



PEDRO DE ALMEIDA AMARAL RAMOS VALENTE

Bachelor in Computer Science

A CUDA BACKEND FOR MARROW AND ITS OPTIMISATION VIA MACHINE LEARNING

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
July, 2022



DEPARTMENT OF
COMPUTER SCIENCE

A CUDA BACKEND FOR MARROW AND ITS OPTIMISATION VIA MACHINE LEARNING

PEDRO DE ALMEIDA AMARAL RAMOS VALENTE

Bachelor in Computer Science

Adviser: Hervé Miguel Cordeiro Paulino
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

July, 2022

A CUDA backend for Marrow and its Optimisation via Machine Learning

Copyright © Pedro de Almeida Amaral Ramos Valente, NOVA School of Science and Technology, NOVA University Lisbon.

The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

ACKNOWLEDGEMENTS

I would like to thank the following people, without whom I don't think I would have been able to finish this thesis:

My adviser Prof. Hervé P. for his exceptional help and guidance when I needed it, and his infinite patience when I was being unproductive.

Prof. João M. Lourenço for providing the \LaTeX template[14] used to write this thesis.

My friends and fellow MSc colleagues who always supported me and kept me motivated.

My family, specially my mother who allowed me to go through my masters degree and helped me focus.

*“You cannot teach a man anything; you can only help him
discover it in himself.” (Galileo)*

ABSTRACT

In the modern days, various industries like business and science deal with collecting, processing and storing massive amounts of data. Conventional CPUs, which are optimised for sequential performance, struggle to keep up with processing so much data, however GPUs, designed for parallel computations, are more than up for the task.

Using GPUs for general processing has become more popular in recent years due to the need for fast parallel processing, but developing programs that execute on the GPU can be difficult and time consuming. Various high-level APIs that compile into GPU programs exist, however due to the abstraction of lower level concepts and lack of algorithm specific optimisations, it may not be possible to reach peak performance. Optimisation specifically is an interesting problem, optimisation patterns very rarely can be applied uniformly to different algorithms and manually tuning individual programs is extremely time consuming.

Machine learning compilation is a concept that has gained some attention in recent years, with good reason. The idea is to have a model trained using a machine learning algorithm and have it make an estimate on how to optimise an input program. Predicting the best optimisations for a program is much faster than doing it manually, in works making use of this technique, it has shown to also provide even better optimisations.

In this thesis, we will be working with the Marrow framework and develop a CUDA based backend for it, so that low-level GPU code may be generated. Additionally, we will be training a machine learning model and use it to automatically optimise the CUDA code generated from Marrow programs.

Keywords: code generation, Marrow, GPU, CUDA, machine learning, compilation

RESUMO

Hoje em dia, várias indústrias como negócios e ciência lidam com a coleção, processamento e armazenamento de enormes quantidades de dados. CPUs convencionais, que são otimizados para processarem sequencialmente, têm dificuldade a processar tantos dados eficientemente, no entanto, GPUs que são desenhados para efetuarem computações paralelas, são mais que adequados para a tarefa.

Usar GPUs para computações genéricas tem-se tornado mais comum em anos recentes devido à necessidade de processamento paralelo rápido, mas desenvolver programas que executam na GPU pode ser bastante difícil e demorar demasiado tempo. Existem várias APIs de alto nível que compilem para a GPU, mas devido à abstração de conceitos de baixo nível e à falta de otimizações específicas para algoritmos, pode ser impossível obter o máximo de eficiência. É interessante o problema de otimização, pois na maior parte dos casos é impossível aplicar padrões de otimização uniformemente em diferentes algoritmos e encontrar a melhor maneira de otimizar um programa manualmente demora bastante tempo.

Compilação usando aprendizagem automática é um conceito que tem ficado mais popular em tempos recentes, e por boas razões. A ideia consiste em ter um modelo treinado através com um algoritmo de aprendizagem automática e usa-lo para ter uma estimativa das melhor otimizações que se podem aplicar a um dado programa. Prever as melhores otimizações com um modelo é muito mais rápido que o processo manual, e trabalhos que usam esta técnica demonstram obter otimizações ainda melhores.

Nesta tese, vamos trabalhar com a *framework* Marrow e desenvolver uma *backend* de CUDA para a mesma, de forma a que esta possa gerar código de baixo nível para a GPU. Para além disso, vamos treinar um modelo de aprendizagem automática e usa-lo para otimizar código CUDA gerado a partir de programas do Marrow automaticamente.

Palavras-chave: geração de código, Marrow, GPU, CUDA, aprendizagem automática, compilação

CONTENTS

List of Figures	xii
List of Tables	xiii
Glossary	xv
Acronyms	xvi
1 Introduction	1
1.1 Motivation	1
1.2 Problem	2
1.3 Solution	2
1.4 Contributions	3
2 State of the Art	4
2.1 GPU	4
2.2 CUDA	6
2.2.1 Basics	7
2.2.2 From CUDA to Hardware	8
2.2.3 Kernel Optimisation	9
2.2.4 Host Optimisation	11
2.3 Machine Learning Compilation	12
2.3.1 Feature Extraction and Reduction	13
2.3.2 Machine Learning Models	14
2.3.3 Validation	16
2.3.4 Exploration Techniques	16
2.3.5 Related Works Using Machine Learning Compilation	17
2.4 Concluding Remarks	20
3 Marrow	21
3.1 Using the API	21

3.2	Structure and Execution	23
3.2.1	Example Execution	25
4	A CUDA Backend	27
4.1	Management	27
4.1.1	Memory	28
4.1.2	Synchronization	28
4.1.3	Example	29
4.2	Kernel Execution	30
4.2.1	Map	32
4.2.2	Reduction	36
4.2.3	Scan	37
4.2.4	Filter	38
5	Machine Learning-Assisted Compilation	39
5.1	Optimisations	39
5.1.1	Thread Coarsening	40
5.1.2	Index Partitioning	42
5.2	Features	43
5.2.1	Generating the Dataset	44
5.3	Model	47
5.3.1	Issues using OpenNN	47
5.4	Integrating Predictions With Generated Kernels	49
6	Evaluation	52
6.1	Objectives	52
6.2	Testing methodology	54
6.3	Testing environment	55
6.4	Assess the performance of the CUDA backend	55
6.4.1	How does the Marrow CUDA backend perform, compared to just CUDA?	56
6.4.2	How does the Marrow CUDA backend perform, compared to the preexisting OpenCL backend?	58
6.5	Assess the accuracy of the autotuner	59
6.5.1	Datset generation context	59
6.5.2	Overall results	60
6.5.3	Thread coarsening model	61
6.5.4	Index generation model	61
6.6	Assess the effectiveness of the autotuner	62
6.6.1	How high is the speedup gained by kernels optimised using the autotuner?	62
6.6.2	How long does the autotuner take to make a prediction?	63

6.6.3	Is the whole execution faster with the autotuner?	63
6.7	Concluding remarks	64
7	Conclusions and Future Work	67
7.1	Conclusions	67
7.2	Future Work	68
	Bibliography	69
	Appendices	
	Annexes	

LIST OF FIGURES

2.1	NVIDIA GA102 GPU architecture diagram.	5
2.2	GA102 streaming multiprocessor diagram.	6
2.3	Examples of warp memory access patterns, the arrows represent a thread access and the greyed out blocks the segments that will be loaded.	10
3.1	Illustration of Marrow’s internal structure.	23
3.2	Illustration of the expressions generated by the example in Listing 2, represented as Abstract Syntax Trees (ASTs).	25
3.3	Table representing how the example in Listing 2 would be executed in Marrow, with the flow of execution going from left to right.	26
4.1	Illustration of how the example in Listing 2 would be executed in Marrow, while using the CUDA backend.	29
5.1	Example with a container of size 85 being processed by a map skeleton with a thread coarsening of 8.	42
5.2	Schematic of the 2 models used in the autotuner.	48
6.1	Box charts of relative kernel execution times for expressions within the dataset, optimised using the autotuner and using optimal parameters.	65
6.2	Box charts of relative kernel execution times for expressions outside the dataset, optimised using the autotuner and using optimal parameters.	66

LIST OF TABLES

2.1	Summary of related works in Machine Learning Compilation	17
6.1	Testing environment machines.	55
6.2	Execution times of the Marrow CUDA backend, relative to pure CUDA (positive number means a speed down and negative means a speed up).	56
6.3	Kernel execution times of the Marrow CUDA backend, relative to pure CUDA (positive number means a speed down and negative means a speed up).	57
6.4	Execution times of the Marrow CUDA backend, relative to the Marrow OpenCL backend (positive number means a speed down and negative means a speed up).	58
6.5	Table with the execution time (in nanoseconds) of each operation in each testing machine, and the assigned cost class.	60
6.6	Accuracy values of the autotuner model predicting optimisation parameters for expressions present in the dataset used during training.	60
6.7	Accuracy values of the autotuner model predicting optimisation parameters for expressions outside of the dataset used during training.	61
6.8	Relative execution time of expressions optimised using the autotuner.	63

LIST OF LISTINGS

1	Two snippets of code that perform the addition of two vectors, one is a sequential implementation on the CPU (lines 1-5) and the other a parallel implementation on the GPU using CUDA (lines 8-28).	8
2	An example of performing some calculations with Marrow.	22
3	Marrow's method of launching CUDA kernels.	31
4	Definition of modular CUDA functions to be passed as kernel parameters.	33
5	Marrow maps' CUDA kernel signatures.	34
6	Example of how Marrow's code should translate to CUDA kernels, after all the template specialisations.	35
7	Code snippet of the CUDA filter kernel implementation.	38
8	Code snippet of the 1 dimensional map kernel with optimisations.	41
9	Code snippet of the dataset generator.	46
10	Code snippet of a solution to run an expression using optimisations only known at runtime.	50
11	Definition of the generic cache entry, and a specialised entry in the cache.	51

GLOSSARY

classification	Attribution of one or more predetermined values (classes) to something, based on its' characteristics. 14
float	Single-precision floating-point number. 21
pragma	A directive to access compiler specific extensions during the compile time of a program, typically used in C/C++. 17
struct	In several programming languages, a struct is a programmer defined data type which can hold objects of different types. 24 , 30 , 32 , 49

ACRONYMS

API	Application Programming Interface 21 , 22 , 23
AST	Abstract Syntax Tree xii , 19 , 25
CGI	Computer-generated Imagery 1
CPU	Central Processing Unit 1 , 21 , 24 , 39 , 43
GPU	Graphics Processing Unit 1 , 21 , 23 , 24 , 25 , 26 , 27 , 39 , 43 , 64 , 68
NaN	Not a Number 49
SIMD	Single Instruction Multiple Data 5
SVM	Support Vector Machine 15

INTRODUCTION

1.1 Motivation

Nowadays, data on weather, traffic, businesses, people, entertainment and many other fields is being collected and processed in absurdly large quantities, for example, CERN seek to maximize data collection in their research and experiments [6], and companies like Facebook, Twitter and Netflix employ the use of big data [28] for their services. While there may not be many issues with storing extreme amounts of data, [Central Processing Units \(CPUs\)](#) struggle to keep up with processing it in a reasonable amount of time. The processing of data is done with complex algorithms, like machine learning models, meaning not only the number of computations to be done is huge, but each individual computation may be very large as well. Thus, there is a need for powerful processors that can perform parallel computations on a massive scale. It is possible to simply increase processing power with a larger amount of CPUs, but that's not a very efficient solution energy wise, instead the [Graphics Processing Unit \(GPU\)](#), built for the purpose of parallel computing, has been chosen to handle many of those tasks, as it can perform them much more efficiently.

Computation using the GPU has greatly increased in popularity in recent years, previously it was mostly used for graphical computations such as video games and [Computer-generated Imagery \(CGI\)](#) rendering. While graphics processing is the main and original purpose of the GPU, it is capable of general computations, specially massively parallel computations. With the increase in demand for parallel operations over massive amounts of data in high-performance computing and other applications, with which the CPU has been struggling to keep up with, the use of GPUs has increased due to its suitability for the task. Even GPU manufacturers have noticed and reacted to this trend, NVIDIA's more recent GPU architectures have additional components that increase performance in AI computations [20] [19].

Creating a program that takes advantage of the GPU requires the use an API such as DirectX[17], Vulkan[5], OpenCL[23] or CUDA[18], which can be very difficult and daunting to use. Some APIs, such as Futhark[13], SkePU 2[8] and Dandelion[21], are

high-level, meaning they abstract away many implementation details, in order to make programming with the GPU easier, but at the cost of control, the API will either generalise or guess many implementation decisions which may not always be ideal. On the other hand, the lower-level APIs can be extremely verbose, doing less guess work and leaving most implementation details for the programmer to deal with. Additionally, there are some aspects one may have to take into account when working with a GPU, these are communication management, memory management and kernel code optimisation, all of which can greatly affect performance.

Despite the potential of not being able to achieve peak performance, programmers using GPU APIs may prefer to stick with those whose programming model is simpler or more familiar in favour of writing more concise and easier to maintain code. Unfortunately, it may be very difficult to make GPU tools be similar to the generally more accessible CPU tools in the ways they are worked with, since the hardware is not the same, and does not function in the same way.

1.2 Problem

Many tools and frameworks are built on top of lower level APIs, to enable GPU code execution through its own programming model. Some are made as wrappers, abstracting difficult concepts through high-level functions and constructs. Others do it by procedurally generating code of a low-level API, based on high-level code. Compared to wrapper APIs, this approach may be a bit more complex to implement by the framework developers, but there may be a lot more room for optimisation due to higher flexibility, as the generated code can be altered before compilation. A lot of challenges are involved with simply generating working code, simply translating CPU functions into GPU functions, line by line, does not work well, the code generator has to be aware of the context of each operation.

When implementing code generation, creating programs that work properly isn't the only hurdle to be overcome. Even more difficult than that is generating working and well optimised code, as not only the generated code has to execute correctly, the generator also has to be aware of which coding patterns should be avoided and which should be preferred for the architecture. Procedurally optimising code is extremely difficult, generally, optimisations cannot be applied uniformly to every program, they are specific to each programs specific computation and data flow. Usually, finding the best optimisations for a program is a manual process, and it takes very long to do it for even just a single program.

1.3 Solution

Using machine learning models to optimise programs is not a new concept, and it may be a solution worth considering to help with the optimisation of generated code. After

training a model, it can make predictions on what compiler flags should be used, or how to alter the generated code or even help with program scheduling during runtime. As long as programs are properly characterized and a sufficiently large training set is used to train the model, predictions should be mostly accurate, as demonstrated by various works in the field.

In this thesis we will be working with Marrow, an algorithmic skeleton based parallel programming framework, which heavily focuses on utilising the GPU. Marrow is one such framework that procedurally generates low-level code based on high-level code, and we will be implementing a new backend for Marrow, based in CUDA, so that it can generate and compile CUDA programs. We will be comparing how the CUDA backend generated programs perform compared to Marrow's other GPU backend, OpenCL.

Optimisation is obviously a big focus for a project like this, will be working to make sure that generated are automatically optimised to utilise available resources as efficiently as possible. However, besides implementing procedural optimisations, we will be developing a machine learning framework to further optimise the generated programs, by training models to predict the best optimisation parameters. After trained, the performance of programs optimised with the model will be evaluated and compared to programs optimised through default parameters and equivalent handwritten CUDA programs.

1.4 Contributions

The main contributions of our work are:

- A new functional backend for Marrow, based in CUDA
- An auto-tuner powered by a machine learning model for Marrow
- Several optimisations to be applied by the auto-tuner
- Evaluation of the CUDA backend and of the machine learning model

STATE OF THE ART

This chapter has the aim of presenting relevant technologies that will be utilised in this thesis, as well as related works such as projects making use of machine learning compilation and other high-level frameworks utilising the GPU, similarly to Marrow.

The first section is dedicated to the GPU, presenting its purpose, detailing how it works. In this document the presented architecture will be that of NVIDIA GPUs, this is because the API that will be studied and used is CUDA, which is exclusive to NVIDIA. In the second section will be presented the CUDA API, what it is, the fundamentals of working with CUDA and how to optimise CUDA programs to properly utilise the GPU.

The last section is dedicated to Machine Learning Compilation, explaining the concept and why it exists, as well as detailing different methods and algorithms that can be used. At the end, several related works in the field are explored and studied.

2.1 GPU

A GPU is a computer component specialised for, as the name implies, graphics processing or rendering, which consists of thousands of independent and somewhat simple tasks over large amounts of data. To be able to render in real time, a GPU has a few thousands of processors (named CUDA cores for NVIDIA GPUs), similar to a simplified CPU core with less capabilities, that can perform render operations in parallel.

Because a GPU is a very complex piece of hardware, in this document we will focus only on the components that are relevant to the work to be developed in our thesis. NVIDIA has developed and released many GPUs of different architectures over the years, Figure 2.1 presents the diagram of one of their most recent GPUs. While the internal structure may differ between GPUs, most should still be relatively similar and be composed of:

VRAM or video random access memory (not in the diagram), it's the largest and slowest memory of the GPU, accessible by all cores and by the host;



Figure 2.1: NVIDIA GA102 GPU architecture diagram.

L2 cache a memory considerably smaller, but also faster than the VRAM, it's accessible to all cores and is the last cache searched before an access to the VRAM is necessary, every access to VRAM will go through the L2 cache first;

GPCs or graphics processing clusters, these are composed of TPCs or texture/thread processing clusters, which in turn are composed of memory controllers and SMs or streaming multiprocessors.

The internal structure of a streaming multiprocessor (in this particular case of an SM of the GA102 GPU) is depicted in Figure 2.2, and as with the rest of the architecture, it may be different in other GPUs, but still generally similar. Each SM contains:

L1 cache which can be accessed by all of the cores inside the SM and compared to the L2 cache it is much faster and smaller, it can be programmed to be used as shared memory;

Read-only cache that is hardware-managed and like the L1 cache, is faster and smaller than the L2 cache and can be accessed by all the cores inside the SM;

SPs or streaming processors of 32 cores each, which follow the [Single Instruction Multiple Data \(SIMD\)](#) model, that is, all of its cores execute physically in parallel.

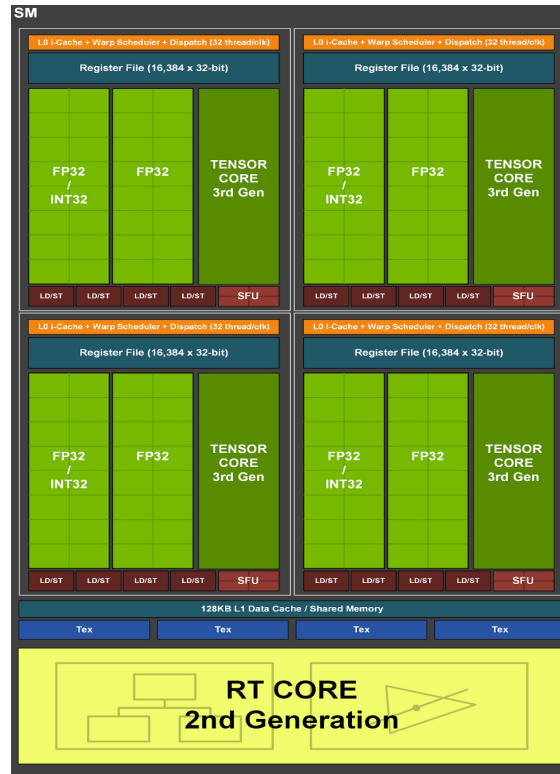


Figure 2.2: GA102 streaming multiprocessor diagram.

More recent GPUs such as those of the Turing[20] or Ampere[19] architectures (which is the case for the GA102 GPU) have additional components inside their SMs that can dramatically boost performance for some specific workloads:

Tensor cores optimised for AI computations such as matrix operations, providing a significant performance boost to AI neural network training and inferencing;

Ray tracing cores optimised to process bounding volume hierarchy traversal and intersection of scene geometry, allowing rendering with ray tracing in real time.

2.2 CUDA

Compute Unified Device Architecture or CUDA[18] is an API and a parallel computing platform developed by NVIDIA, used for general purpose computing on GPUs. When working with CUDA directly, developers may use C, C++ (which will be used in this thesis) and Fortran, otherwise some languages like Python and MATLAB can use CUDA *under the hood* during runtime and some parallel programming frameworks such as OpenACC[26] and OpenCL[23] may compile to CUDA.

2.2.1 Basics

In CUDA, the host has the role of orchestrating GPU operations, it tells the GPU what to do, and when calling asynchronous operations, it should also specify when they can be carried out to avoid conflicts. The GPU processes parallel computations, which assume the form of a computational kernel, and memory allocations or transfers according to the host specification, it handles memory operations and synchronisation within the kernels independently. Writing and launching a kernel is fundamentally the same as writing and calling a regular function, with the exceptions that in the definition the return value is always void and some additional decorators, `__global__` or `__device__`, are added to tell the compiler that the function is to be compiled into GPU machine code, and when calling, some special kernel parameters are added to specify the number of blocks and threads per block launched with the kernel (will be explained in more detail in Section 2.2.2). Kernel code is written as the execution of a single thread, similarly to code inside a *for* loop, but with every iteration running in parallel. To identify a thread within the whole kernel during runtime, CUDA provides constants for the index of the block and the size of each block, which should be the same as the values specified when launching the kernel, and an additional constant for the index of the thread within the block.

