

Politecnico di Milano
Facoltà di Ingegneria dell'Informazione



Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

**Parallel, GPU-based, computing: analysis of
the state of the art and preliminary
experiments to support modeling of
heterogeneous, GPU and CPU-based
applications**

Relatore: Prof Elisabetta Di Nitto

Tesi di Laurea di: Matteo Franchi 883237

Anno Accademico 2019-2020

*dedico questo lavoro alla mia famiglia, che mi ha
sempre supportato durante tutti i miei anni di studio.*

Ringraziamenti

Ringrazio la professoressa Elisabetta Di Nitto per la grande disponibilità e pazienza che ha avuto nei confronti, e per la possibilità di lavorare su questa tesi.

Ringrazio poi Michele Guerriero, che con il suo lavoro di dottorato di ricerca ci ha permesso di partire da una solida base, e lo ringrazio inoltre per la sua disponibilità ad aiutarci a tal proposito.

Piacenza, 18 Novembre 2020

Sommario

In un mondo in cui i Big Data e l'Intelligenza Artificiale sono sempre più importanti, la richiesta di potenza computazionale non è mai stata così abbondante. Per rispondere a tale richiesta, la "computazione eterogenea" si propone con un principio fondamentale: che ogni operazione venga calcolata dall'hardware più adatto, ossia l'hardware che riesca a completare l'operazione nei tempi più rapidi possibili e con la maggiore efficienza. Per quanto possa risultare chiaro tale principio, esso è tutt'altro che semplice da realizzare, soprattutto per quanto riguarda la progettazione del software che dovrebbe lavorare con sistemi eterogenei. Decidere manualmente quali parti di codice debbano essere eseguite su quali device richiede molto tempo e una conoscenza profonda delle architetture dei singoli device. Per di più, il codice da produrre deve essere pensato con una prospettiva di computazione parallela, cosa che si è storicamente dimostrata essere difficile e controintuitiva per la mente umana. La questione della sincronizzazione è altrettanto complicata quando si parla di componenti di sistemi eterogenei, dato che essi hanno bisogno di comunicare tra di loro e di accedere spesso ad una memoria condivisa. Nonostante gli sforzi di molte imprese nel mondo per creare soluzioni che facilitino tali processi, abbiamo ancora bisogno di poter utilizzare strumenti che ci permettano di lavorare in maniera efficace nell'ambito dei sistemi eterogenei; strumenti che ci possano permettere di astrarre dalla logica di basso livello delle diverse architetture, ma che allo stesso tempo possano mantenere un livello sufficiente di espressività e correttezza.

Definizione di sistema eterogeneo

Prima di procedere oltre, soffermiamoci su come definiamo un sistema eterogeneo: qualsiasi sistema computazionale che sia formato da componenti hardware

differenti può essere considerato un sistema eterogeneo. Tutti i nostri computer portatili moderni, per esempio, appartengono alla categoria di sistemi eterogenei, dato che contengono un processore (CPU) e una scheda grafica (GPU). Entrambi questi componenti sono costruiti per poter eseguire computazioni, ma la loro architettura è largamente differente, e il codice pensato per poter essere eseguito su uno non può essere direttamente eseguito sull'altro. Data questa definizione, la computazione eterogenea è semplicemente l'atto di eseguire software capace di sfruttare la potenza computazionale di un intero sistema di componenti eterogenee.

Applicazioni potenziali

Come abbiamo già accennato inizialmente, il nostro principale incentivo a condurre ricerca nel campo della computazione eterogenea è la performance: ogni tipo di hardware è progettato per essere efficiente nel eseguire solamente una frazione delle operazioni che troviamo nei software moderni, sicché la soluzione migliore non può che essere quella di fare eseguire ogni operazione al componente hardware che può farlo nel minor tempo possibile. Oltretutto, in buona parte dei casi il componente hardware più veloce a completare una operazione è allo stesso tempo quello più efficiente in termini di consumo energetico, cosa che rendere ancora più conveniente questo tipo di approccio. Tra i settori in cui possiamo sfruttare il vantaggio di questo tipo di potenza computazionale troviamo:

- **Intelligenza Artificiale (AI).** Negli ultimi tempi, un grande interesse è emerso per l'Edge-AI, ossia la pratica di posizionare l'hardware che deve processare gli algoritmi AI vicino ai sensori che acquisiscono i dati, riducendo così consumi energetici e costi di comunicazione tra componenti. Data la natura distribuita di sistemi del genere, viene naturale posizionare l'hardware più veloce nei punti più adatti [1][7].
- **Machine Learning (ML).** Considerata una sottocategoria dell'Intelligenza Artificiale, il Machine Learning si basa sul training di modelli, cosa che richiede un alto numero di iterazioni da eseguire su grandi set di dati. Tali iterazioni consistono in numerose operazioni simili tra di loro, che possono essere efficientemente calcolate dall'hardware

più efficiente in ciò, risparmiando così un grande quantitativo in tempo una volta completate tutte le iterazioni [10].

- **Reti Neurali (NN)**. Molti ambiti in cui la sicurezza è prioritaria, come la guida autonoma per i veicoli o l'atterraggio guidato per i velivoli sono basati sulle cosiddette reti neurali profonde (Deep Neural Networks, DNN), le quali richiedono un grande quantitativo di potenza computazionale, che può essere fornita attraverso l'uso di sistemi eterogenei [16][3].

Obbiettivi della tesi

Date le numerose sfide e gli ostacoli che emergono nel lavorare con sistemi eterogenei, abbiamo deciso con la nostra tesi di indagare su come potremmo contribuire a risolvere alcuni di essi, soprattutto per quanto riguarda la creazione di framework facili da utilizzare e ed efficaci nel scrivere software destinato a sistemi eterogenei. Questo ci porta a ciò che vogliamo fare con la nostra tesi, descrivibile in due fasi principali:

- Come prima cosa, analizzeremo una serie di framework e piattaforme che possono essere considerate stato dell'arte nel campo della computazione eterogenea, assieme ad alcuni altri strumenti che ci aiuteranno nello sviluppo del progetto pratico.
- In secondo luogo, proveremo ad affrontare parte delle problematiche riscontrate nella prima fase, attraverso lo sviluppo di un modesto progetto, il cui scopo principale sarà quello di combinare gran parte di ciò che avremo visto nella prima sezione in uno strumento di modellazione che ci aiuterà a generare software per sistemi eterogenei.

Struttura della tesi

La tesi è suddivisa nei seguenti capitoli:

- Nel **Capitolo 1, Introduzione**, viene intrdotto il contesto, il soggetto e gli obbiettivi della tesi.

-
- Nel **Capitolo 2, Stato dell'Arte**, viene presentata una selezione di framework e piattaforme all'avanguardia che interessano la nostra ricerca.
 - Nel **Capitolo 3, Impostazione di ricerca del problema**, vengono presentate le principali sfide e problematiche che concernono il campo della computazione eterogenea.
 - Nel **Capitolo 4, Progetto logico della soluzione del problema**, viene dettagliato il progetto della tesi, che punta a risolvere parte delle problematiche dell'ambito della computazione eterogenea.
 - Nel **Capitolo 5, Valutazione del progetto**, viene giudicato il progetto attraverso la riproduzione di un esempio pratico.

La tesi si conclude con il **Capitolo 6, Conclusioni e lavori futuri**, dove vengono discussi i risultati ottenuti e presentati sviluppi futuri.

Table of Contents

1	Introduction	1
1.1	Definition of heterogeneous systems	2
1.2	Potential applications	2
1.3	Scope of the thesis	3
1.4	Structure of the thesis	4
2	State of the art	5
2.1	An introduction to GPUs	6
2.2	CUDA	7
2.3	SafeGPU	13
2.4	Tensorflow	15
2.5	Flink	17
2.5.1	Combining Flink and Tensorflow	20
2.6	StreamGen	22
2.7	Conclusions on the state of the art	24
3	Research problem setup	27
3.1	Parallelism	27
3.2	Concurrent Memory Access	29
3.3	Scope of the project	30
3.3.1	Reasons of choice	30
3.3.2	Requirements	31
3.3.3	Machine-learning features	31
3.3.4	Relation between challenges and goals	32
4	Design of the problem solution	35
4.1	Working with Flink and Tensorflow	36

TABLE OF CONTENTS

4.2	The extension of StreamGen: StreamGen-Tensorflow	36
4.2.1	UML profile extension	37
4.2.2	Code generator logic update	40
5	Project evaluation	45
5.1	Goal of the evaluation	45
5.2	Application description	45
5.3	Implementation	47
5.4	Evaluation	48
6	Conclusions and Future Works	51
	Bibliography	53

Chapter 1

Introduction

”In a world with Big Data and diverse workloads, the need for compute seems endless — and now with the influence of AI through new and merged workloads, the need for compute may prove to be endless. The hunger for compute has spurred an onslaught of product competing to help accelerate our workloads – all helping usher in the era of Heterogeneous computing.”

James Reinders - ”The future of computing is heterogeneous, CPU and friends”

As Big Data and Artificial Intelligence become more and more prominent, the need for computing power and performance is as high as it ever was. To that end, heterogeneous computing offers a way forward with a simple principle: make each operation be calculated by the best hardware, that is, the one that does it in the fastest and most efficient way. However, as simple as it may seem, it is equally complex to put it in practice, and, chief among the challenges, is programming the software that is able to take advantage of heterogeneous systems. Manually deciding which piece of code has to run on which piece of hardware is time-intensive and requires a deep understanding of the underlying system. On top of that, the code needs to be designed with a parallel-computing perspective, something that has proven to be extremely hard and counterintuitive for the human mind. Synchronization is another challenging aspect when dealing with heterogeneous devices, since they need to communicate with each other and access some form of shared memory. Despite the great effort that is put by numerous companies worldwide to develop solutions that can facilitate this process, we are still lacking the tools

that would enable us to work effectively within the heterogeneous computing paradigm; tools that, at the same time, would be able to abstract from the low-level logic of different architectures, but also maintain a sufficient amount of expressiveness and correctness.

1.1 Definition of heterogeneous systems

Before we go further, let's clarify on what we consider to be an heterogeneous system: any computing system that is comprised of different hardware components can be considered to be a heterogeneous system. All of our modern laptops, for instance, belong to the category of heterogeneous systems, since they are fitted with a Central Processing Unit (CPU) and a Graphical Processing Unit (GPU). Both components are tasked to perform computations, yet their architecture is vastly different, and code designed to run on one cannot be simply fed to other in a straightforward way. Given this definition, heterogeneous computing is simply the act of executing software that can simultaneously take advantage of the various hardware of an heterogeneous system.

1.2 Potential applications

As we hinted right at the beginning, our main incentive for research in the field of heterogeneous computing is performance: each type of hardware is designed to be efficient in executing only a fraction of the tasks that we need to perform in our everyday software, so the optimal solution must be based on running each task on the piece of hardware that can do it in the smallest time possible. On top of that, it is often the case that the fastest device for running a particular task consumes the least amount of power, which emboldens the case for switching to a heterogeneous approach even more.

Many are the fields that could take advantage of this kind computational power, such as:

- **Artificial Intelligence (AI)**. Recently, there's been much interest in the concept of Edge-AI, which is the practice of putting the hardware tasked to perform AI algorithms near the sensors that acquire the data, thus reducing power consumption and data communication costs. Given

the distributed nature of this approach, it comes natural to put the most fitting processing unit in the right spot [1][7].

- **Machine Learning** (ML). Machine learning, which is considered a subset of AI, is all about training models, which requires an extensive number of iterations on large datasets. Numerous, similar operations can effectively be calculated by the processing unit that gets the job done in the least amount of time, potentially saving hours and hours on training iterations [10].
- **Neural Networks** (NN). Many safety-critical fields, such as autonomous driving for cars or guided landing for aircraft, are based on Deep Neural Networks (DNN) and require a high amount of computational power, which can be provided with the use of heterogeneous systems [16][3].

