Interpreter and Transpiler for Simple Expressions on Nvidia GPUs using Julia

Daniel Roth

MASTERARBEIT

eingereicht am

Fachhochschul-Masterstudiengang

Software Engineering

in Hagenberg

im Januar 2025

Advisor:

DI Dr. Gabriel Kronberger

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere. This printed copy is identical to the submitted electronic version.

Hagenberg, January 1, 2025

Daniel Roth

Contents

Declaration iv

Abstract vii

Kurzfassung viii

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Research Question . . . 2
- 1.3 Thesis Structure . . . 2

2 Fundamentals and Related Work 4
- 2.1 Equation learning . . . 4
- 2.2 GPGPU . . . 6
  - 2.2.1 Programming GPUs . . . 8
  - 2.2.2 PTX . . . 16
- 2.3 Compilers . . . 17
  - 2.3.1 Transpilers . . . 19
  - 2.3.2 Interpreters . . . 20

3 Concept and Design 21
- 3.1 Requirements . . . 21
- 3.2 Architecture . . . 23
  - 3.2.1 Pre-Processing . . . 23
  - 3.2.2 Interpreter . . . 24
  - 3.2.3 Transpiler . . . 27

4 Implementation 30
- 4.1 Technologies . . . 30
- 4.2 Expression Processing . . . 30
- 4.3 Interpreter . . . 30
- 4.4 Transpiler . . . 30

5 Evaluation 31
- 5.1 Test environment . . . 31
- 5.2 Results . . . 31

5.2.1 Interpreter 31
5.2.2 Performance tuning 31
5.2.3 Transpiler 31
5.2.4 Performance tuning 31
5.2.5 Comparison 32

6 Conclusion **33**
6.1 Future Work 33

References **34**
Literature 34
Online sources 39

Abstract

This should be a 1-page (maximum) summary of your work in English.

Kurzfassung

An dieser Stelle steht eine Zusammenfassung der Arbeit, Umfang max. 1 Seite. ...

Chapter 1

Introduction

This chapter provides an entry point for this thesis. First the motivation of exploring this topic is presented. In addition, the research questions of this thesis are outlined. Lastly the methodology on how to answer these questions will be explained.

1.1 Background and Motivation

Optimisation and acceleration of program code is a crucial part in many fields. For example video games need optimisation to lower the minimum hardware requirements which allows more people to run the game, increasing sales. Another example where optimisation is important are computer simulations. For those, optimisation is even more crucial, as this allows the scientists to run more detailed simulations or get the simulation results faster. Equation learning or symbolic regression is another field that can heavily benefit from optimisation. One part of equation learning, is to evaluate the expressions generated by a search algorithm which can make up a significant portion of the runtime. This thesis is concerned with optimising the evaluation part to increase the overall performance of equation learning algorithms.

The following expression $5 - \text{abs}(x_1) * \text{sqrt}(x_2)/10 + 2\hat{}x_3$ which contains simple mathematical operations as well as variables x_n and parameters p_n is one example that can be generated by the equation learning algorithm, Usually an equation learning algorithm generates multiple of such expressions per iteration. Out of these expressions all possibly relevant ones have to be evaluated. Additionally, multiple different values need to be inserted for all variables and parameters, drastically increasing the amount of evaluations that need to be performed.

In his blog, Sutter (2004) described how the free lunch is over in terms of the ever-increasing performance of hardware like the CPU. He states that to gain additional performance, developers need to start developing software for multiple cores and not just hope that on the next generation of CPUs the program magically runs faster. While this approach means more development overhead, a much greater speed-up can be achieved. However, in some cases the speed-up achieved by this is still not large enough and another approach is needed. One of these approaches is the utilisation of Graphics Processing Units (GPUs) as an easy and affordable option as compared to compute clusters. Especially when talking about performance per dollar, GPUs are very

inexpensive as found by Brodtkorb et al. (2013). Michalakes and Vachharajani (2008) have shown a noticeable speed-up when using GPUs for weather simulation. In addition to computer simulations, GPU acceleration also can be found in other places such as networking (S. Han et al., 2010) or structural analysis of buildings (Georgescu et al., 2013).

1.2 Research Question

With these successful implementations of GPU acceleration, this thesis also attempts to improve the performance of evaluating mathematical equations, generated at runtime for symbolic regression using GPUs. Therefore, the following research questions are formulated:

- How can simple arithmetic expressions that are generated at runtime be efficiently evaluated on GPUs?
- Under what circumstances is the evaluation of simple arithmetic expressions faster on a GPU than on a CPU?
- Under which circumstances is the interpretation of the expressions on the GPU or the translation to the intermediate language Parallel Thread Execution (PTX) more efficient?

Answering the first question is necessary to ensure the approach of this thesis is actually feasible. If it is feasible, it is important to evaluate if evaluating the expressions on the GPU actually improves the performance over a parallelised CPU evaluator. To answer if the GPU evaluator is faster than the CPU evaluator, the last research question is important. As there are two major ways of implementing an evaluator on the GPU, they need to be implemented and evaluated to finally state if evaluating expressions on the GPU is faster and if so, which type of implementation results in the best performance.

1.3 Thesis Structure

In order to answer the research questions, this thesis is divided into the following chapters:

Chapter 2: Fundamentals and Related Work
In this chapter, the topic of this thesis is explored. It covers the fundamentals of equation learning and how this thesis fits into this field of research. In addition, the fundamentals of General Purpose GPU computing and how interpreters and transpilers work are explained. Previous research already done within this topic is also explored.

Chapter 3: Concept and Design
Within this chapter, the concepts of implementing the GPU interpreter and transpiler are explained. How these two prototypes can be implemented disregarding concrete technologies is part of this chapter.

Chapter 4: Implementation
This chapter explains the implementation of the GPU interpreter and transpiler.

The details of the implementation with the used technologies are covered, such as the interpretation process and the transpilation of the expressions into Parallel Thread Execution (PTX) code.

Chapter 5: Evaluation

The software and hardware requirements and the evaluation environment are introduced in this chapter. All three evaluators will be compared against each other and the form of the expressions used for the comparisons are outlined. The comparison will not only include the time taken for the pure evaluation, but it will also include the overhead, like PTX code generation. Finally, the results of the comparison of the GPU and CPU evaluators are presented to show which of these yields the best performance.

Chapter 6: Conclusion

In the final chapter, the entire work is summarised. A brief overview of the implementation as well as the evaluation results will be provided. Additionally, an outlook of possible future research is given.

With this structure the process of creating and evaluating a basic interpreter on the GPU as well as a transpiler for creating PTX code is outlined. Research is done to ensure the implementations are relevant and not outdated. Finally, the evaluation results will answer the research questions and determine if expressions generated at runtime can be evaluated more efficiently on the GPU than on the CPU.

Chapter 2

Fundamentals and Related Work

The goal of this chapter is to provide an overview of equation learning or symbolic regression to establish common knowledge of the topic and problem this thesis is trying to solve. First the field of equation learning is explored which helps to contextualise the topic of this thesis. The main part of this chapter is split into two sub-parts. The first part is exploring research that has been done in the field of general purpose computations on the GPU (GPGPU) as well as the fundamentals of it. Focus lies on exploring how graphics processing units (GPUs) are used to achieve substantial speed-ups and when and where they can be effectively employed. The second part describes the basics of how interpreters and compilers are built and how they can be adapted to the workflow of programming GPUs. When discussing GPU programming concepts, the terminology used is that of Nvidia and may differ from that used for AMD GPUs.

2.1 Equation learning

Equation learning is a field of research that can be used for understanding and discovering equations from a set of data from various fields like mathematics and physics. Data is usually much more abundant while models often are elusive which is demonstrated by Guillemot (2022) where they explain how validating the models against large amounts of data is a big part in creating such models. Because of this effort, generating equations with a computer can more easily lead to discovering equations that describe the observed data. Brunton et al. (2016) describe an algorithm that leverages equation learning to discover equations for physical systems. A more literal interpretation of equation learning is demonstrated by Pfahler and Morik (2020). They use machine learning to learn the form of equations. Their aim was to simplify the discovery of relevant publications by the equations they use and not by technical terms, as they may differ by the field of research. However, this kind of equation learning is not relevant for this thesis.

Symbolic regression is a subset of equation learning, that specialises more towards discovering mathematical equations. A lot of research is done in this field. Using genetic programming (GP) for different problems, including symbolic regression, was first described by Koza (1994). He described that finding a computer program to solve a problem for a given input and output, can be done by traversing the search space of all solutions. This fits well for the goal of symbolic regression, where a mathematical

expression needs to be found to describe a problem with specific inputs and outputs. Later, Koza (2010) provided an overview of results that were generated with the help of GP and were competitive with human solutions, showing how symbolic regression is a useful tool. In their book Symbolic Regression, Kronberger et al. (2024) show how symbolic regression can be applied for real world scenarios. They also describe symbolic regression in great detail, while being tailored towards beginners and experts alike.

Keijzer (2004) and Korns (2011) presented ways of improving the quality of symbolic regression algorithms, making symbolic regression more feasible for problem-solving. Bartlett et al. (2024) describe an exhaustive approach for symbolic regression which can find the true optimum for perfectly optimised parameters while retaining simple and interpretable results. Alternatives to GP for symbolic regression also exist with one proposed by Jin et al. (2020). Their approach increased the quality of the results noticeably compared to GP alternatives. Another alternative to heuristics like GP is the usage of neural networks. One such alternative has been introduced by Martius and Lampert (2016) where they used a neural network for their equation learner with mixed results. Later, an extension has been provided by Sahoo et al. (2018). They introduced the division operator, which led to much better results. Further improvements have been described by Werner et al. (2021) with their informed equation learner. By incorporating domain expert knowledge they could limit the search space and find better solutions for particular domains. One drawback of these three implementations is the fact that their neural networks are fixed. An equation learner which can change the network at runtime and therefore evolve over time is proposed by Dong et al. (2024). Their approach further improved the results of neural network equation learners. In their work, Lemos et al. (2022) also used a neural network for symbolic regression. They were able to find an equivalent to Newton's law of gravitation and rediscovered Newton's second and third law only with trajectory data of bodies of our solar system. Although these laws were already known, this research has shown how neural networks and machine learning in general have great potential. An implementation for an equation learner in the physics domain is proposed by Sun et al. (2023). Their algorithm was specifically designed for nonlinear dynamics often occurring in physical systems. When compared to other implementations their equation learner was able to create better results but has the main drawback of high computational cost. As seen by these publications, increasing the quality of generated equations and also increasing the speed of finding these equations is a central part in symbolic regression and equation learning in general.