Types such as integer and floating-point numbers, and even structures, may be passed freely as parameters to a kernel, however data with variable size, such as arrays and strings, needs to be allocated and initialised on the GPU before the kernel using them is launched, the pointer to the data inside the GPU and its size are then passed as parameters. Furthermore, to obtain the results of a kernel, the required memory should be pre-allocated and its pointer, and most likely its size, passed as parameters to the kernel, so that it can write the results into the data. After the kernel is finished executing, the result data may be explicitly copied from the GPU to the host memory.

Memory operations in CUDA are done similarly to the way they're done in C. Allocations are done simply by calling *cudaMalloc* and passing an empty pointer, which will be used for the new memory, and the size of the allocation in bytes. Memory transfers are done by calling *cudaMemcpy*, which functions identically to a regular *memcpy*, but with an added parameter to indicate the direction of the copy (from host to GPU or vice versa). Just like in C, GPU memory in CUDA has to be freed manually by calling *cudaFree*.

There are a few alternatives for memory management provided by CUDA. Using pinned memory, allocations are always done on host memory, shared by both the CPU and GPU, instead of GPU memory, which means there is no need to copy the memory over to the GPU, however the accesses may be much slower. This type of allocation is only recommended for use in systems with integrated GPUs, where the CPU and GPU share the same physical memory. Unified memory is the second alternative, allocations made this way create a block of coherent memory between CPU and GPU, by allocating on both host memory and GPU memory, changes in unified memory are synchronized by the driver, eliminating the need for explicit memory copies. This type of memory doesn't

```
1 void cpuAdd(int* a, int* b, int* res, int length) {
2     for(int i = 0; i < length; i++) {
3         res[i] = a[i] + b[i];
4     }
5 }
6
7
8 __global__ void kernelAdd(const int length, int* a, int* b, int* res) {
9     int index = threadIdx.x + blockIdx.x * blockDim.x;
10    if(index < length) {
11        res[i] = a[i] + b[i];
12    }
13 }
14
15 void gpuAdd(int* a, int* b, int* res, int length) {
16     int *d_a, *d_b, *d_c;
17     int size = length * sizeof(int);
18     int tpb = 512; //threads per block
19     int num_blocks = length % tpb ? length / tpb + 1 : length / tpb;
20     cudaMalloc((void*)&d_a, size);
21     cudaMalloc((void*)&d_b, size);
22     cudaMalloc((void*)&d_c, size);
23     cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
24     cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
25     kernelAdd<<<num_blocks, tpb>>>(length, d_a, d_b, d_c);
26     cudaMemcpy(res, d_c, size, cudaMemcpyDeviceToHost);
27     cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
28 }
```

Listing 1: Two snippets of code that perform the addition of two vectors, one is a sequential implementation on the CPU (lines 1-5) and the other a parallel implementation on the GPU using CUDA (lines 8-28).

have as much potential for optimization, since the control over when transfers happen is lost, it is targeted for code that is meant to be more readable and maintainable, and where optimization isn't as much of a focus. Both pinned and unified memory still need to be manually freed after use.

Listing 1 presents a sequential implementation of a function versus its CUDA counterpart. Immediately noticeable is that the CUDA implementation is considerably larger than the sequential. Even code strictly inside the kernel, which should be equivalent to one iteration, has a few more operations to identify the thread and check if it should even do the computation, because when a kernel is launched with blocks of threads instead of an arbitrary number of threads that might be needed for a computation, more threads than necessary may be used.

2.2.2 From CUDA to Hardware

Just like with a regular program, it's important to understand how CUDA code translates to hardware. Threads are independent sequences of instructions and a GPU is made to be

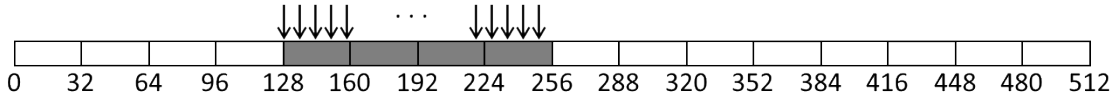
able to process thousands of threads in parallel, however as described in 2.2.1, a program running on the GPU is not executed with any arbitrary number of threads, it is instead executed through blocks of threads, all with the same size, which make up a grid that is executed by the GPU. The maximum number of threads per block is defined by the architecture. Each block is assigned to a SM and is executed only by that one SM, all the threads of a block are divided into warps of 32 threads and executed in parallel by a SP inside the SM. Warps work as the stepping unit for SMs to process, and only a number of them can be processed at a time depending on the architecture, if a block generates an amount of warps higher than the SM can process at a time, they can't all be executed in parallel and some additional iterations will be needed. Additionally, if a block does not have a number of threads multiple of 32, more resources than are necessary will be used because even if less than 32 cores are needed, a full warp is executed. No more than one warp can be processed by a SP at a time, and it will only be freed once all of the threads from the current warp are finished.

2.2.3 Kernel Optimisation

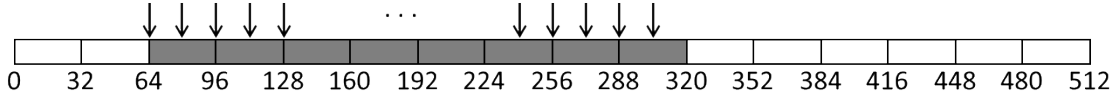
There are many ways to write code that produces the same result, and just like with any other programming language or API, when writing CUDA or working a GPU in general, some patterns should avoided, while others favoured. Special care should be taken when coding kernels as some regular programming patterns can result in very poor use of the GPU resources and a few favourable patterns may not be obvious or intuitive to use. The following optimisations focus on kernel code and patterns used for GPU computations:

- **Avoid branching :** Warps follow the SIMD model, all threads execute the same operation at the same time, when branching occurs, each and every branch must be processed individually by every thread. Because of this, branching is something that should be handled with care when writing a kernel, it can lead to multiple threads inside a warp waiting for a few to finish, or just make everything slower with a large amount of branching due to all the different paths being processed individually. In order to avoid these issues, a kernel should be implemented in such a way that there is the least amount of branching possible, and/or in such a way that branching happens only between warps instead of threads, as warps are independent from each other, even when executing the same kernel.
- **Access to Global Memory :** Data inside any kind of memory in the GPU, be it global or cached, has a minimum amount of granularity. It is divided into segments of constant size and trying to access any data in memory requires the entire segment in which it resides to be loaded. A warp being 32 threads means that a memory access in code usually translates to 32 memory accesses by all the threads (branching code is an exception). If multiple threads try to access addresses inside the same segment in memory, a single transaction is enough to satisfy all those threads as the

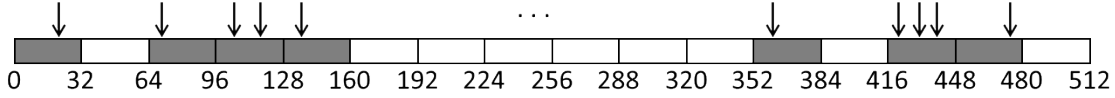
Ex. 1: Coalesced Access Pattern



Ex. 2: Stride Access Pattern



Ex. 3: Random Access Pattern



Ex. 4: Same Word Access Pattern

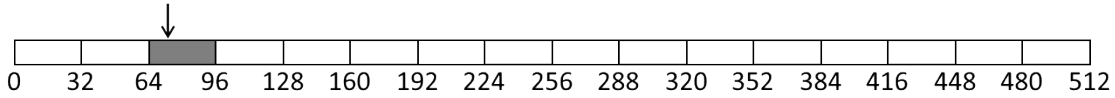


Figure 2.3: Examples of warp memory access patterns, the arrows represent a thread access and the greyed out blocks the segments that will be loaded.

segment only needs to be loaded once. The "order" in which threads access memory is irrelevant, all that matters is which addresses are requested by the threads in a warp, that is, thread X accessing A and thread Y accessing B leads to the same memory transactions as thread X accessing B and thread Y accessing A. In Figure 2.3 there are different access patterns exemplified, in the first example is depicted the ideal coalesced pattern, which is all threads accessing a contiguous block of memory, with it only the requested data is loaded, or the requested data plus one extra segment if the data is misaligned, an example of this pattern would be threads accessing elements of an array in order. Example 2 represents a stride pattern which could be exemplified by threads accessing the same member of different elements in an array of structures and example 3 represents a random pattern where each thread accesses a random position in memory, both have accesses spread out resulting in whole segments being loaded when only a small portion of data in them is needed, which should be avoided. Example 4 is a pattern where every thread is accessing the same position in memory, it should be avoided for the same reasons as examples 2 and 3, as only a small portion of a segment may be needed, but all the 32 bytes are loaded.

As implied with the stride pattern, using an array of structures is something that should be avoided, because the members of the structure will be padded in the memory, when threads of a warp access a member of different elements in the array, due to the padding the addresses are more spread out, resulting in more segments being loaded. Just like when programming with the CPU, using a structure of arrays instead is more efficient as the member arrays are contiguous in memory, when threads access different positions of member arrays, the amount of segments

loaded is minimal. It's not always possible to avoid some access patterns, but there are a few methods to mitigate their performance loss, like using the read-only cache for the data or staging the loads via shared memory.

Shared memory in CUDA is used in two different ways: statically if the size of the memory is known before runtime and always the same for every kernel execution, and dynamically if the size is only known when the kernel is launched. To use shared memory statically, all that is needed is to add the decorator `__shared__` to the type definition of a variable, like an array of constant size, inside the kernel. To use shared memory dynamically, first an array of undefined size, decorated with `extern __shared__`, is defined inside the kernel, then where the kernel is called, a third special parameter is added after the number of threads per block, defining the size of shared memory in bytes.

With CUDA, the read-only cache may be used by either, having pointer arguments of a kernel decorated with `__restrict__` in its definition, this tells the compiler no aliasing will occur with the respective memory, or by using the `__ldg()` intrinsic whenever the memory is accessed in the kernel. Obviously, memory using the read-only cache should not be written to, as the cache is incoherent with writes.

- **Thread Coarsening :** Launching threads comes with a cost, and sometimes that cost may be greater than the total cost of each thread's computation, meaning more time and resources are being spent launching threads than it is on computations. In those scenarios, each thread should be doing more work to compensate for the launch times, the most common solution is to make each do the work of two threads or more. For example, a kernel where each thread only increments an element of an array by 1 is likely to have too few computations per thread. To make it more efficient, the numbers of launched threads is divided by two and each thread increments two elements. Unlike the other kernel optimisations previously mentioned, thread coarsening isn't something that every kernel benefits from, the programmer should try and evaluate different coarsening factors to find the best value for a kernel.

2.2.4 Host Optimisation

Optimisation in CUDA isn't restricted to kernel code, even if the kernels are properly optimised, it may still be possible to utilise the available resources more efficiently. When orchestrating kernel execution and memory operations, the host should overlap communication with computation.

Transferring data to the GPU can be overlapped with transferring data from the GPU to the CPU, memory transfers and kernel executions are also tasks that can be overlapped, and even multiple kernels may be executed in parallel. When working with independent kernels, the time it takes to transfer data to the GPU may be mitigated by having memory

copied over while another kernel is executing, or transferring the output data into host memory while starting another kernel at the same time. Furthermore, while a GPU is working, it's also possible for the host device to perform other independent computations with the CPU to further increase efficiency.

With CUDA, parallelism between CPU computations, memory transfers and kernel executions is achieved through the use of asynchronous functions and streams. Memory copies have an asynchronous variant that returns immediately after getting called and can be associated with a stream. Kernels may also be associated with streams, through the use of a fourth special parameter, to be executed asynchronously. A stream is an independent flow of computation, every operation associated with one is synchronized within the same stream, and executed parallel to CPU computations and other operations associated with a different stream. To synchronize asynchronous operations and streams, the CPU may wait for the stream to finish all its associated operations, or create an event, which may be associated to a stream between other operations and act as a checkpoint, the CPU can then wait for the event to be triggered.

2.3 Machine Learning Compilation

Compilers have improved over the years, however the gains in performance aren't significant, as they struggle to keep up with advances in hardware and changes in programming paradigms. In face of this issue surged the concept of *autotuning* a compiler through the use of machine learning algorithms.

AI programs that are based on machine learning may vary greatly in the way that they function, but have some common aspects. When implementing one such program, first the features of what the program is analysing are selected, that is, what will be the "input", some learning algorithms require the data used for training to have a label or a "score" attached, and if the number of features is too high, some form of feature reduction should be used. After that, a learning algorithm is chosen to model a problem for the program, it is what defines how the program learns from data and what kind of prediction it can make from it, the performance and output vary with different models. After the implementation of the program, it is trained with a large amount of examples and then tested to verify its accuracy.

When compiling any program, beyond general improvements to the code, there are several optional optimisations offered by the compiler and while these can result in better performance, they may also have the opposite effect, depending on the program itself, which optimisations are applied and in which order they are applied. Before compiling, the effect of any selection of optimisations applied in any order to a program is unknown to a compiler, which is why those optimisations are optional and disabled by default.

Typically, finding which optimizations to use with the compiler is a manual process done through trial and error, this makes machine learning an interesting solution to help

with the issue, as it can make a prediction of the best optimizations for a given program, speeding up the process.

2.3.1 Feature Extraction and Reduction

A feature is a measurable property or a characteristic representative of some item we want to analyse, represented by a number so that it can be easily processed by a machine learning algorithm. A group of distinct features is bundled in a vector that is given as input to the algorithm, each feature should be independent and discriminant of the respective item, so that the vector is a suitable representation of it, different Items should not have the same feature vector. Very large feature vectors should be avoided as they may heavily slow down the algorithm and even reduce the accuracy of the results.

Within the context of machine learning compilation, the item we want to analyse is a program we want to compile and its features are dependant on the method of analysis. Static features denote properties observed through the source of the program, dynamic features denote properties that can only be obtained at runtime, such as performance counters, program inputs and memory values. Each type of feature has advantages and disadvantages, it's also possible to take an hybrid approach using both static and dynamic features to analyse a program. In this thesis, programs will be analysed solely through static features, as such, we will only focus on those.

Static features are collected before or during compilation, meaning the program does not need to execute, besides being an advantage by itself, it also assures that for the same program the feature vector is always the same, which is not necessarily true for dynamic features, due to randomness added by things like OS scheduling or different hardware. Static features are collected by parsing the initial source code, an intermediate representation of the compiler, the compiled machine code or any combination of the three, with an intermediate representation and the machine code having the advantage of getting basic optimisations applied to them by the compiler. Features can be extracted directly from the code or intermediate representation parse, but there is also the option of building a graph representation and extracting features from that, as they expose data and control dependencies for all operations.

As mentioned earlier, large feature vectors should be avoided, but programs can be extremely long and not be properly classifiable with a smaller amount of features. The number of features in the feature vector needs to be reduced without discarding relevant information, to do this there are two popular techniques:

- **Principal Component Analysis (PCA):** This technique aims to create a "summary" of the original feature vectors and eliminate the features of least relevance. First a number of feature vectors of different programs is analysed, new feature vectors are made with their elements being linear combinations of the original features, the new vectors are then sorted by the variance of each feature and have the first N elements, named principal components, selected to be in the final vector.

- **Exploratory Factor Analysis (EFA):** This technique aims to find correlation between different features and some hidden variables and form a new vector based on those variables, essentially forming a "compressed" version of the original vector. First a number of feature vectors of different programs is analysed, the correlation between the features and each hidden variable is obtained through the maximum likelihood method and the new vector is made with all the variables.

2.3.2 Machine Learning Models

Machine learning models are the result of training a learning algorithm through a large set of examples. The learning algorithms are able to identify patterns in training data, with which they build a model, and through that model they can make predictions on new data. Generally, models are divided into 3 subcategories:

Supervised Learning, models in this category require the feature vector of each item within the training set to be paired with a label in order for the algorithm to learn relations between features and labels, allowing it to predict the label of new unlabelled data;

Unsupervised Learning, models in this category are able to learn from data alone, without any additional information, for example, clustering models identify similarities between different items and group them together;

Reinforcement Learning, models in this category don't have a defined training or testing phase, instead, algorithms in this category interact with an environment over a period of time, while trying to find the best pattern of interaction (named policy) according to a reward function.

For this thesis, we want a model that can predict the set of compiler optimisations that provide the best speed-up for a program, so we will only focus on suitable algorithms and techniques. Optimisations are known before a new program is analysed and can be seen as discrete values, and we want to predict a set of optimisations, as such, this can be interpreted as problem of [classification](#), which we can model with a variety of supervised learning algorithms. To train the model, for each program whose feature vector is part of the training set, several different compiler optimizations sequences are used for compiling the program and have their performance measured, the best sequences are then used as labels for the feature vector. Some learning algorithms that can be used with this approach:

K-Nearest Neighbours is an algorithm that doesn't build an internal model, but instead stores every item used for training, when predicting on a new item, it will compare its features to those of every training item, and chose the class based on the K training items with the most similar features.

Linear Classifiers build models that make predictions using a linear combination of the features in the feature vector of a new item, the way the model is built depends on the exact algorithm. Some examples of Linear Classifiers are [Support Vector Machines \(SVMs\)](#), Logistic Regression and Naive Bayes Classifier.

Decision Trees have their model built as a tree structure, with each of the nodes being an if-then rule and each leaf a classification or a probability distribution of the classes, or a number if the tree is being used for regression. There are many different Decision Tree algorithms, and they can be expanded into a new one, such as a Random Forest or a Rotation Forest, by generating multiple Decision Trees using different subsets of the feature vector, having the final prediction be the mode of the classes predicted by each individual tree for classification, or the average of the individual predictions for regression.

Neural Network is an algorithm inspired by animal brains, the generated model is composed of artificial neurons (software constructs that mimic organic neurons) arranged into layers, each neuron takes one or more weighted inputs, the sum of which is fed to a local sigmoid function, the resulting value is then sent to the neurons in the next layer. The first layer's input is the feature vector and the last layer's output is the prediction.

Training a supervised model using only good compiler sequences as labels for the feature vectors is a method that has been done and tested before with successful results[4], but because the model only learns from the best examples, without any less or even bad performing ones, there may be the risk of it becoming biased and predicting suboptimal optimization sequences. To avoid this, a different solution may be tested using unsupervised learning algorithms, which can learn with any compiler sequence. Evolutionary algorithms are a subset of unsupervised learning which learn by randomly generating multiple models or "individuals", testing them according to a fitness function and selecting the best performing ones. After that a new generation of models is created, based on the selected models and with some "mutations", the new models are then tested and selected again, this process is repeated until some condition is met, like obtaining a suitable model or passing a number of iterations. For our specific problem, the fitness function may be based on the execution time of programs compiled with the optimisation sequence predicted by the model. Different evolutionary algorithms differ in the way they represent a model and how they evolve it. An example of one such algorithm is Neuro Evolution of Augmenting Topologies[22] or NEAT, it is an algorithm that generates and evolves neural networks. The evolution process is done by altering the weight variables of the neurons and the overall structure of the network.

The last approach may not be very practical, the fitness function may be very expensive and cause training to be extremely slow, as it depends on compiling a program and executing it. If this is indeed an issue, instead of directly measuring the performance of a

program for the fitness function, an auxiliary model trained to predict execution times may be used for the fitness function.

Theoretically, a solution for the problem of finding the best set of compiler optimizations using a reinforcement learning algorithm is possible, however those are more suited for problems where the objective is to interact with an environment over time, which is not the case for our problem, as such, they won't be considered for our solution.

2.3.3 Validation

A model should be trained with a large dataset to ensure the problem is properly modelled, but even then some issues may arise, the model may not generalise properly and predict on new data poorly due to too much noise, in other words, it may be overfitted. Another possible issue is selection bias, which happens if the samples in the training set are not properly randomised, and as such, the dataset is not a good representation of the population. To assess these issues and the overall accuracy of a model, validation is used.

Validation techniques are very similar in the way they work, one such technique is cross-validation. There are a few variations of cross-validation, the one that will be discussed in this document specifically is k-fold cross-validation: the process begins with dividing the training set into N equal sized subsets and shuffling them, the new dataset is then divided into K equal sized subsets, for each subset, a model is trained using the other subsets and then tested with the current one. When testing, the model is attributed a score based on the number of correct and incorrect classification for supervised learning classification algorithms, or on the results of a fitness function for unsupervised evolutionary algorithms, the scores of all the K models are then averaged to obtain the accuracy of the model trained with the whole training set.

2.3.4 Exploration Techniques

The amount of different sequences for every compiler optimization in every possible order can be an extremely large number, such that it becomes infeasible to predict the ideal compiler sequence with a model. For this reason, most studies in the field of compiler optimisation elect to disregard the order of optimisations, and focus only on finding the best set. By ignoring order, the optimisation space becomes much more manageable, but may still be hard to traverse, if the compiler has a large enough pool of optimisations, without a proper exploration strategy.