1.3 Scope of the thesis

Given the many challenges that still affect the world of heterogeneous computing, we decided with our thesis to see if we could contribute to solving some of them, especially when it comes to finding an effective and easy-to-use framework for writing software destined for heterogeneous systems. This brings us to what we aiming to do, which is twofold:

- First, we will analyze a number of frameworks and platforms that can be considered to be state-of-the-art in the field of heterogeneous computing, along with some other tools that will come in handy later on for our subsequent project.
- Secondly, we will try to deal with part of the issues and limitations encountered at first by developing a modest project, whose main goal would be to combine much of what we studied in the first part into a model-driven tool that would help us generate software meant to run on heterogeneous systems.

1.4 Structure of the thesis

The thesis is divided into the following chapters:

- **Chapter 1, Introduction**, introduces the context, subject and scope of our thesis.
- **Chapter 2, State of the art**, presents a selection of state-of-the art frameworks and platforms that pertain to our research.
- **Chapter 3, Research problem setup**, lays out the main challenges and issues that characterize the field of heterogeneous computing.
- **Chapter 4, Design of the problem solution**, details the project of the thesis, which aims at solving part of the discussed problem of heterogeneous computing.
- **Chapter 5, Project evaluation**, describes the evaluation of the project, which is conducted through a practical example.

The thesis is concluded with **Chapter 6, Conclusions and Future Works**, where we discuss the obtained results and future developments.

Chapter 2

State of the art

In this day-and-age, it's hard not to be near a device with heterogeneous components at any given time: laptops, workstations and even smartphones are nowadays included with CPUs and GPUs. The ever-growing pervasiveness of heterogeneous systems in our professional and private lives has led companies, developers and researchers to put great effort into working on modern solutions that can help us design software for these systems. In this chapter, we are going to analyze some of the leading technologies, including more experimental solutions, that try to tackle the many challenges that can be found when working with heterogeneous systems. In particular, our selected projects will primarily focus on the relationship between CPUs and GPUs, which are, arguably, the most common and historical duo that we can find in our everyday-life computing machines. The introduction of GPUs, during the Nineties, can indeed be seen as the first major attempt at improving the computational power of machines with the use of an heterogeneous assortment of processing units. On top of that, in recent years, developers have found renewed interest in GPUs, which have shown great potential in working with a number of tasks outside of the graphical area, promising to be crucial in several general-purpose contexts. Considering the focus we've just described, our selection of state-of-art subjects will comprise a set of solutions for working with GPUs, with the addition of some tools for facilitating software development that show potential compatibility with the former.

Below is a list of the state-of-the-art frameworks presented in this chapter:

- **CUDA**: a parallel computing platform and application programming

interface (API) model created by Nvidia [8][5].

- **SafeGPU**: an experimental tool that facilitates the development of software for general-purpose GPUs, using a design-by-contract methodology [9].
- **Tensorflow**: an interface for expressing and executing machine learning algorithms, released by Google [10].
- **Flink**: an open-source, unified stream-processing and batch-processing framework developed by the Apache Software Foundation [4].
- **StreamGen**: an experimental program that helps to design streaming applications via an UML-based modeling language [11].

2.1 An introduction to GPUs

A Graphical Processing Unit is a type of hardware that specializes in performing a very high number of simple, parallel calculations, like the ones that are typically found when rendering graphics (images, videos, 3D models). As such, they differ from CPUs by having thousands of Arithmetic Logic Units (ALUs, the basic workers for calculations), compared to the general 4 to 8 units that we commonly see in modern CPUs (Fig 2.1).



Figure 2.1: An architectural comparison between CPUs and GPUs. [12]

Despite the raw computational power, GPUs cannot handle the complex and various tasks required to run modern Operative Systems and their supported programs, and need to be supervised by a CPU in order to render their

services properly. Developers can already benefit from a number of low-level interfaces and APIs when writing code that wants to involve the GPU in their computation, such as OpenGL or DirectX. Newer solutions like CUDA, which we'll see in a moment, are aiming to facilitate the interaction between CPUs and GPUs not just for graphical tasks, but for more general range of tasks, which could be effectively accelerated by the computational power of GPUs.

2.2 CUDA

The need for working with hardware of different architectures isn't new: since the introduction of the first Graphical Processing Units, software developers are required to take advantage of heterogeneous systems when programming their code. Nowadays GPUs are very prominent in supporting graphical software, but their usefulness doesn't stop there: for a while now it has become quite clear that this type of hardware can effectively be used for a wide range of non-graphical applications. Nvidia Corporation, who rose as a GPU-manufacturing company, is very well aware of such potential, and for some years now has been working on an API that could help with this endeavor. Its name is CUDA and its purpose is to allow software developers and engineers to use modern Nvidia GPUs as General-Purpose GPUs [8]. A GPU is treated as General-Purpose when it's used for computations that would normally be assigned to CPUs.

The most effective way of describing how the CUDA architecture operates is through a basic example in CUDA C++ that adds the elements of two arrays with a million elements each [5]. The example also shows how CUDA can easily improve on the performance of simple tasks by applying small modifications to the code.

```
1 #include <iostream>
2 #include <math.h>
3
4 // function to add the elements of two arrays
5 void add(int n, float *x, float *y)
6 {
7     for (int i = 0; i < n; i++)
8         y[i] = x[i] + y[i];
9 }
10
11 int main(void)
12 {
13     int N = 1<<20; // 1M elements
14
15     float *x = new float[N];
16     float *y = new float[N];
17
18     // initialize x and y arrays on the host
19     for (int i = 0; i < N; i++) {
20         x[i] = 1.0f;
21         y[i] = 2.0f;
22     }
23
24     // Run operation on 1M elements on the CPU
25     add(N, x, y);
26
27     return 0;
28 }
```

Listing 2.1: standard vector addition without CUDA.

We start with the basic scenario: two vectors are generated and passed to a classic add operation, so the CPU takes care of adding each pair of cells, one pair at a time. If we want to involve the GPU in the computation, we need to change the add function into a *kernel*, that is, a function that can be called by the CPU and is then run on the GPU. We do this by adding the specifier `_global_` to the function.

```
1 // CUDA Kernel function to add the elements of two arrays on
   the GPU
2 _global_
3 void add(int n, float *x, float *y)
4 {
5     for (int i = 0; i < n; i++)
6         y[i] = x[i] + y[i];
7 }
```

Listing 2.2: the specifier `global` is added.

The CUDA architecture provides a Unified Memory system between CPU and GPU, which is an essential part in delegating work to the GPU. To allocate data in unified memory we use `cudaMallocManaged()`, which returns a pointer that we can access from *host* (CPU) code or *device* (GPU) code. At the end of the task, memory is freed with `cudaFree()`.

```
1 // Allocate Unified Memory -- accessible from CPU or GPU
2 float *x, *y;
3 cudaMallocManaged(&x, N*sizeof(float));
4 cudaMallocManaged(&y, N*sizeof(float));
5
6 ...
7
8 // Free memory
9 cudaFree(x);
10 cudaFree(y);
```

Listing 2.3: memory management in CUDA.

Last thing to do is to launch the kernel using a triple angle bracket syntax `<<< >>>`, which specifies the number of thread blocks and the number of threads per block that we want to deploy (more on that later).

State of the art

```
1 #include <iostream>
2 #include <math.h>
3 // Kernel function to add the elements of two arrays
4 __global__
5 void add(int n, float *x, float *y)
6 {
7     for (int i = 0; i < n; i++)
8         y[i] = x[i] + y[i];
9 }
10
11 int main(void)
12 {
13     int N = 1<<20;
14     float *x, *y;
15
16     // Allocate Unified Memory - accessible from CPU or GPU
17     cudaMallocManaged(&x, N*sizeof(float));
18     cudaMallocManaged(&y, N*sizeof(float));
19
20     // initialize x and y arrays on the host
21     for (int i = 0; i < N; i++) {
22         x[i] = 1.0f;
23         y[i] = 2.0f;
24     }
25
26     // Run kernel on 1M elements on the GPU
27     add<<<1, 1>>>(N, x, y);
28
29     // Wait for GPU to finish before accessing on host
30     cudaDeviceSynchronize();
31
32     // Free memory
33     cudaFree(x);
34     cudaFree(y);
35
36     return 0;
37 }
```

Listing 2.4: vector addition run on GPU (one thread only).

Using a GeForce GT 750M, it takes 411ms to run this version of the function. So far we used one thread, so the next logical step would be to use a bunch of threads in order to speed up the process.

```
1 add<<<1, 256>>>(N, x, y);
```

By only changing the number of threads we won't have the expected result, since it will do the computation once per thread, rather than spreading it across the parallel threads. To do it properly, we need to modify the kernel by introducing some keywords: `threadIdx.x` contains the index of the current thread within its block, and `blockDim.x` contains the number of threads in the block. We need to modify the loop so that it iterates with the correct pacing.

```
1 _global__
2 void add(int n, float *x, float *y)
3 {
4     int index = threadIdx.x;
5     int stride = blockDim.x;
6     for (int i = index; i < n; i += stride)
7         y[i] = x[i] + y[i];
8 }
```

The task is completed by the same GPU in 3.2ms, which is a substantial increase in speed, but we can still improve the efficiency by using more than one thread block. CUDA GPUs are grouped into many parallel processors, called Streaming Multiprocessors (or SM) and each SM can run multiple concurrent thread blocks, which together make up what is known as the *grid*. The overall system of threads and blocks is organized as follows:

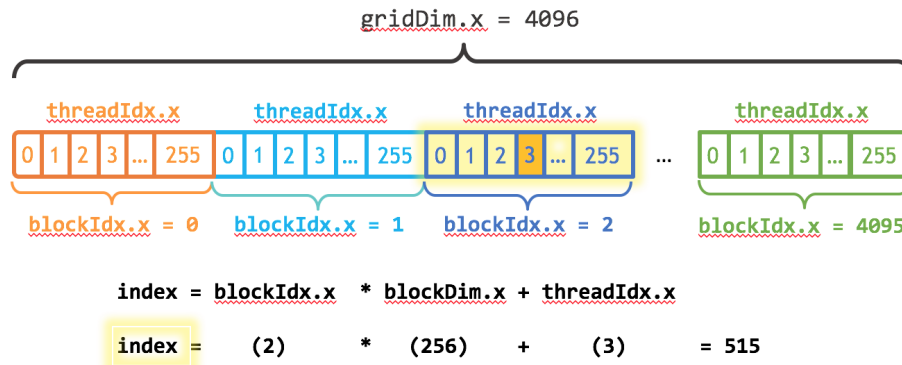


Figure 2.2: How threads are indexed in the CUDA grid.

The kernel has to be modified again to account for multiple blocks:

```

1 __global__
2 void add(int n, float *x, float *y)
3 {
4     int index = blockIdx.x * blockDim.x + threadIdx.x;
5     int stride = blockDim.x * gridDim.x;
6     for (int i = index; i < n; i += stride)
7         y[i] = x[i] + y[i];
8 }

```

To maximize the speed of computation we will use enough blocks so that there will be one thread for each pair of cells that we want to sum up, which is simply done by dividing the vector size with the threads of each block (and round up the result).

```

1 int blockSize = 256;
2 int numBlocks = (N + blockSize - 1) / blockSize;
3 add<<<numBlocks, blockSize>>>(N, x, y);

```

To no surprise, the task is performed in the fastest speed of 0.68ms. As shown through the example, simple changes to normal coding of functions are required in order to involve the GPU in the computation, and basic variables allow to intelligently distribute the workload on the threads.

2.3 SafeGPU

Although CUDA relieves the developer of some of the low-level technicalities involved in the programming of parallel, gpu-runnable code, it still requires the user to manually design kernels (and deal with memory allocation). This problem is tackled by SafeGPU [9], a programming approach developed by Alexey Kolesnichenko, that aims to make GPGPUs accessible through high-level libraries for object-oriented languages, while maintaining the performance benefits of lower-level code. To ensure the optimality and correctness of the final code, SafeGPU operates on a design-by-contract methodology, which derives from the Eiffel language: programs are built using preconditions, post-conditions and invariants which specify the properties that should hold before and after the execution of methods. Contract-checking generally adds a big enough overhead to runtime computation that many developers prefer using it for debug only, but SafeGPU solves the issue by making the GPU itself run the checks on runtime. SafeGPU is still in early development and features simple linear algebra optimizations to code.