As described earlier, the goal of equation learning is to find an expression that fits a given set of data. The data usually consists of a set of inputs that have been applied to the unknown expression and the output after the input has been applied. An example for such data is described by Werner et al. (2021). In one instance they want to find the power loss formula for an electric machine. They used four inputs, direct and quadratic current as well as temperature and motor speed, and they have an observed output which is the power loss. Now for an arbitrary problem with different input and outputs, the equation learner tries to find an expression that fits this data (Koza, 1994). Fitting in this context means that when the input is applied to the expression, the result will be the same as the observed output. In order to avoid overfitting Bomarito et al. (2022) have proposed a way of using Bayesian model selection to combat overfitting and reduce the complexity of the generated expressions. This also helps with making the expressions

more generalisable and therefore be applicable to unseen inputs. A survey conducted by Dabhi and Chaudhary (2012) shows how overfitting is not desirable and why more generalisable solutions are preferred. To generate an equation, first the operators need to be defined that make up the equation. It is also possible to define a maximum length for an expression as proposed by Bartlett et al. (2024). Expressions also consist of constants as well as variables which represent the inputs. Assuming that a given problem has three variables, the equation learner could generate an expression as seen in 2.1 where x_n are the variables and O is the output which should correspond to the observed output for the given variables.

$$O = 5 - \text{abs}(x_1) * \text{sqrt}(x_2)/10 + 2\hat{}x_3 \tag{2.1}$$

A typical equation learner generates multiple expressions at once. If the equation learner generates 300 expressions and each expression needs to be evaluated 50 times to get the best parametrisation for each of these expressions, the total number of evaluations is $300 * 50 = 15\,000$. However, it is likely that multiple runs or generations in the context of GP need to be performed. The number of generations is dependent to the problem, but assuming a maximum of 100 generations, the total number of evaluations is equal to $300 * 50 * 100 = 1\,500\,000$. These values have been taken from the equation learner for predicting discharge voltage curves of batteries as described by Kronberger et al. (2024). Their equation learner converged after 54 generations, resulting in evaluating 800 000 expressions. Depending on the complexity of the generated expressions, performing all of these evaluations takes up a lot of the runtime. Their results took over two days on an eight core desktop CPU. While they did not provide runtime information for all problems they tested, the voltage curve prediction was the slowest. The other problems were in the range of a few seconds and up to a day. Especially the problems that took several hours to days to finish show, that there is still room for performance improvements. While a better CPU with more cores can be used, it is interesting to determine, if using Graphics cards can yield noticeable better performance or not, which is the goal of this thesis.

2.2 General Purpose Computation on Graphics Processing Units

Graphics cards (GPUs) are commonly used to increase the performance of many different applications. Originally they were designed to improve performance and visual quality in games. Dokken et al. (2005) first described the usage of GPUs for general purpose programming (GPGPU). They have shown how the graphics pipeline can be used for GPGPU programming. Because this approach also requires the programmer to understand the graphics terminology, this was not a great solution. Therefore, Nvidia released CUDA[1] in 2007 with the goal of allowing developers to program GPUs independent of the graphics pipeline and terminology. A study of the programmability of GPUs with CUDA and the resulting performance has been conducted by Huang et al. (2008). They found that GPGPU programming has potential, even for non-embarassingly parallel problems. Research is also done in making the low level CUDA development simpler. T. D. Han and Abdelrahman (2011) have described a directive-based language to make

[1] https://developer.nvidia.com/cuda-toolkit

development simpler and less error-prone, while retaining the performance of handwritten code. To drastically simplify CUDA development, Besard et al. (2019b) showed that it is possible to develop with CUDA in the high level programming language Julia[2] with similar performance to CUDA written in C. In a subsequent study W.-C. Lin and McIntosh-Smith (2021) found, that high performance computing (HPC) on the CPU and GPU in Julia performs similar to HPC development in C. This means that Julia can be a viable alternative to Fortran, C and C++ in the HPC field. Additional Julia has the benefit of developer comfort since it is a high level language with modern features such as a garbage-collector. Besard et al. (2019a) have also shown how the combination of Julia and CUDA help in rapidly developing HPC software. While this thesis in general revolves around CUDA, there also exist alternatives by AMD called ROCm[3] and a vendor independent alternative called OpenCL[4]. If not specified otherwise, the following section and its subsections use the information presented by Nvidia (2025b) in their CUDA programming guide.

While in the early days of GPGPU programming a lot of research has been done to assess if this approach is feasible, it now seems obvious to use GPUs to accelerate algorithms. GPUs have been used early to speed up weather simulation models. Michalakes and Vachharajani (2008) proposed a method for simulating weather with the Weather Research and Forecast (WRF) model on a GPU. With their approach, they reached a speed-up of 5 to 2 for the most compute intensive task, with little GPU optimisation effort. They also found that the GPU usage was low, meaning there are resources and potential for more detailed simulations. Generally, simulations are great candidates for using GPUs, as they can benefit heavily from a high degree of parallelism and data throughput. Köster et al. (2020b) have developed a way of using adaptive time steps on the GPU to considerably improve the performance of numerical and discrete simulations. In addition to the performance gains they were able to retain the precision and constraint correctness of the simulation. Black hole simulations are crucial for science and education for a better understanding of our world. Verbraeck and Eisemann (2021) have shown that simulating complex Kerr (rotating) black holes can be done on consumer hardware in a few seconds. Schwarzschild black hole simulations can be performed in real-time with GPUs as described by Hissbach et al. (2022) which is especially helpful for educational scenarios. While both approaches do not have the same accuracy as detailed simulations on supercomputers, they show how a single GPU can yield similar accuracy at a fraction of the cost. Software network routing can also heavily benefit from GPU acceleration as shown by S. Han et al. (2010), where they achieved a significantly higher throughput than with a CPU only implementation. Finite element structural analysis is an essential tool for many branches of engineering and can also heavily benefit from the usage of GPUs as demonstrated by Georgescu et al. (2013). Generating test data for DeepQ learning can also significantly benefit from using the GPU (Köster et al., 2022). However, it also needs to be noted, that GPUs are not always better performing than CPUs as illustrated by Lee et al. (2010), so it is important to consider if it is worth using GPUs for specific tasks.

[2] https://julialang.org/

[3] https://www.amd.com/de/products/software/rocm.html

[4] https://www.khronos.org/opencl/

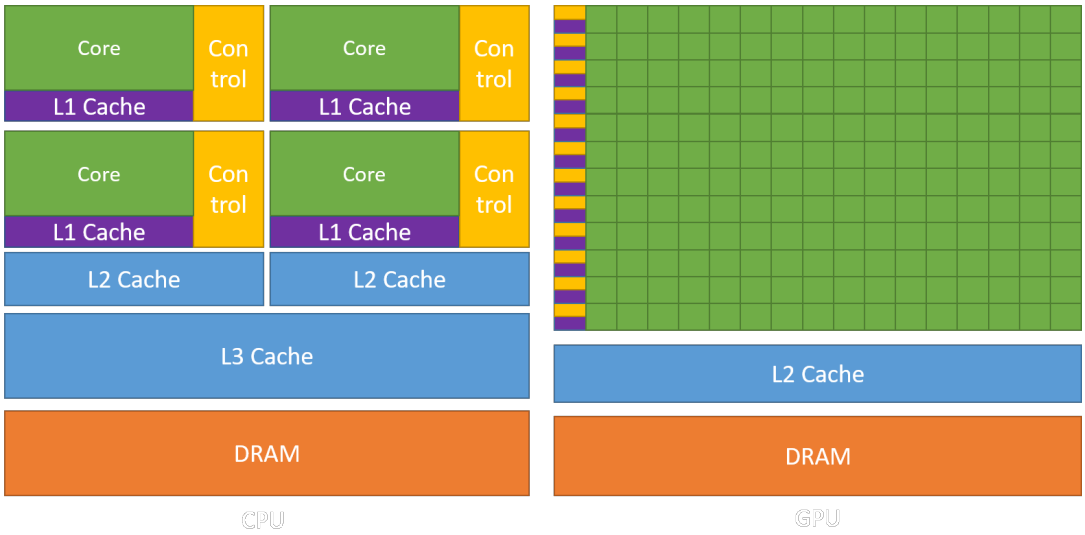


Figure 2.1: Overview of the architecture of a CPU (left) and a GPU (right). Note the higher number of simpler and smaller cores on the GPU (Nvidia, 2025b).

2.2.1 Programming GPUs

The development process on a GPU is vastly different from a CPU. A CPU has tens or hundreds of complex cores with the AMD Epyc 9965[5] having 192 cores and twice as many threads. To demonstrate the complexity of a simple one core 8-bit CPU Schuurman (2013) has written a development guide. He describes the different parts of one CPU core and how they interact. Modern CPUs are even more complex, with dedicated fast integer and floating-point arithmetic gates as well as logic gates, sophisticated branch prediction and much more. This makes a CPU perfect for handling complex control flows on a single program strand and on modern CPUs even multiple strands simultaneously (Palacios & Triska, 2011). However, as seen in Section 2.2, this often is not enough. On the other hand, a GPU contains thousands or even tens of thousands of cores. For example, the GeForce RTX 5090[6] contains a total of 21 760 CUDA cores. To achieve this enormous core count a single GPU core has to be much simpler than one CPU core. As described by Nvidia (2025b) a GPU designates much more transistors towards floating-point computations. This results in less efficient integer arithmetic and control flow handling. There is also less Cache available per core and clock speeds are usually also much lower than those on a CPU. An overview of the differences of a CPU and a GPU architecture can be seen in Figure 2.1.