Exploration techniques are different ways to analyse and compile code, with the objective to reduce the optimisation space in some capacity, either directly, or by simplifying the targeted code. They can be used in conjunction with a machine learning to explore the optimisation space. The three main exploration techniques are:

- **Iterative Compilation:** The optimisation space is divided into multiple different

Table 2.1: Summary of related works in Machine Learning Compilation

Reference	Year	Target Architecture	Main Model	Prediction	Type of Features
[31]	2009	GPU (CUDA)	Regression Trees	Best Optimisation Set	Hybrid
[3]	2014	CPU (gcc)	Bayesian Network	Best Optimisation Set	Dynamic
[7]	2013	Any	KNN	Best Optimisation Set	Static
[15]	2014	GPU (OpenCL)	Neural Network	Coarsening Value	Static (IR)
[9]	2017	GPU (OpenCL)	Neural Network	Execution Time	Compiler Flags
[11]	2014	GPU (OpenCL)	Random Forest	Use Local Memory	Static
[10]	2017	GPU (OpenCL)	Random Forest	Use Local Memory	Static
[30]	2014	GPU (OpenCL)	Decision Trees	Use GPU or CPU	Parametrised Static
[25]	2017	GPU (OpenCL)	SVM	Schedule Kernels	Static (IR)

combinations which are then iteratively evaluated, in the end the best one, according to some function, is returned.

- **Adaptive Compilation:** Also known as profile-guided optimisation, this technique compiles and analyses the output program during runtime to find hotspots and other problematic areas in the code. Those sections are targeted for optimisation and then the program is recompiled, the program may be evaluated again afterwards.
- **Non-Iterative Compilation:** This technique focuses on finding optimisation solutions for specific problems, reducing the optimisation space by looking at one or more specific optimisations and either, ignoring them, give them a constant value or have some way of procedurally generate their values, so that those optimisations don't need to be considered when exploring the optimisation space.

2.3.5 Related Works Using Machine Learning Compilation

In our project, we intend to predict the best set of compiler optimisations directly with a model and, because we are working a GPU, evaluated programs will be characterised only through static features. Selecting which algorithm to use and which features to extract from programs is no trivial task, and isn't something that should be taken lightly as it can drastically affect the end result. This subsection is dedicated to discussing other machine learning compilation related works which focus on similar ideas, we explored those that also attempt to find the best set of optimisations or focus on GPU programs to understand which algorithms may work better for our specific purpose, and also explored those that focus on extracting static features. A summary of the discussed works can be seen in Table 2.1.

There are many different ways to auto-tune a compiler with, or without, the use of machine learning. Using a model to predict the best set of optimisations is one of the simpler approaches, it has already been studied with some success. Liu et al. [31] developed a source-to-source compiler, G-ADAPT, which optimizes a CUDA program and makes it adaptive to different inputs, and obtained significant speed-ups in some benchmarks. Programs are outfitted with various `pragmas` and is fed to the compiler, along with a set of typical inputs, the program is then characterised through data flow, loop and *pragma*

analysis. Using iterative compilation, the compiler finds the best set of parameters (which are for execution configuration, code transformation and algorithmic decisions) for each given input, while bundling all the input adapted binaries into a single program, and builds a database of tuples containing input and best parameters. After finishing the database, a Regression Tree is trained using the tuples and the resulting model integrated into the program, during runtime, the model will predict which adaptation to use for a new input.

Ashouri et al. [3] propose a framework that predicts the probability distribution function of optimisations for the GCC compiler, and then optimises programs through iterative compilation by sampling the distribution. Programs are characterized through dynamic features, reduced with PCA, and the model used is a Bayesian Network trained using tuples of program features and the respective optimisation sets which maximise performance, found by sampling the optimisation space with an uniform distribution. Using this approach, the authors were able to achieve speed-ups of up to 3, averaging 1.5, relative to standard GCC optimisation passes.

Collins et al. [7] implemented MaSiF, a tool designed to auto-tune programs making use of parallel skeleton frameworks (like Marrow). Optimisation with this tool is done through a technique using eigenvectors, obtained by applying PCA to the optimisation space, and a K-Nearest Neighbours model to select optimal parameter values from similar programs. The predictor is trained with tuples of sample program static features, which are dependent on the skeleton framework being used, and their respective near optimal optimisation set (within 5% of maximum performance), obtained through an exhaustive search or subspace sampling. The tool was experimentally shown to significantly outperform manual tuning done by experts in the tested frameworks.

The concept of auto-tuning programs running on the GPU is, understandably, not as popular compared to regular CPU programs, but has been studied before, specially recently with the rise of general purpose GPU computing. Magni et al. [15] propose a solution to the problem of selecting a value for thread coarsening in OpenCL, using a machine learning. For every training program, six versions are made with different coarsening factors and compiled, with static features collected from a compiler intermediate representation. After compilation, every version is run to register performance data, they are ordered by increasing coarsening factor and the first five are each associated with a binary value that is true if the next version has better performance, and false otherwise. A Neural Networked is then trained with the five versions' features and their respective binary for every training program, such that the model can predict if it is beneficial to increase the coarsening factor on new programs. Using this approach, a positive average speed-up was observed on the tested GPU architectures.

Falch and Elster [9] explored auto-tuning parametrized code, and unlike most other works explored for this thesis, used models trained for individual benchmarks/programs, the features used for the models were the optimisation sets. The method of optimisation was to first parametrize OpenCL code with several implementation options that could be

changed at compile time, then compile and run several instances of the program with randomly chosen parameters, the execution time is paired with the parameter set and used to train a model to predict execution time based on input parameters. Three different learning algorithms were tested, Neural Networks, Support Vector Regression and Regression Trees, the overall best performing one was the Neural Networks. After training a model, it could be used to predict the execution time of every parameter configuration and pick the best performing one, or used to only narrow down the search space due to model inaccuracy. The model tested at the end is able to achieve a very low mean relative error find parameter configurations only marginally lower than the best.

Han and Abdelrahman [11] developed a model to predict if local memory should be used for OpenCL programs, more specifically kernels. Kernels are characterized through static features and the model used for prediction is a Random Forest. The authors opted to train the model through synthetic benchmarks, created by using a template and different launch configurations, resulting in a massive training dataset. When tested with other benchmarks and real world programs, the model predicted if local memory should be used with 95% accuracy.

Feature extraction isn't something that should be neglected, and because we are using only static features for this thesis, we explored different features to extract and different ways to do it in works that focus on characterizing GPU programs. Han and Abdelrahman [10] compared the use of synthetic benchmarks versus real programs in training machine learning models. The authors begin by developing a metric for the goodness of training programs, use it to measure real programs and then demonstrating their accuracy. The problem used as a basis for comparison is local memory optimisation for GPU programs, which they have shown to be hard to predict [11], the models used for prediction are Random Forests, trained with static program features. Synthetic benchmarks are generated with a template to increase the training goodness metric and then used to train a model, the same way real programs were used before. The synthetic model achieves significantly greater accuracy than the one trained with real programs. The synthetic benchmarks were also used to demonstrate that accuracy of a model increases with the size of the dataset used in training.

Wang et al. [30] used machine learning to decide if a parallel algorithm should be executed on the CPU or the GPU. OpenMP [27] programs are taken in and translated into OpenCL kernels, with some optimisations applied. An AST is generated from the kernels and used to extract features to be used for the prediction, including parametrized features. At runtime, the parametrized features are updated and then used, along with the others, to predict which implementation will be used. The model used for prediction is a Decision Tree and this approach achieves large speed-ups compared to solutions that focus only on one implementation, and other GPU code generators.

Taylor et al. [25] present an approach to increase the efficiency of a programs time, memory or power use, by scheduling OpenCL kernel at runtime using a machine learning

predictor. The model used is a SVM, and programs used in training are executed exhaustively to find their best processor configuration. Features are static, but extracted during runtime, as OpenCL uses a just-in-time compiler, they are taken from the intermediate representation of the compiler. The final model is able to achieve over 93% performance relative to a perfect predictor.

There were other works we explored in the literature, but the ones mentioned were the closest to our own objectives. All of them had their models fall under the supervised learning umbrella and achieved some success that way, so we will be taking the same approach, being influenced the most by the works predicting the best overall optimisation set, or specific GPU program optimisations. Regarding program characterisation, some work has already been done to extract features from programs using the framework we will be working with, and fortunately similar methods have been used in the studied works successfully. Additionally, the method of training programs with synthetic benchmarks, like in [11] and [10], is an interesting concept and may be useful to our work.

2.4 Concluding Remarks

GPUs are extremely complex pieces of hardware and developing programs for them can be very daunting, there are many low-level concepts that have to be kept in mind while doing so. The basics of CUDA are similar to C, as it is based on it, however more advanced concepts like some of its functionalities and optimisation patterns are quite different. Optimisation in CUDA is closely related to the way GPUs work internally and may differ greatly between algorithms, there aren't many generic optimisations that can apply to every program.

Machine learning in compilation has been rising in popularity, due to the fact that manually optimising programs is an extremely time consuming task. Models trained using various learning algorithms and methods can give a good estimate, in a reasonable amount of time, of how to properly optimise a program, given its characterisation as input. We explored many works, related to our own, in the literature, and through them gained some insight on how to tackle auto-tuning.

MARROW

Marrow[16] [1] is a high-level C++ framework for parallel programming, with a heavy focus on high-performance GPU execution. Before the work described in this thesis, Marrow had two working backend implementations, a sequential backend in regular C++ that runs solely on the CPU, and a parallel backend using OpenCL, which takes advantage of the GPU. As one of the objectives of this thesis, a third working backend, implemented using CUDA, has been developed and added.

Marrow's programming model is one of algorithmic skeletons, data is represented as containers provided by the Application Programming Interface (API), such as arrays, vectors, matrices, scalars and tensors, which can then be manipulated through the use of high-level API functions. In the background, the framework efficiently translates the container operations into backend kernels and orchestrates their execution in such a way that minimises execution time, without breaking any data dependencies between the containers.

3.1 Using the API

Working with Marrow containers and functions is akin to performing operations on primitive types like integers and floats, something like creating a new array from the sum of each corresponding element of two other arrays, in code, is as simple as adding the two arrays and use the result to initialise the new one, assuming the length of the arrays matches. The API also allows for operations between two different types of containers, like a matrix and an array, if the length of the array matches one of the dimensions of the matrix and the values within them are of convertible types¹. There are several operators that can be used, including every C++ operator that can be used between values of primitive types: plus, minus, division, multiplication, modulus, logical and, etc... Additionally, unary functions such as the square root and sin may be called, receiving a container as an argument, to perform the operation on all container elements.

¹https://en.cppreference.com/w/cpp/types/is_convertible

```
1 using namespace marrow;
2
3 array<int, 1000000> a;
4 array<int, 1000000> b;
5 array<int, 1000000> c;
6
7 a.fill(3);
8 b.fill([](int i) { return i; });
9 c.fill([](int i) { return 999999 - i; });
10
11 vector<int> aux = a * b;
12 scalar<int> inner_product = reduce<plus<int>>>(aux);
13
14 auto red = reduce<multiplies<int>>>(a - b);
15 auto mult = b * c;
16 vector<int> x = mult - red;
```

Listing 2: An example of performing some calculations with Marrow.

Besides arithmetic operations between containers and functions performed on each container element, the [API](#) also provides other common functions [24] that can be performed on a single container:

Reduce performs a specified operation, like a sum, on every element of an array or vector and returns the resulting scalar. The function can also be used on a matrix, in which case, it will perform the operation on every element along an axis, returning the resulting array;

Scan for every element of an array or vector, a specified operation will be performed on all elements up to the current element, inclusive, and returns the resulting array or vector, depending on the original container;

Sort reorders the elements of a container, according to the specified condition, and returns the resulting container;

Filter returns a vector containing only the elements of the original container that satisfied the specified condition, alternatively, a bitmap may be used as an argument instead of a condition.

Listing 2 presents an example of how Marrow is used in practice by a programmer, in this instance, to calculate the inner product of two arrays and to do some other arbitrary calculations, that serve to demonstrate a coding style which may be optimised by Marrow. Skipping over declarations, lines 7-9 serve to initialise the first three arrays, *a* with every element set to 3, *b* with every element's value matching its index and *c* with the same pattern but reversed. The inner product of *a* and *b* is calculated in lines 11 and 12, first by multiplying each corresponding element of *a* and *b*, and then adding every element of the result together. Lines 14-16 perform some other similar calculations, but the result

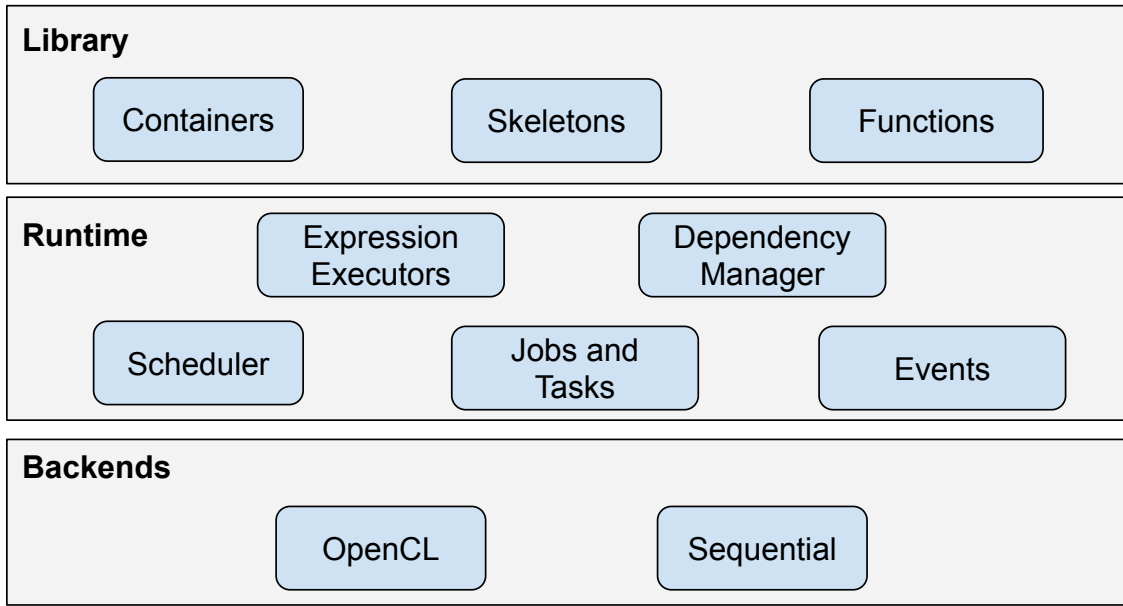


Figure 3.1: Illustration of Marrow's internal structure.

of each operation, except for the last, has its type deduced by the compiler instead of being specified explicitly, what this means is that the type of the results aren't actually containers, but expressions. Sequences of operations done this way may be more efficient due to some possible optimisations, which will be explained in Section 3.2.

3.2 Structure and Execution

A programmer using Marrow is able to make use of containers, skeletons and some other functions provided by the [API](#), however that is only a portion of the elements composing Marrow. While the translation from skeletons and functions to backend code is done mostly at compile time, with little performance overhead, their order and time of execution is dynamically managed at runtime by internal components. By having execution orchestrated at runtime, Marrow can use the hardware more efficiently, processing independent operations in parallel whenever possible and reducing the amount of costly memory transfers, to and from the [GPU](#). This also means that skeletons and functions may not be executed immediately after being called, but only once they are required.

Figure 3.1 presents the components of Marrow and how they relate to each other, as mentioned earlier, the API only exposes the library components to the programmer, while everything else is used internally, in the background. Each runtime component has its own purpose:

Scheduler decides the order of execution of jobs and tasks, restricted by the dependency manager. Forces the execution of incomplete jobs and tasks if some new operation

in the code is being called and depends on them;

Expression Executors execute backend code, according to the skeletons and functions used;

Jobs and Tasks represent various operations, like skeleton executions and memory transfers/allocations, and are associated to an event that is triggered once the respective job or task is completed;

Dependency Manager asserts if a job or task can be executed, by going through its dependencies and checking if they have already been executed, using events;

Events are declared in a generic class, used by the other components to assert if a job or task is completed, can be waited on if necessary.

The runtime components call generic functions and structures that are inherited and implemented by each backend. Each individual backend has its own implementation for memory allocations and transfers, retrieving system information, handling parallel execution, events and skeletons/functions. The backend, and therefore its implementation, is chosen via a compilation flag.

During compilation, calls to skeletons and functions generate a direct conversion to backend code, specific to the context they are used in through the use of C++ templates, which expression executors then call at runtime. One way Marrow optimises their execution, is by having the calls actually return a [struct](#) representing the calculation, instead of their result, which can in turn be used as an argument for other skeletons/functions, that will then return an expanded expression [struct](#), containing the new operation appended to the previous calculations. At runtime, only when an expression is assigned as the value of a container, will Marrow start running the respective generated executors. This allows for independent operations within the expression to be executed in parallel and reduces the need for memory transfers, as host data only needs to be updated at the end of a sequence of operations, instead of transferring data back and forth with every step.

After a sequence of operations finishes execution, the resulting data in the [GPU](#) needs to be transferred to host memory, so that when the respective container is accessed again by the [CPU](#), old or uninitialised data isn't read. However, Marrow will continue execution as normal, even if the memory transfer hasn't occurred yet. When a container is modified by an operation executed on the [GPU](#), it is marked as dirty until its data is updated. By doing this, unrelated operations can run without waiting, and maybe even do so while memory is transferred in the background. When some operation depends on a container marked as dirty, execution will block and wait for the container data to be updated.

Marrow overlaps tasks such as computations and memory transfers through task parallelism. How it is implemented is specific to the backend, for example, a basic C++

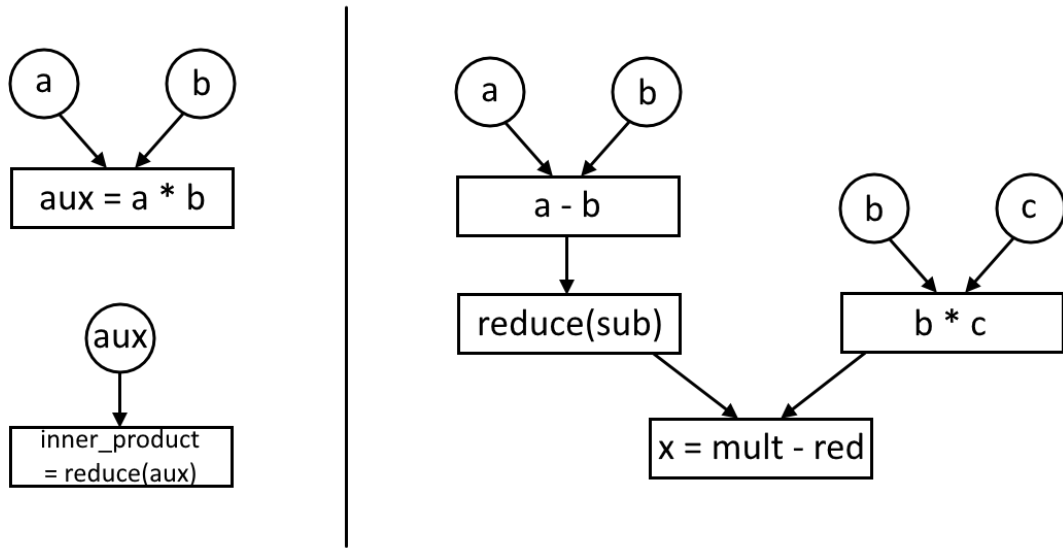


Figure 3.2: Illustration of the expressions generated by the example in Listing 2, represented as [ASTs](#).

backend may have each stream represented by a different thread. By treating operations and transfers as tasks, with dependencies between each other, it may be possible to process independent task in parallel.

3.2.1 Example Execution

With a better understanding of how Marrow works internally, it becomes more apparent why declaring the result of an operation in a sequence of calculations as *auto*, effectively declaring it as an expression type, may be beneficial. Looking back at the example in Listing 2, the expressions generated from the calculations in the code would be like the ones presented in Figure 3.2. The expression on the right could easily be optimised, by having the left and right branches executed in parallel, as they are independent from each other, and there would only be memory operations at the start to transfer *a*, *b* and *c* to the GPU, and at the end, to transfer *x* out of the GPU. Furthermore, the last subtraction and the multiplication in the right-side expression can be fused [29] into a single kernel, that depends on the reduction operation. If the result of the multiplication was declared as a container, then such an optimisation could not be applied and two kernels would be generated, instead of one, not to mention the cost of allocating an extra memory buffer to hold the result (assuming it would not be use elsewhere).