The following snippet of code exemplifies how SafeGPU takes care of kernel generation and allows contract-checking:

```

1 matrix_transpose_vector_mult (matrix: G_MATRIX[DOUBLE]: vector:
   G_VECTOR[DOUBLE]): G_MATRIX[DOUBLE]
2   require
3     matrix.row = vector.count
4   do
5     Result := matrix.transpose.right_multiply (vector)
6   ensure
7     Result.rows = matrix.columns
8     Result.columns = 1
9   end

```

Listing 2.5: an operation on a matrix and a vector with SafeGPU

The method takes as input a matrix and a vector, transposes the matrix, multiplies it with the vector and returns the resulting vector. The core of the operation is done with the keyword **.transpose** concatenated with **.right_multiply**, but the construction of the relative kernels is delegated to SafeGPU, which is also able to compose different kernels in case there's room

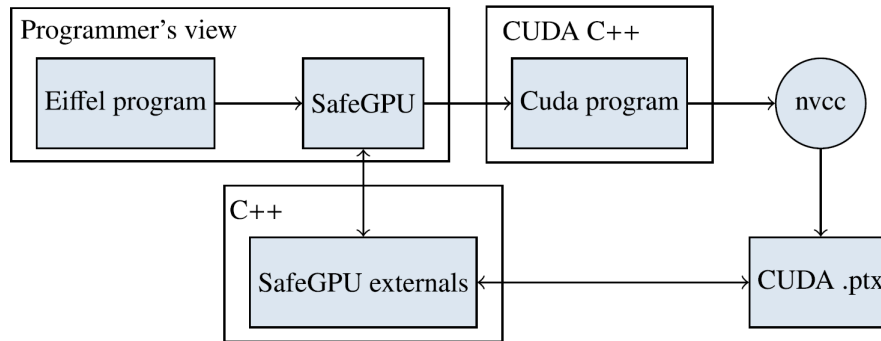


Figure 2.3: The architecture of SafeGPU.

for optimization. The correctness of the function is checked before and after the operation: the **require** section states that the rows of the matrix have to be the same length as the dimension of the vector, while the **ensure** section makes sure that the result of the operation is a vector of appropriate size.

The integration of SafeGPU with CUDA is described in Figure 2.3: the programmer utilizes SafeGPU by writing code in Eiffel, which naturally interfaces with C++ for initialization, data transfers and device synchronization (thanks to its built-in mechanisms), then the resulting kernel is generated by the `nvcc` compiler (the Nvidia compiler for CUDA) into a `.ptx` file, containing a CUDA module that the library uses to launch the kernel.

Generation of kernels in SafeGPU is based on a deferred execution model: instead of generating a kernel for each method and executing them one at a time, kernels are added to a list of pending actions (one list for each different collection) whose execution is triggered by specific calls. Once a trigger is observed, an execution plan is generated in the form of a directed acyclic graph (DAG), representing data and kernels in two different types of nodes, with edges as the dependencies between them. The optimizer runs through the DAG and merges nodes and dependencies wherever possible. For instance, in Figure 2.4 we see that the dot product method, which is composed of component-wise multiplication of two vectors and summation of each element, is merged into a single kernel from two starting kernels.

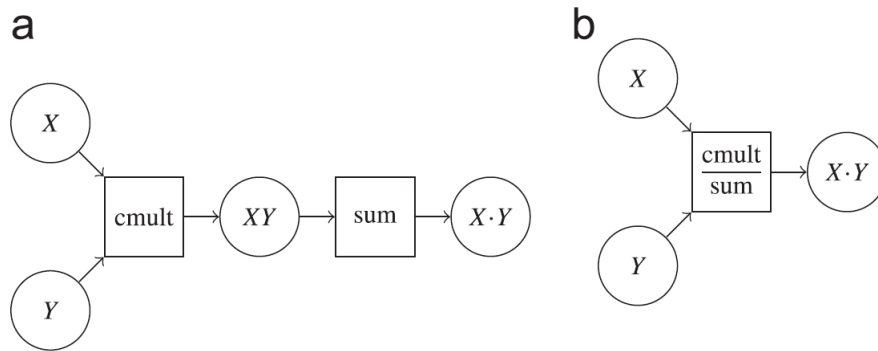


Figure 2.4: Kernel optimization of the dot product method.

SafeGPU is an interesting approach to the challenge of making heterogeneous applications in a simplified and correct way, but at the same time using the contract-based Eiffel language makes it significantly less versatile than what can be normally achieved with more mainstream OO languages, although there's currently an effort to extend the API for C#.

2.4 Tensorflow

We can easily demonstrate how valuable GPUs can be outside of graphics rendering by taking a look at the field of machine learning, whose massive potential is only impaired by the amount of computation that asks to be provided with. Google Brain, a prominent research team at Google, released in 2015 and is currently working on Tensorflow [10], an API for expressing and executing machine learning algorithms.

Similarly to SafeGPU, TensorFlow visualizes its workflow with a directed graph, composed of several nodes and edges. Each node represents an operation and can have multiple ingoing and outgoing edges, which represent the flow of data in the form of arrays of elements called tensors. An example is provided with Figure 2.5.

Tensorflow defines a *Kernel* as a particular implementation of an operation that can run on a specific type of device, such as CPUs or GPUs. In order for a Client to utilize Tensorflow and execute a graph, his application first has to create a *Session*, then initialize an arbitrary number of *Runs* on such session, with each run having a specified set of output nodes: the idea behind this

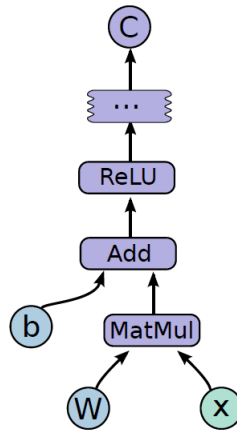


Figure 2.5: An example of a Tensorflow graph.

```

import tensorflow as tf

b = tf.Variable(tf.zeros([100]))           # 100-d vector, init to zeroes
W = tf.Variable(tf.random_uniform([784,100],-1,1)) # 784x100 matrix w/rnd vals
x = tf.placeholder(name="x")             # Placeholder for input
relu = tf.nn.relu(tf.matmul(W, x) + b)   # Relu(Wx+b)
C = [...]                                # Cost computed as a function
                                          # of Relu

s = tf.Session()
for step in xrange(0, 10):
    input = ...construct 100-D input array ... # Create 100-d vector for input
    result = s.run(C, feed_dict={x: input})    # Fetch cost, feeding x=input
    print step, result
  
```

Figure 2.6: An example of a Tensorflow application.

system is that different runs can traverse the graph through different nodes, depending on the selected output nodes, although the typical use of Tensorflow is done by executing millions of identical runs that propagate through the entire graph (which is typical for training ML algorithms). Figure 2.6 shows the code relative to the graph of Figure 2.5, including its execution through a session.

An important aspect of Tensorflow is how the execution of a graph is handled in practice. In the simple case of single-device environment, the algorithm only has to compute the order of execution of nodes, which is determined by their dependencies: every node has a counter that tells how many nodes must first be traversed to have the correct data as input, and when the counter reaches 0 the node is added to a "ready" queue. The creation of kernels for

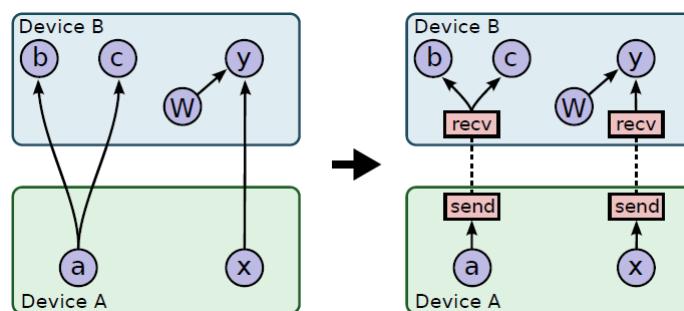


Figure 2.7: Cross-device communication in a Tensorflow graph.

the execution of a node is delegated to the device assigned to such node. In case we want to run the graph on a multitude of devices, it's imperative to determine which node is assigned to which device, and how these devices communicate with each other. For the former task, the algorithm essentially uses a greedy heuristic that examines the effects on the completion time by placing the node on each possible device, then the device with lowest completion time is selected. Once every node is assigned to a device, the graph is partitioned into a set of sub-graphs, one per device. Every edge from one device to another one is replaced with edges to go into either Sender or Receiver nodes, as shown in Figure 2.7. This method not only allows to isolate the communication part inside the Sender and Receiver implementations, but we also ensure that the data is sent and allocated on memory only once per source-destination pair. In case of a distributed system, communication is implemented with TCP or RDMA protocols.

2.5 Flink

In the age of Internet, many modern-day applications require to deal with a constant flow of data, which means that the throughput of a system is an essential aspect for the correct functioning of an application. As such, streaming applications can heavily benefit from the high computation power that heterogeneous systems can offer. The Apache Software Foundation, among a multitude of different programs, provides developers with Apache Flink, an open-source, distributed stream-processing framework [4]. Flink excels at working with many common cluster resource managers such as Hadoop YARN,

State of the art

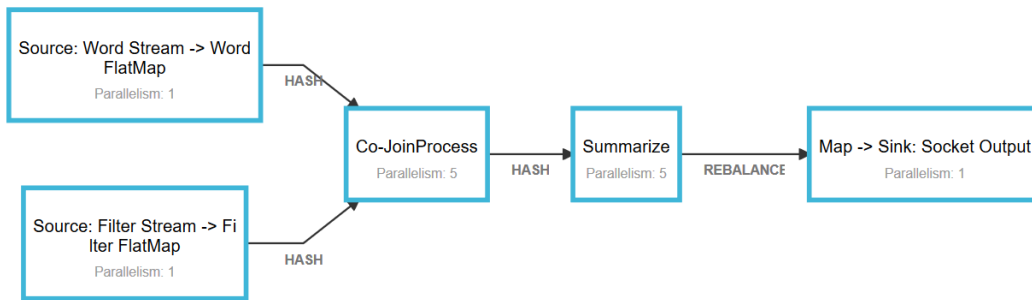


Figure 2.8: An example of a Flink pipeline.

Apache Mesos, and Kubernetes, and is able to create stateful, fault-tolerant applications that take advantage of parallelized processing.

Characteristics

The first basic concept of Flink is the distinction between bounded and unbounded streams of data: while a bounded stream, also called batch, has a definite start and end, an unbounded stream never terminates and provides data as it's generated (possibly from physical sensors, user input etc.). Any Flink application can be represented with a pipeline, where incoming data enters the first segment of the pipeline called *Source*, is then processed and transformed in each of the intermediate segments (called Transformations), and eventually exits the pipeline through the rightmost segment called *Sink*. Figure 2.8 shows an example of a Flink pipeline.

The execution of Flink applications, also called jobs, is supervised by the *Job Manager*, which is responsible for scheduling tasks, recovery, coordination, and the overall bigger picture of a job. The Job Manager distributes the workload on a number of *Task Managers*. Each Taskmanager can execute many tasks on different threads, can communicate with each other (even remotely, through TCP connections) and has its own memory space. Figure 2.9 illustrates what was just described.

