient to use single core, due to the configuration of core on the hardware it may not even be possible to run single core alone.

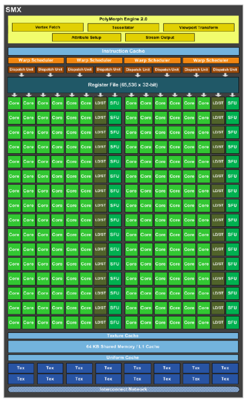
GPUs implement the manycore architectures in a specific way on the hardware. It groups the hardware cores into what is called compute units, and each computing units contains resources to be used by core executing. Each core inside a computing unit requires various resource. These resource can be number of registers, local cache and other potential platform specific resources. The limited number of resources are managed by the scheduler in order to distribute across running core. The local memory located inside each computing unit is a programmable cache that has much faster bandwidth compared to the main memory of the GPU.

Underneath the GPU architectures' seamless adoption of data parallelism there is the technique called SIMT. It is very similar to SIMD extension employed in CPUs where a densely packed data can be processed using the same instruction in one cycle. The difference in SIMT is that there are a group of threads which are issued the same instructions while having so called local resources like number of registers that are dedicated to each thread. Terminological name for this group of threads differs from vendor to vendor.

In Nvidia CUDA architecture this group of thread is named *warp* and AMD calls it *wavefront*. This type of computing is very susceptible to branches, because in the case of threads in the group take different paths



(a) Nvidia Kepler Architecture



(b) Compute Unit

Figure 2.6

of the branch and started to execute different instructions, GPU scheduler executes the threads that are in the same branch path together and then moves to the other path to execute the rest of the threads. Hence, the more there are conditional branches that the condition varies between the threads in a the same warp/wavefront, the more instructions are needed to be issued in multiple clock cycles. This phenomena is called thread divergence, and could hurt the performance unless it is taken into consideration.

The memory hierarchy and principals adopted by the memory configurations are also different compared to CPUs. Even though there are multiple levels of caches similar to CPU, the whole memory system is tuned for high throughput rather than latency. As a result of this, a single access to a memory cell of a single core requires hundreds of clock cycles. In order to maintain the throughput and saturate the memory bus, the scheduler tries to hide the memory transfer latency by switching to a different thread when threads required to access the memory.

SIMT technology plays a huge role for the high performance GPGPU. Because single instruction fetching is sufficient to run all the threads in a warp/wavefront, instruction fetching overhead is improved drastically. Moreover, assuming the memory access pattern of the instructions executing on the cores are adequate, the memory controller only issues single memory access for the whole thread group and data that is fetched is distributed to the threads. This is called coalesced memory access and addresses that are accessed by the threads must be proximity to each other in linear memory address space rather than accessing random locations. The exact number for how distant the access can be so that it is considered coalesced access is architecture depended.

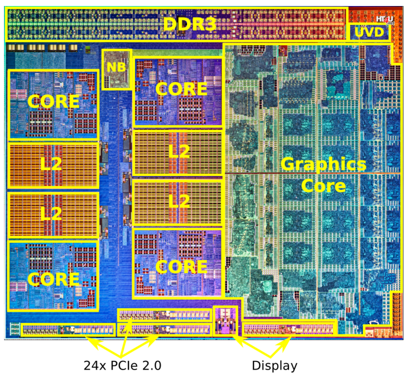
2.2.3 Heterogeneous Architectures

By dictionary, the meaning of the word *heterogeneous* is something composed of parts or elements that have diverse characteristics and widely dissimilar. Heterogeneous computer architectures are not differ from that definition. In the context of computer architecture, heterogeneity means that within the same platform, there are at least two different kinds of processors. The differences can be how the cores of the processor execute instructions, power and performance characteristic or can be configuration of memory systems. The definition is broad, as long as there are two or more architectures based on different principles at some point.

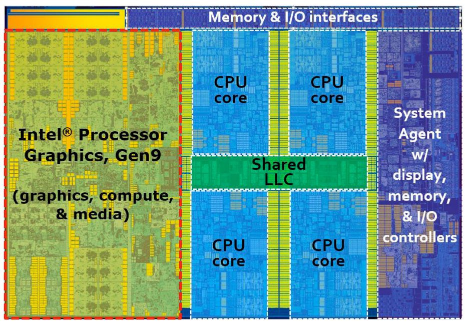
The platforms which have GPUs either discrete or iGPU, can be considered heterogeneous platforms since graphics processors and conventional processors have quite different considerations for computing architectures as described above. The presence of diverse programmable hardware on the same machine brings unique opportunities and approaches to solve a computation problem. Along with these opportunities, it becomes harder for the correct approach to be taken. Usually, ad-hoc solutions that are specific to platforms employed. These solutions are usually either inapplicable for other heterogeneous architectures or sensitive to the differences between architectures. Throughout this dissertation, we will focus on platforms where CPU and GPU reside together. However, most of the technique will be applicable to other heterogeneous platforms. Otherwise, it will be explicitly stated that the specific consideration may not be applicable.

One of the major aspects of the heterogeneous architectures is how the inter homogeneous processor communication has been implemented. For the architectures in consideration for this Thesis, there are two major examples. In order to benefit from heterogeneous workloads, hardware vendors like AMD and Intel developed platforms with CPU and GPU together as a SoC solutions. This kind of pairing and being in the same silicon package as depicted in Figure 2.7 resulted in both processor to share the main memory. Therefore, data sharing problem between the processors become a less serious issue. It is clear form Figure 2.7 that processor vendors are dedicate almost half of their die are to graphical processing and hardware acceleration units.

In contrast to integrated GPU solution, discrete GPU platforms requires separate communication bus that is generally PCI express bus to talk to main processor. Consequently, data transfers necessary for data that needs to be processed by GPU may pose problems depending on intensity of the computation. Because of this narrow bandwidth problem,



(a) Amd Llano die (2011)


Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

(b) Intel Skylake die (2015)

Figure 2.7

certain algorithms which have more data to transfers than to compute on discrete device may prove to be a futile effort to accelerate [4].

2.3 Related Works

Although OpenCL defines the execution of the application that is portable between the devices conforming the OpenCL standard, it does not guarantee the performance to be optimal. Especially, moving applications to different types of architectures like from CPU to GPU may result significant loss of performance, this is the reason why OpenCL is not considered performance portable.

Heterogeneous performance portability represents a challenging research issue. Since there are many approaches to address the issues of heterogeneity of hardware, choosing the right methods for the right problem proves to be a difficult problem, especially when the goal is not only application-independent but also platform-independent. There are numerous works that has been recently done, each one of them dealing with problems of application autotuning and problems of heterogeneous platforms from certain angles. The work of this dissertation is based on some of the work described below, and other scientific outputs are there either due to being inspirational to the Thesis ideas or because of their relevancy.

In Glinda framework presented by [11], a specific application is analyzed and used to drive new techniques to implement load balancing and autotuning. The application field was acoustic ray tracing from Aeronautics, and it may contain some imbalanced workload which depends on application specific parameters. The author argues that this application's case is genuinely suitable for the heterogeneous computing, since the imbalanced workloads fit naturally to heterogeneous platforms where tasks can be divided according to not only architecture characteristics of underlying hardware but also the tendencies of the separate tasks.

In the application which is evaluated by the author, not all the tasks may contain the same workload and whole goal of the framework is to detect this non uniform work distribution and understand where to cut data set in order to benefit from heterogeneous platform. The method employed is sampling the data space to detect *peaks* where data processing time is much more varied due to application specific reasons. However, this approach exposes sampling rate parameters that can effect the outcome of the detection. Even tough, it is not explored different applications may require different sampling rates. Hence, sampling rate of the data space may dependent on application behavior.

Later, task layer part of the framework's autotuning tries to balance the workload between CPU and GPU using iterative methods rather than analytically techniques without considering application dependent methods. Therefore, the situation where one of the devices does not have enough data while other is overwhelmed is avoided. In data layer of the autotuning, the author discusses how the work-group size thus, the number of work-groups effects the performance. The autotuner of the framework very simply just tries values that are power of 2. This is a good estimate since most of the hardware that supports OpenCL has compute units containing multiple of 2^n amount of resources such as register count and local memory size.

In this work, we will also try to address the problems of heterogeneous computing in Section 3.1, however, we will be looking at it from a different perspective with marginally changed problem.

There is a work done by Alok Prakash et al. [10] which shares much similar vision and goal with this dissertation compared to [11]. In this work, the authors explored how to partition the data to execute OpenCL kernel across CPU and GPU. An embedded platform called Exynos 5422 SoC has been used, moreover since the platform exposes not only a way to measure energy consumption but also adjusting voltage and frequency settings.

The platform is pretty unique in terms of heterogeneity and it is heterogeneous in two different ways, it contains two quad-core Arm processor where one of them is A7 which is more suitable for power efficiency while other one is A15 that is advertised as performance oriented architecture. Along side with CPUs, there is also embedded GPU on the same package.

The reason why this configuration is interesting is the authors classify the heterogeneity of this platform in to two separate terminology. They considered heterogeneity brought to table with two types of ARM cores as *performance heterogeneity*, while the CPU-GPU pairing is called *functional heterogeneity*. If we need to compare, the previous work was considering heavily on functional heterogeneity and rely on that to resolve the imbalanced workloads[11].

Unlike the test case used in [11], the benchmark that authors have chosen consists of uniform workloads rather than imbalanced workloads. In their work, it is clear that utilization of both CPU and GPU brings improvement over opting for one type of processor in terms of both execution time and energy consumption. This shows the fact that, even if the computation in question doesn't have any asymmetric subtasks, the additional processing power of any computing devices that is capable of data parallel computing is a plus when utilized properly. This fact makes heterogeneous platforms invaluable in the context of data parallel workloads.

A important remark about the work in [10] that is the techniques used for partitioning the data into two sets so that two different devices can processes these partitioned subsets. They devise an algebraic technique to decide splitting point shown in Algorithm 1. In the pseudo-code, *gw_size* is OpenCL global work size and *wg_size* is OpenCL work-group size.

Algorithm 1 Pseduo-code for data splitting [10]

```
1: $splittingPoint \leftarrow (gw\_size * split\_fraction)/wg\_size$
2: $globalWorkSizeCPU \leftarrow splittingPoint * wg\_size$
3: $offsetCPU \leftarrow 0$
4: $globalWorkSizeGPU \leftarrow (gw\_size/wg\_size - splittingPoint) * wg\_size$
5: $offsetGPU \leftarrow splittingPoint * wg\_size$
```

The main concerning point about this approach is the same work-group size has been used for both CPU and GPU. As a result of this, global work space will be divided into work-groups uniformly as represented in Figure 2.8. This may result in using sub optimal work-group size for one of the devices. Because very often different devices does not prefer the same work-group size mainly due to architectural differences. Therefore, this will be one of the points that we will try to address and discussed in Chapter 3.