It is important to note that not every operation should have its result declared as *auto*. Obviously, the last operation needs to have the result declared as a container for the expression to be executed, but besides that, if the result of an operation is declared as *auto*

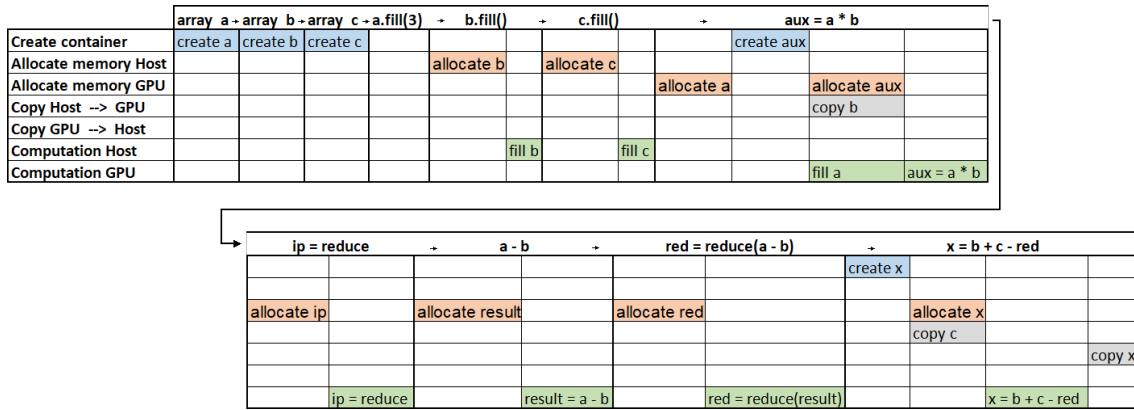


Figure 3.3: Table representing how the example in Listing 2 would be executed in Marrow, with the flow of execution going from left to right.

and used in more than one chain of operations, that operation and all others contained within its result expression would be executed once for each chain that uses the result. This can result in a large performance hit if the result expression has many operations, or if it is used in many different chains. If an operation has its result used in more than one chain of operation, then it probably is better to have the result declared as a container, to avoid such issues.

Figure 3.3 displays how Marrow would handle and execute the example in Listing 2, assuming Marrow was running a GPU backend with a working implementation of task parallelism. As explained, Marrow optimises the execution so that c is only copied into the GPU when it is needed to calculate x , and the calculations of $mult = (b * c)$ and $x = (mult - red)$ are fused into a single kernel. Something that may seem odd is that container a isn't copied to the GPU, but calculated with a kernel instead. As the container is initialised with a flat value that can easily be passed as a kernel parameter, it is more efficient to fill the container using a kernel, directly on the GPU. Another benefit of this is, if the user doesn't ever access a container initialised this way, said container won't even be allocated on the host, and will exist solely on the GPU. Unfortunately, it may not be possible to the same with containers initialised using a lambda function, so those will have to be allocated and initialised on the host, and then copied into the GPU.

A CUDA BACKEND

In this chapter we present how we added a CUDA backend to Marrow. To that end, we add a new configuration to the build system and created implementations for each of Marrow’s generic classes and structures. These fit somewhat nicely with CUDA’s programming model, however there were a few instances where some workarounds were needed in order to achieve the desired functionality: Section 4.1 elaborates on the memory management and host-GPU synchronization operations, Section 4.2 presents how Marrow expressions are transformed into CUDA kernels

4.1 Management

General management is fairly straight forward with CUDA in Marrow, specially due to the fact that, for the purpose of this thesis, the backend assumes that there is only one device (GPU) to be used.

As explained in Section 3, Marrow overlaps the execution of independent operations whenever possible, and for that, there’s the concept of task parallelism. Any objects related to task parallelism are initialised with the backend, using the value of task parallelism degree, that is received as an argument in the backend’s constructor function. The value dictates how many tasks can be processed at the same time. For the CUDA implementation, task parallelism can be achieved through the use of CUDA streams, passed on to kernels and memory operations by the Marrow runtime. Thus, during the backend initialisation, a vector of CUDA streams is initialised, with the vector size/number of streams matching the task parallelism degree.

CUDA doesn’t need its context to be initialised, so besides initialising the task parallelism degree, nothing else needs to be done in the backend initialisation. When terminating a program, if every CUDA related resource is properly managed, the only thing necessary to do in the backend’s destructor is destroying the task parallelism streams with *cudaStreamDestroy*, but to ensure nothing is missed, we reset the device context with *cudaDeviceReset*.

Besides initialisation and termination, the backend definition also requires the ability of retrieving information regarding several capabilities of the backend, such as the maximum number of dimensions of threads a device can launch. This information can be found by querying the maximum grid size of a kernel with *cudaGetDeviceProperties*, which returns an array of 3 numbers (x , y and z), if only z has a value of 0, 2 dimensions are supported, if both y and z are 0, then only one dimension is supported. Other retrievable information is the capability of processing images and the number of the devices, which for this backend, is not possible and is always one, respectively. Lastly, the backend should be able to provide information regarding compute capabilities, found directly by querying with *cudaGetDeviceProperties*. These are used to define the maximum values for the thread block size in each dimension, and some other device limits.

4.1.1 Memory

Regarding memory related operations, we made the decision to use standard memory allocations and deal with transfers between host and device memory manually, instead of using unified memory, which would handle the memory transfers for us, due to the higher potential for optimisation. With that said, the definition of the backend requires the implementation of functions to allocate memory, free memory, transfer data to the device and transfer data from the device. All of these translate directly to their CUDA counterparts: *cudaMalloc*, *cudaFree* and *cudaMemcpyAsync* with the respective directional flag. All memory transfers should be asynchronous, with an event and stream associated to them, so the synchronous and immediate version, *cudaMemcpy*, is not used.

Additionally, the backend needs to be able to query the memory capacity and available memory of the device, both of which can be done with *cudaMemGetInfo*.

There is one more memory operation in the backend definition, however it is incompatible with CUDA, which is the allocation of a *const* memory block. Marrow can optimise some expressions with data that won't be written to by allocating *const* memory, but with CUDA, it's not possible to dynamically allocate this type of memory, or read-only memory for that matter, which is similar. In CUDA, both *const* and read-only memory can only be statically allocated, that is, have their size determined during compilation. For compatibility, the function performs a simple allocation in global device memory.

4.1.2 Synchronization

As Marrow can execute several tasks simultaneously, there is an obvious need to properly handle the dependencies between tasks, in order to only execute tasks after all of their dependencies have finished executing. Part of the solution to that, are events, which are assigned to a task and triggered upon its completion. A process may be notified of a task completion by either waiting on the event, or via a callback assigned to the event.

Cuda events are associated with a stream and triggered when the execution of the stream their assigned to reaches them. This works perfectly fine for the concept of task

parallelism within Marrow, memory operations and kernels are assigned to a stream, and then an event with the purpose of signalling their completion is created and assigned to the same stream immediately after, functioning as intended by the definition. Marrow events can also be waited on, and CUDA provides exactly that same functionality as well, with *cudaEventSynchronize*.

Finally, Marrow events may have callbacks associated to them, these callbacks are triggered as soon as the event completes and notify the dependency manager that the computation as concluded and the produced results are available. For this functionality, the stream associated with the event, which is the same one associated to the kernel the event represents, is saved in the constructor, and when a callback is being set, a host function is en-queued to the event's stream, using *cudaLaunchHostFunc*. The host function is only called after the event is triggered, which in turn happens only after the kernel represented by the event finishes execution.

4.1.3 Example

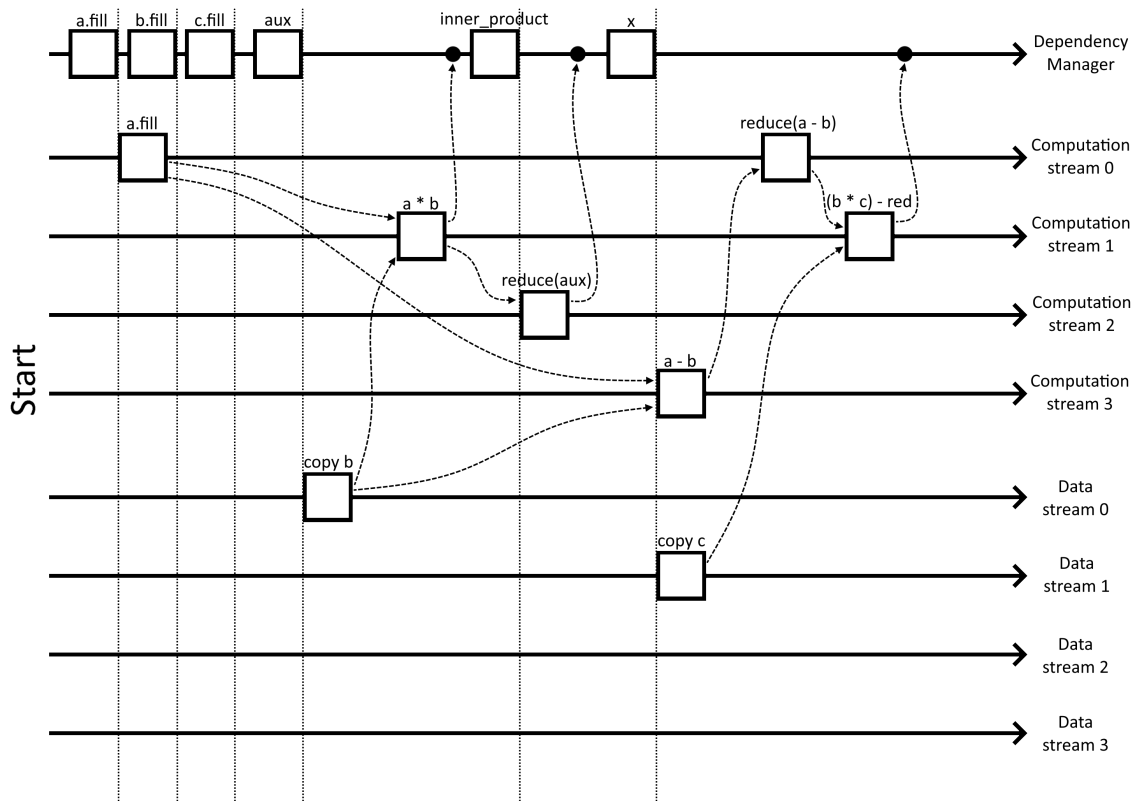


Figure 4.1: Illustration of how the example in Listing 2 would be executed in Marrow, while using the CUDA backend.

Going back to the example of Listing 2, with the CUDA backend, the execution flow would look like the graphic depicted in Figure 4.1. The execution in the figure assumes

that the backend was initialised with a task parallelism of 4. As can be observed, every new task is assigned to the next stream so they can execute in parallel if possible. The dashed arrows in the graphic indicate dependencies between tasks, and that a task has finished and notified the dependency manager when pointing to a dot in the timeline.

4.2 Kernel Execution

With the backend managing itself according to the Marrow definition, or at least close enough to it, all that is left is having it capable of handling the actual computations, that is, launching kernels that can compute Marrow operations. To do this, an implementation is made for each of Marrow's skeletons: map, reduction, scan and filter. While each of them represents the same type of computation, two similar computations of the same type can still be vastly different regarding the low level operations required, depending on the containers and the function (the operator) used for the computation. In Marrow, the containers and the function of a skeleton are handled with C++ templates internally, and fortunately, CUDA kernels can also be defined as templates, meaning a single kernel definition written in code can generate several similar kernels, each handling one specific computation. Having kernels defined as templates also means that all of the code generation happens at compile time, avoiding the overhead of writing a separate kernel file and having to read it, to execute the kernel, as is the case with OpenCL.

Some complications come up regarding the operator function as a kernel parameter. Passing a function, in the skeletons that use one (map, reduce and scan), as a template parameter, does not work directly with CUDA kernels. For a kernel to be passed an arbitrary function, it has to be compiled to device code and its symbol (pointer) has to be retrieved from the device before being passed to the kernel. The first condition is easily satisfied by decorating every function that can be passed to a kernel with a `__device__` flag. The second condition simply requires the use of `cudaMemcpyToSymbol`, which finds the device function pointer and copies it into a variable. The process is exemplified by the `run` function of the first struct (line 9) in Listing 3, assuming the type of the template argument `Func` is a function compiled in the device.

Marrow offers a variety of operators to be used in computations, they are each implemented as the call operator of a matching `struct`, that is used as a template parameter. For the CUDA backend, the call operators are all decorated with the flag to be compiled in device code, but they aren't passed directly to the kernel. Due to all the different types of parameters, return types and the operators used in computations, kernels are passed a "modular" function, adjusted to the computation using templates. For the reduction and scan skeletons, the function passed has a simple definition, like the one presented in line 40 of Listing 4, which is practically a direct call to the operator. The function passed to the map skeleton is a bit more complicated due to its versatility, the number of arguments is variable and there are a few other template arguments. How it is defined will be explained in Section 4.2.1.

```

1  template<...>
2  struct cuda_kernel {
3      /**
4       * Launch a kernel
5       * @tparam CodeOptimizationStages The code optimization stages
6       * @tparam Func Function to be applied by the kernel
7       */
8      template<typename CodeOptimizationStages, typename Func>
9      static void run(const Func& func, std::vector<size_type>& thread_grid, const
        ↳ std::vector<size_type>& thread_block_size, const cudaStream_t stream, ...) {
10         using IndexGeneration = get_index_generation_stage<CodeOptimizationStages>;
11         constexpr auto Coarsening = get_coarsening_factor<CodeOptimizationStages>;
12         Func dfunc;
13         cudaMemcpyFromSymbol(&dfunc, func, sizeof(Func));
14         some_kernel<IndexGeneration, Coarsening><<<thread_grid[0], thread_block_size[0], 0,
        ↳ stream>>>(dfunc, ...);
15     }
16 };
17
18 /**
19  *
20  * @tparam Expr Marrow expression to be executed
21  * @tparam CodeOptimizationStages Sequence with the code optimizations stages
22  * @tparam Types Types of the values to be used in Expr
23  * @tparam Sizes Size of each dimension required by the values to be used in Expr
24  */
25 template<typename Expr, typename CodeOptimizationStages, typename Types, typename Sizes,
        ↳ typename Enable = void>
26 struct cuda_kernel_executor {};
27
28 template<typename CodeOptimizationStages, typename LHS, typename RHS, typename... Sizes,
        ↳ typename ResultType, typename... Types>
29 struct cuda_kernel_executor<assignment<LHS, RHS>, CodeOptimizationStages,
        ↳ sequence<Sizes...>, sequence<ResultType, Types...>> {
30
31     static marrow::runtime::event* run(const std::vector<size_type>& thread_grid, const
        ↳ std::vector<size_type>& thread_block_size, const cudaStream_t stream, ...) {
32         if constexpr (sizeof...(Sizes) > 1)
33             cuda_kernel<CodeOptimizationStages, ...>::run(pfunc<ResultType, RHS, NDims,
        ↳ index_type*, Types...>, thread_grid, thread_block_size, stream, ...);
34         else
35             cuda_kernel<CodeOptimizationStages, ...>::run(pfunc<ResultType, RHS, NDims,
        ↳ index_type, Types...>, thread_grid, thread_block_size, stream, ...);
36         return new marrow::runtime::cuda::cuda_event(stream);
37     }
38 };

```

Listing 3: Marrow’s method of launching CUDA kernels.

Both of the functions that are passed to kernels have their type defined by another template type (*cuda_func* in line 7 of Listing 4). This is because of an issue which we came across, it wasn't possible to use the inherent type of the functions to define the variable which *cudaMemcpyToSymbol* would copy to. To work around this, we use a template which uses the containers of the computation to deduce the correct type.

In lines 17 and 18, we have the type of the function which will be passed to the map kernel, it is defined by the result type, which is the type of the elements of the result container, the expression type, which is used to deduce operations the function will perform, the number of dimensions of the computation, the index type, which for expressions of 1 dimension is an integer number representing an index, and for expressions in higher dimensions it is a pointer to an array of integers, because for expressions of 2 or more dimensions, lower dimensional containers may be present in the calculation, and so more indexes must be passed to the function, one for each type of container, more on this in the following Subsection 4.2.1. Finally, the function type is also defined by the argument types, which depends on the operands of the expression, vectors arrays, matrices, etc.. are represented by a pointer to their element type, while scalars are represented by their element type.

In lines 39 and 40, is the type of the function passed to the scan and reduction skeletons. This one is much more straight forward than the one used for maps, as it is only defined by the operator to be used, the return type, which is the type of the elements of the result container, and the input types, which is the type of the elements of the input container. As mentioned earlier, passing an operator type of function directly as a kernel parameter wasn't working, so in lines 26 to 31 we have a `struct` with a function that uses the operator received in the template parameters. This helper function is what is then passed to the kernel.

4.2.1 Map

The map skeleton is probably the one that is used the most, as it encompasses any and all calculations done with an operator between two containers. It is simple in concept, yet the implementation isn't the most straightforward. There is no real restriction to the amount of dimensions a container can have, and the map skeleton has to be able to process any container, meaning it should be able to compute any number of dimensions. Additionally, as mentioned in Section 3.1, the containers in an operator computation can have a different number of dimensions each, however they do need to have matching sizes. Lastly, Marrow optimises maps by fusing sequences of maps, similarly to other works [29], into a single computation to reduce kernel launches, for example, adding more than two containers in the same line will always result in a single kernel launch. This means that the number of container arguments for a single kernel is two or more.

The implementation of the core computation is split into 3 kernels, 6 after Subsection 5.1.1, with each handling a different number of dimensions. The first kernel computes a

```

1  /**
2   * Type of a Marrow CUDA function
3   * @tparam ResultType Type of the result
4   * @tparam Types Types of the arguments
5   */
6  template<typename ResultType, typename... Types>
7  using cuda_func = ResultType (*)(Types...);
8
9  /**
10   * Pointer to a Marrow CUDA function to be used by the map skeleton
11   * @tparam ResultType Type of the result
12   * @tparam Expr Expression of the function's body
13   * @tparam NDims Number of dimensions of the values being operated in Expr
14   * @tparam IndexType Type of the indexes being used to access the data on the execution
15   * ↪ of Expr
16   * @tparam Types Types of the arguments
17   */
18  template<typename ResultType, typename Expr, size_type NDims, typename IndexType,
19   ↪ typename... Types>
20  static __device__ constexpr cuda_func<ResultType, IndexType, Types...> pfunc =
21   ↪ cuda_kernel_executor<Expr, types<Types...>::template run<ResultType, IndexType,
22   ↪ NDims>;
23
24  /**
25   * Helper struct to use the call operator of Function
26   * @tparam Function Struct with a call operator
27   * @tparam OutType Type of the result
28   * @tparam InType Type of the arguments
29   */
30  template<typename Function, typename InType, typename OutType>
31  struct function_helper {
32      static __device__ inline OutType run(const InType &x, const InType &y) {
33          return Function()(x, y);
34      }
35  };
36
37  /**
38   * Pointer to a Marrow CUDA function to be used by the reduction and scan skeletons
39   * @tparam Function Struct with a call operator
40   * @tparam OutType Type of the result
41   * @tparam InType Type of the arguments
42   */
43  template<typename Function, typename InType, typename OutType>
44  static __device__ constexpr cuda_func<OutType, const InType &, const InType &> rfunc =
45   ↪ function_helper<Function, InType, OutType>::run;

```

Listing 4: Definition of modular CUDA functions to be passed as kernel parameters.

```
1  template<typename Func, typename Type, typename... Types>
2  __global__ void run1dim(Func func, const_size_type size0, Type lhs, Types... arguments);
3
4  template<typename Func, typename Type, typename... Types>
5  __global__ void run2dim(Func func, const_size_type size0, const_size_type size1, Type
   ↪ lhs, Types... arguments);
6
7  template<typename Func, typename Type, typename... Types>
8  __global__ void run3dim(Func func, size_type* sizes, Type lhs, Types... arguments);
```

Listing 5: Marrow maps' CUDA kernel signatures.

single dimension, the second 2, and the third any higher number of dimensions. Which kernel is used, is dictated by the number of dimensions of the container with the highest number of dimensions in the computation. If it is 1, the first kernel is used, if it's 2 the second one is used, and 3 or more uses the third kernel. For example, a computation adding an array to the rows of a matrix will call the second kernel. Before taking index generation related optimisations into consideration, which will be discussed in Subsection 5.1.2, the number of dimensions that each kernel can handle is the only thing differentiating them. Without any optimisations, the threads of all 3 kernels are launched only in 1 dimension.

Each thread in a kernel executes the function for one position of the result container, so the minimum amount threads that need to be launched is the total size of the result container. The thread grid of a kernel is decided the following way: while the number of threads required is lower than the highest possible block size for the device, only one thread block is launched, with its size being at least 32 or the closest power of 2 higher than the number of threads required. If the threads required surpass the maximum block size, then the block size for the kernel will be the maximum, and the number of blocks is the closest multiple of the maximum block size higher than the threads required.

In Listing 5 we have the kernel signature of the 3 map versions, each of them receives the function to be performed on the operands, the size of each axis of the computation (the 3 dimensional kernel receives the sizes as an array), the result container and the operand arguments. The dimension sizes serve to, first, stop outlier threads from calculating out of bounds values, and second, calculate the indexes of each different container type present in the expression, for each thread in the kernel. Kernels 1 and 2 can receive the dimension sizes directly as arguments, but the third, because it can handle any number of dimensions, 3 or higher, would have to receive the sizes as a variadic argument, which is not possible, as kernels can only have 1 variadic argument. To get around the issue, before the kernel is launched, an allocation is made on the device, the sizes are copied to it and the pointer to the allocation is passed as an argument.