Practice

From a more operational perspective, Flink programs can be written in common languages such as Java and Scala, and all share a number of elements, which will be referenced below from Listing 2.6. The following

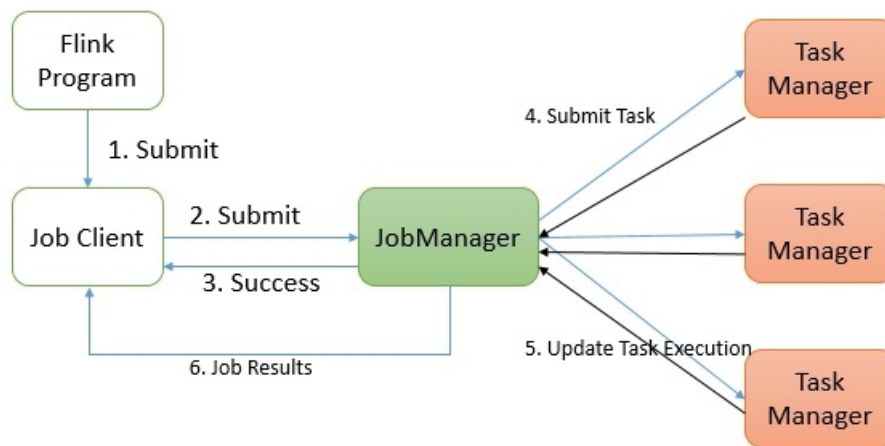


Figure 2.9: Flink Execution Graph

piece of code is a Flink program that simply counts the number of words typed by user in real time. First of all, the execution environment is the common context (either local or remote) in which all applications operate, allowing various tools for controlling the job execution and for interacting with the external environment (line 5 to 6). The next common element is a Flink source, from which data is taken into the pipeline. In this case, data is received in the form of text from user input (line 11). Every meaningful Flink application also applies some transformations to the received data (line 12 to 20) and usually makes use of the processed data through a Flink sink, in this case a simple write to text (line 22 to 25). Finally every application terminates by calling the appropriate method for executing the generated environment (line 27).

```

1 public class StreamingWordCount {
2
3     public static void main(String[] args) throws Exception {
4
5         final StreamExecutionEnvironment env =
6             StreamExecutionEnvironment
7                 .getExecutionEnvironment();

```

```
8      //uncomment the below if you want to set the default
9      parallelism for the project.
10     //env.setParallelism(1);
11
12     DataStream<String> text = env.socketTextStream("localhost",
13     9999);
14     DataStream<WordToken> tokens = text
15     .flatMap(new LineSplitter())
16     .setParallelism(4);
17
18     DataStream<WordCount> counts = tokens
19     .keyBy("word")
20     .timeWindow(Time.seconds(3))
21     .apply(new WordCounter())
22     .setParallelism(4);
23
24     counts
25     .keyBy("word")
26     .writeAsText("/home/utente/word-count-output.txt")
27     .setParallelism(1);
28
29     JobExecutionResult result = env.execute();
30     System.out.println("EXECUTION TIME: " + result.
31     getNetRuntime(TimeUnit.SECONDS));
32 }
```

Listing 2.6: WordCount example on Flink with Java

2.5.1 Combining Flink and Tensorflow

Experimenting with Flink and Tensorflow is going to play a crucial role in the making of the project, as we will explain in the following chapters. A potential benefit of merging the two technologies would be to generate a Flink pipeline capable of providing machine learning functionalities, which could be managed by adapting Tensorflow methods (that utilize Tensorflow libraries) and calling them during the execution of specific Flink transformations.

Existing work

Before delving into the feasibility of coupling these two elements, we searched for third-party solutions that already managed to do so. We found that the most complete and polished solution was an open-source project developed by Eron Wright called Flink-Tensorflow, presented during the annual conference Flink Forward of 2017 [19]. Flink-Tensorflow is a library that facilitates the use of machine-learning algorithms in Flink applications, and it comes with a couple of examples. One of these examples consists in a Flink pipeline that continuously receives images as input and recognizes them on the fly, using a pre-trained machine-learning model. The model can be utilized by the program thanks to a number of methods provided by the Tensorflow library. As noted by the same developer during his presentation at the aforementioned conference, the application's biggest limit is that it doesn't supervise the training of the model required for the labeling operation, but it only manages to apply the model to a series of inputs.

Limitations

This kind of limitation can be traced back to the nature of the two components that we wanted to combine in the first place: while Tensorflow is versatile enough to be able to provide its tools in an external environment, a pipelined and automated framework like Flink is hardly suitable for the training of machine-learning algorithms, since it is generally based on a series of trials, errors and polishes that can only be really performed manually by a human person. As a result, the only feasible level of compatibility we can expect from Flink and Tensorflow is the sort that we can find in the example that was just described: a Flink pipeline that uses a pre-trained model for prediction and recognition purposes. Nevertheless, this kind of application can still be interesting to investigate and potentially useful to develop. For a better idea of the final result, Figure 2.10 show an example of an image being recognized and labeled by this kind of application.

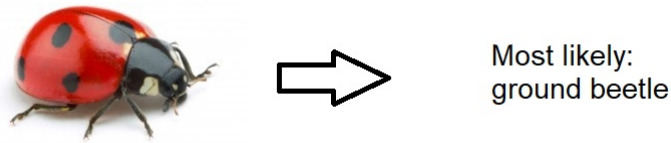


Figure 2.10: An example of image recognition.

2.6 StreamGen

In order to fully appreciate what this thesis aims to achieve, it's essential to delve into our next and final element of the state of the art, which was developed as part of a professional doctorate by Michele Guerriero, from the Politecnico di Milano. The application is called StreamGen and its main function is to automate the generation of fully-functioning Flink (and Spark) applications, starting from an UML-based modeling system [11]. The aforementioned functionality is divided into two main modules: StreamUML and StreamCGM.

StreamUML

StreamUML is a Domain-Specific Modeling Language (DSML) in the form of a UML profile, that allows to design a streaming application by constructing an appropriate UML Information Flow Diagram. The UML profile provides the ontology through which a developer is able to create such diagrams. In other words, it contains a series of stereotypes that can be arbitrarily combined. Every stereotype is mapped onto a piece of code, so when a diagram is finished, the resulting application will be generated depending on what stereotypes were used.

StreamCGM

The second module, StreamCGM, is the code generator that takes care of transforming the diagrams that were built with StreamUML into the actual code, using a language called Aceleo. As previously hinted, every component of a diagram is analyzed and converted into whatever code the generating stereotype is bound to, but the conversion isn't monolithic: for instance, during

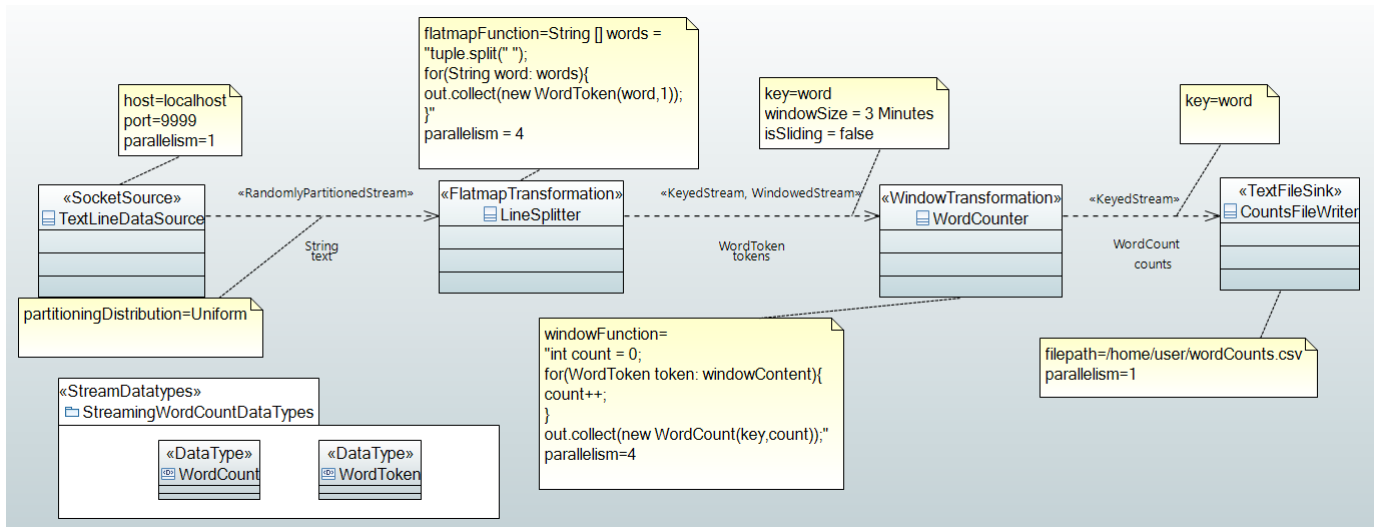


Figure 2.11: UML diagram of the WordCount example.

the code translation of a particular element of the UML diagram, the labeling of variables is strictly related to where such element was positioned in the graph, which inbound data is receiving and what parameters were specified by the user during the diagram design phase.


Example

To show the system in practice, we can see how we arrive to the final code of the WordCount example, which was conveniently presented in the Flink section with listing 2.6. Let's look at the UML diagram that originates the code with figure 2.11: every part of the code (source, transformations and sink) is clearly identified with the 4 nodes of the diagram, and the data, whose type is changed with each transformation, is passed along the arrows. The yellow windows are meant to show the parameters that are specified for each object and the two elements in the bottom left corner are included to define the classes of datatype that the data is transformed into during each transformation phase.

The complete Aceleo code in charge of converting the UML we just presented can be hardly coalesced into a page, so we will only take a look at the part that takes care of the first element of the diagram, the SocketSource. Figure 2.12 shows how the template of Aceleo code takes the information from the UML diagram to generate the final code in Java: while the type "String" and the

State of the art

```
[template public generateFlinkSocketSource(aClass : Class)]
  DataStream<[getOutputsConveyed(aClass) ->first()/]> [getOutputNames(aClass) ->first()/] =
  env.socketTextStream("[getStereotypeProperty(aClass, 'SocketSource', 'host')/]", [getStereotypeProperty(aClass, 'SocketSource', 'port')/]);
[/template]
```



```
DataStream<String> text = env.socketTextStream("localhost", 9999);
```

Figure 2.12: An example of conversion from Acceleo to Java code.

name "text" are extrapolated from the outgoing arrow of the Source element, the host name and the port number are taken from the parameters specified by the user in the object itself.

StreamUML and StreamCGM are designed in a way that allows them to be easily extended, which will come in handy for the development of our thesis project.

2.7 Conclusions on the state of the art

Each of the modern tools that we've just analyzed brings something to the table that can help us in developing software for heterogeneous systems, yet, as any cutting-edge technology, is bound to have a number of limitations. Let's summarize them:

- **CUDA** allows us to have great control on the distribution of workload in a heterogeneous context, but still operates at a low level of abstraction: kernels have to be manually constructed and memory must be explicitly allocated. Moreover, it can only be utilized by Nvidia graphic cards. Nevertheless, it is a good basis onto which develop higher-level tools and libraries, as we've seen with SafeGPU and Tensorflow.
- A more experimental solution, such as **SafeGPU**, tries to automate the process through the explicit use of contract-based semantics, which could be considered a double-edged sword: the design-by-contract methodology can give you a solid way to ensure the program's correctness, but it can become very expensive to maintain in larger projects, and can possibly

2.7 Conclusions on the state of the art

be incapable of asserting the correctness of some instances (like loop terminations). Either case, SafeGPU is in the early stages of development, so the scope and the set of available functionalities is still limited.

- **Tensorflow** has a vast library of functions for all kinds of tasks: Text, Images, Tabular, Video etc. On top of that, it's easy to get started with models based on neural networks, thanks to the graphical support that Tensorflow provides ... but at the same time it has its own model and terminology, so the learning curve is still quite significant, and the current documentation is lacking when moving away from beginner-level concepts.
- **Flink's** pipelined approach to streaming applications is promising, yet, being a relatively new technology, the API support is still limited and there aren't many open-source projects or community discussions. In other words, Flink needs to be used and explored more thoroughly if we really want to grasp its full potential and limitations.
- The core idea behind **StreamGen** is solid and sufficiently fleshed out: as long as you stick with its UML profile, you can build a number of Flink and Spark applications with ease by focusing on the modeling aspect. Consequently, StreamGen biggest limitation depends entirely on the level of expressiveness provided by its profile, which is so far relatively small. The question then becomes: how easily could we expand such profile? Will it be able to scale naturally or will it become harder to add more and more stereotypes?

In other words, there's still a long way to go if we want to comfortably program robust and optimized heterogeneous applications with a high level of abstraction, which clearly comes as no surprise: working with parallel, heterogeneous and distributed systems surely is going to be among the greatest challenges and opportunities for the IT sector over the coming decades.

Chapter 3

Research problem setup

After looking at some of the recent technologies in the field of heterogeneous computing, it is clear that programming software meant to run on a set of heterogeneous devices is fundamentally hard, and many are the challenges that arise from doing so. This chapter describes what we believe are the most important ones and, finally, introduces the scope of our practical project.