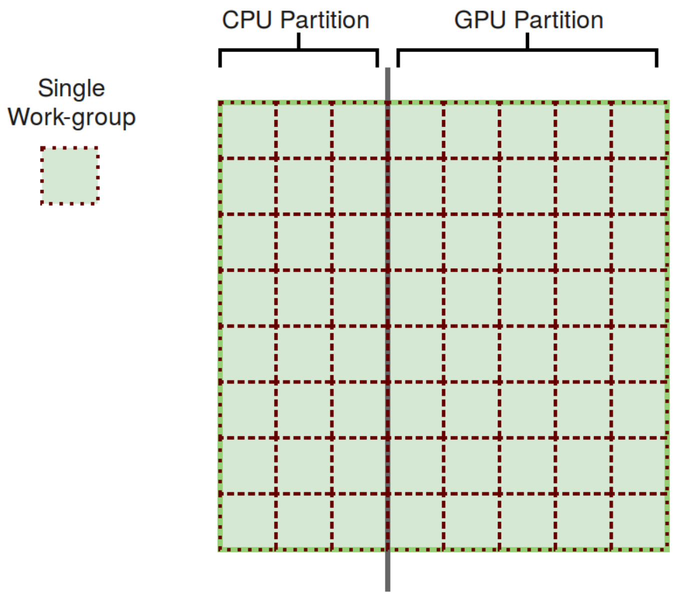


Figure 2.8: Common work-group size, partitioning

Cummins et al. [2] discuss the same issue in their paper as mentioned above about the work-group sizes. In the work, they presented a machine learning workflow to predict the appropriate work-group size for a given kernel and architecture using classifiers and regression. The problem of feasible work-group sizes is addressed by using OpenCL querying framework and setting the maximum allowed work-group size of the architecture. The trained system is expected to return a close to optimal The feature set is defined in terms of architectural and kernel code statistics gathered using LLVM compiler. Since both architecture and OpenCL kernel is included for the feature set, the outcome of the prediction system will find a suitable work-group for a given platform and application.

As a result of [2], the work-group sizes tuned by the trained machine performed well for prediction performant sizes compared to a baseline except with Logistic Regression there are some over-fitting to the features. This is a good example of how autotuning assisted with machine intelligence can improve performance. The whole tuning phase in the work was statically done, therefore there is no run-time overhead besides acquiring the right work-group size using trained model which is minuscule. While the experiments are executed on wide range of platforms, the work focused single class of applications. Thus, it is hard to say it is completely application-independent study, especially comparing to [10] where Polybench is used as a more generic benchmark.

Another approach embraced by autotuning frameworks is using a custom Domain Specific Language (DSL) either embedded inside the original program code or separate file that is fed into the tuning engine. OrCL[1], uses the embedded DSL concept and is an OpenCL code generator backend implemented within Orio framework. It has its own syntax to specify tunable parameters for a given computation that has loop-based implementation. Using these parameters that are defined by annotating the source code, OrCL generates variant of the original source code as OpenCL kernels and then assesses newly generated kernel functions. Furthermore, framework generates the host side code skeleton in order to be able to asses the generated kernels, again employing the information from OrCL's annotations. In this way, for each OpenCL device, it is possible to have a modified version of the original source code that is much more suitable to the specific hardware.

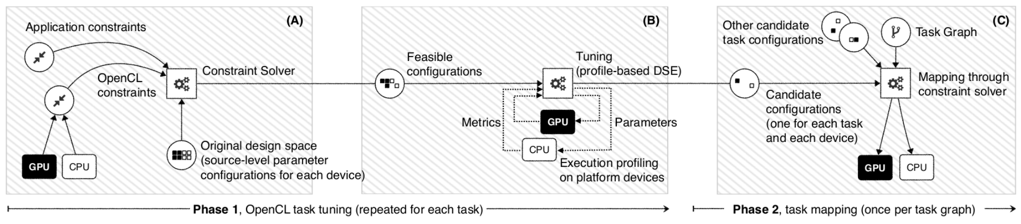

Application constraints
(A)
OpenCL constraints
Constraint Solver
GPU
CPU
Original design space (source-level parameter configurations for each device)
(B)
Feasible configurations
Tuning (profile-based DSE)
Metrics
GPU
Parameters
CPU
Execution profiling on platform devices
(C)
Other candidate task configurations
Task Graph
Mapping through constraint solver
Candidate configurations (one for each task and each device)
GPU
CPU
Phase 1, OpenCL task tuning (repeated for each task)
Phase 2, task mapping (once per task graph)

Figure 2.9: The DSE workflow for autotuning OpenCL application

While previously discussed works focuses on single OpenCL kernel as a task, Edoardo Paone et al. [9] investigate *inter-task parallelism* on heterogeneous devices in their work. The main idea is applications that are composed of multiple OpenCL kernels can be represented as a task graph where each task represent an OpenCL kernel function. Moreover, the work uses *feed-forward cut-sets* to create pipeline stages, then maps the different stages to separate OpenCL devices in order to benefit from heterogeneity of the platform.

Similar to previous approaches tuning of kernels is a crucial about getting performance since sub optimal configuration can effect performance greatly. In [9], profiling-based DSE is used after limiting the exploration space using constraint programming (as shown Phase 1A in Figure 2.9). Since the parameters and constraints are necessary inputs, it is expected from programmer to come up with reasonable constraints therefore making whole system a semiautomatic workflow. As reported by the authors, the constraint solving helps to reduce total design space to a manageable size. Thus, the whole tuning time has been shortened significantly .

All these previous works inspired and lead to the proposed methodology of this Thesis. The Thesis contributions consist of partial automation of pruning processes described in [9] as an improvement and secondly, introducing a new approach to split the computation work of a single task to different OpenCL devices in the presence of heterogeneous platform. Constrast to the work in [9] where multiple indivisible tasks are mapped to different devices, in this work we will be considering spliting a single task.

Chapter 3

Proposed Approach

In this chapter, technical details about the methods and approaches that are taken regarding to problems laid out in Chapter 2 will be introduced.

OpenCL applications are generally performance-oriented. At the end, what is desired is mostly some degree of acceleration for the application's execution time. OpenCL allows to achieve speed up by parallelising the application in a specific fashion where developer has to make some decisions on the parameters that are provided by OpenCL. Even though, the developer knows the best parameters and configurations for their application and target platform, it may not be clearly defined in an OpenCL project or it may very well be subject to change in the near future. Therefore, autotuning techniques are common to address the problem of finding either optimal or close to optimal configuration parameters. These parameters can belong to either OpenCL standard or application domain.

Full automatic tuning is pretty hard problem due to the lack of knowledge of the parameters especially if parameters are deeply connected to application domain, therefore the tuning framework needs ad-hoc knowledge on the topic. Thus, semi-automatic tuning of application with limited input from the domain expert or developer is the preferred approach.

3.1 Methodology

3.1.1 Kernel Autotuning

The procedure for autotuning of an OpenCL application in order to get optimum performance without concerns of underlying architecture of the platform requires a set of parameters that define characteristics of the machine. In the case of OpenCL, these platform specific parameters are stated by the OpenCL standard itself. Other than the platform parameters, there are also parameters that are defined by the application domain. But due to the number of parameters design exploration space can be huge and become impractical to search optimal parameter configuration using brute force methods. Therefore autotuning frameworks usually employ machine learning methods to deal with the large exploration spaces such as presented in [2].

However, most of the configurations are not feasible in the sense that the kernel may not even launch or may fail during execution, due to implausible configuration parameters like work-group sizes with local memory usage which exceeds maximum local memory available. These failed attempts of kernel launches do not provide any information about the sample that has been taken from design space. Hence effort and time are wasted on these infeasible configurations.

In the work of [9], the solution to large exploration space problem has been given by the user provided parameters and constraints related to both OpenCL platform and application domain. While OpenCL parameters come from the definitions included in the standard itself, application specific parameters are unique to each case of application.

This creates a problem in terms of building a workflow that is as automatic as possible since some inputs expected from the user. Furthermore, very often the application specific parameters are tightly related to OpenCL platform parameters. Thus, complicating the pruning process for an user of workflow (as shown in Figure 2.9).

An important point is even though OpenCL parameters come from platforms that application supposed to run on, the constraints regarding to OpenCL standard are fixed. The assumptions and principles of behaviors established by standards apply without differentiating underlying hardware architectures that are present. In this way, standard can enforce its rules over all the conformant products. Therefore these constraints are redundant for user to enter. Even worse, user may enter values incorrectly, since it is well-known that reliability of humans are not the best.

Constraint solver in the Phase 1, Block A of 2.9 uses the following OpenCL platform parameters:

- maximum work-group size for each dimensions of global work size.
- maximum total number of work-group a NDRange kernel launch can contain.
- number of compute units on the OpenCL device.
- local memory size of the device.

As an improvement upon previous work done in [9], just before the constraint solver stage we added an OpenCL platform parameter extractor which collects the parameters specified above. Then, to provide a common set of constraints that matches the rules established by the OpenCL standard. Hence, application developer needs to provide constraints that are only related to application. The common constraints along with their reasoning of inclusion are:

- *Each dimension of the global work size must be divisible by the corresponding work-group size.* This restriction guarantees that number of work-groups are integer and no work-group is covering partial of global work grid.

$$\begin{aligned} global_x \bmod workgroup_x &== 0 \\ global_y \bmod workgroup_y &== 0 \\ global_z \bmod workgroup_z &== 0 \end{aligned} \tag{3.1}$$

- *Total work-group size must be less than or equal to maximum work-group size.* Each OpenCL platform has a total work-group size limit which is independent of maximum work-group size of any dimension.

$$\begin{gathered} total\_wg\_size = workgroup_x * workgroup_y * workgroup_z \\ total\_wg\_size <= max\_total\_wg\_size \end{gathered} \tag{3.2}$$

- *Number of work-group should be equal or greater than number of compute units.* Having fewer number of work-group than number of compute units doesn't break the functionality of the kernel, however device will be under utilized due to not enough work-groups to be mapped to compute units as shown in Figure 3.1.

$$\begin{aligned} global_x/workgroup_x+ \\ global_y/workgroup_y+ \\ global_z/workgroup_z >= num\_compute\_units \end{aligned} \tag{3.3}$$

- *Local memory usage of one work-group must be less than local memory size.* Otherwise, work-group will certainly fail with OpenCL error named CL_OUT_OF_RESOURCE, due to not being able to allocate enough local memory for the kernel function.

$$local\_mem\_usage <= local\_mem\_size \tag{3.4}$$

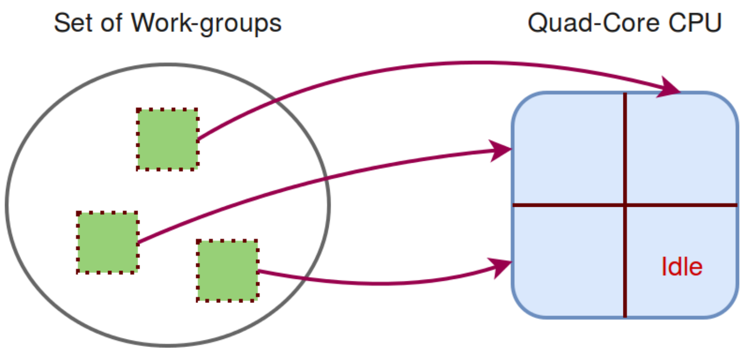


Figure 3.1: Mapping work-group to compute units
Lack of work-groups cause ineffective mapping to CPU with 4 compute units, due to idling computing units.