As previously mentioned, the function passed to the map kernel (*pfunc* in line 18 of Listing 4) is dependant on several template parameters, used to deduce its type and what it does. Once called, for each container, the index from the array that matches the

```

1  //user code
2  array<int, 1000> a;
3  matrix<int, 1000, 500> m;
4  matrix<int, 1000, 1000> result = m + a + 3;

```

```

1  // Translation of the user code into a specialised CUDA kernel
2  __global__ void rund2dim(int(*func)(index_type*, int*, int*, int), size_type size0,
   ↪ size_type size1, int* p_result, int* p_m, int* p_a, int p_c) {
3      index_type indexes[3];
4      const auto index01 = blockIdx.x * blockDim.x + threadIdx.x;
5      const auto size01 = size0 * size1;
6      if (index01 < size01) return;
7      indexes[0] = index01; // 2 dimensional linear index
8      indexes[2] = index01 / size0; // second dimension (Y axis)
9      indexes[1] = index01 - indexes[2] * size0; // first dimension (X axis)
10     p_result[indexes[0]] = func(indexes, p_m, p_a, p_c);
11 }

```

Inlining the code for

```
func(indexes, p_m, p_a, p_c)
```

we have:

```
map<call<plus>, map<call<plus>, matrix<...>, array<...>>, int>(indexes, p_m, p_a, p_c)
```

which applies the functions to the respective arguments:

```
plus(plus(get<matrix<...>>(indexes, pm), get<array<...>>(indexes, p_a),
        get<int>(indexes, p_c))
```

by computing the correct index from the matrix and array templates, we have:

```
plus(plus(p_m[indexes[0]], p_a[indexes[1]]), p_c)
```

which, by inlining the functions' implementation, finally translates to:

```
p_m[indexes[0]] + p_a[indexes[1]] + p_c
```

Listing 6: Example of how Marrow's code should translate to CUDA kernels, after all the template specialisations.

container's geometry (for 1 dimensional maps, the process is simplified as there's only 1 index) is used to retrieve the correct element from the container, finally the operator is called for all the elements and the result returned. In Listing 6 can be seen how Marrow roughly translates user code into kernels. In the examples kernel call, *size0* would be 1000, *size1* = 500, *p_result* would be a pointer to *result*'s data, *p_m* to *m*, *p_a* to *a* and *p_c* would be 3. How the indexes are calculated will be discussed in more detail later in Subsection 5.1.2.

All 3 kernels are very similar in the way they work: calculate index(es) and then call the function. It's the process of calculating indexes that differs for each one. The first kernel only calculates one index, which is the same as the identifier of the thread running the kernel (the thread id). The second, also calculates the thread's id, which matches the 2 dimensional index, then from it, and the dimension sizes, calculates the indexes for

Algorithm 1 Method to calculate all lower dimensional indexes using the highest dimensional index.

```
NDims  $\leftarrow$  number of dimensions of the expression  
sizes  $\leftarrow$  array of size NDims filled with the size values of each dimension  
indexes  $\leftarrow$  [NDims + 1]  
indexes[0]  $\leftarrow$  thread's id  
total_size  $\leftarrow$  sizes[0] * sizes[1] * ... * sizes[NDims - 1]  
base_index  $\leftarrow$  indexes[0]  
total_size  $\leftarrow$   $\frac{\text{total\_size}}{\text{sizes}[\text{NDims}-1]}$   
indexes[NDims]  $\leftarrow$   $\frac{\text{base\_index}}{\text{total\_size}}$   
NDims  $\leftarrow$  NDims - 1  
while NDims > 1 do  
    base_index  $\leftarrow$  base_index - indexes[NDims + 1]  $\times$  total_size  
    total_size  $\leftarrow$   $\frac{\text{total\_size}}{\text{sizes}[\text{NDims}-1]}$   
    indexes[NDims]  $\leftarrow$   $\frac{\text{base\_index}}{\text{total\_size}}$   
    NDims  $\leftarrow$  NDims - 1  
end while  
indexes[1]  $\leftarrow$  base_index - indexes[2]  $\times$  sizes[0]
```

the X and Y axes. The last kernel is a bit more interesting due to the varying number of dimensions. The first step is like the others, to calculate the thread id, which matches the index in N dimensions, with N being the number of dimensions of the calculation. With the N dimensional index, it is possible to iteratively calculate the rest of the lower dimension indexes, as demonstrated in Algorithm 4.2.1, which is done using a statically unrolled loop.

4.2.2 Reduction

The implementation for the reduction skeleton is based on Harris' reduction [12]. The algorithm had already been implemented for the OpenCL version of the skeleton, so for CUDA, the OpenCL kernels were adapted into CUDA kernels. The algorithm is designed to use the device resources as efficiently as possible, and to achieve this, the blocks of the thread grid are completely independent of each other. Each of them is assigned to a unique "slice" of the input container and perform a local reduction of the elements in that slice, then at the end of the kernel, the local reduction is written to a position matching the block id in the output container. The kernel is then executed again, taking the output container of the previous iteration as input. This process repeats until the input container can be reduced by a single thread block, in which case only 1 value is saved in the output, and thus the calculation is finished. While this skeleton may need to launch a kernel multiple times, depending on the size of the container being reduced, unlike the map skeleton, the reduction manipulates only one container, meaning the implementation is a bit more straightforward.

The kernel implementation begins by allocating a shared memory block, which will

be used by all of its threads to speed up memory accesses. Then each thread preforms the reduction of the first 2 elements of it's slice, if possible, and puts the result in the shared allocation. After that, it goes through a cycle that reduces elements of increasing indexes, until the end of the slice is reached. As the indexes used start as a continuous sequence, and are always incremented by the same value within the block, the memory accesses are kept coalesced. Finally, the values saved in shared memory are reduced, and the result saved in the output container.

Marrow also supports the reduction of a 2 dimensional matrix into an array, but Harris' reduction is designed to preform the reduction of an array into a single value. This is an issue that has already been solved in the OpenCL kernel implementation, in which a variant of of Harris' reduction was made. The key difference in the 2D to 1D reduction variant, is that each thread block is assigned to a slice of a row of the matrix container, instead of a slice of the whole container. When the matrix has small enough rows, that only one thread block is necessary to reduce each, another specialised variant of the 2D to 1D kernel is used to compute the final container returned by the skeleton. The two kernels were ported from OpenCL to CUDA.

4.2.3 Scan

Like the reduction skeleton, scan is applied to only one container, and may also require launching kernels multiple times recursively. It also can only be applied to 1 dimensional containers. The full execution of the scan skeleton, for a container that cannot be processed using a single thread block, requires two different kernels to be executed in sequence. This is because some extra operations are necessary to process large containers, and before those operations can be executed, every thread needs to be synced, which can only be achieved by finishing a kernel and starting an new one. Large containers also require the skeleton to be called recursively on an auxiliary container returned by the first kernel. Once again, this skeleton already had a kernel implementation in OpenCL that could be ported to CUDA.

The first kernel works similarly to the reduction kernel, a unique slice of the container is assigned to a thread block, and each block will perform a local scan for their slice. At the end of the kernel, each block writes their local scan result to the output container, as well as the last element of the local scan to an auxiliary array. For containers that fit in a thread block, the execution ends here, as the local scan is already the correct output.

Large containers continue the execution, after the first kernel finishes, a scan is performed on the auxiliary array, resulting in an array of values that each thread block can use the operator function on for their local scan. The second kernel receives the output and auxiliary containers returned by the first kernel, and each thread applies the operator function to its respective element in the output container and to the value of the previous thread block in the auxiliary container, replacing the value of the element with the result.

```
1 template<typename Type>
2 __global__ void cuda_filter_kernel(const bool *g_ibit, Type * g_odata, const unsigned
  ↳ *g_iscan, const Type *g_idata, const unsigned long n) {
3     const unsigned long tid = threadIdx.x + blockDim.x * blockIdx.x;
4     if(tid < n && g_ibit[tid])
5         g_odata[g_iscan[tid]] = g_idata[tid];
6 }
```

Listing 7: Code snippet of the CUDA filter kernel implementation.

4.2.4 Filter

The filter skeleton is one of the more simple implementations, because most of the necessary work is done other by skeleton implementations. It is a compound of 3 operations, the map, the scan and a new one to finish the filter computation.

In high-level Marrow code, a filter is called with a container to be filtered, and the same container with a condition applied to it. The container with the condition applied to it is an operator function, and translates to a map that returns a bit map, by applying the condition to every element of the container.

With the bit map of elements satisfying the condition, a new container with size matching the amount of passing elements needs to be allocated, and the position of the elements in the container need to be calculated. Before allocating the new container, a scan with the plus operator is executed on the bitmap, and the result stored on a temporary container. Due to the way scan works, when we access the position of an element that satisfies the condition in the temporary container, we get the value of its index in the final container of the filter. The size of the final container can be obtained by checking the value of the last position in the temporary array, and thus, the final container can be allocated.

The skeleton is finished with a kernel that checks every position of the original array, if the matching index in the bitmap is true, then the value in the original contained is copied to the new one, with the position matching the value found in the temporary container. This process is exemplified in Listing 7, where *g_odata* is the new, final container, *g_ibit* the bitmap calculated with the map, *g_iscan* the temporary container calculated with scan and *g_idata* the original container.

MACHINE LEARNING-ASSISTED COMPILE

In this chapter we will be discussing what optimisations we are adding, how we're integrating them and to which kernels they apply. We will also explore the autotuning model, what it will predict and how, as well as a few road blocks related to the framework used, the features used and the generation of synthetic programs to use for training.

Marrow already has an autotuning module, which generates a synthetic dataset and trains a model with it to predict if a computation expression should execute on the [CPU](#) or on the [GPU](#) with OpenCL. With this thesis, more autotuning capabilities are being added to Marrow, but focused on the new CUDA backend. The intent is still to train a model with a synthetic dataset, but instead, it will predict the best optimisation parameters for an expression. The dataset generation and model training are both done offline. When a new expression is being executed, the model predicts its best optimisation parameters so that it can run optimised. To avoid having to load the model and do a prediction every time, the prediction results are stored in a cache.

5.1 Optimisations

The map skeleton is the target for the optimisations, as it is used so frequently, and it's more feasible to add modular optimisations to it, compared to the others. For the implementation of the optimisations, their parameters are received by the kernels as template arguments. This way, the optimisations can be applied statically. Before being passed to a kernel, the optimisations are contained in a generic template type representing a sequence of optimisation stages. Whenever needed, any stage can be statically extracted from the sequence, to be used directly. Listing 3 presents an example of optimisation parameters being passed to a kernel after being extracted from the stage sequence in lines 10-14.

Originally, we had considered 4 different optimisation options: Thread coarsening, index partitioning, use of read-only memory and use of local memory. After careful

consideration of the possible performance gains versus time and effort for the implementation of each optimisation, we opted to create implementations for thread coarsening and index partitioning.

5.1.1 Thread Coarsening

The first optimisation added is thread coarsening, in other words, how much work is assigned to each thread. In concept it is extremely simple, divide all the work by the amount of coarsening, launch as many threads as necessary for the "reduced" work and each thread does their original work, multiplied by coarsening. In practice, the implementation is done mostly like that, but there are a few exceptions that have to be dealt with. Also, for the sake of simplicity, the amount of coarsening is restricted to 5 static, multiple of 2 values: 1 (no optimisation, default), 2, 4, 8 and 16. The implementation of this optimisation in kernels will accept any integer number (values lower than 1 are interpreted as no optimisation), but the aforementioned range of restricted values is what the autotuner will be able to predict. Those values may be changed or have more of them added in the future.

For the map skeleton, each thread doing more work means calculating more positions of the result container. To achieve this without loops, a recursive template function is used to statically unroll each coarse iteration, presented in Listing 8, it receives the coarsening value as a template parameter to dictate the number of iterations. As function arguments it receives the operator function, the index, the size (or indexes and sizes, this function has an implementation for each of the 3 kernels), the index stride, and all the containers. At each step it calculates one position of the result container with the current index, then increments the index(es) with the stride and if there is a next step, calls itself. The stride is the total size divided by the coarsening value, and having all threads increment the index(es) by the stride at each step ensures the data accesses of each warp are all coalesced within the block, for the most part. For the second and third kernels, when accessing a lower dimension container, a warp can have one portion of its threads access the end of the container and the rest accessing the beginning.

A new version of the original 3 map kernels is implemented, and it is the one that is launched if the coarsening value is higher than 1. This is done because the coarsening version has a few extra operations to deal with edge cases, and even if there are no edge cases for 1 coarsening, each thread would still check if it is one. Edge cases happen when the total size in a computation isn't a multiple of the coarsening value, and when that's the case, there are some positions of the result container left unprocessed by the coarse function, and some threads have to calculate one more position to ensure a correct result.

Figure 5.1 illustrates how a container of size 85 would be processed, with a coarsening value of 8, using this implementation of coarsening. As 85 isn't a multiple of 8, some extra work needs to be done by a few threads. Everything up to position 79 of the container would be processed by the regular pass using the *multi_run* function, but threads 0-4

```

1 struct map1dim {
2     template<unsigned Iterations, typename Func, typename Type, typename... Types>
3     static __device__ inline void multi_run(Func func, index_type& index, index_type
        ↪ stride, Type lhs, Types... arguments) {
4         index += stride;
5         lhs[index] = func(index, arguments...);
6         if constexpr(Iterations > 1) {
7             multi_run<Iterations - 1>(func, index, stride, lhs, arguments...);
8         }
9     }
10 };
11
12 template<typename IndexGeneration, unsigned Coarsening, std::enable_if_t<(Coarsening >
        ↪ 1)>* = nullptr, typename Func, typename Type, typename... Types>
13 __global__ void run1dim(Func func, const_size_type size0, Type lhs, Types... arguments) {
14     index_type index = IndexGeneration::run<1>(threadIdx, blockIdx, blockDim, size0);
15     const size_type stride = size0 / Coarsening;
16     const size_type coarse_size = stride * Coarsening;
17     if(index < (size0 - coarse_size)) {
18         index_type ext_index = index + coarse_size;
19         lhs[ext_index] = func(ext_index, arguments...);
20     }
21     if (index >= stride) {
22         return;
23     }
24     lhs[index] = func(index, arguments...);
25     map1dim::multi_run<Coarsening - 1>(func, index, stride, lhs, arguments...);
26 }

```

Listing 8: Code snippet of the 1 dimensional map kernel with optimisations.

process one extra position each so that the kernel outputs a correct result.

As each thread is doing more work, before the thread grid is calculated, the total number of threads necessary for the calculation is divided the thread coarsening value, to reduce the amount of threads "wasted". Regarding concerns with warps and the number of threads calculating one extra position, the implementation is made such that threads with an *id* smaller than R , R being the number of extra positions to be calculated, calculate one extra position, which means the memory access is coalesced. Furthermore, currently the maximum value R can reach is 15 (largest coarsening value is 16, largest value R can reach is coarsening - 1), so at most, only one warp calculates extra positions. If, eventually, a coarsening value larger than 32 is used, the number of warps calculating an extra position is:

$$numberofwarps = \frac{R + 31}{32}$$

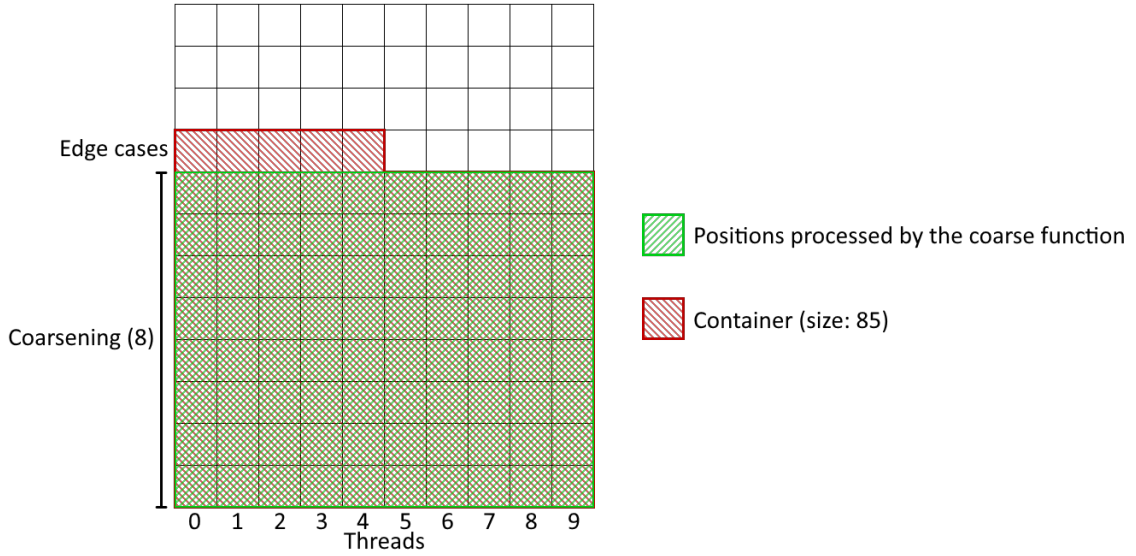


Figure 5.1: Example with a container of size 85 being processed by a map skeleton with a thread coarsening of 8.

5.1.2 Index Partitioning

To optimise a kernel with index partitioning means to have different options in launching threads and calculating the indexes of a thread. The method of launching threads along a single dimensional grid stays as an option, but a new option is added on top of it. The new option being launching threads along 1, 2 or 3 dimensional grids depending on the kernel launched and calculating the X , Y and Z indexes using the matching members of *threadIdx*, *blockIdx* and *blockDim*, then higher dimensional indexes from those values. This is the opposite of the first method which begins by calculating the thread id, the highest dimension index, and going down from there.

This new method does come with the drawback of not being able to perform calculations above 3 dimensions. This is because the threads launched along multiple dimensions do so in a way that "fits" the container resulting from the computation. For example, if the result of a computation is a container of dimensions (4, 4, 7), then the kernel threads are launched with a grid configuration of (1, 1, 1) blocks and (8, 8, 8) block size, which is enough to cover every dimension. CUDA only supports thread grid configurations of up to 3 dimensions, and it is not possible "fit" a number of dimensions into a lower one, thus computations with more than 3 dimensions cannot be executed using this optimisation.

To implement this optimisation, every index partitioning method is represented by its own struct, and each struct has 3 member functions that implement the calculation of indexes for each of the 3 kernels. The struct is then what is passed as a template parameter. As shown in line 14 of Listing 8, the kernel receives the struct type and can directly call the member function, with its respective dimension passed as a template parameter.

When calculating the thread grid, the type of index generation can be statically queried to decide how the threads will be launched.

5.2 Features

As mentioned in the beginning of the chapter, Marrow is already capable of autotuning, using a model to predict if a computation expression should be executed by the [CPU](#) or by the [GPU](#) with an OpenCL kernel. For that, there already is a framework to extract relevant information, in other words features, from an expression, such as: the total number of memory reads, memory writes, shared memory reads and shared memory writes (shared memory being a type of memory that every thread in a thread block can access), the total amount of allocated memory and allocated shared memory, and finally the number of operations of a certain class with integer and floating point numbers. Some of the features in the framework were adapted to be extracted from expressions executed with the CUDA backend. Due to the way shared memory is implemented with CUDA, that is, kernels need to be made specifically to use shared memory, we opted to not use shared memory as it would severely complicate the kernel implementations, and so, features like shared memory reads and writes, and allocated shared memory didn't make sense to be adapted, and were left unable to be extracted.

Most of the features adapted to CUDA should be intuitive to understand, but the number of class operations with integers and floats might require an explanation. There are several operations that can be used in Marrow kernels (addition, division, sin, etc.), but they don't all perform the same. They can somewhat be grouped up with those that perform similarly, like *plus* and *minus* should have a similar performance cost, and so should *less* and *greater*, etc... and that is the concept of an operation cost class. Ideally, every single operation would be extracted out of expressions and used as a feature, however that could result in a large amount of very similar dataset entries being generated, potentially providing redundant information to the model. With our method, only one operation needs to be used to represent a number of different operations, making the dataset generation process more simple and efficient, and the generated dataset remains diverse. We believe extracting the operation classes used in an expression, for both integers and floating point numbers, as the performance hit is different depending on the type, should provide relevant information for a model to use in its prediction, and achieve a result similar or better than if each operation was extracted and used as a feature for the autotuner model. With that said, this method is still being tested and at this point is just a proof of concept, and so the implementation does not work for operations on types other than integers and floats.

The autotuning model will be predicting the best optimisation parameters for an expression, given its features. This means a representation of the optimisation parameters, that the model can learn with during the training phase and then predict, is needed. The index partitioning optimisation can be interpreted as a binary choice, as there are

only 2 possible values, and the thread coarsening optimisation as the choice of 1 out of 5 possible values, each representing a thread coarsening value (1, 2, 4, 8, 16). Both optimisations can be represented together as a sequence of binary values, the first representing which method is using for index partitioning, and the rest which value is used for thread coarsening.

Thus, for any expression, the features will be:

- Number of reads;
- Number of writes;
- Allocated memory (in bytes);
- Number of operations of each cost class for integers and floating point numbers.

During training, the following values, representing the best optimisations for an expression, are also used:

- Index partitioning (binary);
- A value for each possible thread coarsening numbers (all binary).