Despite these drawbacks, the sheer number of cores, makes a GPU a valid choice when considering improving the performance of an algorithm. Because of the high number of cores, GPUs are best suited for data parallel scenarios. This is due to the SIMD architecture of these cards. SIMD stands for Sinlge-Instruction Multiple-Data and states that there is a single stream of instructions that is executed on a huge number of data

[5] https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9965.html
[6] https://www.nvidia.com/en-us/geforce/graphics-cards/50-series/rtx-5090/

streams. Franchetti et al. (2005) and Tian et al. (2012) describe ways of using SIMD instructions on the CPU. Their approaches lead to noticeable speed-ups of 3.3 and 4.7 respectively by using SIMD instructions instead of serial computations. Extending this to GPUs which are specifically built for SIMD/data parallel calculations shows why they are so powerful despite having less complex and slower cores than a CPU. It is also important to note, that a GPU also always needs a CPU, as the CPU is responsible for sending the data to the GPU and starting the GPU program. In GPGPU programming, the CPU is usually called the host, while the GPU is usually called the device.

Thread Hierarchy and Tuning

The thousands of cores on a GPU, also called threads, are grouped together in several categories. This is the Thread hierarchy of GPUs. The developer can influence this grouping to a degree which allows them to tune their algorithm for optimal performance. In order to develop a well performing algorithm, it is necessary to know how this grouping works. Tuning the grouping is unique to each algorithm and also dependent on the GPU used, which means it is important to test a lot of different configurations to achieve the best possible result. This section aims at exploring the thread hierarchy and how it can be tuned to fit an algorithm.

At the lowest level of a GPU exists a Streaming Multiprocessor (SM), which is a hardware unit responsible for scheduling and executing threads and also contains the registers used by these threads. An SM is always executing a group of 32 threads simultaneously, and this group is called a warp. The number of threads that can be started is virtually unlimited. However, threads must be grouped in a block, with one block typically containing a maximum of 1024 threads but is often configured to be less. Therefore, if more than 1024 threads are required, more blocks must be created. Blocks can also be grouped into thread block clusters which is optional, but can be useful in certain scenarios. All thread blocks or thread block clusters are part of a grid, which manifests as a dispatch of the code run on the GPU, also called kernel (AMD, 2025b). All threads in one block have access to some shared memory, which can be used for L1 caching or communication between threads. It is important that the blocks can be scheduled independently, with no dependencies between them. This allows the scheduler to schedule blocks and threads as efficiently as possible. All threads within a warp are guaranteed to be part of the same block, and are therefore executed simultaneously and can access the same memory addresses. Figure 2.2 depicts how threads in a block are grouped into warps for execution and how they shared memory.

A piece of code that is executed on a GPU is written as a kernel which can be configured. The most important configuration is how threads are grouped into blocks. The GPU allows the kernel to allocate threads and blocks and block clusters in up to three dimensions. This is often useful because of the already mentioned shared memory, which will be explained in more detail in Section 2.2.1. Considering the case where an image needs to be blurred, it not only simplifies the development if threads are arranged in a 2D grid, it also helps with optimising memory access. As the threads in a block, need to access a lot of the same data, this data can be loaded in the shared memory of the block. This allows the data to be accessed much quicker compared to when threads are allocated in only one dimension. With one dimensional blocks it is possible that threads

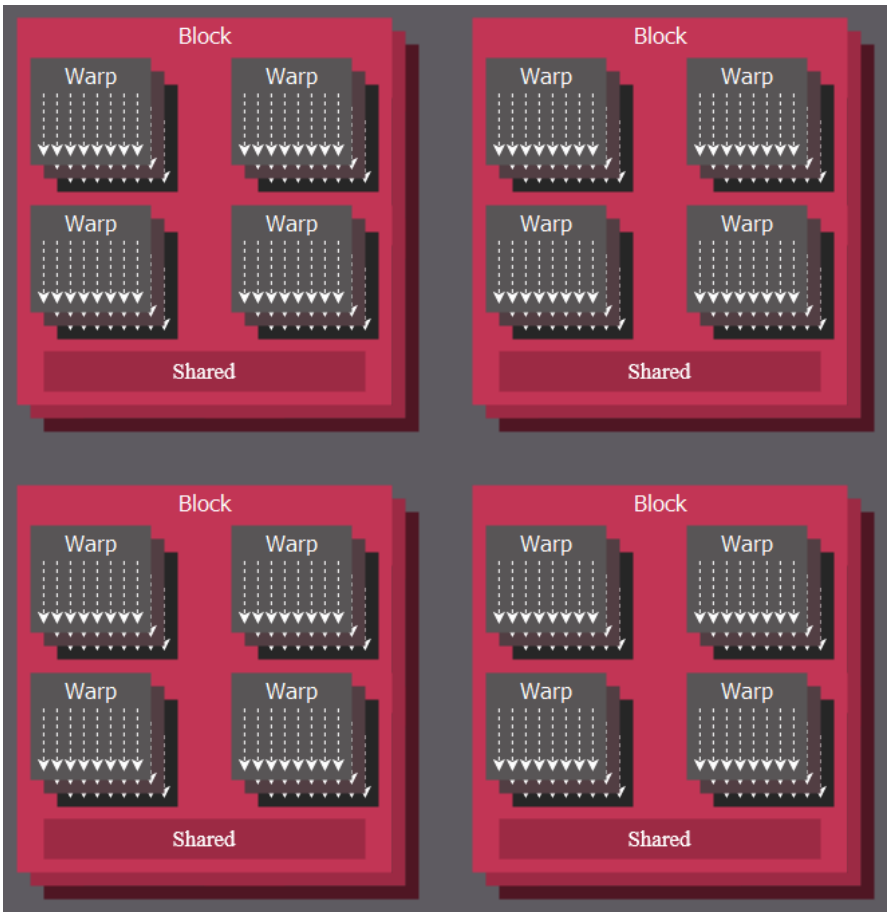


Figure 2.2: An overview of the thread hierarchy with blocks being split into multiple warps and their shared memory (AMD, 2025b).

assigned to nearby pixels, are part of a different block, leading to a lot of duplicate data transfer. The size in each dimension of a block can be almost arbitrary within the maximum allowed number of threads. However, blocks that are too large might lead to other problems which are described in more detail in Section 2.2.1.

All threads in a warp start at the same point in a program, but with their own instruction address, allowing them to work independently. Because of the SIMD architecture, all threads in a warp must execute the same instructions and if threads start diverging, the SM must pause threads with different instructions and execute them later. Figure 2.3 shows how such divergences can impact performance. The situation described by the figure also shows, that after the divergence the thread could re-converge. On older hardware this does not happen and leads to T2 being executed after T1 and T3 are finished. In situations where a lot of data dependent thread divergence happens, most of the benefits of using a GPU likely have vanished. Threads not executing the same instruction is strictly speaking against the SIMD principle but can happen in reality, due to data dependent branching. Consequently, this leads to bad resource utilisation, which in turn leads to worse performance. Another possibility of threads being paused

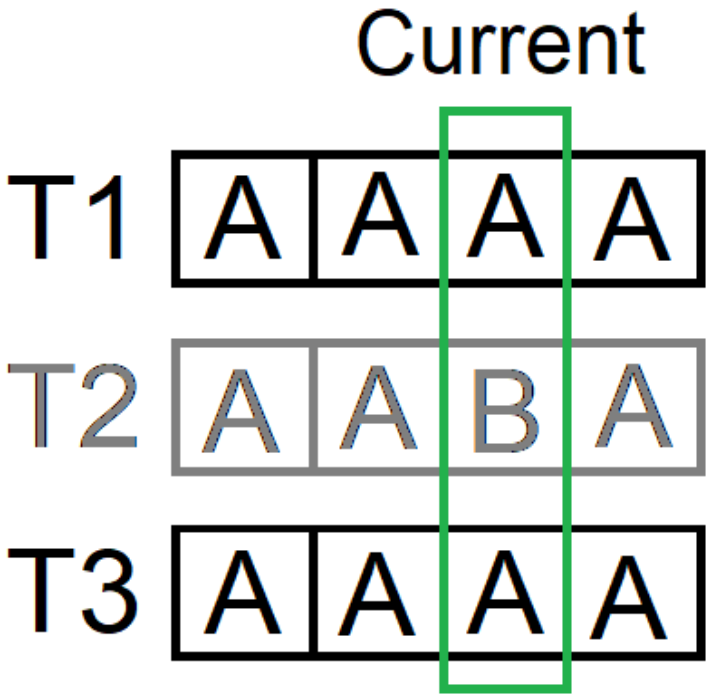


Figure 2.3: Thread T2 wants to execute instruction B while T1 and T3 want to execute instruction A. Therefore T2 will be an inactive thread this cycle and active once T1 and T3 are finished. This means that now the divergent threads are serialised.

(inactive threads) is the fact that sometimes, the number of threads started is not divisible by 32. In such cases, the last warp still contains 32 threads but only the threads with work are executed.

Modern GPUs implement the so called Single-Instruction Multiple-Thread (SIMT) architecture. In many cases a developer does not need to know the details of SIMT and can develop fast and correct programs with just the SIMD architecture in mind. However, leveraging the power of SIMT can yield substantial performance gains by re-converging threads after data dependent divergence occurred. A stack-less re-convergence algorithm was proposed by Collange (2011) as an alternative to the default stack-based re-convergence algorithm. Their algorithm was able to achieve higher performance than the default one. Another approach for increasing occupancy using the SIMT architecture is proposed by Fung and Aamodt (2011). They introduced a technique for compacting thread blocks by moving divergent threads to new warps until they re-converge. This approach resulted in a noticeable speed-up between 17% and 22%. Another example where a SIMT aware algorithm can perform better was proposed by Köster et al. (2020a). While they did not implement techniques for thread re-convergence, they implemented a thread compaction algorithm. On data-dependent divergence it is possible for threads to end early, leaving a warp with only partial active threads. This means the inactive threads are still occupied and cannot be used for other work. Their thread compaction

tackles this problem by moving active threads into a new thread block, releasing the inactive threads to perform other work. With this they were able to gain a speed-up of roughly 4 times compared to previous implementations. Adapting Multiple-Instruction Multiple-Data (MIMD) programs with synchronisation to run on SIMT architecture can be a difficult task, especially if the underlying architecture is not well understood. A static analysis tool and a transformer specifically designed to help avoid deadlocks with MIMD synchronisation is proposed by ElTantawy and Aamodt (2016). In addition, they proposed a hardware re-convergence mechanism that supports MIMD synchronisation. A survey by Khairy et al. (2019) explores different aspects of improving GPGPU performance architecturally. Specifically, they have compiled a list of different publications discussing algorithms for thread re-convergence, thread compaction and much more. Their main goal was to give a broad overview of many ways to improve the performance of GPGPU programming to help other developers.