For application specific parameters, it is possible to include C-style definitions inside OpenCL kernel code and expose those definitions as parameters in MiniZinc either decision variables or plain parameters. This allows application programmer to introduce parameters unique to the application domain. Moreover, using constraint it is possible to form a relationship between newly introduced parameters and OpenCL platform parameters.

After the pruning phase of the exploration space, the rest of the configuration points which satisfy all the constraints are executed in a brute force fashion. In order to reduce the interference of other present processes in the operating system, the same configuration has been run multiple times and average of the best runtimes is calculated. Results of the run stored in a table with their configuration.

3.1.2 Kernel Data Splitting for Heterogeneity

While tuning and running OpenCL kernels on OpenCL devices are fine, multiple computation architecture is present most of today's computing platforms. Therefore, being able to handle this heterogeneity is crucial for achieving higher performance systems. Therefore, like many previous works [11][10][9], we investigate how to exploit heterogeneity. In contrast to work done by [9], We will not focus on mapping of separate kernels to different devices. Rather than that, the main focus of the work is splitting huge OpenCL tasks into smaller chunks then map those partial computations to different OpenCL devices running them simultaneously.

There are a number of problems to be addressed in order to achieve this kind of splitting of OpenCL kernels to devices.

1. Finding the right split point for an heterogeneous platform requires performance knowledge a prior.

2. After splitting, the work-group sizes can be not only inappropriate for performance but also ill-advised, therefore kernel execution may fail in one or more devices.

Split Point Decision

In order to find split point, we used the exploration information collected from tuning phase. The best runtime candidates are chosen according to their execution times.

$$exe\_speed_i = 1/exe\_time_i \quad i = 0, \ldots, N-1s \tag{3.5}$$

First we convert the execution times into speed values with the Equation 3.5 where N is the number of devices present on the platform. In this way it is easier to reason about the performance variation across different devices since the greater the speed value is, the higher performance of the device.

Then these individual device speeds for the OpenCL kernel are used to calculate *split factor* for each device (equation 3.6). These split factors define how much of the computation will take place on each OpenCL device. Therefore, sum of all split factors must reach to 1.0, otherwise, there would be residue computation that is missing for the output of the task.

$$split\_factor_i = exe\_speed_i/(\sum_{j=0}^{N-1} exe\_speed_j) \quad i = 0, \ldots, N-1s \tag{3.6}$$

It is easy to observe that $\sum_{i=0}^{N-1} split\_factor_i = 1.0$ is satisfied by substituting $split\_factor_i$ by the equation 3.6.

Work-group Size Adjustments

Because split factor is computed using total number of data to be split, it doesn't consider any dimensionality of the global work size. However, this doesn't create problem as long as splitting cut has been applied to each dimension separately as different option and measured the execution time with each dimensional cut (Figure 3.2).

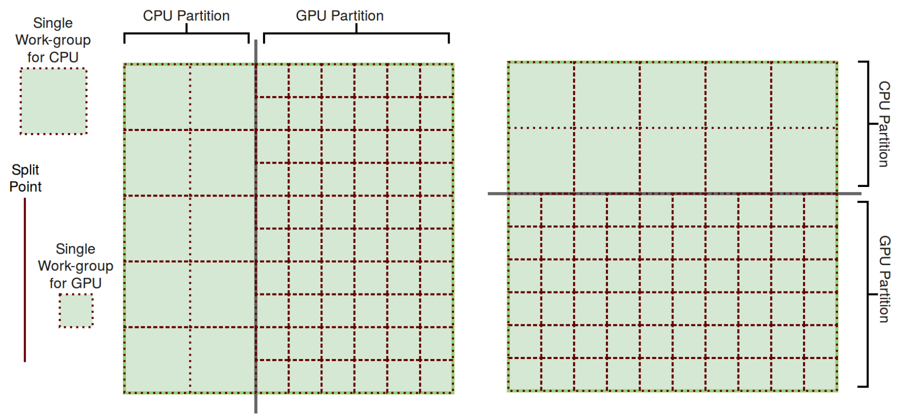


Figure 3.2: Global work space splitting illustration on two different dimensions

While split factor is calculated from performance measurements to give good hint how to load balance between different devices in heterogeneous environments, dividing global work size using split factor results in partitions with fractional work sizes which is totally impractical to be used in OpenCL's context.

For example, let's consider global work size is 512 work items and split factors are 0.4 and 0.6 for two devices respectively in an CPU-GPU heterogeneous platform. Using this scenario, CPU needs to handle 204.8 work-items and GPU has a share of 307.2 work-items.

This problem is only one-side of the coin, the more important problem is keeping the optimum work-group sizes found by autotuner for each device. Moreover, since maximum performance work-group sizes can be contrasting for different devices on the same platform, it is not possible to use a single '*optimal*' size as used in [10].

Addressing this problem requires adjusting split factors while taking into account the optimal work-group sizes. But the more deviation from the original splif factors there is, the further away splitting is from optimal load balancing. Therefore it is desired to make minimum adjustments.

$$((workgroup_i \bmod workgroup_j) * (workgroup_j \bmod workgroup_i)) = 0 \quad \forall i, j \in [0, \ldots, N-1] \quad \text{where i and j are integers} \tag{3.7}$$

Additionally, if work-group sizes are not multiple of each other, hence failing to hold following equation 3.7, would results incompatible work-group sizes between different OpenCL devices. In order to deal with this problem, we devise a method allowing devices to compute work-items redundantly as minimum as possible, while preserving their desired work-group sizes. This is achieved by overlapping the split partitions of each device.

$$G = wg_1 * \gamma_1 + wg_2 * \gamma_2 + \cdots + wg_{N-1} * \gamma_{N-1} \tag{3.8}$$

Equation 3.8 shows how the global work size decomposes such that multiple devices are contributing to computation. The G in the equation is global work size and γ is number of work-groups. Each $wg_i * \gamma_i$ component represents work size mapped to a certain device. While the equation represents global work size, it only considers one dimension of the global sizes and work-group sizes. As mentioned before, all the splitting operation is applied for each dimension separately in case of multi-dimensional work space, then multiple heterogeneous configurations created for each dimensional split to be executed and measured. Therefore, all the equations are based on one dimensional split cut.

$$wg_i * \gamma_i = G * S_i \tag{3.9a}$$

$$\gamma_i = \frac{G * S_i}{wg_i} \tag{3.9b}$$

$$\hat{\gamma_i} = \lfloor \gamma_i \rfloor \tag{3.9c}$$

$$\hat{S_i} = \frac{wg_i * \hat{\gamma_i}}{G}, \quad \text{for} \quad i = 0, \ldots, N-1 \tag{3.9d}$$

$$S_R = \sum_{i=0}^{N-1} S_i - \sum_{j=0}^{N-1} \hat{S_j}$$

where $\hat{S_i}$ and $\hat{\gamma_i}$ are reduced split factor and number of groups and S_R is residue split factor

(3.9e)

Methodologically, number of work-groups need for each device is calculated using devices' optimal work-group sizes (Eq. 3.9b). It is important to note that at this stage numbers of work-groups are real values so they are not fit to be used by OpenCL framework. In order to deal with this issue, numbers of work-groups are reduced to an integer number (Eq. 3.9c). Then, corresponding reduced split factors recalculated in order to find the residue (Eq. 3.9d and Eq. 3.9e).

$$\tilde{\gamma}_i = \lceil \frac{G * S_R}{wg_i} \rceil \tag{3.10a}$$

$$\tilde{S}_i = \frac{wg_i * \tilde{\gamma}_i}{G}, \quad \text{for} \quad i = 0, \ldots, N-1 \tag{3.10b}$$

This S_R, residue split factor, represents how much work has been left after adjusting for optimal work-groups. And since we want to this residue work to be done with minimum redundant work effort, it is required to choose the device that is best suited. To be able to assess which device is more fitting our situation, equation 3.10b calculates for each device how much work necessary to finish the residue. Then we choose the device with minimum work which is represented with $\tilde{S}_i$. Next, we add the residue fraction of the workload to the chosen devices reduced split factor, ending up with the device's final split factor. For all the other devices, reduced split factors are used.

Using this method, it is possible to preserve desired work-group sizes while adjusting split factor minimally. We achieve this first reducing the number of work-groups conservatively per device, then choosing appropriate device for the residue work to be computed on.

Additional benefit of this technique is that if equation 3.7 holds, there will be no overlap between the works of platform devices. Hence, no redundant and wasted computation will be present. This type of outcome is observed because when work-group sizes are multiple of each other hence reduction on the larger size can perfectly be accommodated by smaller work-group size.

3.2 Novelty in the Approach

The previous works mentioned either focuses on autotuning extensively or try to exploit heterogeneity except [9] where the authors suggest a full workflow where efficient autotuning and heterogeneous computing come together. While their DSE workflow takes heterogeneous platforms into consideration, a task graph where multiple OpenCL kernels that are independent from each other is required to take advantage of heterogeneity. This is due to mapping of computations to the platform is on the task level in terms of granularity.

In this work, we take it one step further and introduce a technique that will exploit multiple device platforms in the case of single task. Additionally, we utilize the valuable information explored in the previous autotuning phases of the workflow.

3.3 Development

All the management of the different parts of the workflow (Figure 3.3) implemented by python 3.4.

3.3.1 Workflow, Pruning Phase

The first part of the workflow (Pruning phase of 3.3), has multiple components that needed to be realized. *OpenCL Standard Constraints* which has been discussed in Section 3.1.1 is implemented as MiniZinc constraint model. *Application Related* parameters and constraints are expected from developer as MiniZinc file that contains both decision data and constraint model. Then, using *#include* MiniZinc features, OpenCL constraints and application parameters and constraints are merged together to form the final constraint model needed for MiniZinc constraint solver.

The only information that is absent for the solver is the missing platform parameters needed for the platform constraints in the constraint model. And those parameters are gathered using OpenCL Querying API provided by the standard. For each device present on the system, *clGetDeviceInfo* function is called to get desired device information to be used as parameter. Later, these parameters processed to be compatible with MiniZinc language, thus they can be used as decision variables for constraint model.

MiniZinc solver has multiple backends that is suited and optimized for different constraint solving problems. In this work we are using *mzn-g12fd*, it is the backend for finite domain constraint solving. This backend is the most suitable for us because the configuration space is composed of finite numbers which define OpenCL parameters.

In constraint model, *solve satisfy* is used in order to find configuration points that satisfy all the constraints. Then solver is launched via command-line with *–all-solutions* argument. This argument flag forces MiniZinc to return all solutions that satisfies constraints, otherwise MiniZinc returns the first solution that it finds.