5.2.1 Generating the Dataset

To avoid the long and tedious task of coding several programs performing varying calculations, that translate to various expressions, which are then used to train the model with their features, we opted to procedurally generate expressions. In addition to saving time, generating synthetic expressions yields a much higher number of different expressions, which should hopefully also yield a much bigger and more varied dataset, and thus a better trained model.

The generation of different expressions is done, like most things in Marrow, using templates. Several parameters are used in the specification of an expression: the data size and type, used to specify the size and element type of the processed containers, the number of dimensions, which specifies the type of containers used in the generated expressions, one number to dictate the maximum amount of operations of each cost class done in generated expressions and a number dictating the percentage of operations used to generate expressions with random configurations. As explained earlier in the previous Subsection (5.2), we're using the concept of cost classes when extracting features. When generating the dataset, each cost class is "represented" by one operator (for example, addition could represent both addition and subtraction, if they're both in the same class), so generated generated expressions will all use the same operator for the same cost class.

There are 2 components to the dataset generator: the expression generator and the setup. The expression generator receives the data type and the number of dimensions

as template parameters, and with those and the number of operations per cost class, retrieved from the setup, it generates valid expressions with varying operations performed. When the number of dimensions is higher than 1, expressions are generated for every combination of container with the provided number of dimensions, and containers of dimensions as high or lower than the provided number of dimensions. For example, if the provided number of dimensions is 3, the expressions generated operate on: only containers of 3 dimensions, containers of 3 and 2 dimensions, or containers of 3 and 1 dimensions.

The setup component of the dataset generator receives as arguments the data size of the containers, the initial number of measures, the maximum number of measures, the minimum standard deviation of the measures and an output stream pointer, which is where the dataset entries will be written into. In order to guarantee the containers used in expressions are all "compatible" with each other, the data size parameter will dictate the length of all axes of the containers, regardless of the dimension. The total size of a container will then be $total_size = data_size^{container_dimensions}$. For example, with a data size of 100, an array will have a length of 100 (same as its total size), but a matrix will have the length of both its axes be 100 (so it will have a total size of $100 \times 100 = 10000$).

Listing 9 presents a simplified code snippet of the dataset generator. Lines 2 to 16 set up the execution: lines 5 to 10 initialise the execution parameters; lines 13 to 16 initialise the output file and print the names of each feature in the first line of the dataset file, which are used later when training the autotuning model. Lines 19 to 21 are the ones that generate the dataset. Entries are generated for data sizes of all orders of magnitude from min_data_size to max_data_size . Moreover, for each data size considered, the generator also ranges all dimensions and types defined. For now we are only considering types *int* and *float*. Consider example

```
autotuning::generate_dataset<float, 3>(
    autotuning::cuda::cuda_setup<number_ops>(1000000, ...))}
```

The code line generates entries for expressions of 3 dimensions, that operate on containers of floats, each being either a 3 dimensional tensor of size $1000000 = 100 \times 100 \times 100$, a matrix of size $10000 = (100 \times 100)$ or an array of size 100. Note that, the higher the values of *number_ops*, the higher is the number of combinations of cost class operations. The value is set to a pre-processor constant *NUMBER_OPS* that may be defined at compilation time. All combinations will be generated for both integer and float types. Having more operations for each cost class would be highly beneficial for the dataset, as it would be bigger with more entries, but unfortunately there were complications compiling with higher values in the available hardware.

After an expression is generated, an instance of it is optimised and executed for every combination of optimisation parameters. Each instance is executed multiple times, with each execution being timed. The number of times an instance is executed depends of the

```
1  /**
2   * usage command [-i initial_measures] [-m max_measures] [-d minimum_std_deviation] [-s
   ↪ min_data_size] [-S max_data_size]
3   */
4  int main(int argc, char argv[]) {
5      // Setup parameters
6      int initial_measures = ...; // command line argument, if given, or default value 20
7      int max_measures = ...; // command line argument, if given, or default value 200
8      double minimum_std_deviation = ...; // command line argument, if given, or default
   ↪ value 0.1
9      std::size_t min_data_size = ...; // command line argument, if given, or default value
   ↪ 1000
10     std::size_t max_data_size = ...; // command line argument, if given, or default value
   ↪ 10000000
11
12     // Setup output file
13     constexpr auto features_file = literal( "features.csv");
14     auto output_file = get_file_path(std::string(cuda_directory + path_separator) +
   ↪ std::string(features_file));
15     std::ofstream output(output_file, std::fstream::out);
16     autotuning::cuda::cuda_features<void>::print_names(output);
17
18     // Generate the dataset
19     constexpr auto number_ops = NUMBER_OPS; // default is 5
20     for (std::size_t size = min_data_size; size <= max_data_size; size *= 10)
21         // for all dimensions Dim and all types Type: generated via template functions
22         autotuning::generate_dataset<Type, Dim>(
   ↪ autotuning::cuda::cuda_setup<number_ops>(size, initial_measures, max_measures,
   ↪ minimum_std_deviation, &output));
23
24     output.close();
25 }
```

Listing 9: Code snippet of the dataset generator.

parameters passed to *cuda_setup*, taking the code snippet as an example, each instance is executed at least 50 times, and keeps being executed until the standard deviation of execution times reaches a value below *minimum_std_deviation* (default 0.1), up to *max_measures* (default 200) times. After every instance has finished executing, their average execution times are compared and the one with the shortest time has its features (expression specification and the binary sequence representing the optimisation parameters) saved to the dataset file.

5.3 Model

The prediction model, and everything related to it, like evaluation, are implemented in their own dedicated class for ease of use. The class has functions for creating a model from scratch, reading and parsing the dataset file, training the model with the parsed dataset, making a prediction, saving the model on the disk and creating a new model from a saved file. There are few more functions dedicated to evaluating the model's performance, but those will be discussed in the next Chapter.

We have opted to use a neural network to make predictions, as this algorithm has been used successfully in similar works, using the OpenNN [2] library, which was already used to create the model for the OpenCL autotuning in Marrow. Using the library made the implementation fairly straight forward. A neural network object is created and stored in a member variable when a new model is created, with the inputs and outputs defined to match the features, and the type of predictions is set to classification, as we want the model to output the probability of each class (optimisation parameter). The library provides functions to read and parse the dataset file, making the process simple as well, all that is needed is specifying which features are inputs and which are outputs, and how the inputs should be scaled. When training the network, several parameters have to be set: the loss method, which is the cross entropy error for classification problems, the optimisation method, which specifies how the network updates its internal parameters, the maximum number of epochs, which is how many times the network learns from the dataset, and the loss goal, if the network reaches a certain degree of accuracy predicting the dataset, it will stop learning before the maximum number of epochs is reached.

Our autotuner is composed of 2 separate, fully connected neural network models, both performing classification, the first, predicts which index generation method should be used, and the second predicts which value of coarsening should be used, 1, 2, 4, 8 or 16. To train our models, we opted to use a maximum of 30000 epochs, a loss goal of 0.001 and for the optimisation method we used ADAM. Due to issues that will be discussed in the next Subsection (5.3.1), we had to use two separate neural networks for the model, one for each optimisation parameter. Given the low amount of inputs and outputs for the prediction, we opted to have a single hidden layer composed of 64 neurons, for both networks. An illustration of both models can be seen in Figure 5.2.

The prediction function receives an expression as an argument, extracts its features and passes them to the neural network. The prediction made by the network is then converted into a number array containing the "index" of the index partitioning method and the value for thread coarsening, which is easier to work with later, and returned.

5.3.1 Issues using OpenNN

Unfortunately, there were a few road blocks using OpenNN, either due to some functionality failing or not existing. The problem represented by predicting the optimisation

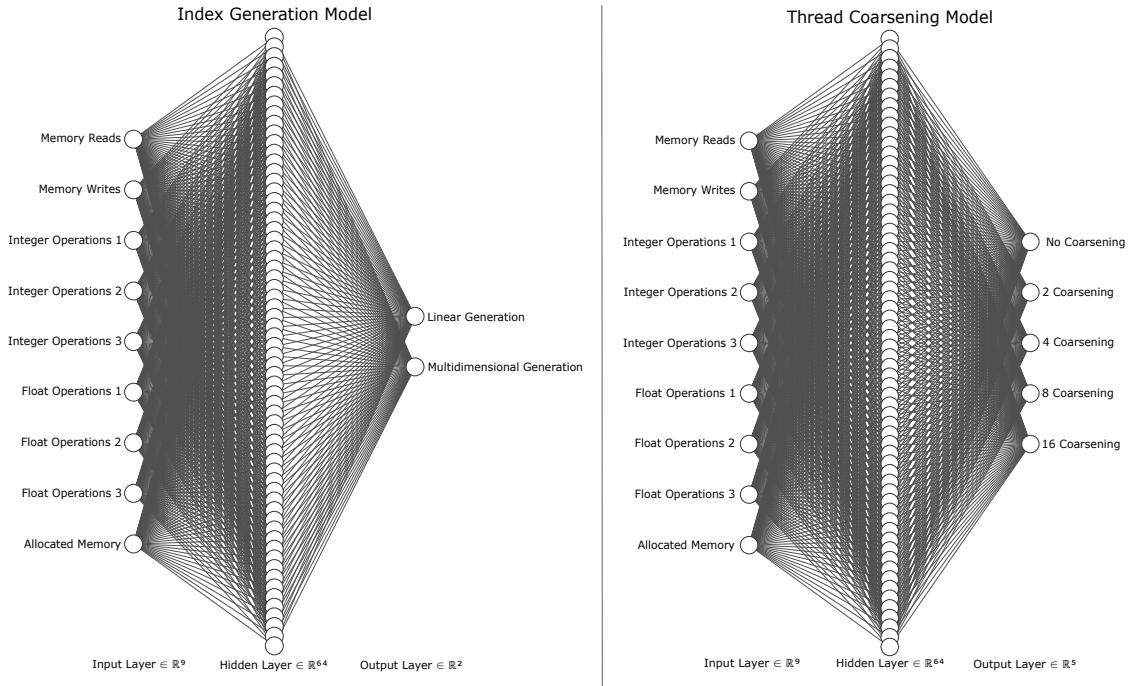


Figure 5.2: Schematic of the 2 models used in the autotuner.

parameters is one of classification, where more than one class can be the answer, and OpenNN does not support this type of problem, at least not directly. There are 3 solutions to get around this, the first being training and using 2 separate networks to predict each optimisation parameter individually, but there is the potential of a performance hit using 2 networks, but that also depends highly on the configuration of the neural network. The second solution is using a single model that predicts 1 out of every parameter combination, but this solution does not scale very well if more index partitioning methods, coarsening values and/or completely new optimisation parameters are added in the future, as the number of classes could drastically increase. The last solution is trying to adapt the prediction into returning two classes. Because the result of a prediction is the chance of a class being the solution, and not a binary value, the chances can be observed as subgroups corresponding to each optimisation. The issue with this solution is it can have less accuracy, as the network isn't made to predict multiple classes, only 1. We decided to test both the first and third solutions, and ended using the first, as the third results in faulty predictions.

Another issue we came across was with saving and loading a model from a file. OpenNN provides this functionality already, a network can have its structure and parameters saved into a XML file and a new network can be created with the file. Saving the network works without any problems, the issue is with loading. The parameters of are floating point numbers, and they can reach really small values, and as such, when

saving a network, some parameters can be written in scientific notation into the XML file. The function that parses a file into a network, reads the numbers saved in scientific notation incorrectly and sets the respective parameter as a **Not a Number (NaN)** value, making the resulting network not only different from the original, but also unable to make a prediction at all, always returning NaN values in its predictions.

Lastly, a more minor issue is that the OpenNN header has to be hidden from other files, as it would cause ambiguity issues with Marrow containers when included in a header. The issue is easily resolved by implementing the functions that depend on OpenNN in a source file.

5.4 Integrating Predictions With Generated Kernels

A programmer using Marrow does not make explicit calls to the autotuning model to optimise an expression, that is something automatically handled in the background by Marrow. A pre-trained autotuner is provided with Marrow, and at runtime, during the program initialisation the autotuner is loaded from memory as a static object, then when a new expression is being scheduled for execution, its features are computed¹, and then the autotuner is used to predict the best optimisation parameters. The result of the prediction made by the autotuner can only be known at runtime, but as explained earlier in Section 5.1, optimisations are applied statically at compile time using templates, so it is necessary to use some work around to dynamically decide which template specification of the optimised expression is executed. A possible solution for this issue is exemplified in Listing 10, this solution does have the penalty of having to compile every specialisation of the optimised kernel, increasing compile times and memory usage.

Running the model, as well as deciding the optimisation specification at runtime has an impact on performance, but this only happens in the first execution of an expression. When a new prediction is done, it is saved in a cache file. If the program is recompiled, an entry for the expression is detected in the cache, and the expression is statically optimised with parameters saved from the prediction. How this is achieved is exemplified in Listing 5.4, a generic entry `struct` is defined in the first header, along with functions to write new entries into the cache (not presented). The second header contains an entry for a specification of optimisation parameters for an expression, which was written after the same expression was run for the first time, and can now be used at compile time. This means that for this cache, entries are added every time a new expression is executed, but the entries on the cache can only be used after a program is recompiled after being run.

Besides the static cache, Marrow also has a dynamic cache, which works similarly to the static cache. Every time an expression is executed, if it has not been optimised by the static cache, it checks if it already has an entry in the dynamic cache. If it does have an entry, then it is executed, using a method such as the one presented in Listing

¹Expression features are "extracted" during compile time, but need to finish being computed at runtime using the expression data size, effectively they are parametrized features, similar other works [30].

```
1  template<typename CodeOptimizationStages, typename Expr>
2  static inline void run(Expr&& expr) {
3      using Execution = execution<backend::CUDA, Expr, CodeOptimizationStages>;
4      get_executor<backend::CUDA, Execution>::executor::run(std::forward<Expr>(expr));
5  }
6
7  template<typename Expr, unsigned Coarsening>
8  static inline void dynamic_stages_run(Expr&& expr, const unsigned indexing) {
9      if(!indexing) {
10         using CodeOptimizationStages = typename
11             ↳ autotuning::cuda::generate_stages<combination<Coarsening, 0>>::type;
12         run<CodeOptimizationStages>(std::forward<Expr>(expr));
13     } else {
14         using CodeOptimizationStages = typename
15             ↳ autotuning::cuda::generate_stages<combination<Coarsening, 1>>::type;
16         run<CodeOptimizationStages>(std::forward<Expr>(expr));
17     }
18 }
19
20 template<typename Expr>
21 static inline void dynamic_stages_run(Expr&& expr, const unsigned indexing, const
22     ↳ unsigned coarsening) {
23     switch(coarsening) {
24     default:
25         return dynamic_stages_run<Expr, 1>(std::forward<Expr>(expr), indexing);
26     case 2:
27         return dynamic_stages_run<Expr, 2>(std::forward<Expr>(expr), indexing);
28     case 4:
29         return dynamic_stages_run<Expr, 4>(std::forward<Expr>(expr), indexing);
30     case 8:
31         return dynamic_stages_run<Expr, 8>(std::forward<Expr>(expr), indexing);
32     case 16:
33         return dynamic_stages_run<Expr, 16>(std::forward<Expr>(expr), indexing);
34     }
35 }
```

Listing 10: Code snippet of a solution to run an expression using optimisations only known at runtime.

10, with the retrieved optimisation parameters from the dynamic cache. If there is no entry already in the dynamic cache, then the expression is being executed for the first time during runtime, and so the predicted optimisation parameter values are stored in the dynamic cache. The obvious advantage of this is that a program does not need to be recompiled for the entries in the dynamic cache to be used.

```
1  //static_cache.hpp
2  /**
3   * An entry of the static cache
4   * @tparam Execution The definition of the execution (of type marrow::execution)
5   */
6  template<typename Execution>
7  struct static_cache_entry;
8
9  template<backend Backend, typename AST, typename CodeOptimizationStages>
10 struct static_cache_entry<execution<Backend, AST, CodeOptimizationStages>> {
11     static constexpr bool cached = false;
12     static constexpr long long exectime = 0;
13     static constexpr marrow::backend backend = Backend;
14     static constexpr int block_size = 0;
15     using code_optimization_stages = CodeOptimizationStages;
16 };
17
18 ...
19
20 //cache.hpp
21 #include "static_cache.hpp"
22
23 template<>
24 struct static_cache_entry<execution<backend::CUDA, AST<...>>>{
25     constexpr bool cached = true;
26     constexpr long long exectime = ...
27     constexpr marrow::backend backend = backend::CUDA;
28     using code_optimization_stages = executors::code_optimisation_stages<...>;
29 };
```

Listing 11: Definition of the generic cache entry, and a specialised entry in the cache.

EVALUATION

In this chapter, we will elaborate on what we will be evaluating, how we are going to do it and finally discussing the results obtained.

The core of the work done in this thesis is Marrow's CUDA backend and the autotuner for the CUDA backend, with the latter being a bit more highlighted. The CUDA backend will be compared to Marrow's OpenCL backend and to functionally equivalent, handmade, CUDA programs. The autotuner will be evaluated in its prediction accuracy and effectiveness, as well as in its overall performance.

6.1 Objectives

In order to properly assess the performance of each component, we will be employing a methodology of Goal, Question, Metric. For each of the components we present one or more goals, followed by questions that must be answered in order to achieve the goal. To "answer" the questions, each of them has an associated metric that will be measured in applicable tests. The goals are:

Assess the performance of the CUDA backend To achieve this we ask: How does the Marrow CUDA backend perform, compared to the preexisting OpenCL backend? How does the Marrow CUDA backend perform, compared to just CUDA? For both of these questions, the metric to be measured is execution time. The first goal is to gauge the performance of the CUDA backend, thus we are going to directly compare the average execution time of varied calculations made with Marrow using the CUDA backend, versus marrow using the OpenCL backend, and a pure CUDA implementation of the same calculations. While the main metric of this objective is the execution time of the whole program, we will also be measuring and evaluating the execution time of just the kernels run by Marrow's CUDA backend and kernels implemented with just CUDA. These should provide some good insight, the whole execution times on how well Marrow's CUDA backend performs and how much overhead it has, and the kernel execution times on how well expressions are translated into kernels.

Assess the accuracy of the autotuner For this goal, we ask: How accurate is the index generation model? How accurate is the thread coarsening model? An important metric of any machine learning model is how accurate its predictions are, and that is what we will be measuring. In the context of the CUDA autotuner, this metric is measured by the success rate in predicting the best optimisation parameters. We will be counting each time a parameter predicted is the same as the best, so that we may calculate the accuracy in predicting each individual parameter. Additionally, we will be measuring the accuracy in predicting both parameters correctly and the overall accuracy, using the following formulas:

$$\text{indexGenerationAccuracy} = \frac{\text{correctIndexGenerationPredictions}}{\text{totalPredictions}}$$

$$\text{threadCoarseningAccuracy} = \frac{\text{correctThreadCoarseningPredictions}}{\text{totalPredictions}}$$

$$\text{perfectAccuracy} = \frac{\text{perfectPredictions}}{\text{totalPredictions}}$$

$$\text{accuracy} = \frac{\text{correctIndexGenerationPredictions} + \text{correctThreadCoarseningPredictions}}{\text{totalPredictions} \times 2}$$

In these formulas, *totalPredictions* is the total amount of predictions, *correctIndexGenerationPredictions* the amount of times the index generation parameter prediction matches the best, *correctThreadCoarseningPredictions* the amount of times the thread coarsening parameter prediction matches the best and *perfectPredictions* the amount of times both optimisation parameters predicted match the best.

Assess the effectiveness of the autotuner Even if the autotuner makes good predictions, that doesn't necessarily mean it will always result in performance gains, thus we ask: How high is the speedup gained by kernels optimised using the autotuner? How long does the autotuner take to make a prediction? Is the whole execution faster with the autotuner? Regarding the first question, while it is highly beneficial to train the model of the autotuner to be as accurate as possible, within the the context of our solution, the model can still be useful even if it is not making perfect predictions every time, as long as the expressions optimised using the autotuner gain enough of a speedup. Thus, another important metric of the CUDA autotuner is the speedup gained from optimised expressions. We will be collecting kernel execution times with the default optimisation parameters, with the best parameters and with the parameters predicted by the autotuner, for every expression we evaluate. This way we can calculate the average speedup of expressions optimised using the autotuner, and the average speedup gained using a perfect autotuner, for comparison. For the second and third questions, another metric is needed, it being the time it takes for the autotuner models to finish a prediction. Running the autotuner takes

some time, and if the optimised expression doesn't gain enough of a speedup, the whole performance may be slower than simply running the expression with no optimisations. Although the static and dynamic caches exist to avoid running the autotuner more than once for the same expression, the performance of the first execution is still worth looking into, as any change made to an expression may require running autotuner again.

6.2 Testing methodology

For the first goal, we are running a series of different computations, ranging from simple array additions to complex operation sequences. The computations will be executed using both the OpenCL and CUDA backends, several times each. The average run time is then used to compare the OpenCL and CUDA executions directly. Additionally, a hand coded CUDA version of some computations is implemented and executed as well, to assess how much performance is lost using Marrow compared to just CUDA. As the intent is to evaluate the CUDA backend alone, all the calculations use the default optimisation parameters. Additionally, each computation will be tested with a varying range of data sizes, which should allow us to observe the launch overhead and how execution times grow with the size of a computation, for each implementation.