Memory Model

On a GPU there are two parts that contribute to the performance of an algorithm. The one already looked at is the compute-portion of the GPU. This is necessary because if threads are serialised or run inefficiently, there is nothing that can make the algorithm execute faster. However, algorithms run on a GPU usually require huge amounts of data to be processed, as they are designed for exactly that purpose. The purpose of this section is to explain how the memory model of the GPU works and how it can influence the performance of an algorithm. In Figure 2.4 the memory layout and the kinds of memory available are depicted. The different parts will be explained in this section.

On a GPU there are multiple levels and kinds of memory available. All these levels and kinds have different purposes they are optimised for. This means that it is important to know what they are and how they can be best used for specific tasks. On the lowest level threads have registers and local memory available. Registers is the fastest way to access memory but is also the least abundant memory with up to a maximum of 255 32-Bit registers per thread on Nvidia GPUs and 256 on AMD GPUs (AMD, 2025a). However, using all registers of a thread can lead to other problems which are described in more detail in Section 2.2.1. On the other side, the thread local memory is significantly slower than registers. This is due to the fact, that local memory is actually stored in global memory and therefore has the same limitations which are explained later. This means it is important to try and avoid local memory as much as possible. Local memory is usually only used when a thread uses too many registers. The compiler will then spill the remaining data into local memory and loads it into registers once needed, drastically slowing down the application.

Shared memory is the next tier of memory on a GPU. Unlike local memory and registers, shared memory is shared between all threads inside a block. The amount of shared memory is depending on the GPU architecture but for Nvidia it hovers at around 100 Kilobyte (KB) per block. While this memory is slower than registers, its primary use-case is communicating and sharing data between threads in a block. If all threads in a block access a lot of overlapping data this data can be loaded from global memory into faster shared memory once. It can then be accessed multiple times, further increasing performance. Loading data into shared memory and accessing that data has to be done

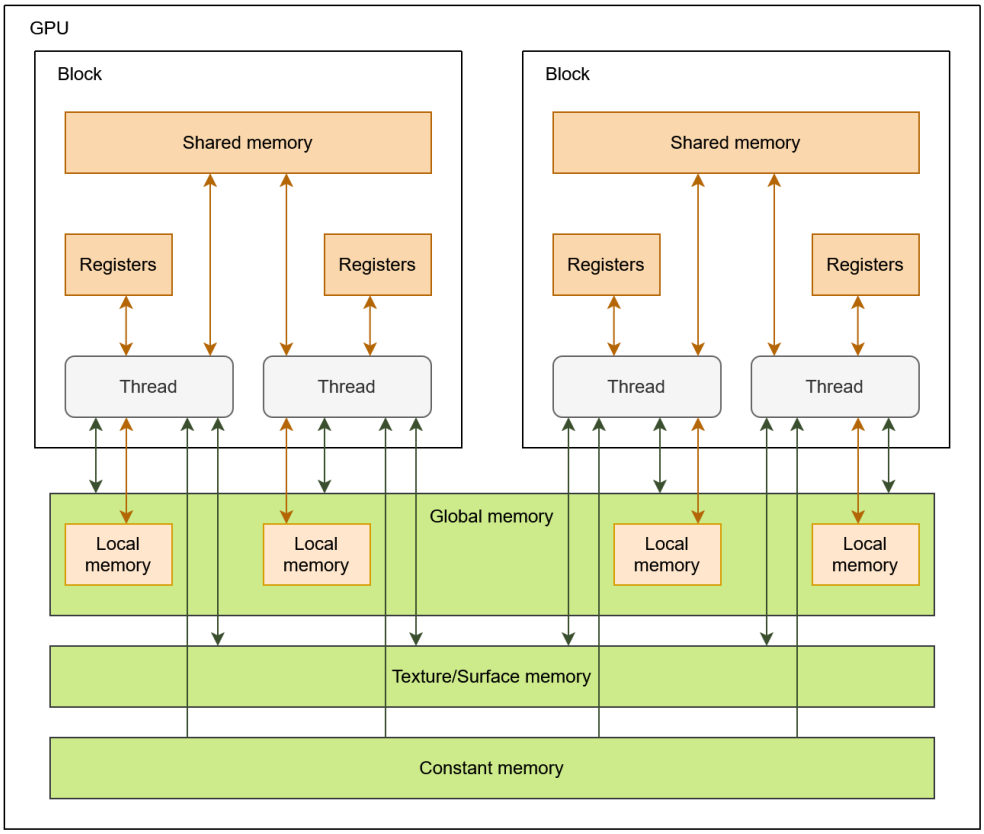


Figure 2.4: The layout of the memory in the GPU. The connections between the memory regions can be seen as well as the different kinds of memory available.

manually. Because shared memory is part of the unified data cache, it can either be used as a cache or for manual use, meaning a developer can allocate more shared memory towards caching if needed. Another feature of shared memory are the so-called memory banks. Shared memory is always split into 32 equally sized memory modules also called memory banks. All available memory addresses lie in one of these banks. This means if two threads access two memory addresses which lie in different banks, the access can be performed simultaneously, increasing the throughput.

The most abundant and slowest memory is the global memory and resides in device memory. A key constraint of device memory and therefore global memory is, that can only be accessed in either 32, 64 or 128 byte chunks. This means if a thread wants to access 8 bytes from global memory, alongside the 8 bytes, the 24 bytes after the requested 8 bytes are also transferred. As a result, the throughput is only a fourth of the theoretical maximum. Therefore, it is important to follow optimal access patterns. What these optimal patterns are, are architecture dependent and are described in the according sections in the CUDA programming guide.

A small portion of device memory is allocated to constant memory. Constant memory is accessible by all threads and as the name implies, can not be written to by threads. It can be initialised by the CPU when starting a kernel if needed. As constant memory

Compute Capability	8.9	10.x
Max. number of threads per block	1 024	
Warp size	32 threads	
Max. number of warps per SM	48	64
Max. number of blocks per SM	24	32
Max. number of threads per SM	1 536	2 048
Number of 32-bit registers per SM	64 000	
Max. number of 32-bit registers per block	64 000	
Max. number of 32-bit registers per thread	255	
Max. amount of shared memory per SM	100 Kilobytes	228 Kilobytes
Max. amount of shared memory per block	99 Kilobytes	227 Kilobytes

Table 2.1: A simplified version of the technical specifications for the Compute Capabilities 8.9 and 10.x (Nvidia, 2025b). These correspond to the Nvidia Ada Lovelace and Blackwell microarchitectures.

has a separate cache, it can be used to speed-up data access for constant and frequently accessed data.

Another special kind of memory is the texture and surface memory. According to AMD (2025b) texture memory is read-only memory, while surface memory can also be written to, which is the only difference between these two kinds of memory. Nvidia does not explicitly state this behaviour, but due to the fact that accessing textures is only performed via caches, it is implied that on Nvidia GPUs, texture memory is also read-only. As the name implies, this kind of memory is optimised for accessing textures. This means that threads of the same warp, accessing data which is spatially close together, will result in increased performance. As already mentioned, surface memory works the same way, with the difference, that it can be written to. It is therefore well suited for manipulating two- or three-dimensional data.

Occupancy

Occupancy describes the utilisation of a GPU. A high occupancy means, that there are Warps executing, or in other words, the cores are occupied with work. This is important, as a low occupancy means that the GPU is waiting for work to be scheduled and is therefore idle. As a result, it is desired to achieve high occupancy in order to increase the performance of an algorithm. It needs to be noted, that occupancy is not the only option for improving performance. As it is possible for the GPU to have a high occupancy while performing a lot of unnecessary or redundant work or utilising compute-resources that are slower. An example for the latter would be developing an algorithm that uses 64-bit floating point (FP64) numbers while 32-bit floating point (FP32) numbers would have sufficient accuracy. Because GPUs tend to have fewer FP64 compute-resources than they have FP32 compute-resources, performing FP64 operations will take longer. However, despite these drawbacks, having low occupancy will very likely result in performance degradation while high occupancy will either improve performance or do no harm otherwise. Ways of achieving high occupancy will be outlined in this section as most other performance problems can be solved algorithmically.

When starting a kernel, the most important configuration is the number of threads and thread blocks that need to be started. This is important, as this has other effects on occupancy as well. In table 2.1 the most notable limitations are presented that can affect occupancy. These limitations need to be considered when choosing a kernel configuration. It is important to note, that depending on the GPU and problem, the occupancy tuning might differ, and the same approach might perform well on one GPU but perform poorly on another GPU. Therefore, the things discussed here are only guidelines. Tools like Nvidia Nsight Compute[7] and Nsight Systems[8] are essential for performance tuning. Nsight compute also contains an occupancy calculator which takes a kernel and computes how the configuration performs in terms of occupancy and also lets the developer try out different configurations (Nvidia, 2025c).

In general, it is important to have as many warps as possible ready for execution. While this means that a lot of warps could be executed but are not, this is actually desired. A key feature of GPUs is so-called latency hiding, meaning that while a warp waits for data to be retrieved for example, another warp ready for execution can now be run. With low occupancy, and therefore little to no warps waiting for execution, latency hiding does not work, as now the hardware is idle. As a result, the runtime increases which also explains why high occupancy is not guaranteed to result in performance improvements while low occupancy can and often will increase the runtime.

As seen in table 2.1, there exist different limitations that can impact occupancy. The number of warps per SM is important, as this means this is the degree of parallelism achievable per SM. If due to other limitations, the number of warps per SM is below the maximum, there is idle hardware. One such limitation is the number of registers per block and SM. In the case of compute capability 8.9, one SM can handle $32 * 48 = 1\,536$ threads. This leaves $64\,000/1\,536 \approx 41$ registers per thread, which is lower than the theoretical maximum of 255 registers per thread. Typically, one register is mapped to one variable in the kernel code, meaning a developer can use up to 41 variables in their code. However, if the variable needs 64 bits to store its value, the register usage doubles, as all registers on a GPU are 32-bit. On a GPU with compute capability 10.x a developer can use up to $64\,000/2\,048 \approx 31$ registers. Of course a developer can use more registers, but this results in less occupancy. However, depending on the algorithm using more registers might be more beneficial to performance than the lower occupancy, in which case occupancy is not as important. If a developer needs more than 255 registers for their variables the additional variables will spill into local memory which is, as described in Section 2.2.1, not desirable.