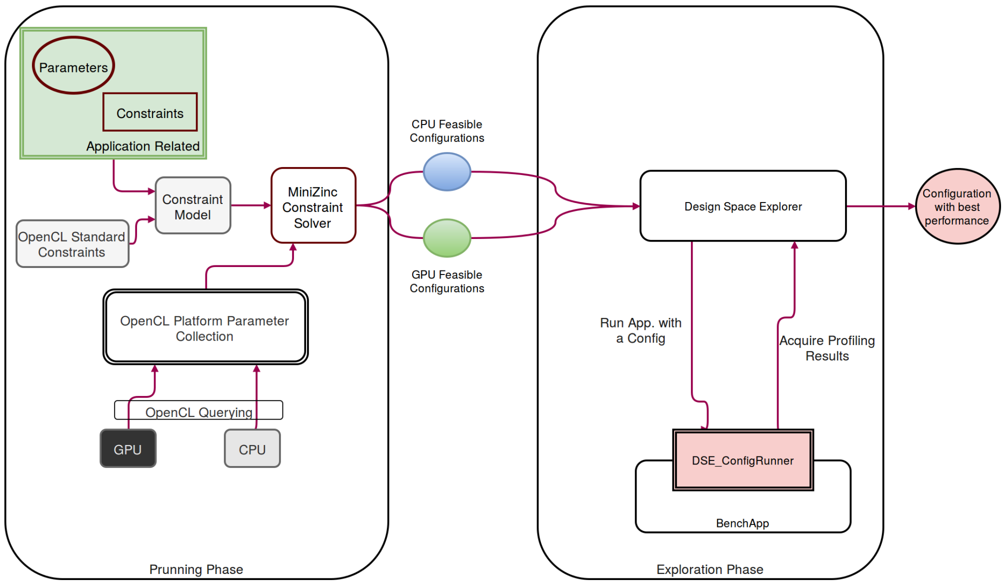

Parameters
Constraints
Application Related
Constraint Model
MiniZinc Constraint Solver
OpenCL Standard Constraints
OpenCL Platform Parameter Collection
OpenCL Querying
GPU
CPU
Prunning Phase
CPU Feasible Configurations
GPU Feasible Configurations
Design Space Explorer
Configuration with best performance
Run App. with a Config
Acquire Profiling Results
DSE_ConfigRunner
BenchApp
Exploration Phase

Figure 3.3: Autotuning and heterogeneous run workflow

3.3.2 Workflow, Exploration Phase

OpenCL provides a host API that is not inherently taking heterogeneous environments into consideration. But standard API provides enough flexibility to implement splitting methods on top of OpenCL host API. Heterogeneous implementations of application may requires ad-hoc solutions specific to each application, however the information needed for achieving heterogeneity like split factors, global work size, desired work-group sizes can be generalized and supplied to the benchmark applications.

A small library that will allow to communicate with the benchmark application developed using C++. The library is show in the Figure 3.3 and named *DSE_ ConfigRunner*. Its main purpose is to reading the configuration file, acquire necessary OpenCL platforms and devices and compile required OpenCL kernels. In this way all the boilerplate code necessary for OpenCL is abstracted away.

To achieve heterogeneity each case is hand modified to use configuration. *clEnqueueNDRange* with offset parameter is used to shift the second or later devices if they are present. offset value is calculated by accommodating the global work sizes of the previously scheduled devices using only the dimension the split operation applied. In a similar manner, *clReadBufferRect* API has been used to copy a region only where the computation output written to host memory.

The configuration file which is a json file, is generated by the python code which handles the design space explorations. Explorer written in python, picks a configuration and produces file in json format that will be supplied to the benchmark application when executed from command-line using argument as *auto_ cl=<config_file_ name>*. An example of this configuration file is present in Appendix B.

Design space explorer first runs benchmarks as non-heterogeneously, then uses the generated configuration file, performance characteristics pair list to decide on the best possible splitting described in Methodology Section with equations 3.5 and 3.6. In order to explore possibility of performance gain on splitting different dimensions, in the explorer starting from the first dimension splitting is applied all the dimensions present for the OpenCL kernel. Later, all the results corresponding different dimensional splits are compared and chosen based on execution time of heterogeneous executions.

The theoretical best performance for heterogeneous executions is also calculated using the split factors and best autotuned execution speeds. These values supplied to the function (Algorithm 2). The function scales all the execution times with respect split factors and takes the maximum

Algorithm 2 Calculates the best theoretical performance estimate

```
1: function EstimateHeteroPerf(splitFactors, exeTimes)
2:     devicePerfs ← empty list
3:     $i \leftarrow 0$
4:     while $i <$ number of devices present do
5:         devicePerfs[i] ← splitFactors[i] * exeTimes[i]
6:     return MAX(devicePerfs)
```

amount assuming all the devices running concurrently in a perfect way. Split factors used here is before adjusting for work-group sizes. Therefore, we can have numerical values to compare our results against a perfectly scaled scenario.

For the best output configuration as shown in Figure 3.3, best results from non-heterogeneous runs and heterogeneous execution on all of the dimensions are compared and configuration with the minimum execution time is chosen. The reason why not only the heterogeneous runs are included is not all devices can run the OpenCL kernels asynchronously. Therefore, it is possible that a synchronization happens due to device limitations leading to launching kernels sequentially. This situation is limited to very old generation devices or the lack of support from OpenCL runtime even though the standard enforces asynchronous behaviour.

Chapter 4

Experiment Results

In this chapter, experimental setup which the proposed methodology has been implemented and assessed, will be explained in detail. In Section 4.1, first the hardware that is used for experiments and measuring will be examined, then the case studies implemented will be explained. And lastly, the experimental results where the proposed techniques applied on the case studies will be introduced and discussed in Section 4.2.

4.1 Experimental Setup

4.1.1 Target Platform

Target case studies that are explained in Section 4.1.2 have been run on two different platforms. Both of the platforms consist of a CPU and a GPU, therefore they are considered heterogeneous platforms.

The followings are the specifications of the platforms:

1. Platform 1 (or PLT1) is a workstation machine with: Hardware:

    - CPU: Intel(R) Xeon(R) CPU E5-1607 (Quad-core @ 3.0Ghz)
    - GPU: Nvidia NVS 300 (16 CUDA cores)

    OpenCL driver versions:

    - CPU: Intel OpenCL driver for Ubuntu 1.2.0.25
    - GPU: Nvidia Linux driver 340.96

2. Platform 2 (or PLT2) is a consumer grade laptop with: Hardware:

    - CPU: Intel(R) Core(TM) i7-2630QM (Quad-core @ 2.0Ghz)
    - GPU: Nvidia GeForce GT 550M (96 CUDA cores)

    OpenCL driver versions:

    - CPU: Intel OpenCL driver for Ubuntu 1.2.0.23
    - GPU: Nvidia Linux driver 304.131

On both platforms, Ubuntu (Linux) 14.04 has been used as operating system. On Platform 2, Intel CPU has an iGPU which doesn't support OpenCL standard. Thus, it was not possible to use that for experimental runs. Another problem is since Platform 2 has two different GPUs, the operating system was getting confused about which GPU to actively use. If the integrated GPU is activated, no OpenCL runtime environment is found for a GPU. In order to solve this issue, we installed on Platform 2 an open source project called Bumblebee. This provided us a command name *optirun* which switches the Nvidia GPU on. Therefore, we had to use this command for every benchmark run on Platform 2.

Another big difference between the platforms is CPU in Platform 2 has

8 logical cores while Platform 1 has 4 logical cores. This is due to CPU in Platform 2 has Intel Hyper-Threading support. In the end, OpenCL runtime sees Platform 1 (CPU) as a 4 compute units device and Platform 2 (CPU) as a 8 compute units device.

4.1.2 Target Case Studies

Convolution

Convolution is a mathematical operation on two functions such that it expresses the amount of overlap of one function as it is shifted over another function. It is widely used in image and signal processing, differential equations, computer vision and statistics. Especially in image and signal processing, discrete form of this operation has been popular due to the ease of implementation on digital systems. The discrete form of the mathematical operation is usually defined like Equation 4.1 and in many fields the two functions are usually considered signal and mask respectively (f and g in Eq. 4.1).

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m] \tag{4.1}$$

If we examine the Equation 4.1, it is possible to see that right hand side of the equation needs to be calculated for each discrete point of $(f*g)$ independently. This property of convolution is crucial because it allows to be computed using data parallelism techniques naturally. Therefore, it is very suitable for OpenCL implementation, too.

In Figure 4.1, a simple OpenCL kernel implementation is shown. This

```
__kernel void CONV( __global float* signal,
                    __constant const float* mask,
                    __global float* outSignal,
                    int maskLength, int signalLength)
{
  int idx = get_global_id(0);
  float out = 0;
  int start_point = idx - maskLength/2;
  for(int i = 0; i < maskLength; ++i)
  {
    int mask_index = start_point + i;
    if(mask_index >= 0 && mask_index < signalLength)
    {
      out += signal[mask_index] * mask[i];
    }
  }
  outSignal[idx] = out;
}
```

Figure 4.1: Basic OpenCL convolution implementation

kernel definition is executed for each work-item in the global work grid. The only difference between separate kernel instances is the outcome of *get_global_id*. In this way, each work-item accesses different region of the memory buffer. The address of the memory buffers are *signal, mask, outSignal* and they are supplied as the arguments of the kernel function. In this particular implementation, there is no exploitation of local memory to take advantage of temporal locality.

One way of taking advantage of local memory is sharing some data needed for computation of a work-item in the same work-group. In Figure 4.2, executions of two work-items are depicted. The buffer shown with upper green array is the input buffer, the other array is the output buffer, while the *C1-C5* are the mask coefficients. It is clear that two consecutive work-items share some overlapped region of the input buffer, therefore it is meaningful to have allocated local memory.