Concerning the second goal, we will perform predictions using the model on dataset entries, expressions whose best optimisation parameters are already known, and check if the predictions match the expected output. Additionally, some more predictions will be made on expressions not present in the dataset, to measure the accuracy of the autotuner for new expressions. To find the best optimisation parameters for new expressions, they will be executed several times for each combination of parameters, and the parameters with the fastest average time are then considered the best, similar to how entries are created during the dataset generation (Section 5.2.1).

As a side note, in practice, dataset expressions have their best parameters added to the static cache during the dataset generation. We check the accuracy of the autotuner for expressions in the dataset to check if the models are over-fitted.

Regarding the first question of the third goal, measuring autotuner optimised kernel speedup is a process somewhat similar to measuring the model's accuracy. For each entry in the dataset, the respective expression is executed with default optimisation parameters, predicted parameters and optimal parameters several times, until a standard deviation of execution times threshold is reached or until the maximum number of executions is reached, and the average time for each parameter combination is saved. For expressions not in the dataset, they are run several times with every parameter combination, and the average execution times for default parameters, predicted parameters and optimal parameters are saved. Finally we compare the 3 average times for each expression.

When assessing the models' accuracy and effectiveness, expressions outside the dataset are generated procedurally, using the same framework that generates the dataset, by creating expressions with different data sizes and number/type of operations.

Table 6.1: Testing environment machines.

Name	Description	Processor	Memory	Graphics	Video Memory	Compute Capability
machine0	Personal laptop	Intel i7-8750H	16GB DDR4 2667MHz	NVIDIA GTX 1050Ti Mobile	4GB GDDR5	6.1

Answering the second question of the last goal is a straightforward process of measuring the average time it takes for the autotuner to return a prediction made by the models. The prediction process should always take the same amount of time regardless of inputs, but even then, we will measure the average prediction time for a few different expressions.

For the last question, we measure the time of several whole executions, including making a prediction and running the optimised expression, the intent of this being to test the worst case scenario, where the autotuner has to be used at runtime. We then time several whole executions that run expressions with default optimisation parameters, and compare the two average execution times for each expression. The number of unique expressions that execute faster with the autotuner, divided by the total number of expressions gives us the relative execution when using the autotuner.

6.3 Testing environment

We had the intention of running the benchmarks on a few different machines, a personal laptop used during development, and a couple of nodes of the cluster provided by the Department of Computer Science of FCT/UNL¹. Unfortunately, due to some constraints and difficulties, testing was only done on one machine, the laptop used for development. The hardware specifications of each machine used are listed in Table 6.1.

Table 6.1 presents the machines used to run the benchmarks described in this chapter, with relevant the hardware specs listed for each. Furthermore, *machine0* is running on Linux Ubuntu 20.04.2 LTS, using g++ compiler version 10.3.0 and CUDA toolkit version 11.2.1.

6.4 Assess the performance of the CUDA backend

In the following tests, we ran 8 different expressions, each with the intent of testing different aspects of the backend:

Basic addition: the intent is to test a very simple operation, an addition between 2 arrays;

Sequence of additions: the intent is to test how a very simple operation over a large number of containers is handled, an addition between 8 arrays;

¹<https://cluster.di.fct.unl.pt>

Modulo comparison: the intent here is to test if the generated kernels optimise out redundant expressions, the expression is the comparison of the modulo of two arrays with a handmade equivalent expression, meaning the result of the comparison is always true;

Complex expression of simple operations: the intent is to see how the backend handles a somewhat complex/large expression of simple operations over 8 containers;

Complex expression of simple operations with container repetition: similar to the last expression, but here the number of arrays used is only 2, repeatedly used throughout the expression;

Sin: here the intent is to see how a single unary operation is handled;

Complex expression of heavier operations: here the intent is to test a more complex expression, with "heavier" operations, over 4 arrays;

Complex expression of heavier operations with a low number of containers: similar to the last expression, but here only 2 containers are used in the expression.

Hopefully these will give some good insight in how the CUDA backend handles different operations, and shines some light on its advantages and drawbacks, compared to pure CUDA and the OpenCL backend.

6.4.1 How does the Marrow CUDA backend perform, compared to just CUDA?

Table 6.2: Execution times of the Marrow CUDA backend, relative to pure CUDA (positive number means a speed down and negative means a speed up).

Expression \ Data size	1000	10000	100000	1000000	10000000
$a + b$	0.63	0.54	0.18	-0.23	-0.22
$a + b + c + d + e + f + g + h$	0.35	0.33	0.07	-0.02	-0.07
$(a \% b) == (a - (a / b) * b)$	0.61	0.62	0.46	0.15	0.19
$((a * b) + (d * c)) / ((e * g) - (h * i))$	0.59	0.33	0.05	-0.01	-0.08
$((a * a) + (a * b)) / ((a * a) - (b * b))$	1.05	0.56	0.28	-0.17	-0.17
$\sin(a)$	0.56	0.60	0.21	-0.17	-0.38
$\text{pow}(\sin(a), \text{sqrt}(b)) / (\exp(c) + \sin(d))$	0.44	0.73	0.28	-0.01	-0.03
$\sin(\exp(\text{pow}(\sin(a), \text{sqrt}(b))))$	0.64	0.55	0.36	-0.05	-0.06

Table 6.2 depicts the relative execution times of Marrow using the CUDA backend, compared to hand written, functionally equivalent, CUDA expressions. Each instance

of an expression combined with a data size was executed and timed 200 times for both implementations. The obtained fractions were calculated by dividing the average of Marrow CUDA backend execution times, by the matching average of CUDA execution times, minus 1. The whole execution was timed, including memory allocations and transfers, and kernel execution, for this table.

We can see that, even if it needs some more work, the CUDA backend implementation for Marrow is somewhat successful. For larger data sizes, the CUDA only implementations aren't substantially faster than Marrow, with some expressions running practically as fast, or even very slightly faster, granted those values could be within the margin of error. The results for smaller data sizes indicate that there may still be some more room for improvement, as there is a clear overhead in the Marrow implementation.

Table 6.3: Kernel execution times of the Marrow CUDA backend, relative to pure CUDA (positive number means a speed down and negative means a speed up).

Expression \ Data size	1000	10000	100000	1000000	10000000
$a + b$	0.01	0.18	0.24	0.04	0.02
$a + b + c + d + e + f + g + h$	0.04	0.89	0.86	0.87	0.91
$(a \% b) == (a - (a / b) * b)$	0.16	0.57	2.22	6.85	11.24
$((a * b) + (d * c)) / ((e * g) - (h * i))$	0.11	0.76	0.83	0.85	0.91
$((a * a) + (a * b)) / ((a * a) - (b * b))$	0.04	0.61	2.03	2.04	2.16
$\sin(a)$	0.04	0.29	0.09	0.21	0.24
$\text{pow}(\sin(a), \text{sqrt}(b)) / (\exp(c) + \sin(d))$	0.61	1.66	2.17	1.78	2.01
$\sin(\exp(\text{pow}(\sin(a), \text{sqrt}(b))))$	0.80	3.07	7.64	6.57	7.34

Similarly to the previous table, in Table 6.3 can be seen the relative kernel execution times of Marrow using the CUDA backend, compared to hand written, functionally equivalent, CUDA expressions. Each instance of an expression combined with a data size was executed and their kernel timed 200 times for both implementations. The obtained fractions were calculated by dividing the average of Marrow CUDA backend kernel execution times, by the matching average of CUDA kernel execution times, minus 1. Only the kernel executions were timed for this table.

Unlike the total execution times, kernel times vary much more wildly. This is because memory allocations are costly operations, which hide the Marrow overhead very well when the data size isn't too small. As the table results suggest, kernel times depend on the number of threads launched, but mostly on what the kernel does. The data size, which affects the number of threads launched, does cause some variation in the relative times, but these remain mostly consistent across the same expression. The relative time variation across different expressions is almost random, but the more simple expressions in Marrow, the first sum and the sin, are closer to their CUDA only counterpart. We

can then deduce, from the consistency across the same expressions, that the cause the variation in different expressions is caused by optimisations done by the CUDA compiler, applied to the CUDA only kernels but not Marrow kernels. This is specially apparent in the third expression, which always results in same value, the CUDA compiler is able to identify this and optimise it into simply returning the value, resulting in the Marrow version of the kernel taking almost 12 times more to execute, compared to the CUDA only implementation.

There may be a number of reasons why the CUDA compiler does not perform the same optimisations on Marrow kernels, but the most likely one may be because in those, the function performed by the kernel is passed as an argument and defined separately. The CUDA only kernels used in these tests have their function hard coded inside, which may facilitate optimisations. There are some optimisations we could apply ourselves to Marrow kernels, such as saving reused container values inside a thread variable to avoid unnecessary global memory reads, but these were not implemented in this thesis. If such an optimisation was implemented, the Marrow version of the 5th expression would probably be closer to its CUDA counterpart, due to its repetitive use of some containers. We can conclude this because the 4th expression, which is similar but doesn't reuse containers, performs better relative to CUDA.

The last expression has the second highest speed down, and the reason why that is, is somewhat puzzling. There's no obvious optimisation that can be made to the expression, but the reason may be similar to the modulo expression. More work and testing is required to arrive to a conclusion regarding this expression's behaviour.

6.4.2 How does the Marrow CUDA backend perform, compared to the preexisting OpenCL backend?

Table 6.4: Execution times of the Marrow CUDA backend, relative to the Marrow OpenCL backend (positive number means a speed down and negative means a speed up).

Expression \ Data size	1000	10000	100000	1000000	10000000
$a + b$	-0.05	-0.04	-0.16	1.05	-0.26
$a + b + c + d + e + f + g + h$	-0.23	-0.14	-0.21	1.18	-0.08
$(a \% b) == (a - (a / b) * b)$	-0.06	-0.04	0.01	1.53	0.02
$((a * b) + (d * c)) / ((e * g) - (h * i))$	-0.12	-0.15	-0.20	1.22	-0.08
$((a * a) + (a * b)) / ((a * a) - (b * b))$	0.19	-0.03	-0.01	1.23	-0.22
$\sin(a)$	-0.12	-0.07	-0.06	0.43	-0.39
$\text{pow}(\sin(a), \text{sqrt}(b)) / (\exp(c) + \sin(d))$	-0.16	0.09	-0.11	1.04	-0.07
$\sin(\exp(\text{pow}(\sin(a), \text{sqrt}(b))))$	-0.06	-0.01	0.10	1.46	-0.01

In Table 6.4 are the relative execution times of Marrow using the CUDA backend,

compared to Marrow using the OpenCL backend running some expressions. Like the other 2 tables, each instance of an expression combined with a data size was executed and timed 200 times for both backends. The obtained fractions were calculated by dividing the average of Marrow CUDA backend execution times, by the matching average of Marrow OpenCL backend execution times, minus 1. The whole execution was timed, including memory allocations and transfers, and kernel execution, for this table.

The OpenCL backend has a bigger overhead compared to the CUDA backend, noticeable by the fact that if the data size isn't large enough, the CUDA backend performs better overall. An odd behaviour happening with every expression, is that when the data size is 1000000, the CUDA backend performs significantly worse than OpenCL.

Some things to note, that are not observable in the results for the sake of a more fair comparison, is that the OpenCL has start time, which is not present in CUDA. This is because OpenCL kernels have to be generated and saved in a file before they can be executed, unlike CUDA which has it's kernels defined in code, having no such need. During runtime, the OpenCL backend has to generate a kernel file if there isn't one already, and then read it from disk, which can take a considerable amount of time, before a kernel is executed for the first time, while the CUDA backend can execute kernel directly straight away with no such overhead. As the intent behind these tests is to compare the average performance of each implementation, OpenCL kernels were run once at the start without measuring the time, to eliminate the first time overhead, however this should still be taken into consideration.

6.5 Assess the accuracy of the autotuner

6.5.1 Dataset generation context

Before discussing the results, we must elaborate on the context in which the dataset is generated. In Subsection 5.2.1 we discussed what expressions are generated, and how they are generated. Part of the process of generating the dataset involves the use of cost classes, but we never mentioned what operations belong to which class, as this is something that has to be asserted through tests.

To assert the cost class of each operation, we ran expressions doing only the respective operation on large containers (container size of 10000000). Each expression was executed 1000 times. After executing everything, the average execution times were classified into 3 different classes, for both integer and float operation, using K-Means clustering. Table 6.5 presents the values obtained. As a side note, the (\leq , \geq) operations represent every comparison operator, we tested only the equals operator, but assume that all of them have the same performance cost. Similarly, only sin was tested, but we assume sin, cos and tan operations all perform similarly.

Concerning the values presented in Table 6.6 and Table 6.7, the overall accuracy values in both tables are more influenced by 1 and 2 dimensional expressions, as the number of

Table 6.5: Table with the execution time (in nanoseconds) of each operation in each testing machine, and the assigned cost class.

Operation	Avg. time in machine0	Cost class
+	1275605	1
-	1286170	1
*	1282447	1
/	1252936	1
<=>	1282017	1
%	1253192	1
+	1286921	1
-	1285628	1
*	1282586	1
/	1256689	1
<=>	1277286	1
Square root	1003798	0
Power	1597486	1
Sin	1176057	0
Exponential	4291320	2

expressions generated in these dimensions is higher than the number of 3 dimensional expressions generated.

6.5.2 Overall results

Table 6.6: Accuracy values of the autotuner model predicting optimisation parameters for expressions present in the dataset used during training.

	Accuracy(Index Generation)	Accuracy(Thread Coarsening)	Accuracy(Perfect)	Accuracy
Overall	68.81%	48.63%	33.08%	58.72%
1 Dim	46.13%	46.21%	19.27%	44.92%
2 Dims	89.92%	36.83%	28.33%	44.13%
3 Dims	70.77%	71.54%	62.69%	71.15%

The autotuner models ended up somewhat decent, but definitely not impressive and in need of improvements, as can be seen by the accuracy values in Table 6.6 and Table 6.7. For expressions inside the dataset, the autotuner only achieves a bit over half overall accuracy and only about 33.08% accuracy in predicting both optimisation parameters

Table 6.7: Accuracy values of the autotuner model predicting optimisation parameters for expressions outside of the dataset used during training.

	Accuracy(Index Generation)	Accuracy(Thread Coarsening)	Accuracy(Perfect)	Accuracy
Overall	62.10%	51.28%	30.49%	56.69%
1 Dim	43.23%	46.53%	19.52%	49.68%
2 Dims	69.68%	39.52%	12.06%	48.85%
3 Dims	79.87%	77.82%	77.69%	78.85%

correctly for an expression. For expressions outside the dataset, the autotuner maintains similar accuracy values, meaning neither of the autotuner models are over-fitted². Interestingly, the autotuner accuracy increases with the number of dimensions in an expression, as can be observed in both tables, this could mean that optimisation parameters are more effective in higher dimensions and thus the autotuner models learn more easily from entries in higher dimensions. One reason we believe the accuracy values aren't great is because of a limited dataset, due to the heavy use of templates, there was an extreme number of specialisations when compiling the dataset generator, such that it was not possible to generate a large number of entries for the dataset.

6.5.3 Thread coarsening model

The model predicting the value for thread coarsening may need some improvements, as overall, it is able to predict the best value nearly half of the time, out of 5 possible values. The model is worse for expressions in 1 and 2 dimensions, it isn't able to predict the best coarsening value even half the time. For 3 dimensional expressions however, the model achieves a much higher accuracy. This is probably because different coarsening values make more of an impact on higher dimension expressions.

6.5.4 Index generation model

The prediction of the index generation stage seems like it greatly needs improvements, as only predicting the best value 3 out of 5 times, for a binary choice, isn't very good. The overall value isn't what should be focused on however, because this optimisation does effectively nothing for 1 dimensional expressions, as the method for generating indices cannot change in 1 dimension, this means the "best" value for those expressions is effectively random, leading to the observed accuracy in such expressions. Expressions in 2 and 3 dimensions are affected by this optimisation, and as such, the accuracy when predicting the best value for those expressions is much higher. Some work is still needed, while the accuracy is indeed high, for a binary decision it could be higher, this lack of

²Over-fitted in this context would mean that a model does not generalise properly after training, it has high accuracy for entries within the dataset, and low accuracy for entries without.

accuracy could be related to the fact that the model is also trained for 1 dimensional expressions when this is completely unnecessary, possibly skewing the model.

6.6 Assess the effectiveness of the autotuner

6.6.1 How high is the speedup gained by kernels optimised using the autotuner?

In Figure 6.1 are the distributions of relative kernel execution times of expressions within the dataset optimised by the autotuner, and of the same expressions executed using optimal optimisation parameters, both compared to expressions executed using default optimisation parameters. The base (no optimisations) and "Perfect" (optimal optimisation parameters) execution times were obtained during the dataset generation and stored. Execution times for expressions optimised with the autotuner were obtained with a framework similar to the one used to generate the dataset. All expressions used in the calculation of these values are present in the dataset used to train the autotuner model.

We can observe that most optimisations don't have a huge impact on kernel performance, but they do improve performance on average and sometimes performance is greatly improved. As suspected earlier, the impact on performance seems to increase with the number of dimensions of an expression. The reason these values are so low is because for most expressions, particularly expressions operating on small containers, the kernel performance does not change, regardless of the optimisations applied. Only expressions operating on large containers will see a noticeable difference on their kernel times, with optimisations. Additionally, while not observable in the distributions, most of the expressions with higher execution times are expressions operating on smaller containers, for which the kernel times vary more wildly, but are less affected by optimisations. For expressions within the dataset, our autotuner's performance is even comparable to the best case, getting similar but slightly worse results.

Similarly, Figure 6.2 has the distributions of relative kernel execution times of expressions outside of the dataset optimised by the autotuner, and of the same expressions executed using optimal optimisation parameters, both compared to expressions executed using default optimisation parameters. The base (no optimisations), best (best optimisation parameters) and "Perfect" (optimal optimisation parameters) execution times used to calculate the table values were obtained using a framework similar to the one used to generate the dataset. All expressions used in the calculation of these values are not present in the dataset used to train the autotuner model.

We can see that on average, the autotuner has a diminished effect on expressions outside of the dataset, and interestingly, even see an inverse trend, where the autotuner is more effective in predicting good parameters for lower dimension expressions. Considering the accuracy values observed earlier and the distributions for expressions inside the dataset, these relative time values are a bit puzzling, but this may be because the

autotuner has more trouble predicting good parameters specifically for expressions that benefit more from optimisations.

6.6.2 How long does the autotuner take to make a prediction?

We wrote a few different expressions, and used the autotuner to make 10000 predictions for each. On average, it takes about 0.2 milliseconds for both of the models of the autotuner to make a prediction, and while this number may seem rather small, for most kernels, except for those operating on very large containers, 0.2 milliseconds is a lot of time. Of course this depends on what the kernel does, but 0.2 milliseconds is more or less how long it takes for a kernel operating on containers with a size in the range of 100000 to 1000000, to finish execution. This is to say, even if the autotuner optimisations are very effective, it should not be worth using the autotuner for expressions with containers of lower size.

6.6.3 Is the whole execution faster with the autotuner?

Table 6.8 presents the average execution time of expressions optimised using autotuner, compared to the average time of expressions running with default optimisations. Like some of the previous tables, each expression instance was executed and timed 200 times, and the fraction obtained is the average autotuner optimised times divided by the matching base times. For this table, all of the execution was timed, including performing the prediction and executing the expression.

Table 6.8: Relative execution time of expressions optimised using the autotuner.

Expression \ Data size	1000	10000	100000	1000000	10000000
$a + b$	0.60	0.63	0.45	-0.47	0.02
$a + b + c + d + e + f + g + h$	0.59	0.55	0.22	-0.47	0.01
$(a \% b) == (a - (a / b) * b)$	0.59	0.65	0.47	-0.47	-0.01
$((a * b) + (d * c)) / ((e * g) - (h * i))$	0.29	0.53	0.23	-0.47	0.01
$((a * a) + (a * b)) / ((a * a) - (b * b))$	0.24	0.60	0.49	-0.45	0.00
$\sin(a)$	0.77	0.72	0.53	-0.32	0.03
$\text{pow}(\sin(a), \text{sqrt}(b)) / (\exp(c) + \sin(d))$	0.62	0.33	0.24	-0.41	0.03
$\sin(\exp(\text{pow}(\sin(a), \text{sqrt}(b))))$	0.64	0.62	0.37	-0.31	0.03

As expected, expressions with small containers did not benefit at all from the autotuner optimisation, as it takes too long to make a prediction. Expressions with large containers can mask the prediction time and, in the case of expressions operating on containers with a size of 1000000, benefit overall from the autotuner optimisations. It is puzzling however, why expressions with the largest containers are unaffected, neither

benefiting nor being hindered by the autotuner. It could be because of some overhead with extremely large blocks of memory on the GPU, but further testing is required to verify this.

In the end, these results demonstrate that even in the worst case, some expressions can still benefit from the autotuner. Furthermore, they are also a good demonstration why the static and dynamic caches are such a good utility for Marrow, by enabling to run optimised expressions without having to run the autotuner, if the program is compiled again, after the first execution, or if the same expression is executed more than once.