Additionally, shared memory consumption can also impact the occupancy. If for example a block needs all the available shared memory, which is almost the same as the amount of shared memory per SM, this SM can only serve this block. On compute capability 10.x, this would mean that occupancy would be at maximum 50% as a block can have up to 1 024 threads while an SM supports up to 2 048 threads. Again, in such cases it needs to be determined, if the performance gain of using this much shared memory is worth the lower occupancy.

Balancing these limitations and therefore the occupancy and performance often requires a lot of trial and error with help of the aforementioned tools. In cases where

[7] https://developer.nvidia.com/nsight-compute
[8] https://developer.nvidia.com/nsight-systems

occupancy is already high and the amount of warps ready for execution is also high, other areas for performance improvements need to be explored. Algorithmic optimisation is always a good idea. Some performance improvements can be achieved by altering the computations to use different parts of the GPU. One of such optimisations is using FP32 operations wherever possible. Another well suited optimisation is to rewrite the algorithm to use as many Fused Multiply-Add (FMA) instructions. FMA is a special floating point instruction, that multiplies two values and adds a third, all in a single clock cycle (Nvidia, 2025a). However, the result might slightly deviate compared to performing these two operations separately, which means in accuracy sensitive scenarios, this instruction should be avoided. If the compiler detects a floating point operation with the FMA structure, it will automatically be compiled to an FMA instruction. To prevent this, in C++ the developer can call the functions ___fadd_ and ___fmul_ for addition and multiplication respectively.

2.2.2 Parallel Thread Execution

While in most cases a GPU can be programmed in a higher level language like C++ or even Julia[9], it is also possible to program GPUs with the low level language Parallel Thread Execution (PTX) developed by Nvidia. A brief overview of what PTX is and how it can be used to program GPUs is given in this section. Information in this section is taken from the PTX documentation (Nvidia, 2025d) if not stated otherwise.

PTX defines a virtual machine with an own instruction set architecture (ISA) and is designed for data-parallel processing on a GPU. It is an abstraction of the underlying hardware instruction set, allowing PTX code to be portable across Nvidia GPUs. In order for PTX code to be usable for the GPU, the driver is responsible for compiling the code to the hardware instruction set of the GPU it is run on. A developer typically writes a kernel in CUDA using C++, for example, and the Nvidia compiler generates the PTX code for that kernel. This PTX code is then compiled by the driver once it is executed. The concepts for programming the GPU with PTX and CUDA are the same, apart from the terminology which is slightly different. For consistency, the CUDA terminology will continue to be used.

Syntactically PTX resembles Assembly style code. Every PTX code must have a `.version` directive which indicates the PTX version and an optional `.target` directive which indicates the compute capability. If the program works in 64 bit addresses, the optional `.address_size` directive can be used to indicate that, which simplifies the code for such applications. After these directives, the actual code is written. As each PTX code needs an entry point (the kernel) the `.entry` directive indicates the name of the kernel and the parameters needed. It is also possible to write helper functions with the `.func` directive. Inside the kernel or a helper function, normal PTX code can be written. Because PTX is very low level, it assumes an underlying register machine, therefore a developer needs to think about register management. This includes loading data from global or shared memory into registers if needed. Code for manipulating data like addition and subtraction generally follow the structure `operation.datatype` followed by up to four parameters for that operation. For adding two FP32 values together and storing them in the register %n, the code looks like the following:

[9] https://juliagpu.org/

```
1 .func loop(.param .u32 N)
2 {
3   .reg .u32 \%n;
4   .reg .pred \%p;
5
6   ld.param.u32 \%n, [N];
7 Loop:
8   setp.eq.u32 \%p, \%n, 0;
9 @\%p bra    Done;
10  sub.u32   \%n, \%n, 1;
11  bra     Loop;
12 Done:
13 }
```

Program 2.1: A PTX program fragment depicting how loops can be implemented.

```
  add.f32   \%n, 0.1, 0.2;
```

Loops in the classical sense do not exist in PTX. Alternatively a developer needs to define jump targets for the beginning and end of the loop. The Program in 2.1 shows how a function with simple loop can be implemented. The loop counts down to zero from the passed parameter N which is loaded into the register %n in line 6. If the value in the register %n reached zero the loop branches at line 9 to the jump target at line 12 and the loop has finished. All other used directives and further information on writing PTX code can be taken from the PTX documentation (Nvidia, 2025d).

2.3 Compilers

Compilers are a necessary tool for many developers. If a developer wants to run their program it is very likely they need one. As best described by Aho et al. (2006) in their dragon book, a compiler takes code written by a human in some source language and translates it into a destination language readable by a computer. This section briefly explores what compilers are and research done in this old field of computer science. Furthermore, the topics of transpilers and interpreters are explored, as their use-cases are very similar.

Aho et al. (2006) and Cooper and Torczon (2022) describe how a compiler can be developed, with the latter focusing on more modern approaches. They describe how a compiler consists of two parts, the analyser, also called frontend, and the synthesiser also called backend. The front end is responsible for ensuring syntactic and semantic correctness and converts the source code into an intermediate representation, an abstract syntax tree (AST), for the backend. Generating code in the target language, from the intermediate representation is the job of the backend. This target code can be assembly or anything else that is needed for a specific use-case. This intermediate representation also makes it simple to swap out frontends or backends. The Gnu Compiler Collection GCC (2025) takes advantage of using different frontends to provide support for many languages including C, C++, Ada and more. Instead of compiling source code for specific machines directly, many languages compile code for virtual machines instead. Notable

examples are the Java Virtual Machine (JVM) (Lindholm et al., 2025) and the low level virtual machine (LLVM) (Lattner & Adve, 2004). Such virtual machines provide a bytecode which can be used as a target language for compilers. A huge benefit of such virtual machines is the ability for one program to be run on all physical machines the virtual machine exists for, without the developer needing to change that program (Lindholm et al., 2025). Programs written for virtual machines are compiled into their respective bytecode. This bytecode can then be interpreted or compiled to physical machine code and then be run. According to the JVM specification Lindholm et al. (2025) the Java bytecode is interpreted and also compiled with a just-in-time (JIT) compiler to increase the performance of code blocks that are often executed. On the other hand, the common language runtime (CLR)[10], the virtual machine for languages like C#, never interprets the generated bytecode. As described by Microsoft (2023) the CLR always compiles the bytecode to physical machine code using a JIT compiler before it is executed.

A grammar describes how a language is structured. It not only describes the structure of natural language, but it can also be used to describe the structure of a programming language. Chomsky (1959) found that grammars can be grouped into four levels, with regular and context-free grammars being the most relevant for programming languages. A regular grammar is of the structure $A = a \mid a\,B$ which is called a rule. The symbols A and B are non-terminal symbols and a is a terminal symbol. A non-terminal symbol stands for another rule with the same structure and must only occur after a terminal symbol. Terminal symbols are fixed symbols or a value that can be found in the input stream, like literals in programming languages. Context-free grammars are more complex and are of the structure $A = \beta$. In this context β stands for any combination of terminal and non-terminal symbols. Therefore, a rule like $A = a \mid a\,B\,a$ is allowed with this grammar level. This shows that with context-free grammars enclosing structures are possible. To write grammars for programming languages, other properties are also important to efficiently validate or parse some input to be defined by this grammar. However, these are not discussed here, but are described by Aho et al. (2006). They also described that generating a parser out of a grammar can be automated. This automation can be performed by parser generators like Yacc (Johnson, 1975) as described in their book. More modern alternatives are Bison[11] or Antlr[12]. Before the parser can validate the input stream, a scanner is needed as described by Cooper and Torczon (2022). The scanner reads every character of the input stream and is responsible for removing white-spaces and ensures only valid characters and words are present. Flex [13] is a tool that allows generating a scanner and is often used in combination with Bison. A simplified version of the compiler architecture using Flex and Bison is depicted in Figure 2.5. It shows how source code is taken and transformed into the intermediate representation by the frontend, and how it is converted into executable machine code by the backend.

[10] https://learn.microsoft.com/en-us/dotnet/standard/clr
[11] https://www.gnu.org/software/bison/
[12] https://www.antlr.org/
[13] https://github.com/westes/flex

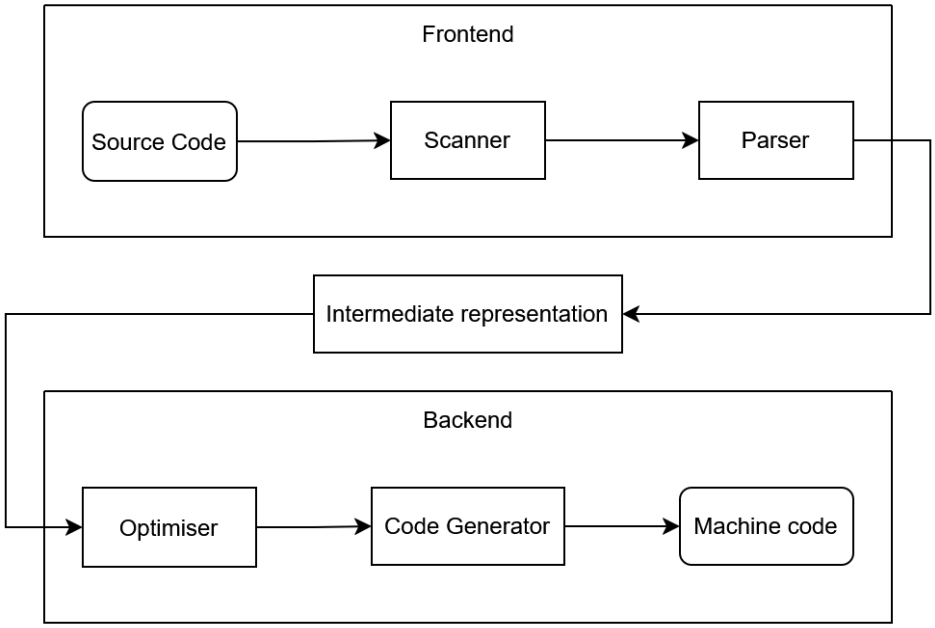


Figure 2.5: A simplified overview of how the architecture of a compiler looks, using Flex and Bison.