Assuming work-group consists of only two work-items as illustrated in

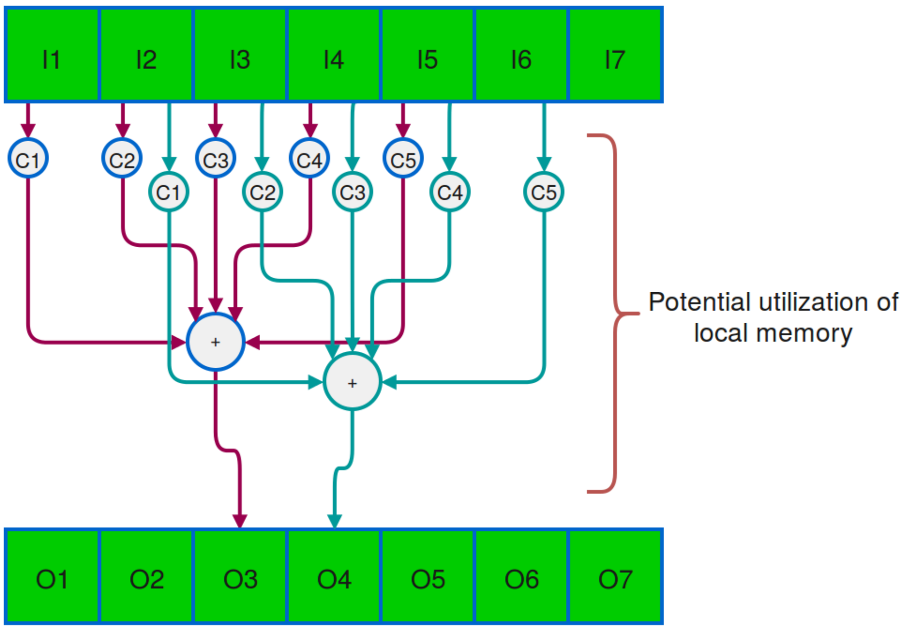


Figure 4.2: Execution of two work-items

Figure 4.2, the required amount of local memory is equal to 6 input data size since we work-items' access pattern include *I1-I6*. In order to utilize local memory, it is needed to be explicitly assigned values by work-items, therefore the first work-item will load *I1-I3* while second one will load *I4-I6*. This leads to certain local memory usage depending on mask size and work-group size which can be formalized as:

$$local\_memory\_usage = (workgroup\_size + mask\_length - 1)$$

This is shown in Figure 4.3, color of the arrows differentiate separate work-items' memory accesses. In this way, *I2-I5* is shared between two work-items and needed only one global memory access rather than two. This may not be significant performance improvement with just two work-items due to number of access to local memory not being few enough compared to global memory accesses. However, with the increased number

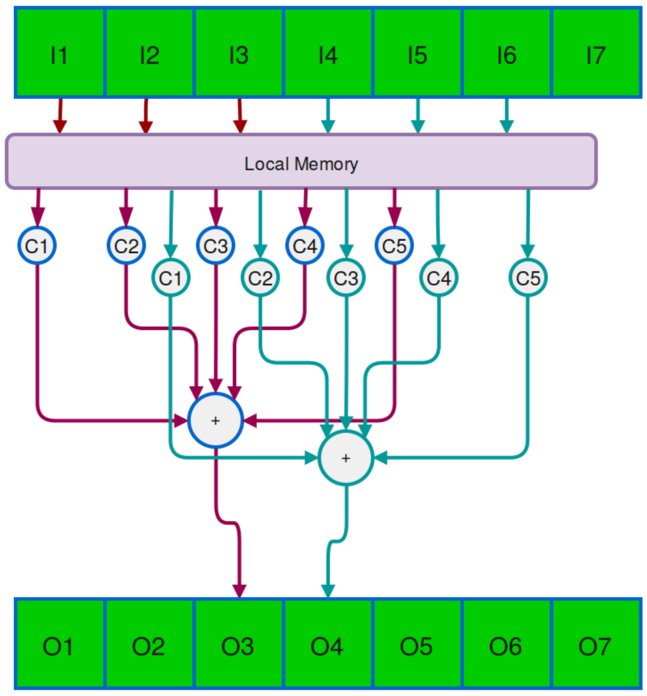


Figure 4.3: Execution of two work-items using local memory

of work-items in a work-group, the usage of local memory saves reasonable amount of global memory bandwidth.

Matrix Multiplication

Matrix multiplication is the other case study that is used in this Thesis to verify the methodology described in Chapter 3.1. It is one of the most common operation in linear algebra and mathematics and engineering that are using linear algebra. The operation has two matrices as inputs and an output matrix where the elements of the output matrix are dot product between rows of the first input matrix and columns of the seconds input matrix. The simplest implementation is given in Algorithm 3.

In the OpenCL implementation of the Algorithm 3, each output el-

Algorithm 3 Simple Matrix Multiplication

```
function MatrixMult(A, B)
    $nRows \leftarrow$ height of A
    $nCols \leftarrow$ width of B
    $dotWidth \leftarrow$ width of A
    $C \leftarrow$ empty matrix nRows x nCols
    for i from 1 to nRows do
        for j from 1 to nCols do
            $sum \leftarrow 0$
            for k from 1 to dotWidth do
                $sum \leftarrow sum + (A_{ik} \times B_{kj})$
            $C_{ij} \leftarrow sum$
    return $C$
```

ement of C matrix is computed by a single work-item. Since matrices have columns and rows, our global work grid is two dimensional as well as work-group sizes. This makes a work-group to calculate a rectangular area of output matrix. In this work, we will focus on square matrix multiplication because they are both common type of matrix multiplication and easier to express and analyze. Therefore, in Algorithm 3, *nRows*, *nCols* and *dotWidth* are equal to each other.

In Figure 4.4, it is possible to see a small example of how matrix multiplication accesses memory. For each single output element of C, one row of A and one column of B required to be accessed. For 4x4 matrix this does not create any problem, since whole matrix can be stored inside the last level cache of computing units easily. However, once the dimensions of matrix increases and it starts to be a problem, especially for matrix B.

Because the memory access pattern is not contiguous, it results in a lot of cache misses.

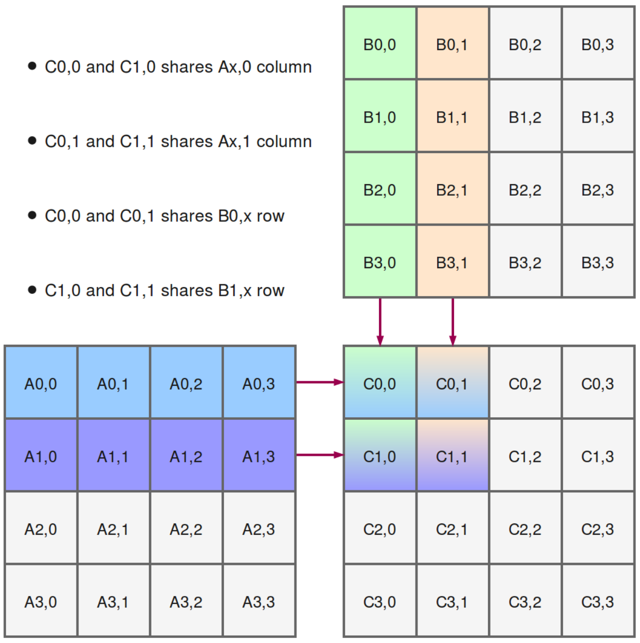


Figure 4.4: Matrix multiplication of 4x4 square matrices

The solution to this problem is the well-known tiled matrix multiplication. It is observable from Figure 4.4 that for output elements on the same row share the same row of A for computation, for column this holds trues as they share the same column n of B. This kind of characteristic can be exploited using local memory and grouping output elements in a way that output elements can share data that has been accessed by previous elements rather than go for the global memory access.

As an example, let's consider the situation in Figure 4.4 and assume that 4x4 input matrices divided into 2x2 tiles hence, 4 tiles per matrix. For each tile we will have one work-group with the same configuration. At the end, there will be 4 work-groups with 2x2 work-items each covering the whole output space of the matrix C. Each work-group will load 2x2 elements from A and B, then do partial computation of its outputs. Then slide the tile to load subsequent tile. The movement of tile is from left to right for A while for B it is top to bottom. After that, work-group will sum the partial computations. Since we store two 2x2 tiles for each work-group, local memory usage in terms of number of elements can be calculated with

$$local\_memory\_usage = 2 \times workgroup\_size_x \times workgroup\_size_y$$

Step by step execution example for the first work-group is detailed below:

1. $C_{i,j} = 0$, prepare output elements for partial results accumulation.

2. Load *A0,0 A0,1 A1,0 A1,1* and *B0,0 B0,1 B1,0 B1,1* from global memory to local memory.

3. Compute partial results per element $C_{i,j} = C_{i,j} + \sum_{k=0}^{\text{tile\_size}} (A_{i,k} * B_{k,j})$ using local memory address space.

4. Move the tile and load *A0,2 A0,3 A1,2 A1,3* and *B2,0 B2,1 B3,0 B3,1* from global memory to local memory. Then do the step 3 again.

The tiled version of the matrix multiplication is used as a case study because local memory usage scenarios are more interesting for our proposed methodology since local memory size of the OpenCL device is a limiting factor on how large work-group can size be.

4.2 Experimental Results & Discussion

4.2.1 Autotuning Space Pruning

In autotuning frameworks tuning time is very often required to be fast even though the application is tuned at design-time rather than runtime. Therefore design space exploration techniques are commonly employed to reduced the time taken to tune the application. In this work, design space is reduced before exploration take places.

In order to understand the impact, it is beneficial to understand the size of the original design space. Since our main contribution in for pruning phase in the workflow (Figure 3.3) comes from automatically collection OpenCL parameters, it is important to discuss the impact of OpenCL parameters to the design space. The main parameter knob presents in the OpenCL standard is work-group size. The range of values that work-group size can take depends on specific OpenCL device. For our target platform described in Section 4.1.1 the work-group size limitations are as follows:

	PLT1		PLT2	
	CPU	GPU	CPU	GPU
Dim0	8192	512	8192	1024
Dim1	8192	512	8192	1024
Dim2	8192	64	8192	64

Table 4.1: Work-group size limits per OpenCL devices

The numbers in Table 4.1 are gathered from OpenCL device querying using *clGetDeviceInfo* API. An important aspect of OpenCL is work-groups that can form multi-dimensional (up to three) groups of work-items. This additional parameters lead to exponential increase of the design space even if only work-group sizes are considered. Table 4.2 show the full design space without considering the proposed techniques.

	PLT1	PLT2
CPU	2^{39}	2^{39}
GPU	2^{24}	2^{27}

Table 4.2: Full design space per device

This huge design spaces are not feasible for exploration. Furthermore, because most of the samples from this design space are ill-conditioned meaning that they are failing either because of not complying to the standard or due to lack of resource at runtime. These failed configuration samples do not provide any information about exploration, therefore their execution attempts are wasted effort and time. Using the methodology described in Section 3.1.1, it is possible to reduced to space semi-automatically to a more reasonable size by eliminating the samples that are out of standard.

Table 4.3: Pruned design space per device and case study

	PLT1		PLT2	
	GPU	CPU	GPU	CPU
Convolution:	17	24	19	24
MatrixMult:	5	7	6	7

In Table 4.3, it has been shown the design space size after pruning exploration space. The application specific parameters and constraints are in Appendix C. This huge reduction in number of configuration points is the result of the constraints that are enforced by the standard. The largest contributor to this decrease is the fact that OpenCL standard requires global work size to be divisible by work-group size for each dimension. Thus, the number of feasible work-group sizes depends on prime factorization of the global work size.

For example, global work size of 1024 will only allow work-groups sizes of power of 2, while global work size of 1000 will allow 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 200, 125, 250, 500, 1000 leading to 15 configurations. The reason behind this is $1000 = 2^3 \times 2^3$ compared to $1024 = 2^10$.

Full-Search Autotuning

Looking at the pruned results, it is observable that the numbers are small enough to be fully explored. We used exhaustive searching methodology without any machine intelligent approach since the exploration space left after pruning is manageable. Hence, the applications are assessed for each work-group sizes in order to find out the fastest one.