3.1 Parallelism

Heterogeneous systems are naturally fitted for working in a parallel manner: the more devices that can work at their most suitable tasks simultaneously, the less time that will be needed to finish all the tasks. Consequently, heterogeneous programming inherits the same problems that arise when designing parallel applications.

Unintuitiveness

The first layer of difficulty that is introduced when switching to parallel applications, is that the vast majority of developers are used to think in a serial way. The initial instinct is to start with a serial approach to a problem, then find the independent operations, decouple them, and finally run them in parallel, but the results are often unsatisfactory: the amount of parallelism may be too low and the process can turn out to be more complicated than expected [2]. This problem becomes even more grave when we move to the debugging

Research problem setup

phase: finding bugs in a parallel application is particularly hard since the order of operations cannot be deterministically foreseen.

Non-determinism

The non-deterministic aspect of software parallelism also brings synchronization problems: reading a piece of data before or after a scheduled write on it completely changes the final result of the operations, which brings the need for explicit synchronization mechanisms, such as locks. The problem with locks is that it's fairly easy to use them improperly, either by drastically reducing the amount of parallelism that can be obtained or even by creating infamous dead-lock situations.

Client-Server

How much and how easy we can exploit parallelism in applications strongly depends on what kind of tasks need to be performed. If we analyze the matter using a client-server metric, we can notice that server-based programs are generally easy to parallelize, since they handle a vast number of independent requests from different users, and they work with databases designed to support concurrent access. Client-based programs, on the other hand, are characterized by non-homogeneous code, fine-grained, complicated interactions and pointer-based data structures. These types of programs are thus much more intricate in terms of data dependency, which is the primary obstacle for parallel computing [6].

Hard cap

Ultimately, the amount of parallelism that can be extracted, and the subsequent speedup will always be limited by the length of the longest series of tasks that must be performed serially, as famously stated by Amdahl's Law [18]. For example, if a program needs 20 hours to complete using a single thread, but a one-hour portion of the program cannot be parallelized, therefore only the remaining 19 hours ($p = 0.95$) of execution time can be parallelized, then regardless of how many threads are devoted to a parallelized execution of this program, the minimum execution time cannot be less than one hour. Hence,

3.2 Concurrent Memory Access

the theoretical speedup is limited to at most 20 times the single thread performance. Fig 3.1 shows the maximum potential speedup through parallelization of a number of cases, varying in amount of parallelizable portion of code and available processors.

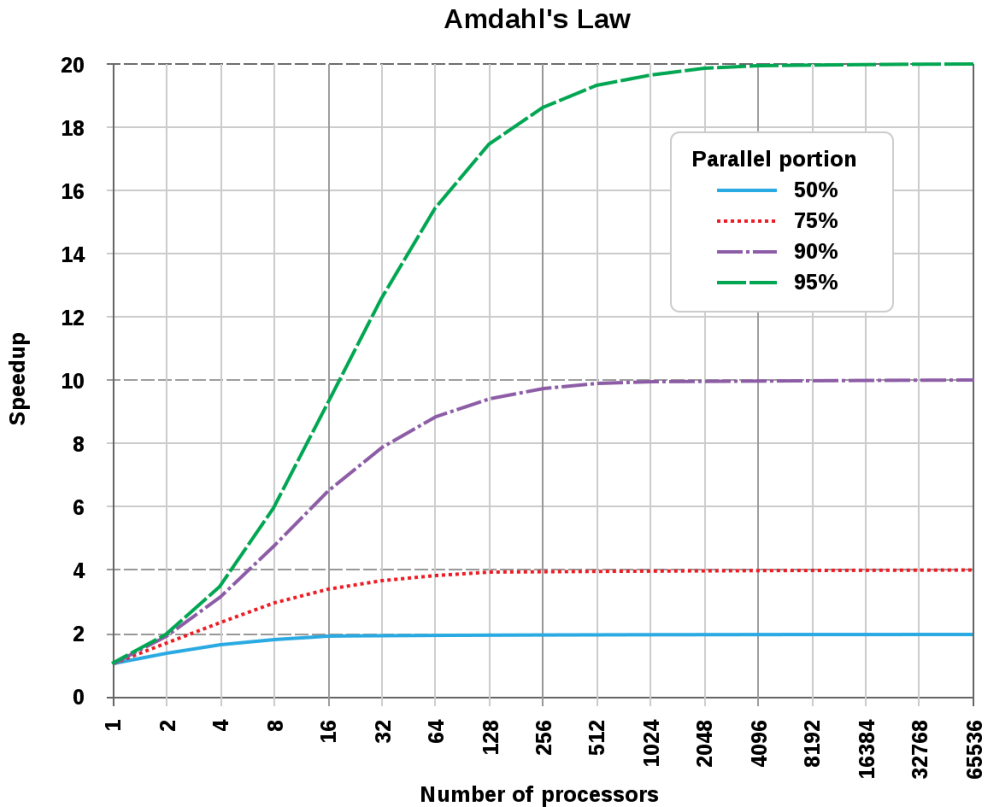


Figure 3.1: Examples of potential speedup based on the parallelizable percentage of code and the available processors.

3.2 Concurrent Memory Access

Memory abstraction

For heterogeneous systems, when it comes to designing either the physical layout of memory or the techniques for memory access, there's really no definitive answer for what the most optimal approach is. Any system that needs to read from or write to a memory component requires some kind of memory abstraction. On a software level, typically, developers work in the context of a shared

Research problem setup

virtual memory, so the level of abstraction is managed and provided by the operating system, but it's hard to say if the approach of a single, flat address space can be suitable for a heterogeneous system in which physical memory is distributed between the devices. It may be more effective to let the developers have some form of explicit influence on how memory is managed, which could differ from layout to layout.

Latency

From a hardware perspective we know that the latency of global communication in a heterogeneous system can be prohibitively high: when multiple processing units, such as CPUs and GPUs, are integrated together and share the same memory through a common bus, memory access requests coming from the GPU can heavily interfere with requests coming from the CPU cores, leading to low system performance and starvation of CPU cores. Unfortunately, state-of-the-art application-aware memory scheduling algorithms are ineffective at solving this problem at low complexity due to the large amount of GPU traffic [14][15].

3.3 Scope of the project

From what we've just seen, working with heterogeneous systems comes with many challenges and, in order to tackle some of them, our project is going to take advantage of what was already accomplished by the state of art. The desired outcome of our project should consist in an extension of StreamGen that would allow it to model and generate Flink applications with GPU-runnable, machine-learning operations, which would be provided by the Tensorflow API.

3.3.1 Reasons of choice

The particular choice of combining these diverse and modern solutions into an inclusive project stems from a number of reasons:

- We are seeking a way to facilitate the process of writing software for heterogeneous systems, which is, as we previously demonstrated, difficult by nature. StreamGen offers us an established framework for generating

runnable code that is centered on an intuitive modeling system, and the generated code is designed to be executed on a second, promising subject, that is Flink.

- Flink is, at its core, a framework for building streaming and distributed applications, the second quality of which is particularly relevant for our intent. There indeed a substantial similarity between a distributed system and a heterogeneous one, in that the former is naturally predisposed to the latter: a system that is already able to coordinate between physically sparse and disjointed devices can very well do so even if those devices are architecturally different (which is, arguably, the most common case in which you would want to work with distributed devices in the first place)
- Tensorflow is an API for designing and executing machine learning algorithms, which represent a very prominent approach in today's software solutions. On top of that, Tensorflow offers the feature to easily redistribute the workload, for the training phase of ML models, to the devices of our choice, be they CPUs or GPUs.

3.3.2 Requirements

The final result of our project should have the following features:

- The user should be able to model UML diagrams in the same way he was able to with the original version of StreamGen, with the additional capacity to choose from a number of new stereotypes that can introduce machine-learning functionalities in the UML diagrams.
- Provided that the designed model was valid, the user should be able to execute StreamGen's code-generation task and obtain a working, Java-code version of the model, ready to be executed on Flink.

3.3.3 Machine-learning features

Tensorflow provides the tools to facilitate the design of two major phases in the machine-learning context:

Research problem setup

- **Training phase:** machine-learning algorithms build a model based on available data in a process that is known as *training*. Models are generally represented by some type of neural network, composed of a series of nodes and edges. The training phase requires multiple iterations in which the learning parameters (weights and biases) of the nodes are tuned in order to improve the model. This phase is very demanding from a computational standpoint, having to iterate many times with possibly huge amounts of data, and one of the biggest features of Tensorflow, that is utilizing GPUs, can have a huge impact on performance times, when applied to this phase. As we saw in section 2.4, moving this phase in the Flink environment has proven to be more difficult than expected, but we still deem that this could be further investigated in future works.
- **Execution phase:** on the other hand, the execution of a model fares much better in a Flink environment than its training part, and Tensorflow is still able to take advantage of GPUs in this phase, even if it is generally less computationally demanding than the training phase. As a result, we will focus our efforts on making sure that our modified version of StreamGen will be able to generate Flink applications that support this type of feature.

3.3.4 Relation between challenges and goals

In the first part of this chapter, we presented the main issues revolving around designing heterogeneous systems, along with the relative software, and at first reading it can be hard to identify the connection between these issues and what we are trying to achieve. The challenges we described are particularly impactful when working at a low level of abstraction: for instance, we saw in Chapter 2 that even when utilizing the CUDA API, it is still necessary to manage memory allocation and de-allocation. However, we can arguably say that the two primary technologies that we are experimenting with, which are Flink and Tensorflow (with StreamGen being the "wrapper" of the two), operate on a higher level of abstraction, as they already manage to alleviate the user from dealing with these issues in part, yet they still present a number of limitations and learning difficulties that we believe can be further improved

3.3 Scope of the project

upon with our practical project. As a result, we can reasonably say that these issues are somehow touched by our efforts, even though indirectly.

Chapter 4

Design of the problem solution

This chapter provides a detailed description of how the thesis project was developed. There are three major components that our project tries to weave together:

- **StreamGen**: a modeling tool that converts UML representations into working Flink applications.
- **Flink**: a streaming data-flow engine that executes programs in a data-parallel and pipelined manner.
- **Tensorflow**: an API used for designing machine-learning applications, which is highly compatible with heterogeneous environments.

The desired, final result of our project should be an extended version of StreamGen, capable of generating Flink applications that incorporate machine-learning functionalities, provided by the Tensorflow API.

The development of the project can be divided into two main parts:

- The first task is to experiment with Flink and Tensorflow. Depending on the level of compatibility between these components, we need to come up with working applications that can be run on Flink and make use of some of the tools provided by Tensorflow.
- The second task is to design an effective extension of StreamGen, so that the latter will be able to generate Flink applications that also utilize Tensorflow tools. The extended version of StreamGen should at least be able to reproduce the working solutions developed in the first part, and possibly a number of unrealized, similar applications.

4.1 Working with Flink and Tensorflow

The part of experimentation between Flink and Tensorflow began with the search for attempts in the sector to effectively combine the two into meaningful applications, finding an interesting project, called Flink-Tensorflow, that we described in 2.5.1. Inspired by what was accomplished by this project and by the potential level of compatibility between Flink and Tensorflow, we used the approach of modifying StreamGen so that it could accommodate the modeling and the generation for this type of application. The first obstacle that had to be cleared was that Flink-Tensorflow was developed with the Scala language, while StreamGen was written in Java, so the example provided by Flink-Tensorflow needed to be ported in Java first. Although the conversion proved to be more challenging than expected, we were eventually able to reproduce the same example in Java.

4.2 The extension of StreamGen: StreamGen-Tensorflow

As described in previous chapters, StreamGen is composed of two main features: a modeling tool that allows the developer to design applications through UML diagrams and a generator module that automatically translates UML diagrams into fully-functioning code. Therefore, the desired extension of StreamGen can be divided as well into two main tasks:

- An extension of **StreamUML**: each of the UML diagrams that are built for modeling our Flink applications is an arbitrary composition of predefined stereotypes that are stored in StreamGen's UML profile. By adding the right entries to the profile, we would be able to model Flink applications that incorporate machine-learning functionalities.
- An update on **StreamCGM**: every stereotype from the UML profile is mapped to a specific Java template in the code generator section. Adding the templates for the new stereotypes is crucial for generating the final code of the applications that were modeled with those stereotypes.