2.3.1 Transpilers

With the concepts already mentioned, it is possible to generate executable code from code written in a programming language. However, sometimes it is desired to convert a program from one programming language to another and therefore the major difference between these use-cases is the backend. A popular transpiler example is TypeScript, which transforms TypeScript source code into JavaScript source code (Microsoft, 2025). Other examples for transpilers are the C2Rust transpiler (Ling et al., 2022) that transpiles C code into Rust code as well as the PyJL transpiler (Marcelino & Leitão, 2022) which transpiles Python code into Julia code. Chaber and Ławryńczuk (2016) proposed a transpiler that takes MATLAB and C code and transforms it into pure and optimised C code for an STM32 microcontroller. An early example for a transpiler has been developed by Intel (1978) where they built a transpiler for transforming assembly code for their 8080 CPU to assembly code for their 8086 CPU. Transpilers can also be used in parallelisation environments, like OpenMP (C.-K. Wang & Chen, 2015). There also exists a transpiler that transforms CUDA code into highly parallel CPU code. Moses et al. (2023) described this transpiler, and they found that the generated code performs noticeably better than doing this transformation by hand. When designing complex processors and accelerators, Register-transfer level (RTL) simulations are essential (L.-T. Wang et al., 2009). In a later study Zhang et al. (2020) have shown how RTL simulations can be performed on GPUs with a speed-up of 20. This led to D.-L. Lin et al. (2023) developing a transpiler to transform RTL into CUDA kernels instead of handwriting them. The compared their results with a CPU implementation running on 80 CPUs, where they found that the transpiled CUDA version was 40 times faster. Using tran-

spilers for software backend and business logic has been proposed by Bastidas Fuertes et al. (2023a). Their approach implemented a programming language that can be transpiled into different programming languages, for usage in a multi-programming-language environment that share some business logic. In another study, Bastidas Fuertes et al. (2023b) reviewed over 600 publications to map the use of transpilers alongside their implementations in different fields of research, demonstrating the versatility of transpiler use.

2.3.2 Interpreters

Interpreters are a different kind of program for executing source code. Rather than compiling the code and executing the result, an interpreter executes the source code directly. Languages like Python and JavaScript are prominent examples of interpreted languages, but also Java, or more precise Java-Bytecode, is also interpreted before it gets compiled (Lindholm et al., 2025). However, interpreters can not only be used for interpreting programming languages. It is also possible for them to be used in GP. Langdon and Banzhaf (2008) have shown how a SIMD interpreter can be efficiently used for evaluating entire GP populations on the GPU directly. In a later work Cano and Ventura (2014) further improved this interpreter. They used the fact that a GP individual represents a tree which can be split into independent subtrees. These can be evaluated concurrently and with the help of communication via shared memory, they were able to evaluate the entire tree. With this they achieved a significant performance improvement over previous implementations. As shown by Dietz and Young (2010), it is even possible to develop an interpreter that can execute MIMD programs on a SIMD GPU. However, as noted by the authors, any kind interpretation comes with an overhead. This means that with the additional challenges of executing MIMD programs on SIMD hardware, their interpreter, while achieving reasonable efficiency, still suffers from performance problems. Another field where interpreters can be useful are rule-based simulations. Köster et al. (2020a) has shown how they implemented a GPU interpreter for such simulations. In addition with other novel performance improvements in running programs on a GPU, they were able to gain a speed-up of 4 over non-interpreted implementations. While publications like Fua and Lis (2020) and Gherardi et al. (2012) have shown, interpreted languages often trail behind in terms of performance compared to compiled languages, interpreters per se are not slow. And while they come with performance overhead as demonstrated by Dietz and Young (2010) and Romer et al. (1996), they can still be a very fast, easy and powerful alternative for certain tasks.

Chapter 3

Concept and Design

To be able to determine whether evaluating mathematical expressions on the GPU is better suited than on the CPU, a prototype needs to be implemented. More specifically, a prototype for interpreting these expressions on the GPU, as well as a prototype that transpiles expressions into code that can be executed by the GPU. The goal of this chapter, is to describe how these two prototypes can be implemented conceptually. First the requirements for the prototypes as well as the data they operate on are explained. This is followed by the design of the interpreter and the transpiler. The CPU interpreter will not be described, as it already exists.

3.1 Requirements and Data

The main goal of both prototypes or evaluators is to provide a speed-up compared to the CPU interpreter already in use. However, it is also important to determine which evaluator provides the most speed-up. This also means that if one of the evaluators is faster, it is intended to replace the CPU interpreter. Therefore, they must have similar capabilities, and therefore meet the following requirements:

- Multiple expressions as input.
- All input expressions have the same number of variables (x_n), but can have a different number of parameters (p_n).
- The variables are parametrised using a matrix of the form $k \times N$, where k is the number of variables in the expressions and N is the number of different parametrisations for the variables. This matrix is the same for all expressions.
- The parameters are parametrised using a vector of vectors. Each vector v_i corresponds to an expression e_i.
- The following operations must be supported: $x+y$, $x-y$, $x*y$, x/y, x^y, $|x|$, $\log(x)$, e^x and $\sqrt{x}$. Note that x and y can either stand for a value, a variable, or another operation.
- The results of the evaluations are returned in a matrix of the form $k \times N$. In this case, k is equal to the N of the variable matrix and N is equal to the number of input expressions.

These requirements mean, that one possible expression that must be able to be

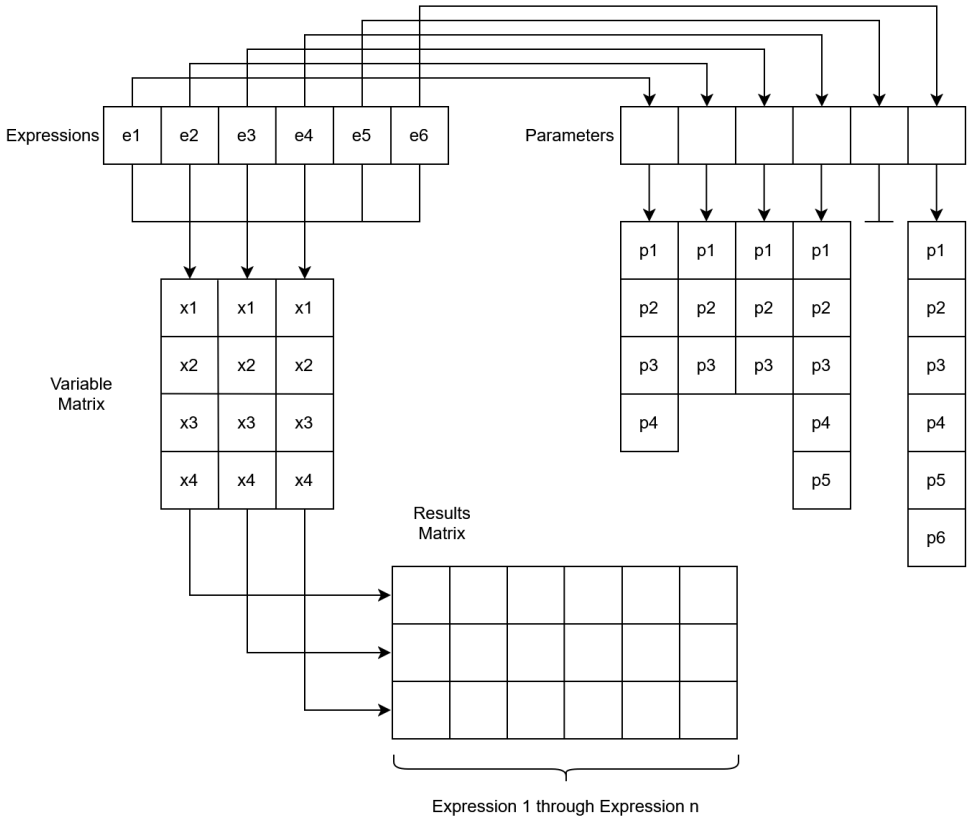


Figure 3.1: This diagram shows how the input and output looks like and how they interact with each other.

evaluated is the following: $\log(e^{p_1}) - |x_1| * \sqrt{x_2}/10 + 2^{x_3}$

With this, the capabilities are outlined, however, the input and output data need to further be explained for a better understanding. The first input are the expressions that need to be evaluated. These can have any length and can contain constant values, variables and parameters and all of these are linked together with the supported operations. In the example shown in Figure 3.1, there are six expressions e_1 through e_6. Next is the variable matrix. One entry in this matrix, corresponds to one variable in every expression, with the row indicating which variable it holds the value for. For example the values in row three, are used to parametrise the variable x_3. Each column holds a different set of variables. In the provided example, there are three variable sets, each holding the values for four variables x_1 through x_4. All expressions are evaluated using all variable sets and the results of these evaluations are stored in the results matrix. Each entry in this matrix holds the resulting value of the evaluation of one expression with one variable set. The row indicates the variable set while the column indicates the expression.

This is the minimal functionality needed to evaluate expressions with variables generated by a symbolic regression algorithm. In the case of parameter optimisation, it is

useful to have a different type of variable, called parameter. For parameter optimisation it is important that for the given variable sets, the best fitting parameters need to be found. To achieve this, the evaluator is called multiple times with different parameters, but the same variables. The results are then evaluated for their fitness by the caller. In this case, the parameters do not change within one call. Parameters could therefore be treated as constant values of the expressions, as the caller and no separate input for them would be needed. However, providing the possibility to have the parameters as an input, makes the process of parameter optimisation easier. This is the reason the prototype evaluators need to support parameters as inputs. Unlike variables, not all expressions need to have the same number of parameters. Therefore, they are structured as a vector of vectors and not a matrix. The example in Figure 3.1 shows how the parameters are structured. For example one expression has zero parameters, while another has six parameters p_1 through p_6. It needs to be mentioned that just like the number of variables, the number of parameters per expression is not limited. It is also possible to completely omit the parameters if they are not needed.

3.2 Architecture

Based on the requirements above, the architecture of both prototypes can be designed. While the requirements only specify the input and output, the components and workflow also need to be specified. This section aims at giving an architectural overview of both prototypes, alongside their design decisions.