To verify the methodology, application described in Section 4.1.2 has been used. Convolution application has been used with 625 width mask and 655360 as a global size. Matrix Multiplication application has been used with 1024x1024 matrices.

On both platforms, work-group sizes on CPU have reached only 4096 rather than the device limit of 8192. This is because local memory usage exceeding the local memory size on the both of the CPUs (Figures 4.5 and 4.7).

Work-group sizes for matrix multiplications are total workgroup sizes. Therefore 8x8 work-group size will result in 64 total work-group size. To-

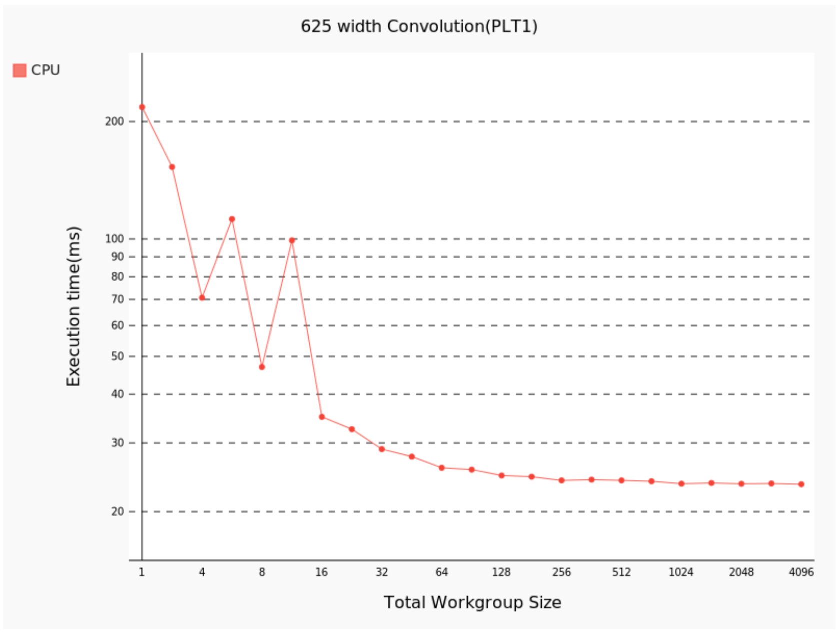


Figure 4.5: Platform 1, Convolution with CPU

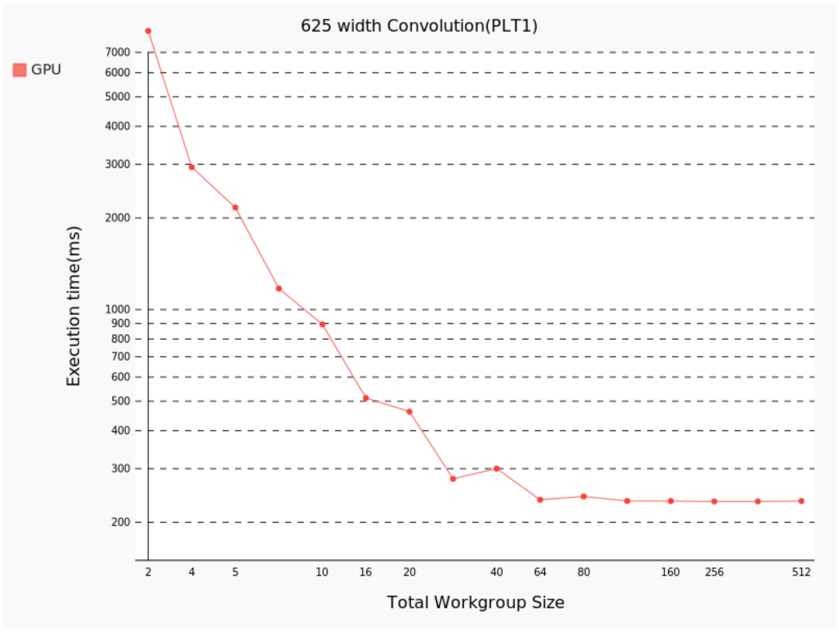


Figure 4.6: Platform 1, Convolution with GPU

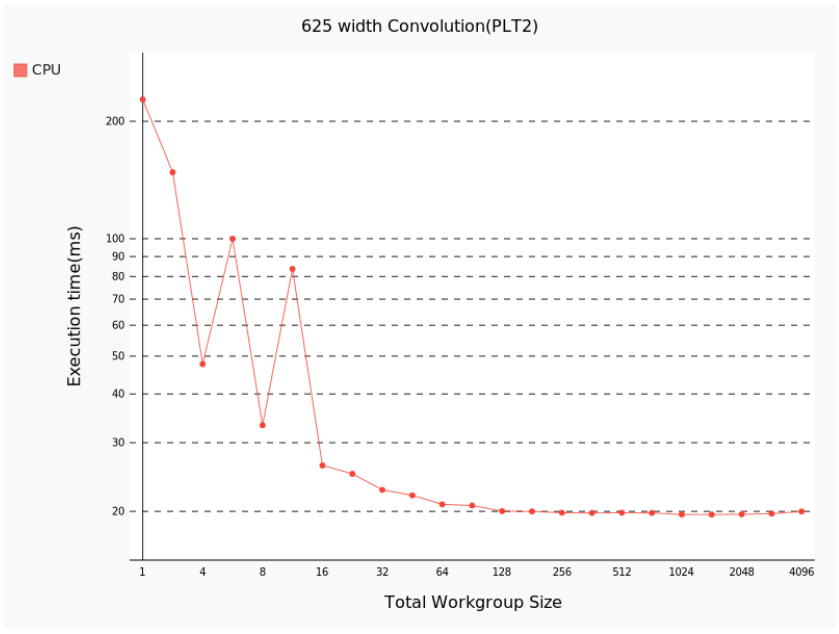


Figure 4.7: Platform 2, Convolution with CPU

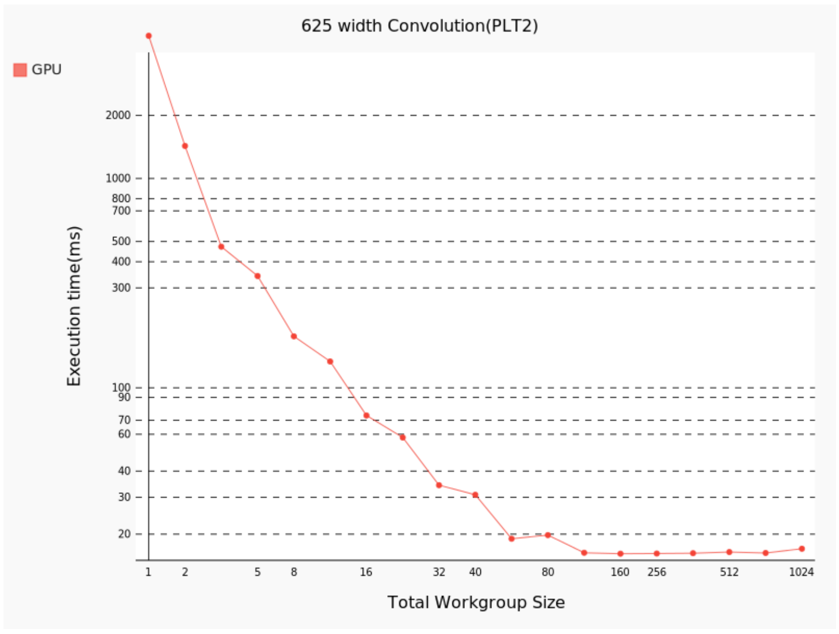


Figure 4.8: Platform 2, Convolution with GPU

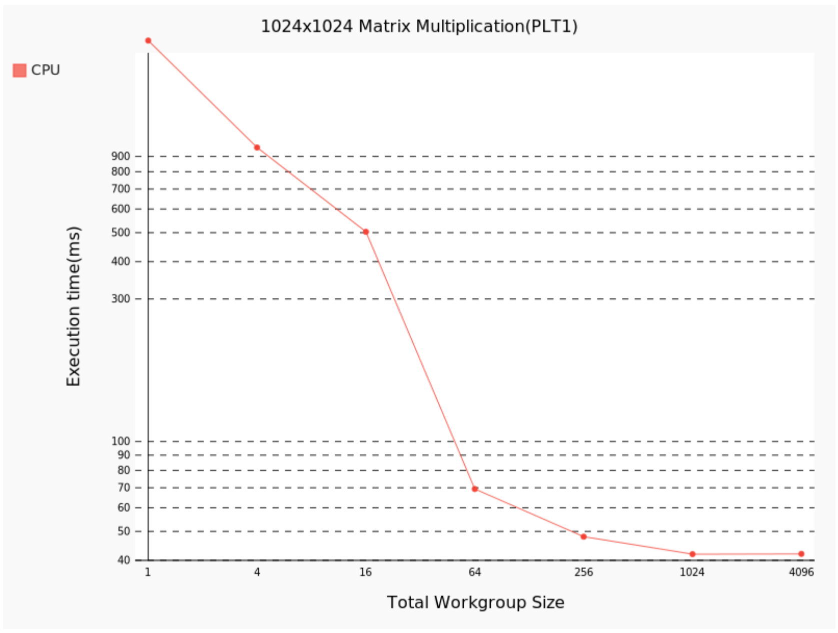


Figure 4.9: Platform 1, Matrix Multiplication with CPU

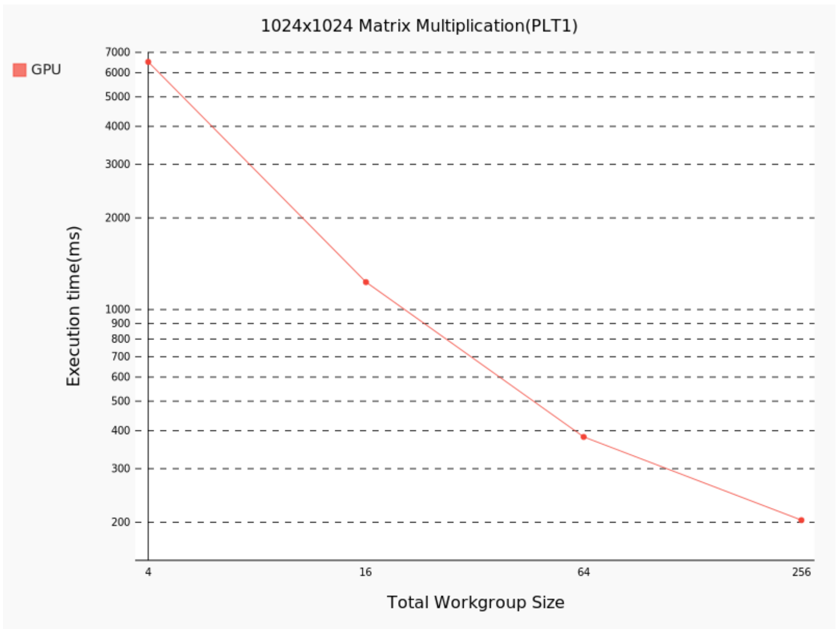


Figure 4.10: Platform 1, Matrix Multiplication with GPU

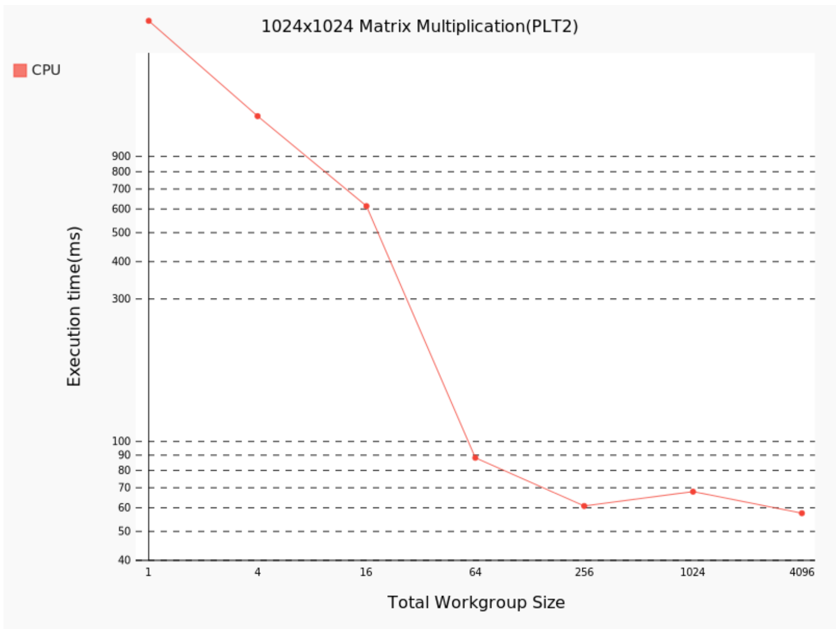


Figure 4.11: Platform 2, Matrix Multiplication with CPU

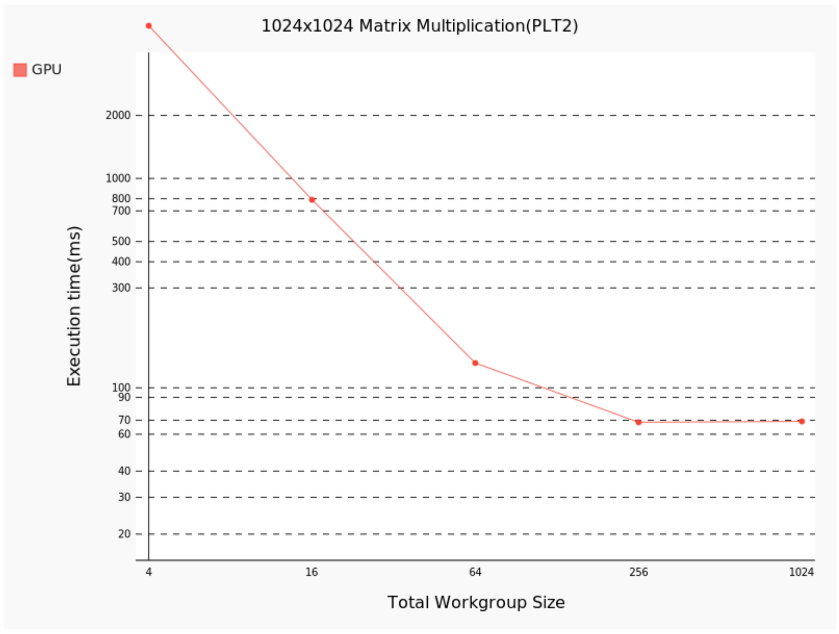


Figure 4.12: Platform 2, Matrix Multiplication with GPU

tal work-group size is crucial in terms of determining the maximum size for each dimension of work-group size. Because of that number of configuration options on Figures 4.9, 4.10, 4.11 and 4.12 significantly lower than convolution case study. Moreover in Figures 4.6, 4.10 and 4.12, work-group sizes do not start from 1. This is because Nvidia has hardware limitation on how many work-groups can be present in one kernel launch even though OpenCL standard does not introduce such a limitations. Therefore those execution runs are discard.

Other important remark is that all the devices on both platforms have benefit from increased work-group size especially compared to really small work-group sizes like 4 or 8. The reason is by increasing the number of work-items in a work-group, OpenCL driver will have potentially higher degree of parallelism and more options of work-items to schedule. On GPUs, the penalty of not being able to utilize parallelism within work-group is greater compared to the CPUs.