4.2.1 UML profile extension

Premise

Before we present what we added to StreamUML, we need to provide a description of the main body to which we are applying our extension. StreamUML can be seen as a dictionary of basic blocks that can be combined in whichever way we choose to form the UML models of our desired applications. As such, it provides a set of stereotypes for the most frequent components that can be found in Flink and Spark streaming applications: sources from which data arrives, transformations that modify the data, and sinks that write the final data or pass them to external programs.

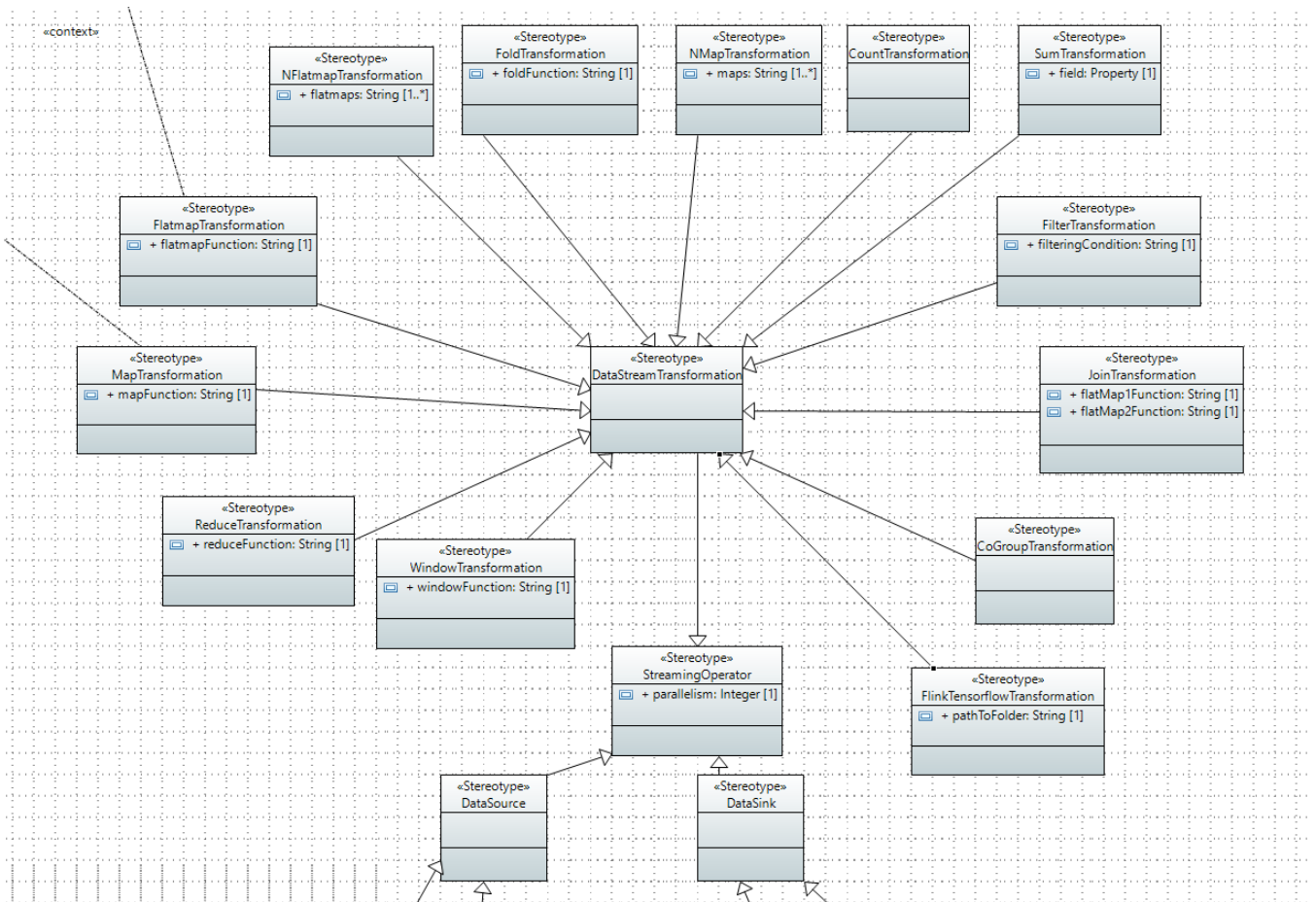


Figure 4.1: StreamUML’s stereotypes for transformations

Design of the problem solution

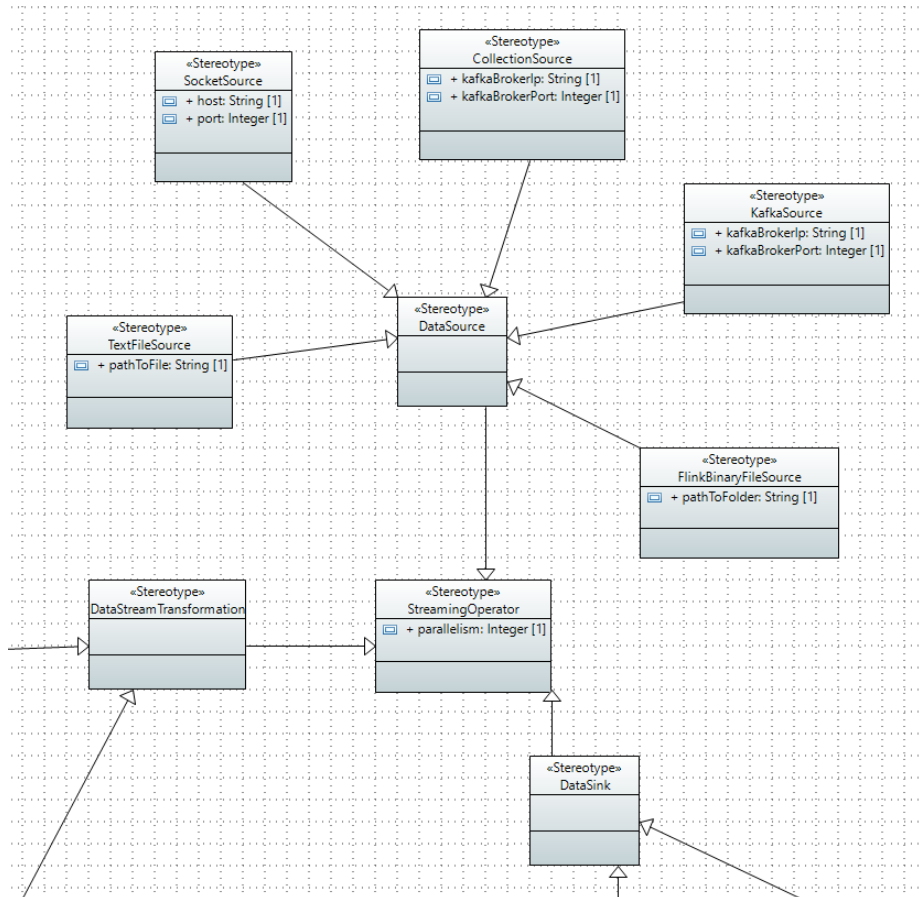


Figure 4.2: StreamUML's stereotypes for sources

Profile Extension

Figure 4.1 captures the section of StreamUML that defines the various types of applicable transformations: every specific instance of a transformation, be it `FlatmapTransformation` or `WindowTransformation`, are derived from the general `DataStreamTransformation`, which in turn is derived from `StreamingOperator`. We can also catch from figure 4.1 that transformations, along with sources and sinks, represent the three different types of streaming operators. Part of our extension of the profile can already be seen in the figure: the stereotype `FlinkTensorflowTransformation` is in fact the new stereotype that enables us to include Tensorflow functionalities in our final Flink applications.

Similarly, figure 4.2 illustrates the set of defined sources: each applicable

4.2 The extension of StreamGen: StreamGen-Tensorflow

stereotype belongs to the family of `DataSource`, which again can be seen as being derived from the `StreamingOperator` type. In this case we introduced the new stereotype `FlinkBinaryFileSource`, which is crucial for reading images and loading them into the Flink pipeline.

Rationale

It's important to describe the decision-making behind the addition of these new stereotypes. After converting the example of Flink-Tensorflow in Java (described in Section 4.1), it was time to create a UML graph that was able to correctly model such application. Logically, it was best to take advantage, as much as we could, of what was already made available with the pre-existing stereotypes of StreamGen, and then come up with new stereotypes whenever a particular aspect of the application could not be modeled otherwise. We wanted to set up a pipeline that received incoming images, so we checked if any of the available source stereotypes could handle that and found that no one was able to. This kind of limitation wasn't really coming from StreamGen itself, but rather from Flink: the API's list of pre-fabricated sources, as of 2020, mainly covers the reading of inbound textual types of file, and not much else. Nevertheless, it still provides a more generic Java interface that could be adapted to create new types of custom sources, so we used that to shape our own personalized source that was capable of reading images as an array of generic bits and load them to the Flink pipeline; hence, the introduction of `FlinkBinaryFileSource`. As for the transformation operator in charge of feeding the image to the Tensorflow model and produce the desired recognition, we clearly had to generate a whole new transformation operator (along with the relative stereotype in StreamGen), so we came up with `FlinkTensorflowTransformation`. Finally, the results of the images being recognized had to be written down and saved into a text file, so we were able to take advantage of the pre-existing sink stereotype `TextFileSink` provided by StreamGen.

Application

We can now take a look at the aforementioned example and see how old and new stereotypes work together to create an application that receives a stream

Design of the problem solution

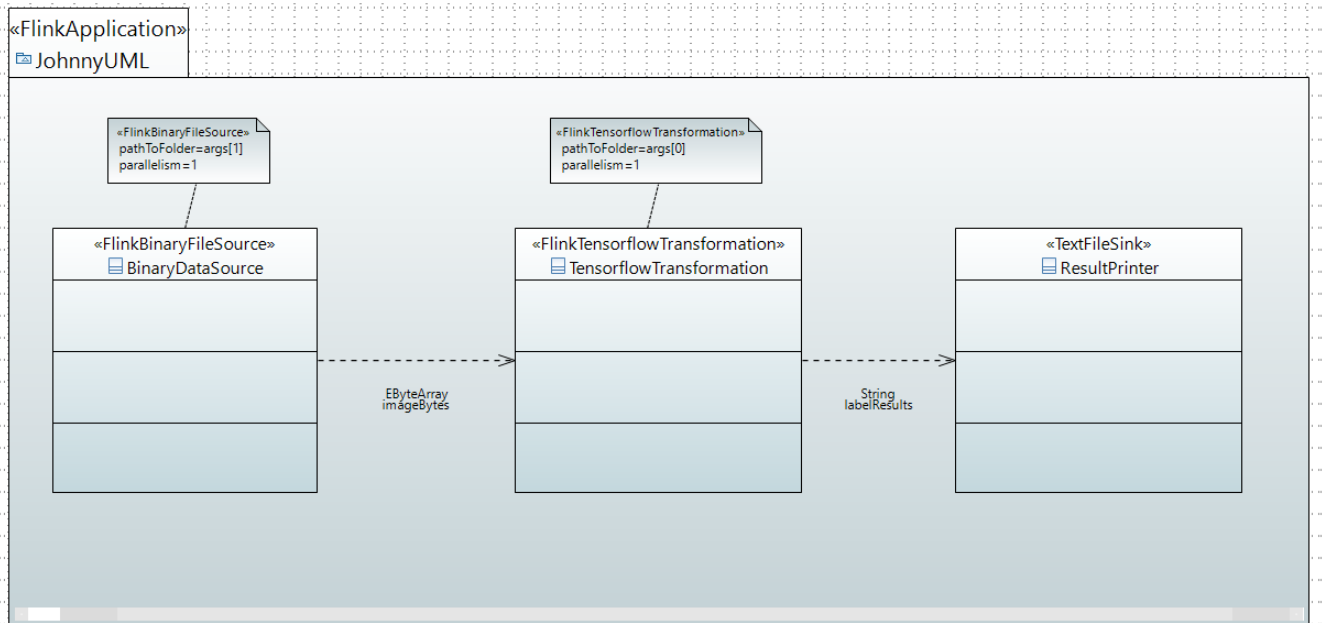


Figure 4.3: UML diagram of the streaming image recognition example

of images, and recognizes them in real time. Fig 4.3 shows the UML diagram that models such example: as previously described, we can see that the pipeline begins with the new stereotype for reading images (`FlinkBinaryFileSource`), which are passed as arrays of bites to the new stereotype that takes care of the recognition process (`FlinkTensorflowTransformation`), which in turn delivers the results to the sink that prints them into a text file (`TextFileSink`). You can notice that the whole pipeline is enclosed in a `FlinkApplication` context, which tells the parser that the resulting application will need to be implemented for working in a Flink environment (as opposed to, for instance, a Spark environment, which is also supported by StreamGen).