A design decision that has been made for both prototypes is to split the evaluation of each expression into a separate kernel or kernel dispatch. As explained in Section 2.2.1, it is desirable to reduce the occurrence of thread divergence as much as possible. Although the SIMT programming model tries to mitigate the negative effects of thread divergence, it is still a good idea to avoid it when possible. For this use-case, thread divergence can easily be avoided by not evaluating all expressions in a single kernel or kernel dispatch. GPUs are able to have multiple resident grids, with modern GPUs being able to accommodate 128 grids concurrently (Nvidia, 2025b). One grid corresponds to one kernel dispatch, and therefore allows up-to 128 kernels to be run concurrently. Therefore, dispatching a kernel for each expression, has the possibility to improve the performance. In the case of the interpreter, having only one kernel that can be dispatched for each expression, also simplifies the kernel itself. This is because the kernel can focus on evaluating one expression and does not require additional code to handle multiple expressions at once. Similarly, the transpiler can also be simplified, as it can generate many smaller kernels than one big kernel. Additionally, the smaller kernels do not need any branching, because the generated code only needs to perform the operations as they occur in the expression itself.

3.2.1 Pre-Processing

The first step in both prototypes is the pre-processing step. It is needed, as it simplifies working with the expressions in the later steps. One of the responsibilities of the pre-processor is to verify that only allowed operators and symbols are present in the given

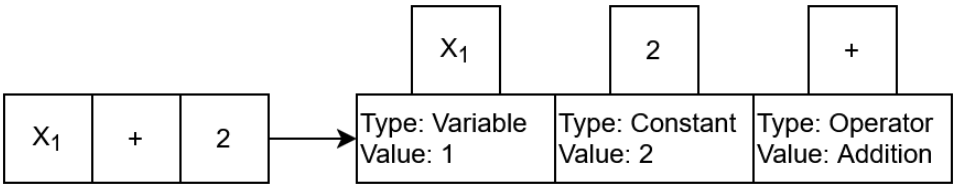


Figure 3.2: This diagram shows how an expression will be transformed in the pre-processing step.

expressions. This is comparable to the work a scanner like Flex[1] performs. Additionally, this step also converts the expression into an intermediate representation. In essence, the pre-processing step can be compared to the front-end of a compiler as described in Section 2.3. The conversion into the intermediate representation transforms the expressions from infix-notation into postfix-notation. This further allows the later parts to more easily evaluate the expressions. One of the major benefits of this notation is the implicit operator precedence. It allows the evaluators to evaluate the expressions token by token from left to right, without needing to worry about the correct order of operations. One token represents either an operator, a constant value, a variable or a parameter. Apart from the intermediate representation containing the expression in postfix-notation, it also contains the information about the types of the tokens themselves. This is all that is needed for the interpretation and transpilation steps. A simple expression like $x + 2$ would look like depicted in figure 3.2 after the pre-processing step.

It would have also been possible to perform the pre-processing step on the GPU. However, pre-processing only one expression can not easily be split into multiple threads, which means one GPU thread would need to process one expression. As described in Section 2.2 a single GPU thread is slower than a single CPU thread and as a result means the processing will also be slower. Furthermore, it wouldn't make sense to process all expressions in a single kernel. This would lead to a lot of thread divergence, which essentially means processing one expression after the other. The SIMT programming model might help with parallelising at least some parts of the processing work. However, the generated expressions can differ a lot from each other and restricting them to be similar and therefore SIMT friendly, would likely reduce the overall quality of the symbolic regression algorithm. Therefore, it does not make sense to perform the processing step on the GPU. This is a typical example of code that is better run on the CPU, also because the parallelisation possibilities of one thread per expression can be applied to the CPU as well. Concepts like caching processed expressions, or caching parts of the processed expressions can also be employed on the CPU. This would not be possible on the GPU, because a GPU can not save state between two kernel dispatches.

3.2.2 Interpreter

The interpreter consists of two parts. The CPU side is the part of the program, that interacts with both the GPU and the caller. An overview on the components and the

[1] https://github.com/westes/flex

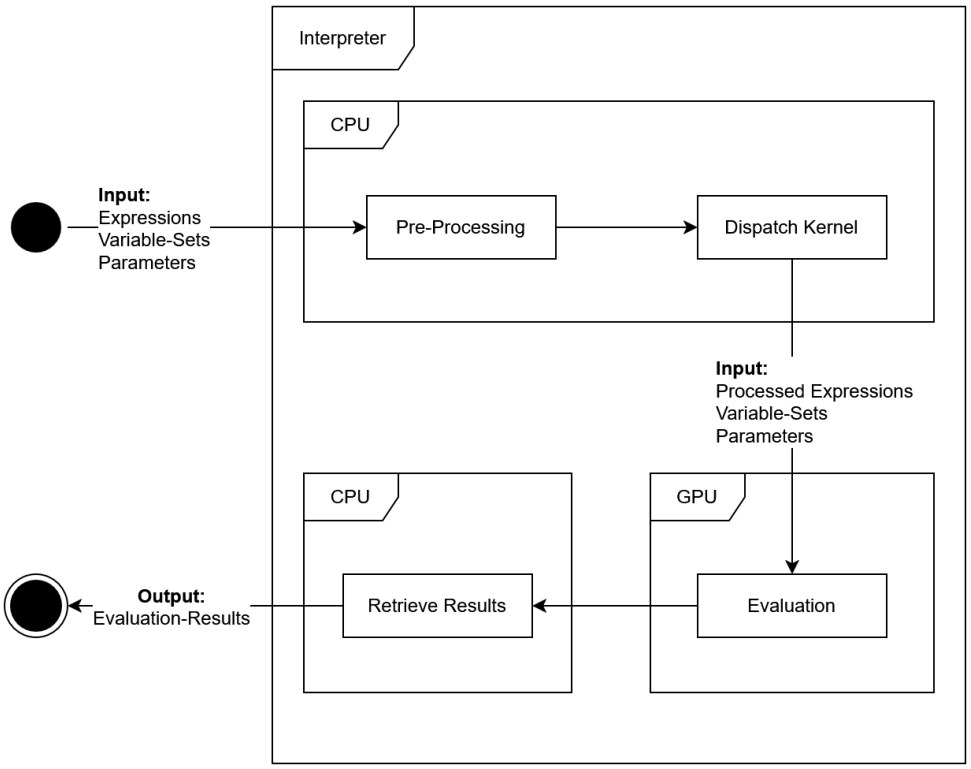


Figure 3.3: This diagram depicts the coarse-grained workflow of the interpreter. It shows how the parts interact with each other and with the system it will operate in.

workflow of the interpreter can be seen in Figure 3.3. Once the interpreter receives the expressions, they are pre-processed. This ensures the expressions are valid, and that they are transformed into the intermediate representation needed for evaluating them. The results of this pre-processing are then sent to the GPU, which performs the actual interpretation of the expressions. Alongside the expressions, the data for the variables and parameters also needs to be sent to the GPU. Once all the data resides on the GPU, the interpreter kernel can be dispatched. It needs to be noted, that for each of the expressions, a separate kernel will be dispatched. As already described, this decision has been made, to ensure, reduce thread divergence and therefore increase performance. In fact, dispatching the same kernel multiple times with different expressions, means, there will not occur any thread divergence as explained later. Once the GPU has finished evaluating all expressions with all variable sets, the result will be stored in a matrix on the GPU. The CPU then retrieves the results and returns them to the caller in the format specified by the requirements.

Evaluating the expressions is relatively straight forward. Due to the expressions being in post-fix notation, the actual interpreter must only iterate over all tokens once and perform the appropriate tasks. If the interpreter encounters a binary operator, it must simply read the previous two values and perform the operation specified by the operator. For unary operators, only the previous value must be read. As already

mentioned, expressions in postfix-notation implicitly contain the operator precedence, therefore no look-ahead or other strategies need to be used to ensure correct evaluation. The Algorithm 3.1 shows how the interpreter works. Note that this is a simplified version, that only works with additions, multiplications and constant values.

Algorithm 3.1: Interpreting an equation in postfix-notation

```
1:  procedure Evaluate(expr: PostfixExpression)
2:      stack ← []
3:      while HasTokenLeft(expr) do
4:          token ← GetNextToken(expr)
5:          if token.Type = Constant then
6:              Push(stack, token.Value)
7:          else if token.Type = Operator then
8:              if token.Value = Addition then
9:                  right ← Pop(stack)
10:                 left ← Pop(stack)
11:                 Push(stack, left + right)
12:             else if token.Value = Multiplication then
13:                 right ← Pop(stack)
14:                 left ← Pop(stack)
15:                 Push(stack, left ∗ right)
        return Pop(stack)
```

If a new operator is needed, it must simply be added as another else-if block inside the operator branch. New token types like variables or parameters, can also be added by adding a new outer else-if block that checks for these token types. However, the preprocessing step also needs to be extended with these new operators and token types. Otherwise, the expression will never reach the evaluation step as they would be seen as invalid. It is also possible to add unary operators like log(). In this case only one value would be read from the stack, the operation would be applied, and the result would be written back to the stack.

The Algorithm 3.1 in this case resembles the kernel. This kernel will be dispatched for every expression that needs to be evaluated, to eliminate thread divergence. Thread divergence can only happen on data dependent branches. In this case, the while loop and every if and else-if statement contains a data dependent branch. Depending on the expression passed to the kernel, the while loop may run longer than for another expression. Similarly, not all expressions have the same constants, operators and variables in the same order and would therefore lead to each thread, taking different paths. However, one expression, always has the same constants, operators and variables in the same locations, meaning all threads will take the same paths. This also means that despite the interpreter containing many data dependent branches, these branches only depend on the expression itself. Because of this, all threads will take the same paths and therefore will never diverge from one another if they execute the same expression.