4.2.2 Heterogeneous Executions

There are some problems with heterogeneous execution in Platform 1. The GPU on Platform 1 is an old generation (2010) processor. Even though, it supports OpenCL 1.1, its hardware is not fully compliant. Therefore, kernel launching is not asynchronous as required by the OpenCL standard. This creates problem in heterogeneous runs because it prevents running two kernels to each device simultaneously. In the end, the kernel launchings happen in sequential order. The performance is between CPU and GPU because some of the workload is shifted to the slower device while the executions are not overlapped (Figure 4.13).

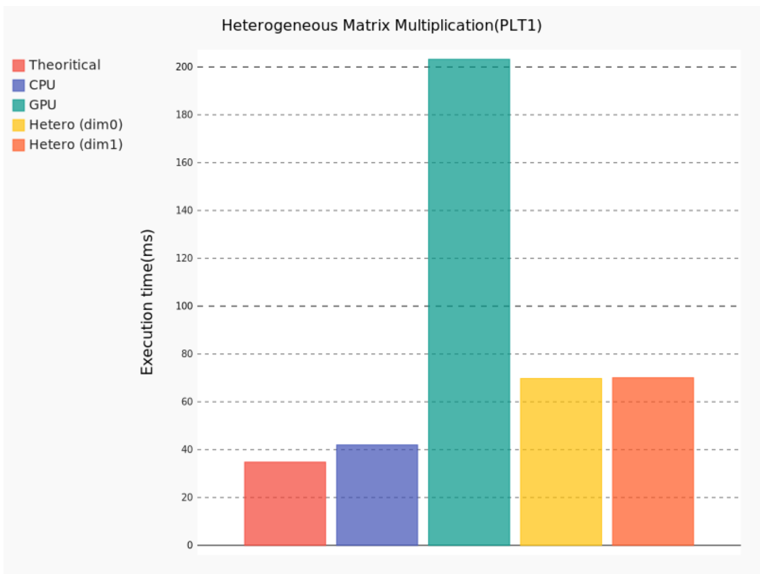


Figure 4.13: Platform 1, Matrix Multiplication with GPU and CPU

The theoretical estimate of the perfect scenario shows that even if the heterogeneous execution had been successful, the speed up would be less than 20 percent. This is due to the fact that GPU of Platform 1 is significantly slower than its CPU. Hence potential heterogeneous execution has diminishing returns.

In the case of convolution application (Figure 4.14), situation is more dire since the performance difference between the processors are even larger. Theoretical estimate only suggests 10 percent improvement. While the actual run suffers due to lack of asynchronous OpenCL kernel launching.

Under the Platform 2, convolution application fails to gain performance benefits due to bug in the Bumblebee command-line tool. This tool normally allows to switch on the discrete GPU, hence it is activated for each application run. But in this case, using the tool to switch on forces

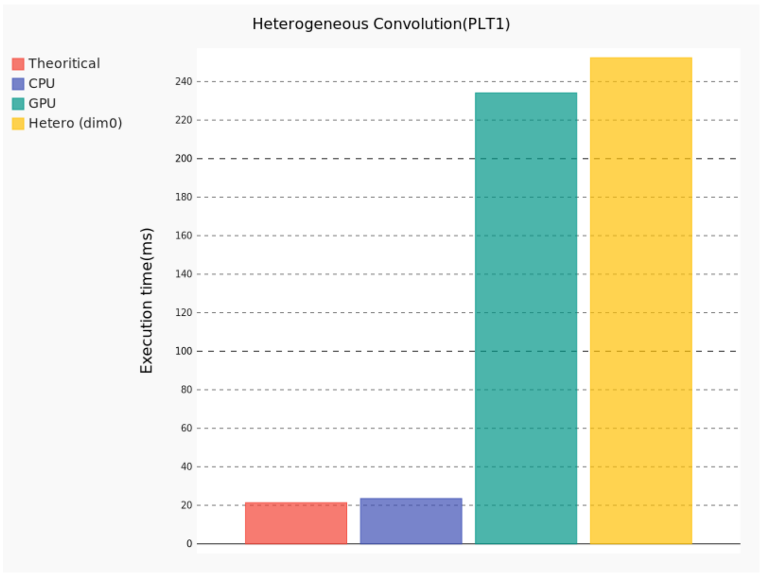


Figure 4.14: Platform 1, Convolution with GPU and CPU

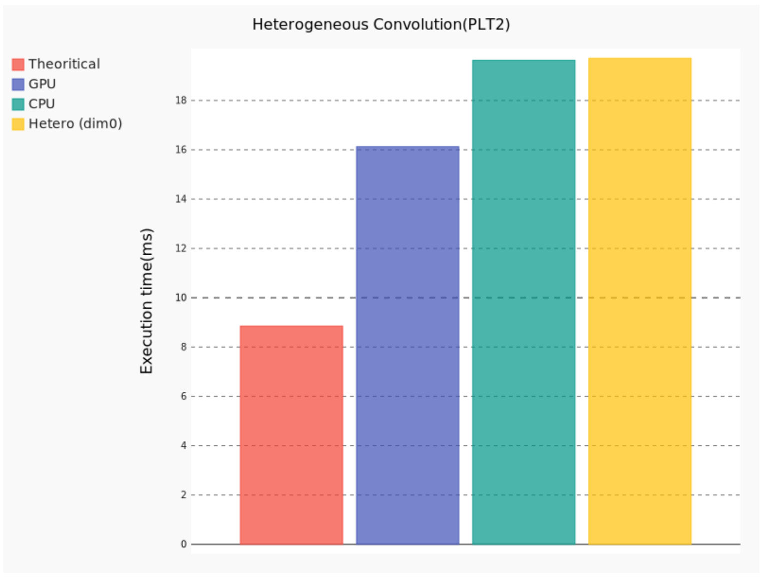


Figure 4.15: Platform 2, Convolution with GPU and CPU

the CPU runtime to execute non-splited version. Because of this, even if the GPU overlaps with CPU, due to long running time of Intel processor, heterogeneity is bounded by CPU execution time (Figure 4.15).