4.2.2 Code generator logic update

Updating the logic that converts the stereotypes found in a model into Java code is very straightforward: every new stereotype introduced in StreamUML must be provided with an Acceleo template. In this context, a template is a piece of pseudo-code that, depending on its relative stereotype in a UML model, is turned into Java code during the conversion phase (section Stream-

4.2 The extension of StreamGen: StreamGen-Tensorflow

Gen in Chapter 2 provides an example of that). Since two new major stereotypes, `FlinkBinaryFileSource` and `FlinkTensorflowTransformation`, were introduced, we needed to add two main templates, one for each. The easiest way of showing how these two new Acceleo templates operate is to present the final code that is generated from the UML diagram from Figure 4.3, and then see how each portion of code corresponds to its relative UML stereotype and Acceleo template.

```
1 public static void main(String[] args) throws Exception {
2
3     StreamExecutionEnvironment env = StreamExecutionEnvironment
4     .getExecutionEnvironment();
5
6     /*FLIK BINARY FILE SOURCE*/
7     ImageInputFormat inputFormat = new ImageInputFormat();
8     DataStreamSource<byte []> imageBytes = env.readFile(
9     inputFormat, args [1], FileProcessingMode.
10    PROCESS_CONTINUOUSLY,1000);
11
12    /*TENSORFLOW TRANSFORMATION*/
13    TensorflowTools t = new TensorflowTools();
14    byte [] graphDef = t.readAllBytesOrExit(Paths.get(args [0], "
15    tensorflow_inception_graph.pb"));
16    List<String> labels = t.readAllLinesOrExit(Paths.get(args
17    [0], "imagenet_comp_graph_label_strings.txt"));
18    DataStream<Tensor<Float>> tensor = imageBytes.map(new
19    MapFunction<byte [],Tensor<Float>>() {
20
21        @Override
22        public Tensor<Float> map(byte [] imageBytes) throws
23        Exception {
24            return t.constructAndExecuteGraphToNormalizeImage(
25            imageBytes);
26        }
27    }).setParallelism(2);
28
29    DataStream<float []> probabilities = tensor.map(new
30    MapFunction<Tensor<Float>,float []>() {
31
32        @Override
```

Design of the problem solution

```
24     public float[] map(Tensor<Float> tensor) throws Exception
25     {
26         return t.executeInceptionGraph(graphDef, tensor);
27     }
28     }).setParallelism(2);
29
30     DataStream<String> result = probabilities.map(new
31     MapFunction<float[],String>(){
32
33         @Override
34         public String map(float[] probabilities) throws Exception
35         {
36             int[] bestIndexes = t.bestThreeProbabilities(
37             probabilities);
38             return new String(String.format("Most likely: %s\n",
39             labels.get(bestIndexes[0])));
40         }
41     });
42
43     /*TEXT FILE SINK*/
44     result
45     .writeAsText(args[1] + "\\output\\results.txt", FileSystem.
46     WriteMode.OVERWRITE)
47     .setParallelism(1);
48     env.execute();
49 }
```


Listing 4.1: Final Java code of the image recognition example

Listing 4.1 contains the working Java code for the image recognition example. The markers on line 5, 9 and 38 show how each portion of code corresponds to a stereotype in the UML diagram of the same example. Let's start by focusing on the code generated by the `FlinkBinaryFileSource` stereotype; we can see from figure 4.4 how the parser generates the final code through the Accileo template: a new input format is created in order to read images that are transferred to a specified folder as series of bytes. The template names the new streaming source by looking at the label of the outgoing edge from the stereotype in the UML diagram (which is called `imageBytes` in our case), then extrapolates the path of the specified folder to monitor for incoming

4.2 The extension of StreamGen: StreamGen-Tensorflow

images again by looking at the stereotype from the diagram and searching for the parameter `pathToFolder` (which is, in our case, `args[1]`, since we want to insert the path of the folder as a parameter when running the example from command line). You can notice that the final portion of template code (assigned to set the number of threads for the operation) didn't trigger since the parallelism, unless changed by parameter in the UML diagram, is set by default to 1.

```
[template public generateFlinkBinaryFileSource(aClass : Class)]
ImageInputFormat inputFormat = new ImageInputFormat();
DataStreamSource<byte[ '[ ' /][ ' ]' /]> [getOutputNames(aClass) ->first()] =
env.readFile(inputFormat, [getStereotypeProperty(aClass, 'FlinkBinaryFileSource', 'pathToFolder')/],
FileProcessingMode.PROCESS_CONTINUOUSLY,1000)
[if getParallelism(aClass, 'FlinkBinaryFileSource').toString().toInteger() > 1 ]
.setParallelism([getParallelism(aClass, 'FlinkBinaryFileSource')/]);
[else]
;
[/if]
[/template]
```



```
/*FLIK BINARY FILE SOURCE*/
ImageInputFormat inputFormat = new ImageInputFormat();
DataStreamSource<byte[]> imageBytes = env.readFile(inputFormat, args[1], FileProcessingMode.PROCESS_CONTINUOUSLY,1000);
```

Figure 4.4: Conversion from the Acceleo source template to the final Java code of `FlinkBinaryFileSource`

The second template introduced for `FlinkTensorflowTransformation`, which is in charge of feeding the Tensorflow model with the images and output a recognition, can be found in Figure 4.5. The template calls for a number of methods that utilize the Tensorflow library in order to pre-process the image bytes, feed them to the model, and generate an output label, which represent the most likely object that resembles the images from a pre-existing database of labeled images.

Design of the problem solution

```
[template public generateFlinkTensorflowTransformation(aClass : Class)
{input : DirectedRelationship = getInputs(aClass)->first();}]
    TensorflowTools t = new TensorflowTools();
    byte[ '[' /][ ']' /] graphDef = t.readAllBytesOrExit(Paths.get([getStereotypeProperty(aClass, 'FlinkTensorflowTransformation',
    'pathToFolder')/], "tensorflow_inception_graph.pb"));
    List<String> labels = t.readAllLinesOrExit(Paths.get(args[ '[' /][ ']' /], "imagenet_comp_graph_label_strings.txt"));
    DataStream<Tensor<Float>> tensor = [input.eGet('name')/].map(new MapFunction<byte[ '[' /][ ']' /],Tensor<Float>>(){

        @Override
        public Tensor<Float> map(byte[ '[' /][ ']' /] imageBytes) throws Exception {
            return t.constructAndExecuteGraphToNormalizeImage(imageBytes);
        }
    });[if getParallelism(aClass, 'FlinkTensorflowTransformation').toString().toInteger() > 1 ]
    .setParallelism([getParallelism(aClass, 'FlinkTensorflowTransformation')/]);
    [else]
    ;
    [/if]

    DataStream<float[ '[' /][ ']' /]> probabilities = tensor.map(new MapFunction<Tensor<Float>,float[ '[' /][ ']' /]>(){

        @Override
        public float[ '[' /][ ']' /] map(Tensor<Float> tensor) throws Exception {
            return t.executeInceptionGraph(graphDef, tensor);
        }
    });[if getParallelism(aClass, 'FlinkTensorflowTransformation').toString().toInteger() > 1 ]
    .setParallelism([getParallelism(aClass, 'FlinkTensorflowTransformation')/]);
    [else]
    ;
    [/if]

    DataStream<String> [getOutputNames(aClass)->first()/] = probabilities.map(new MapFunction<float[ '[' /][ ']' /],String>(){

        @Override
        public String map(float[ '[' /][ ']' /] probabilities) throws Exception {
            int[ '[' /][ ']' /] bestIndexes = t.bestThreeProbabilities(probabilities);
            return new String(String.format("Most likely: %s\n", labels.get(bestIndexes[ '[' /][ ']' /]]));
        }
    });
};
[/template]
```

Figure 4.5: The Acceleo template for FlinkTensorflowTransformation

Chapter 5

Project evaluation

5.1 Goal of the evaluation

In the previous chapter we described the Tensorflow extension that we applied to StreamGen by presenting a practical example of image recognition that could be generated with it; in a similar fashion we are going to evaluate the project by testing if and how the same extension is capable of generating a different example.

Specifically, we are going to verify:

- Whether the new stereotypes introduced are sufficiently generic that they can be used for generating different applications in the same context of machine learning, without having to apply further updates to such extension (such as the introduction of additional stereotypes).
- In case they aren't generic enough, what needs to be added for the extension to generate different examples, and how much of these additions can be utilized on multiple use cases.
- How much the system is scalable when being expanded with supplementary updates.

5.2 Application description

The example we are going to show originates from a project developed by three research units from the Dipartimento di Elettronica, Informazione e Bioingeg-

Project evaluation

neria of Politecnico di Milano, called **SnowWatch**, which is part of the project Sodalite (funded by European Union’s Horizon 2020 research and innovation programme) [17][13].

SnowWatch is an application for deriving the amount of snow coverage of mountains, by extrapolating the necessary data from public images of those mountains. Part of the mechanism that manages to derive the relevant information makes use of machine-learning algorithms; specifically, these algorithms are used extract the skyline silhouette of a particular mountain depicted in a photo, which is a critical information that the program will then need to use for the snow coverage prediction. The process of skyline extraction is what we are going to replicate with our StreamGen extension: when adapted to a Flink environment, the resulting application should be able to receive incoming images of mountains, extract their skylines, and then store them in new image files all during a live execution (as with the image recognition example previously described).

Figure 5.1 shows two instances of skyline extraction from the image of a mountain.

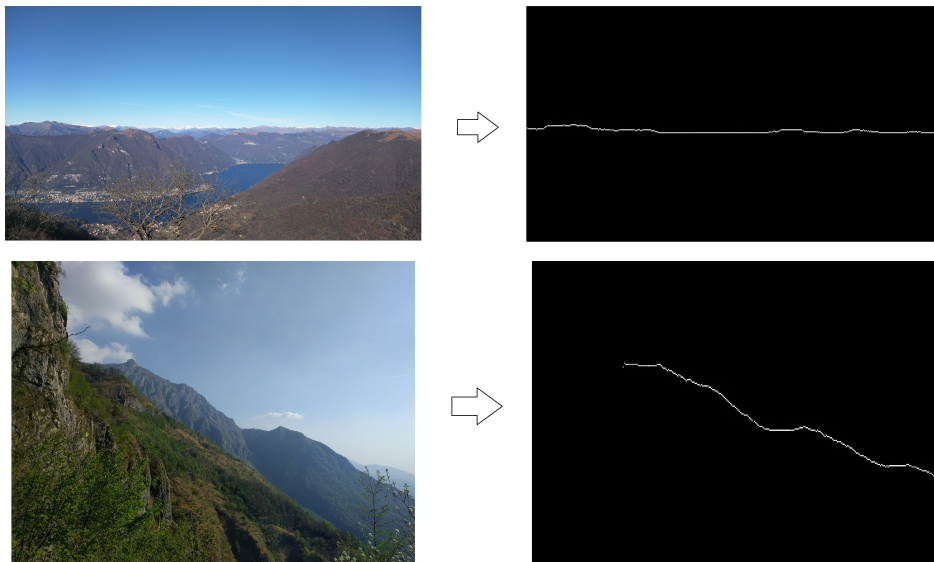


Figure 5.1: UML diagram of the skyline extraction example

5.3 Implementation

In Chapter 3, we explained the purpose and the behavior of the `FlinkTensorflowTransformation` stereotype: the stereotype manages the Tensorflow functionalities required to realize the image recognition example we previously presented. As such, its relative Acceleo template calls a number of common Tensorflow libraries, as well as a class that we created to pre-process the image bytes so that they could be successfully fed to the Tensorflow model. This kind of pre-processing, necessary for every piece of data that needs to be interpreted by a model, is unique for each scenario: our previous case of image recognition required a specific one, and for the same reason we need to add an additional pre-processing algorithm for each new case that we want to replicate. Consequently, we need to add a new pre-processing class (designed for our newest example) and make sure that the same Acceleo template is able to call this class when its stereotype is utilized in the UML model of the skyline extraction application.