6.7 Concluding remarks

In the end, the results didn't turn out quite like we expected, the CUDA backend is a good addition to Marrow, but the kernel implementations still need some work. The autotuner should work but also needs some improving.

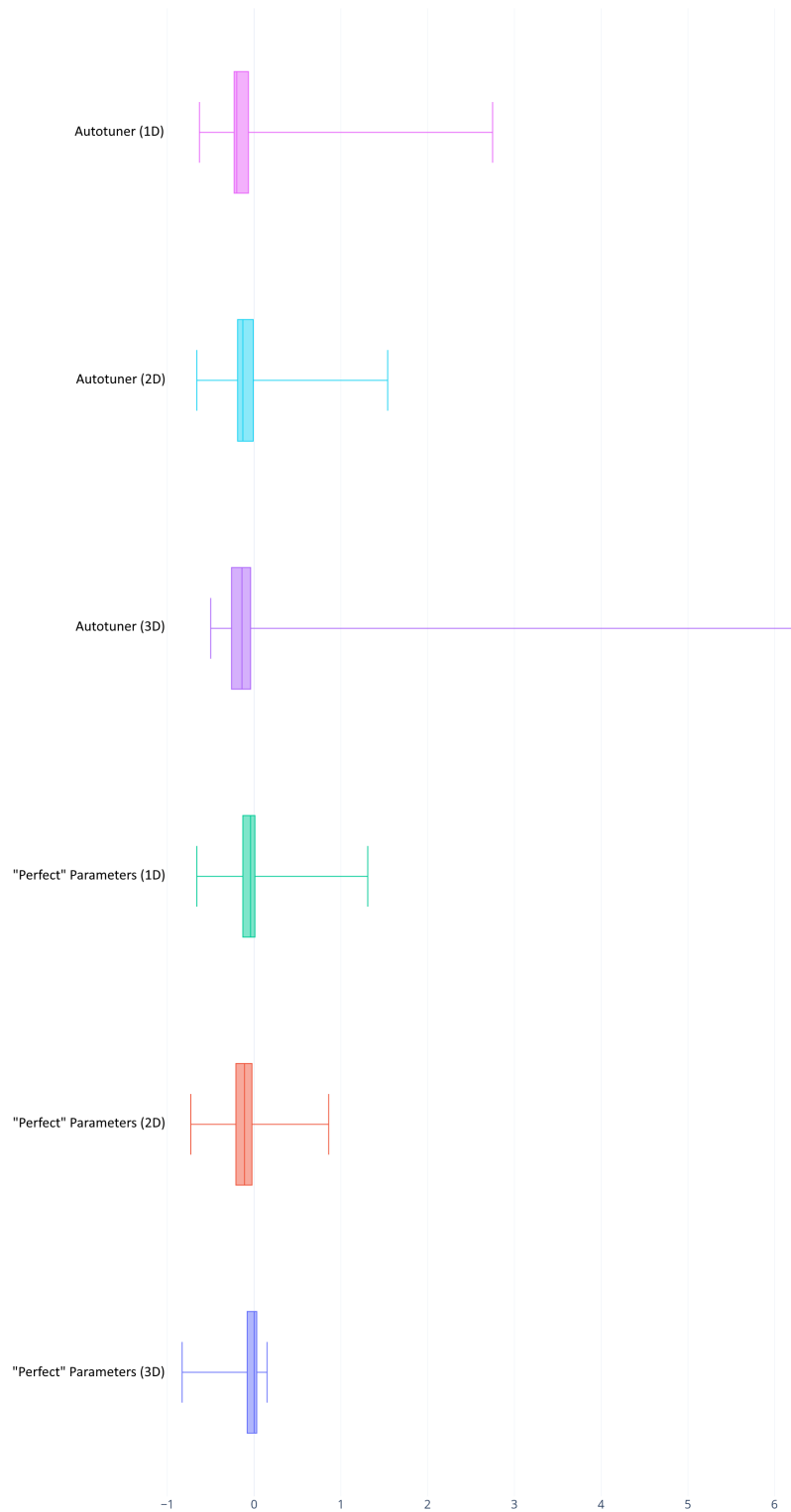


Figure 6.1: Box charts of relative kernel execution times for expressions within the dataset, optimised using the autotuner and using optimal parameters.

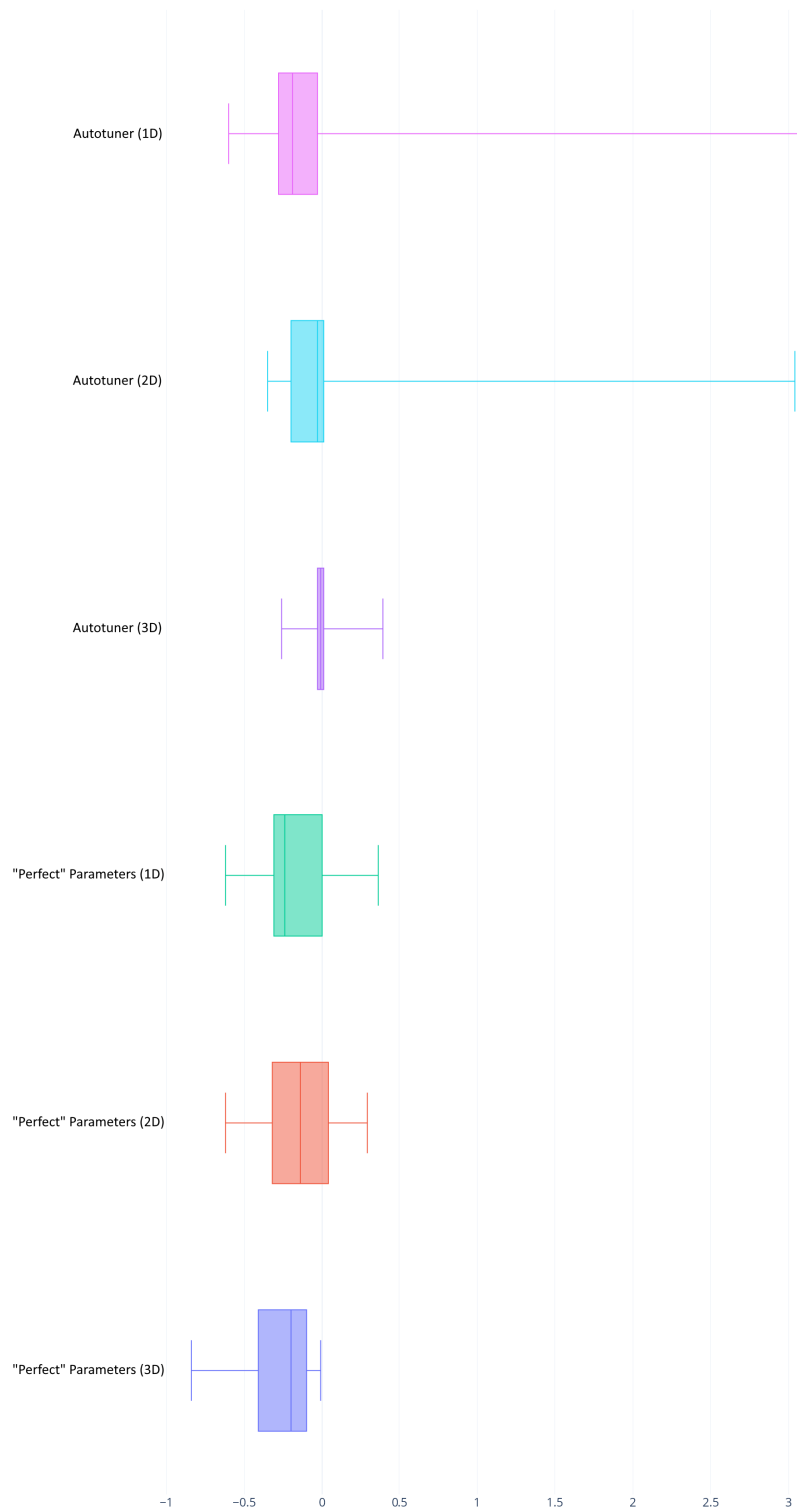


Figure 6.2: Box charts of relative kernel execution times for expressions outside the dataset, optimised using the autotuner and using optimal parameters.

CONCLUSIONS AND FUTURE WORK

In this chapter, we will be presenting the conclusions, assessing if everything we set out to do has been accomplished and analysing the final results. We will also explore the challenges we faced during development, what could be done better and finally, suggest some possibilities of what could be done in the future.

7.1 Conclusions

The two main objectives in this thesis were to implement a CUDA backend to Marrow, and then add an autotuner, based in machine learning, for it. Overall, both of these objectives have been accomplished, however not all of the smaller tasks for each of the objectives has been completed. The CUDA backend does work, it is fully capable of running Marrow expressions and returning their correct result, however some bugs still need to be ironed out. The autotuner uses a neural network that can be trained and used to predict optimisation parameters, and is well integrated into the execution of expressions in Marrow. But the dataset generator is extremely resource intensive to both compile and run, and maybe could be optimised.

There were some roadblocks along the way during the development, initially there was a small attempt to make Marrow compatible with Windows, to ease development, but we realised it would take far more work than its worth. Due to the extremely high use of templates in Marrow, errors could sometimes be harder to identify, slowing down development. Additionally, the dataset generation, as well as functions that run an expression using runtime optimisations parameters, result in a large amount of complex template specifications, which can substantially extend the compilation time and memory usage, this forced the dataset to have a reduced size and needing to be generated in chunks. Finally, there were the issues with saving and loading an OpenNN neural network, discussed earlier in 5.3, which took some time to figure out and implement a work around. There were also a few issues regarding the dataset being incorrectly generated, which led us to very bad results.

In the end, the results were not fully what we expected, the CUDA kernel implementations don't perform optimally and the autotuner isn't as effective as it could be, both in its accuracy and effectiveness. However, both solutions work, the CUDA backend performs nicely overall and the autotuner can be effective for some expressions.

7.2 Future Work

There is still plenty of work that could be done in the CUDA backend, for starters, the current CUDA backend only supports one device, so a new implementation can be made to support multiple GPUs. The CUDA backend supports most functionality, however there is some missing, such as the sort skeleton. Furthermore, the map kernels may be modified to use local memory when the same container is used multiple times in the same expression, and the function passed to the kernels may be modified to potentially allow the CUDA compiler to better optimise it, such as when code is redundant. Finally, it may be possible to mitigate some of the logistical overhead introduced by Marrow, by using sub-allocation, that is, by allocating large chunks of GPU memory in advance, and then when smaller containers are allocated, they are instead assigned a slice of the already allocated memory chunks.

Regarding the CUDA autotuner, a few alternative implementations could be tested against the current one, like trying another library or prediction algorithm entirely. Some new features could also be added and tested. New optimisations could be implemented on the map like using different types of memory, or maybe even on other kernels, and have their parameters predicted as well. Compilation times for the dataset generator are extremely inflated due to all the template specialisations, and may be optimised or implemented differently to reduce compilation times. Lastly, currently the neural networks of the autotuner are trained using and make predictions for all kinds of expressions, however this isn't necessary and may even be detrimental, for example, the method to generate indices cannot change for 1 dimensional expressions, so the network for index generation doesn't need to make predictions for such expressions, and using them during training may cause the predictions to be biased. Thus, the autotuner implementation should be changed so that each network is only trained by, and only makes predictions on relevant expressions.

BIBLIOGRAPHY

- [1] F. Alexandre, R. Marques, and H. Paulino. “On the support of task-parallel algorithmic skeletons for multi-GPU computing”. In: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. Ed. by Y. Cho et al. ACM, 2014, pp. 880–885. DOI: [10.1145/2554850.2555018](https://doi.org/10.1145/2554850.2555018). URL: <https://doi.org/10.1145/2554850.2555018> (cit. on p. 21).
- [2] Artelnics. *OpenNN*. <https://www.opennn.net/>. Last visited in November 2021. 2021 (cit. on p. 47).
- [3] A. H. Ashouri et al. “A Bayesian network approach for compiler auto-tuning for embedded processors”. In: *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. 2014, pp. 90–97. DOI: [10.1109/ESTIMedia.2014.6962349](https://doi.org/10.1109/ESTIMedia.2014.6962349) (cit. on pp. 17, 18).
- [4] A. H. Ashouri et al. “COBAYN: Compiler Autotuning Framework Using Bayesian Networks”. In: *ACM Trans. Archit. Code Optim.* 13.2 (June 2016). ISSN: 1544-3566. DOI: [10.1145/2928270](https://doi.org/10.1145/2928270). URL: <https://doi.org/10.1145/2928270> (cit. on p. 15).
- [5] M. Bailey. “Introduction to the Vulkan Computer Graphics API”. In: *ACM SIGGRAPH 2020 Courses*. SIGGRAPH ’20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450379724. DOI: [10.1145/3388769.3407508](https://doi.org/10.1145/3388769.3407508). URL: <https://doi.org/10.1145/3388769.3407508> (cit. on p. 1).
- [6] CERN. *Machine Learning and Data Analytics*. <https://openlab.cern/ml-da>. Last visited in February 2021. 2021 (cit. on p. 1).
- [7] A. Collins et al. “MaSiF: Machine learning guided auto-tuning of parallel skeletons”. In: *20th Annual International Conference on High Performance Computing*. 2013, pp. 186–195. DOI: [10.1109/HiPC.2013.6799098](https://doi.org/10.1109/HiPC.2013.6799098) (cit. on pp. 17, 18).
- [8] A. Ernstsson, L. Li, and C. W. Kessler. “SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems”. In: *Int. J. Parallel Program.* 46.1 (2018), pp. 62–80. DOI: [10.1007/s10766-017-0490-5](https://doi.org/10.1007/s10766-017-0490-5). URL: <https://doi.org/10.1007/s10766-017-0490-5> (cit. on p. 1).

- [9] T. L. Falch and A. C. Elster. “Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications”. In: *Concurrency and Computation: Practice and Experience* 29.8 (2017). e4029 cpe.4029, e4029. doi: <https://doi.org/10.1002/cpe.4029>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4029>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4029> (cit. on pp. 17, 18).
- [10] T. D. Han and T. S. Abdelrahman. “Use of Synthetic Benchmarks for Machine-Learning-Based Performance Auto-Tuning”. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2017, pp. 1350–1361. doi: [10.1109/IPDPSW.2017.108](https://doi.org/10.1109/IPDPSW.2017.108) (cit. on pp. 17, 19, 20).
- [11] T. D. Han and T. S. Abdelrahman. *Automatic Tuning of Local Memory Use on GPGPUs*. 2014. arXiv: [1412.6986 \[cs.DC\]](https://arxiv.org/abs/1412.6986) (cit. on pp. 17, 19, 20).
- [12] M. Harris et al. “Optimizing parallel reduction in CUDA”. In: *Nvidia developer technology* 2.4 (2007), p. 70 (cit. on p. 36).
- [13] T. Henriksen et al. “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by A. Cohen and M. T. Vechev. ACM, 2017, pp. 556–571. doi: [10.1145/3062341.3062354](https://doi.org/10.1145/3062341.3062354). URL: <https://doi.org/10.1145/3062341.3062354> (cit. on p. 1).
- [14] J. M. Lourenço. *The NOVAthesis L^AT_EX Template User’s Manual*. NOVA University Lisbon. 2021. URL: <https://github.com/joaomlourenco/novathesis/raw/master/template.pdf> (cit. on p. v).
- [15] A. Magni, C. Dubach, and M. O’Boyle. “Automatic Optimization of Thread-Coarsening for Graphics Processors”. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT ’14. Edmonton, AB, Canada: Association for Computing Machinery, 2014, pp. 455–466. ISBN: 9781450328098. doi: [10.1145/2628071.2628087](https://doi.org/10.1145/2628071.2628087). URL: <https://doi.org/10.1145/2628071.2628087> (cit. on pp. 17, 18).
- [16] R. Marques et al. “Algorithmic Skeleton Framework for the Orchestration of GPU Computations”. In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. Ed. by F. Wolf, B. Mohr, and D. an Mey. Vol. 8097. Lecture Notes in Computer Science. Springer, 2013, pp. 874–885. doi: [10.1007/978-3-642-40047-6_86](https://doi.org/10.1007/978-3-642-40047-6_86). URL: https://doi.org/10.1007/978-3-642-40047-6_86 (cit. on p. 21).
- [17] Microsoft Corporation. *Direct3D*. <https://docs.microsoft.com/en-us/windows/win32/direct3d>. Last visited in February 2021. 2021 (cit. on p. 1).

-
- [18] J. Nickolls et al. “Scalable Parallel Programming with CUDA”. In: *ACM Queue* 6.2 (2008), pp. 40–53. DOI: [10.1145/1365490.1365500](https://doi.org/10.1145/1365490.1365500). URL: <https://doi.org/10.1145/1365490.1365500> (cit. on pp. 1, 6).
- [19] NVIDIA Corporation. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. <https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf>. Last visited in January 2021. 2020 (cit. on pp. 1, 6).
- [20] NVIDIA Corporation. *NVIDIA TURING GPU ARCHITECTURE*. <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>. Last visited in January 2021. 2018 (cit. on pp. 1, 6).
- [21] C. J. Rossbach et al. “Dandelion: a compiler and runtime for heterogeneous systems”. In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP ’13, Farmington, PA, USA, November 3-6, 2013*. Ed. by M. Kaminsky and M. Dahlin. ACM, 2013, pp. 49–68. DOI: [10.1145/2517349.2522715](https://doi.org/10.1145/2517349.2522715). URL: <https://doi.org/10.1145/2517349.2522715> (cit. on p. 1).
- [22] K. O. Stanley and R. Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. DOI: [10.1162/106365602320169811](https://doi.org/10.1162/106365602320169811). eprint: <https://doi.org/10.1162/106365602320169811>. URL: <https://doi.org/10.1162/106365602320169811> (cit. on p. 15).
- [23] J. E. Stone, D. Gohara, and G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Comput. Sci. Eng.* 12.3 (2010), pp. 66–73. DOI: [10.1109/MCSE.2010.69](https://doi.org/10.1109/MCSE.2010.69). URL: <https://doi.org/10.1109/MCSE.2010.69> (cit. on pp. 1, 6).
- [24] H. K. Swamy. “Structured Parallel Programming Patterns for Efficient Computation by Michael McCool, Arch D. Robison and James Reinders”. In: *SIGSOFT Softw. Eng. Notes* 37.6 (Nov. 2012), p. 43. ISSN: 0163-5948. DOI: [10.1145/2382756.2382773](https://doi.org/10.1145/2382756.2382773). URL: <https://doi.org/10.1145/2382756.2382773> (cit. on p. 22).
- [25] B. Taylor, V. S. Marco, and Z. Wang. “Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems”. In: *SIGPLAN Not.* 52.5 (June 2017), pp. 11–20. ISSN: 0362-1340. DOI: [10.1145/3140582.3081040](https://doi.org/10.1145/3140582.3081040). URL: <https://doi.org/10.1145/3140582.3081040> (cit. on pp. 17, 19).
- [26] The OpenACC Organization. *OpenACC*. <https://www.openacc.org>. Last visited in January 2021. 2011 (cit. on p. 6).
- [27] The OpenMP ARB (Architecture Review Boards). *OpenMP*. <https://www.openmp.org>. Last visited in February 2021. 2012 (cit. on p. 19).

- [28] T. Vergilio and M. Ramachandran. “Non-functional Requirements for Real World Big Data Systems - An Investigation of Big Data Architectures at Facebook, Twitter and Netflix”. In: *Proceedings of the 13th International Conference on Software Technologies, ICSOFT 2018, Porto, Portugal, July 26-28, 2018*. Ed. by L. A. Maciaszek and M. van Sinderen. SciTePress, 2018, pp. 867–874. DOI: [10.5220/0006825408670874](https://doi.org/10.5220/0006825408670874). URL: <https://doi.org/10.5220/0006825408670874> (cit. on p. 1).
- [29] M. Wahib and N. Maruyama. “Scalable Kernel Fusion for Memory-Bound GPU Applications”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*. Ed. by T. Damkroger and J. J. Dongarra. IEEE Computer Society, 2014, pp. 191–202. DOI: [10.1109/SC.2014.21](https://doi.org/10.1109/SC.2014.21). URL: <https://doi.org/10.1109/SC.2014.21> (cit. on pp. 25, 32).
- [30] Z. Wang, D. Grewe, and M. F. P. O’boyle. “Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems”. In: *ACM Trans. Archit. Code Optim.* 11.4 (Dec. 2014). ISSN: 1544-3566. DOI: [10.1145/2677036](https://doi.org/10.1145/2677036). URL: <https://doi.org/10.1145/2677036> (cit. on pp. 17, 19, 49).
- [31] Yixun Liu, E. Z. Zhang, and X. Shen. “A cross-input adaptive framework for GPU program optimizations”. In: *2009 IEEE International Symposium on Parallel Distributed Processing*. 2009, pp. 1–10. DOI: [10.1109/IPDPS.2009.5160988](https://doi.org/10.1109/IPDPS.2009.5160988) (cit. on p. 17).