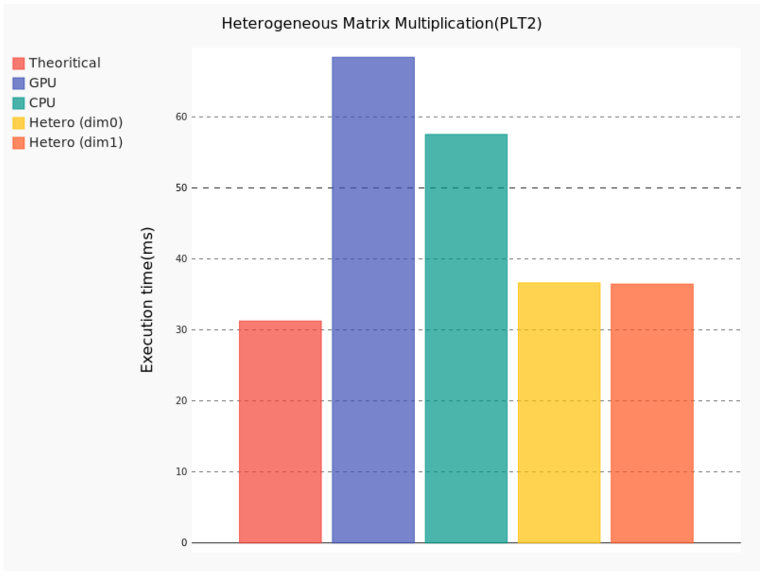


Figure 4.16: Platform 2, Matrix Multiplication with GPU and CPU

Finally in Figure 4.16, we can see an improvement over both CPU and GPU. According to multiple runs of the same matrix multiplication application, CPU is 18 percent faster than GPU. Compared to the Platform 1 the difference between heterogeneous device is not big in terms of performance. While theoretical estimate for heterogeneous run is 84 percent faster than CPU, in real run we did get a 57 percent of speed up compared to fastest of the both heterogeneous devices.

The speed up achieve is the result of two devices successfully running in parallel, therefore their execution times are overlapped. Using our proposed method, it is possible to achieve performance improvement that is just 17 percent less than the theoretical best configuration. Theoretical speed up does not care about work-group sizes, it just uses the best performance values of the configurations which are explored already.

In the Figure 4.13 and Figure 4.16, there are two heterogeneous runs. Since matrix multiplication uses two dimensional global work and work-group sizes, we investigate splitting through different dimensions. As seen on the 4.16, the result is not affected by splitting dimension.

Even though some experiments may fail to exploit heterogeneity, our framework return the configuration that performs best even if it is not the heterogeneous execution.

As mentioned before, time required for autotuning an application is also important. Even though, our methodology relies on being statically tune the application, the less time it takes, the better it is. The tuning times for both applications and platforms are presented below:

	PLT1	PLT2
Convolution	1289.4 s	1018.6 s
Matrix Multiplication	749 s	1172.5 s

Table 4.4: Autotuning elapsed time

Chapter 5

Conclusions

In this Thesis, we have investigated existing approaches about efficient autotuning on heterogeneous platforms from [9] and improved upon this previous work. In order to increase the usability, an automatic OpenCL parameter collector has been integrated before the constraint solving phase. Moreover, a default set of constraints that are enforced by OpenCL standard is added to the constraint solver. This allows application developers to focus on their application parameters, if necessary to introduce new constraints related to their application. Therefore, usability of the workflow is enhanced.

While the work in [9] focuses on multiple tasks that are represented with acycling graph for data flow and then try to find efficient way to map tasks of the graph to hardware devices, in this Thesis we proposed an approach to split a single big task in order to be able to map it onto heterogeneous platforms. The method that we introduced takes advantage of OpenCL kernels that are autotuned for different devices separately. The configurations with optimal work-group sizes are chosen after the exploration and a workload balance point is estimated. With this estimation split factor is decided while taking desired work-group sizes into consideration.

The proposed approach has been assessed on two popular algorithms, namely, convolution and matrix multiplication. In order to run the benchmarks a small library to handle boiler-plate part of OpenCL host and json configuration file.

Experimental results have shown that it is possible to reduce design space and also gain some performance benefits in terms of usage of heterogeneity by using our proposed methodology.

5.1 Future Works

In the future, much more diverse algorithms are planned to be assessed and more recent and HPC-oriented computing platforms will be the target platform to further verify the methodology. There are also places for some methodological improvements. As a next step, it is possible to devise an algorithm to calculate the split factors in a way that favors to preservation of estimated split factor rather than preservation of optimal work-group sizes. This can decrease the gap between theoretical execution time estimate and measured run-time execution time.

Bibliography

[1] 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.

[2] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. Autotuning opencl workgroup size for stencil patterns. *arXiv preprint arXiv:1511.02490*, 2015.

[3] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225. IEEE, September 2011.

[4] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE, 2011.

[5] Khronos Group. The open standard for parallel programming of heterogeneous systems. url: `https://www.khronos.org/opencl/`, [Online; Accessed: Nov. 2015].

[6] Khronos Group. Opencl private memory address space qualifier. url: https://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/private.html, [Online; Accessed: Aug. 2016].

[7] Intel. Intel xeon processor e5 v3 family. url: http://ark.intel.com/products/family/78583/Intel-Xeon-Processor-E5-v3-Family#@All [Online; Accessed: Sep. 2016].

[8] MiniZinc. Medium-level constraint modelling language minizinc. url: http://www.minizinc.org/ [Online; Accessed: Aug. 2016].

[9] E. Paone, F. Robino, G. Palermo, V. Zaccaria, I. Sander, and C. Silvano. Customization of OpenCL applications for efficient task mapping under heterogeneous platform constraints. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 736–741. EDA Consortium, 2015.

[10] Alok Prakash, Siqi Wang, Alexandru Eugen Irimiea, and Tulika Mitra. Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pages 208–215. IEEE, 2015.

[11] Jie Shen, Ana Lucia Varbanescu, Henk Sips, Michael Arntzen, and Dick G Simons. Glinda: a framework for accelerating imbalanced applications on heterogeneous platforms. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 14. ACM, 2013.

Acronyms

AMD	Advanced Micro Devices	23, 26
APU	Accelerated Processing Unit	5
CPU	Central Processing Unit	3–6, 12, 13, 15, 19–24, 26–31, 39, 42, 51–53, 62, 67, 68, 70
CUDA	Compute Unified Device Architecture	5, 12, 23
DSE	Design Space Exploration	7, 8, 33, 46
DSL	Domain Specific Language	32

GPGPU	General-purpose computing on GPU	4, 12, 25
GPU	Graphical Processing Unit	3–6, 15, 22–31, 42, 51, 52, 67, 68, 70
HPC	High Performance Computing	6, 74
iGPU	Integrated Graphical Processing Unit	5, 25, 52
ISA	Instruction Set Architecture	20
SIMD	Single Instruction Multiple Data	20–23
SIMT	Single Insruction Multiple Thread	23, 25
SoC	System on Chip	26, 29

Appendix A

MiniZinc Constraint Models

Common OpenCL Constraints

The MiniZinc constraint model for OpenCL platform constarints:

```
% platform specific parameters
int: max_wg_size_x;
int: max_wg_size_y;
int: max_wg_size_z;
int: num_compute_units;
int: max_total_wg;
int: local_mem_size;

var 1..max_wg_size_x: workgroup_1;
var 1..max_wg_size_y: workgroup_2;
var 1..max_wg_size_z: workgroup_3;

var int: total_num_wg = workgroup_1 * workgroup_2 * workgroup_3;

% global-work-size should be divisible by work-group-size
% (check opencl 2.0 specs might be changed)
constraint (global_1 mod workgroup_1 == 0);
constraint (global_2 mod workgroup_2 == 0);
constraint (global_3 mod workgroup_3 == 0);

% number of work-group should be equal
% or greater than compute_units
constraint ((global_1 div workgroup_1 +
             global_2 div workgroup_2 +
             global_3 div workgroup_3) >= num_compute_units);
```

```
% total work-group size should be multiple of pref_wg_mult
% (warning needs testing, disabled)
% constraint (total_num_wg mod pref_wg_mult == 0);

% total work-group size should be less than max work-group size
constraint (total_num_wg <= max_total_wg);

% local memory usage must be less than local memory size
constraint (local_mem_usage <= local_mem_size);

% use --all-solutions -statistics flags
solve satisfy;
```

OpenCL Platform Parameters Examples

Example OpenCL platform parameters for a Intel CPU and Nvidia GPU:

```
% OpenCL device constraint parameters for: CPU
%device_name: Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz
%device_vendor: Intel(R) Corporation
%cl_version: OpenCL C 1.2

max_wg_size_x = 8192;
max_wg_size_y = 8192;
max_wg_size_z = 8192;

num_compute_units = 8;
max_total_wg = 8192;
local_mem_size = 32768;

% OpenCL device constraint parameters for: GPU
%device_name: GeForce GT 550M
%device_vendor: NVIDIA Corporation
%cl_version: OpenCL C 1.1

max_wg_size_x = 1024;
max_wg_size_y = 1024;
max_wg_size_z = 64;

num_compute_units = 2;
max_total_wg = 1024;
local_mem_size = 49152;
```

Appendix B

Example Configuration Json

```
    "configs": [{
        "split_type": "overlap",
        "kernel_name": "matmul",
        "split_factors": [0.4375, 0.5625],
        "kernel_file": "matmul.cl",
        "task_configs": [{
            "launch_config": {
                "globalsize": [512, 224, 1],
                "groupsize": [32, 32, 1]
            },
            "build_flags": ["TILE_SIZE=32"],
            "hardware": {
                "cl_version": 1.1,
                "dev_type": "GPU",
                "device": "GeForce GT 550M",
                "vendor": "NVIDIA Corporation"
            },
            "priority": 0
        }, {
            "launch_config": {
                "globalsize": [512, 288, 1],
                "groupsize": [16, 16, 1]
            },
            "build_flags": ["TILE_SIZE=16"],
            "hardware": {
                "cl_version": 1.2,
                "dev_type": "CPU",
                "device": "Intel(R) Core(TM) i7-2630QM",
                "vendor": "Intel(R) Corporation"
            },
            "priority": 1
        }], "split_dim": 1 }]
```

Appendix C

Case Study Sources

Convolution MiniZinc Sources

MiniZinc application specific constraints and parameters:

```
% 1D Convolution Example
int: global_1 = 655360;
int: global_2 = 1;
int: global_3 = 1;

int: mask_size = 625;

var int: local_mem_usage = (mask_size + workgroup_1 - 1) * 4;

include "ocl_common.mzn";

% Output database point
output [
"wg_1:"++show( workgroup_1 )++",",
"wg_2:"++show( workgroup_2 )++",",
"wg_3:"++show( workgroup_3 )++",",
"lmem:"++show( local_mem_usage ),
"\n"
];
```

Matrix Multiplication MiniZinc Sources

MiniZinc application specific constraints and parameters:

```
% Matrix Multiplication Example
int: global_1 = 1024;
int: global_2 = 1024;
int: global_3 = 1;

var 1..256 : tile_size;

% application specific, local memory usage in bytes
% for tiled matrix multiplication:
var int: local_mem_usage = (tile_size * tile_size) * 4 * 2;

constraint (workgroup_1 == workgroup_2);
constraint (tile_size == workgroup_1);

include "ocl_common.mzn";

% Output database point
output [
"wg_1:"++show( workgroup_1 )++",",
"wg_2:"++show( workgroup_2 )++",",
"wg_3:"++show( workgroup_3 )++",",
"lmem:"++show( local_mem_usage ),
",tile_size:"++show(tile_size),
"\n"
];
```