Stereotype implementation

Figure 5.2 shows the UML diagram required to model the skyline extraction application: as you can notice, the stereotype in charge of reading the inbound images, `FlinkBinaryFileSource`, was fit to be used in the context of this new example, and the stereotype for applying the Tensorflow functionalities is the same one that we used in the image recognition example. However, a new parameter had to be introduced to the latter stereotype, as we needed to define what type of pre-processing algorithm had to be called for the purpose of the current application (in our case `selectedAlgorithm = skylineExtraction`).

Acceleo template implementation

Unfortunately, we cannot show the source code of the updated template for privacy reasons related to the confidentiality of the SnowWatch project, but suffice it to say that the Acceleo template was modified accordingly: the template contains both the pre-processing algorithm for the image recognition example and the one for the skyline extraction; either of which is selected through an if statement connected to the parameter specified in the stereotype.

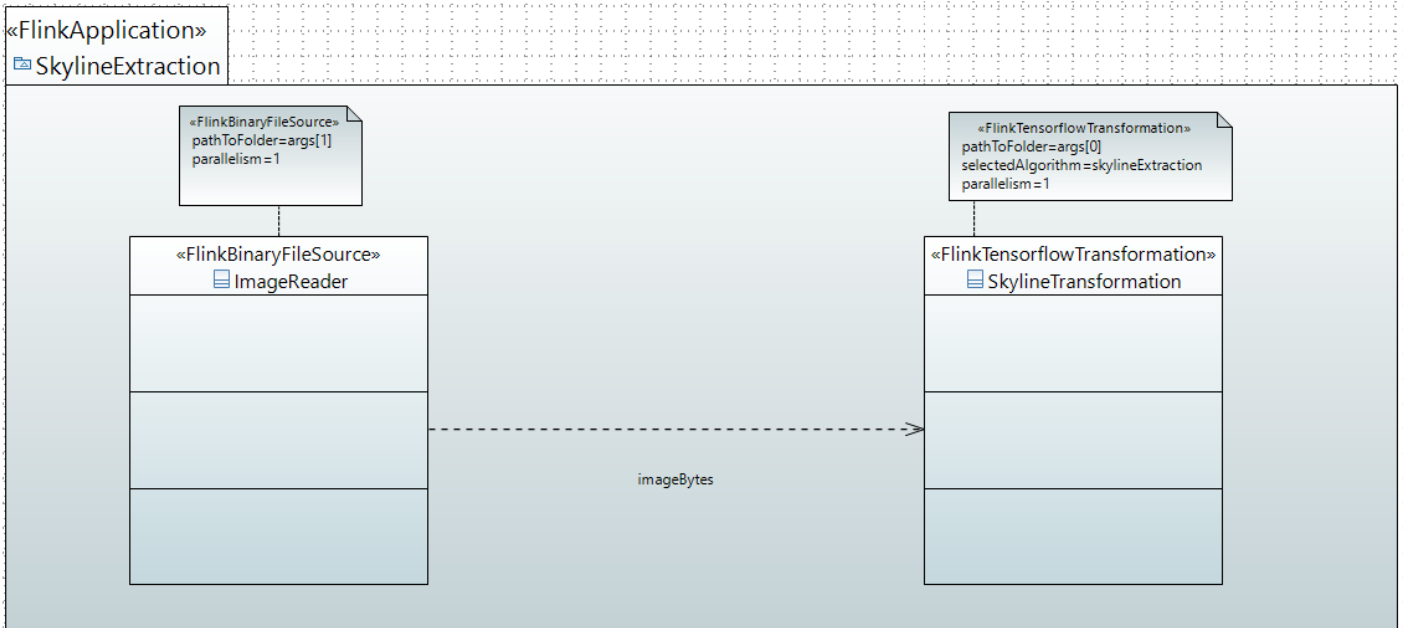


Figure 5.2: UML diagram of the skyline extraction example

5.4 Evaluation

The results of our implementation of a different use case scenario bring us to a number of considerations:

- It is clear that the stereotype that contains the Tensorflow functionalities, which is at the core of our StreamGen extension, suffers from a considerable generalization problem: since every machine-learning model requires a different pre-processing algorithm (that sets up the correct inputs to be fed to the model), every application that makes use of an unprecedented model will not be able to be expressed through StreamGen, unless the stereotype itself is updated to take into account of the new algorithms. On the other hand, the stereotype that manages the reading of input files proved to be sufficiently generic.
- Although lacking in generalization capabilities, the system as a whole shows a good level of scalability: the addition of new stereotypes to

the pre-existing ontology represented by the UML profile was completed without obstacles, as well as the update on the Tensorflow stereotype for the second example. Moreover, the particular choice of enlarging the Tensorflow stereotype (and its template) to accommodate for the latest use case wasn't the only option: as an alternative we could also have decided to add a separate stereotype, so that each stereotype introduced would have been in charge of managing each Tensorflow model execution.

- On a more technical note, the Acceleo template related to the Tensorflow transformation stereotype stores the portion of code that builds the Flink transformation, but not the actual classes that perform the required pre-processing algorithms, as they would be too big to be directly stored by the template itself. At the moment, those auxiliary classes need to be manually provided to the final application after the code generation phase is complete, but a future work would consist in packaging these classes in a library section of the StreamGen project as a Jar file, requiring only from the user to install these files to the local Operative System, before using the Tensorflow functionalities of StreamGen (at which point they would be automatically retrieved via a simple import reference in the main class of the generated code).

Chapter 6

Conclusions and Future Works

Summary

Our main research objective was to analyze the current technological frontier in the field of heterogeneous programming and develop a solution that would advance in some of its most notable challenges.

We discovered that there is plenty of frameworks, platforms and tools that are designed to facilitate the user by elevating from the low level of abstraction that is naturally required to operate with heterogeneous devices (with a particular focus on GPU exploitation), but each of them has its own limitations, indicating that there is still a big margin of potential improvement to be made.

With our developed project we tried to combine the strengths of what we analyzed into a single solution, starting from an experimental modeling tool, StreamGen, that was able to generate Flink applications from UML diagrams, and expanding it with machine-learning capabilities provided by Tensorflow, a popular API developed by the Google Brain Team.

We discussed the results of this experimentation, highlighting the biggest design limitations that we encountered during the process, mainly the compatibility issues between a dataflow framework like Flink and the process of machine-learning model training, and the low generalization capability of the newly introduced stereotypes.

Future Works

Finally, we believe that there is a number of possible future developments to be added to the project:

Conclusions and Future Works

- A more thorough investigation on whether the training features provided by Tensorflow could be feasibly integrated within the pipelined approach related to Flink, as the potential benefit in terms of GPU acceleration could be substantial.
- Working on a way to refactor the Tensorflow transformation stereotype in order to be more generalizable for different machine-learning models and subsequent algorithms.
- A better organization of the dependency system behind our SteamGen-Tensorflow extension, which still needs some manual, post-generation adjustments.

The field of heterogeneous computing has immense potential, especially if we consider the growing computational demands of Big Data and AI systems, and we hope to have contributed to its development.

Bibliography

- [1] Adlink. Heterogeneous computing for artificial intelligence at the edge, 2019. URL <https://532386f9a72d1dd857a8-41058da2837557ec5bfc3b00e1f6cf43.ssl.cf5.rackcdn.com/wp-content/uploads/2019/09/Heterogeneous-Computing.pdf>.
- [2] Brian Bayley. Why parallelization is so hard. URL <https://semiengineering.com/why-parallelization-is-so-hard/>.
- [3] Håkan Forsberg. Safedeeep: Dependable deep learning for safety-critical airborne embedded systems. URL <https://www.mdh.se/en/malardalen-university/research/research-projects/safedeeep-dependable-deep-learning-for-safety-critical-airborne-embedded-syst>
- [4] Apache Software Foundation. Apache flink[®] — stateful computations over data streams. URL <https://flink.apache.org/>.
- [5] Mark Harris. An even easier introduction to cuda, 2017. URL <https://developer.nvidia.com/blog/even-easier-introduction-cuda/>.
- [6] James Larus Herb Sutter. Software and the concurrency revolution, 2005. URL <https://dl.acm.org/doi/10.1145/1095408.1095421>.
- [7] Imagimob. What is edge ai?, March 2018. URL <https://www.imagimob.com/blog/what-is-edge-ai>.
- [8] Edward Kandrot Jason Sanders. Cuda by example, 1987. URL <https://developer.nvidia.com/cuda-example>.

BIBLIOGRAPHY

- [9] Alexey Kolesnichenko. Seamless heterogeneous computing: Combining gpgpu and task parallelism, 2016. URL <https://www.research-collection.ethz.ch/handle/20.500.11850/156142>.
- [10] Paul Barham Eugene Brevdo Zhifeng Chen Craig Citro Greg S. Corrado Andy Davis Jeffrey Dean Matthieu Devin Sanjay Ghemawat Ian Goodfellow Andrew Harp Geoffrey Irving Michael Isard Yangqing Jia Rafal Jozefowicz Lukasz Kaiser Manjunath Kudlur Josh Levenberg Dan Man' e Rajat Monga Sherry Moore Derek Murray Chris Olah Mike Schuster Jonathon Shlens Benoit Steiner Ilya Sutskever Kunal Talwar Paul Tucker Vincent Vanhoucke Vijay Vasudevan Fernanda Vi'egas Oriol Vinyals Pete Warden Martin Wattenberg Martin Wicke Yuan Yu Mart'ın Abadi, Ashish Agarwal and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015. URL <https://research.google/pubs/pub45166/>.
- [11] Damian Andrew Tamburri Michele Guerriero, Elisabetta Di Nitto. Streamgen: Model-driven development of distributed streaming applications. URL <https://www.computer.org/csdl/proceedings-article/mise/2018/573501a057/13bd1sx4Zsp>.
- [12] NYU-CDS. Introduction to gpus. URL <https://nyu-cds.github.io/python-gpu/01-introduction/>.
- [13] Darian Frajberg Sergio Herrera Piero Fraternali, Rocio Torres. Snow use case, 2019. URL <https://www.sodalite.eu/water-availability-prediction-mountains-images>.
- [14] Lavanya Subramanian Gabriel H. Loh Onur Mutlu Rachata Ausavarungnirun, Kevin Kai-Wei Chang. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems, 2012. URL https://users.ece.cmu.edu/~omutlu/pub/staged-memory-scheduling_isca12.pdf.
- [15] Arkaprava Basu Shuai Che and Jonathan Gallmeier. Challenges of programming a system with heterogeneous memories and heterogeneous processors – a programmer's view. URL <https://dl.acm.org/doi/10.1145/2989081.2989097>.

BIBLIOGRAPHY

- [16] Mikael Sjödin. Hero: Heterogeneous systems - software-hardware integration. URL http://www.es.mdh.se/projects/511-HERO_Heterogeneous_systems__software_hardware_integration.
- [17] Sodalite. Initial implementation and evaluation of the sodalite platform and use cases, 2020. URL https://sodalite.eu/sites/default/files/sodalite/public/content-files/articles/D6_2-Initial_implementation_and_evaluation_of_the_SODALITE_platform_and_use_cases.pdf.
- [18] Wikipedia. Amdahl's law. URL https://en.wikipedia.org/wiki/Amdahl's_law.
- [19] Eron Wright. Introducing flink tensorflow, 2017. URL <https://sf-2017.flink-forward.org/index.html?p=1922.html>.