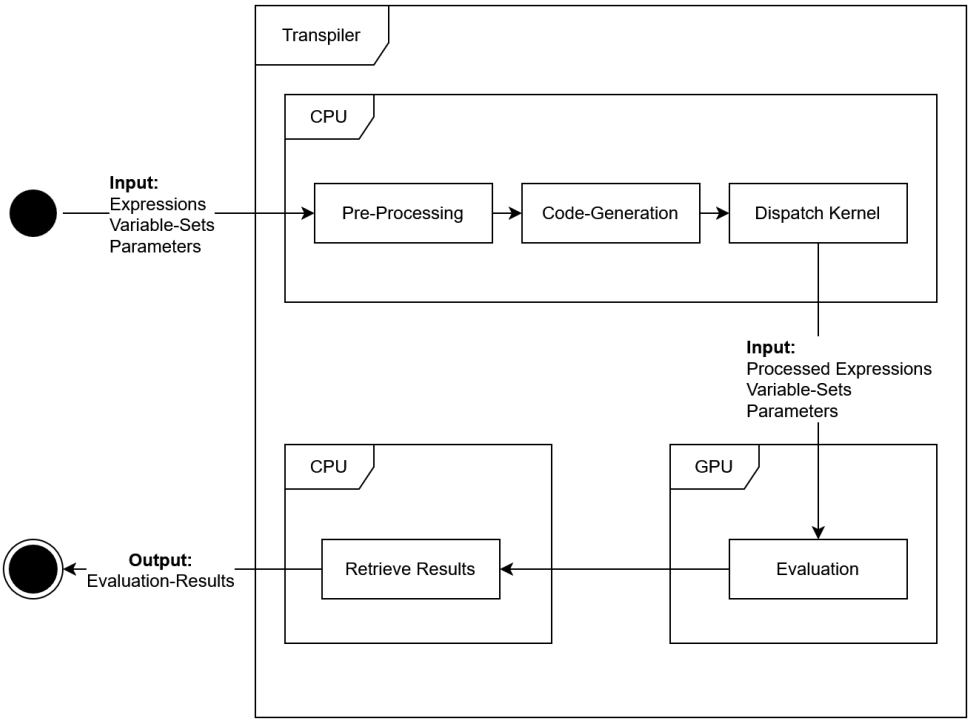


Figure 3.4: This diagram depicts the coarse-grained workflow of the transpiler. It shows how the parts interact with each other and with the system it will operate in.

3.2.3 Transpiler

Similar to the interpreter, the transpiler also consists of a part that runs on the CPU and a part that runs on the GPU. When looking at the component and workflow of the transpiler, as shown in Figure 3.4, it is almost identical to the interpreter. However, the key difference between the two, is the additional code generation, or transpilation step. Apart from that, the transpiler also needs the same pre-processing step and also the GPU to evaluate the expressions. However, the GPU evaluator generated by the transpiler works differently to the GPU evaluator for the interpreter. The difference between these evaluators will be explained later.

Before the expressions can be transpiled into PTX code, they need to be pre-processed. As already described, this step ensures the validity of the expressions and transforms them into the intermediate representation described above. As with the interpreter, this also simplifies the code generation step at the cost of some performance because the intermediate representation has to be generated. However, in this case the benefit of having a simple code generation step was more important than performance. By transforming the expressions into postfix-notation, the code generation follows a similar pattern to the interpretation described above. Algorithm 3.2 shows how the transpiler takes an expression, transpiles it and then returns the finished code. It can be seen that the while loop is the same as the while loop of the interpreter. The main difference is in the operator branches. Because now code needs to be generated, the branches

themselves call their designated code generation function, such as *GenerateAddition*. However, this function can not only return the code that performs the addition for example. This addition, when executed, also returns a value which will be needed as an input by other operators. Therefore, not only the code fragment must be returned, but also the variable in which the result is stored. This variable can then be put on the stack for later use, and the code fragment must be added to the code already generated so that it can be returned to the caller. As with the interpreter, there is a final value on the stack when the loop has finished. Once the code is executed, this value is the variable containing the result of the expression. This value then needs to be stored in the results matrix, so that it can be retrieved by the CPU after all expressions have been executed on the GPU. Therefore, one last code fragment must be generated to handle the storage of this value in the results matrix. This fragment must then be added to the code already generated, and the transpilation process is complete.

Algorithm 3.2: Transpiling an equation in postfix-notation

```
1:  procedure Transpile(expr: PostfixExpression): String
2:      stack ← []
3:      code ← ""
4:      while HasTokenLeft(expr) do
5:          token ← GetNextToken(expr)
6:          if token.Type = Constant then
7:              Push(stack, token.Value)
8:          else if token.Type = Operator then
9:              if token.Value = Addition then
10:                 right ← Pop(stack)
11:                 left ← Pop(stack)
12:                 (valueLocation, codeFragment) ← GenerateAddition(left, right)
13:                 Push(stack, valueLocation)
14:                 Append(code, codeFragment)
15:             else if token.Value = Multiplication then
16:                 right ← Pop(stack)
17:                 left ← Pop(stack)
18:                 (valueLocation, codeFragment) ← GenerateMultiplication(left, right)
19:                 Push(stack, valueLocation)
20:                 Append(code, codeFragment)
21:     codeFragment ← GenerateResultStoring(Pop(stack))
22:     Append(code, codeFragment)
        return code
```

The code generated by the transpiler is the kernel for the transpiled expressions. This means that a new kernel must be generated for each expression that needs to be evaluated. This is in contrast to the interpreter, which has one kernel and dispatches it once for each expression. However, generating one kernel per expression results in a much simpler kernel. This allows the kernel to evaluate the postfix expression from left to right without having to perform any branching. However, while the kernel no longer has to perform branching, it is still needed and therefore only offloaded to the

CPU. There is also a noticeable overhead in that a kernel has to be generated for each expression. In cases like parameter optimisation, many of the expressions will be transpiled multiple times, because the transpiler is called multiple times with the same expressions. Strategies such as caching can be used to improve the performance in such cases.

Both the transpiler and the interpreter have their respective advantages and disadvantages. While the interpreter puts less load on the CPU, the GPU has to perform more work. Much of this work is branching and therefore involves many instructions that are not used to evaluate the expression itself. However, all of this work is done in parallel rather than sequentially on the CPU.

On the other hand, the transpiler performs more work on the CPU. The kernels are much simpler, and most of the instructions are used to evaluate the expressions themselves. Because unlike the GPU the CPU can manage state, concepts such as caches can be employed to reduce the overhead on the CPU. This means, that unnecessary work can be avoided in certain scenarios such as parameter optimisation.

Chapter 4

Implementation

somewhere in here explain why one kernel per expression and not one kernel for all expressions

4.1 Technologies

Short section; CUDA, PTX, Julia, CUDA.jl

Probably reference the performance evaluation papers for Julia and CUDA.jl

4.2 Expression Processing

Talk about why this needs to be done and how it is done (the why is basically: simplifies evaluation/transpilation process; the how is in ExpressionProcessing.jl)

4.3 Interpreter

Talk about how the interpreter has been developed.

UML-Ablaufdiagram

main loop; kernel transpiled by CUDA.jl into PTX and then executed

Memory access (currently global memory only) no dynamic memory allocation like on CPU (stack needs to have fixed size)

4.4 Transpiler

Talk about how the transpiler has been developed (probably largest section, because it just has more interesting parts)

UML-Ablaufdiagram

Front-End and Back-End Caching of back-end results

PTX code generated and compiled using CUDA.jl (so basically the driver) and then executed

Memory access (global memory and register management especially register management)

Chapter 5

Evaluation

5.1 Test environment

Explain the hardware used, as well as the actual data (how many expressions, variables etc.)

5.2 Results

talk about what we will see now (results only for interpreter, then transpiler and then compared with each other and a CPU interpreter)

5.2.1 Interpreter

Results only for Interpreter

5.2.2 Performance tuning

Document the process of performance tuning

Initial: CPU-Side single-threaded; up to 1024 threads per block; bounds-checking enabled (especially in kernel)

Blocksize reduced to a maximum of 256 -> moderate improvement in medium and large Using @inbounds -> noticeable improvement in 2 out of 3

5.2.3 Transpiler

Results only for Transpiler

5.2.4 Performance tuning

Document the process of performance tuning

Initial: CPU-Side single-threaded; up to 1024 threads per block; bounds-checking enabled

Blocksize reduced to a maximum of 256 -> moderate improvement in medium and large Using @inbounds -> small improvement only on CPU side code

5.2.5 Comparison

Comparison of Interpreter and Transpiler as well as Comparing the two with CPU interpreter

Chapter 6

Conclusion and Future Work

Summarise the results

6.1 Future Work

talk about what can be improved

Transpiler: transpile expression directly from Julia AST -> would save time because no intermediate representation needs to be created (looses step and gains performance, but also makes transpiler itself more complex)

References

Literature

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools (2nd edition)*. Addison-Wesley Longman Publishing Co., Inc. (Cit. on pp. 17, 18).

Bartlett, D. J., Desmond, H., & Ferreira, P. G. (2024). Exhaustive symbolic regression. *IEEE Transactions on Evolutionary Computation*, *28*(4), 950–964. https://doi.org/10.1109/TEVC.2023.3280250 (cit. on pp. 5, 6)

Bastidas Fuertes, A., Pérez, M., & Meza, J. (2023a). Transpiler-based architecture design model for back-end layers in software development. *Applied Sciences*, *13*(20), 11371. https://doi.org/10.3390/app132011371 (cit. on p. 20)

Bastidas Fuertes, A., Pérez, M., & Meza Hormaza, J. (2023b). Transpilers: A systematic mapping review of their usage in research and industry. *Applied Sciences*, *13*(6), 3667. https://doi.org/10.3390/app13063667 (cit. on p. 20)

Besard, T., Churavy, V., Edelman, A., & Sutter, B. D. (2019a). Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software*, *132*, 29–46. https://doi.org/10.1016/j.advengsoft.2019.02.002 (cit. on p. 7)

Besard, T., Foket, C., & De Sutter, B. (2019b). Effective extensible programming: Unleashing julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, *30*(4), 827–841. https://doi.org/10.1109/TPDS.2018.2872064 (cit. on p. 7)

Bomarito, G. F., Leser, P. E., Strauss, N. C. M., Garbrecht, K. M., & Hochhalter, J. D. (2022). Bayesian model selection for reducing bloat and overfitting in genetic programming for symbolic regression. *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 526–529. https://doi.org/10.1145/3520304.3528899 (cit. on p. 5)

Brodtkorb, A. R., Hagen, T. R., & Sætra, M. L. (2013). Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, *73*(1), 4–13. https://doi.org/10.1016/j.jpdc.2012.04.003 (cit. on p. 2)

Brunton, S. L., Proctor, J. L., & Kutz, J. N. (2016). Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, *113*(15), 3932–3937. https://doi.org/10.1073/pnas.1517384113 (cit. on p. 4)

Cano, A., & Ventura, S. (2014). GPU-parallel subtree interpreter for genetic programming. *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 887–894. https://doi.org/10.1145/2576768.2598272 (cit. on p. 20)

Chaber, P., & Ławryńczuk, M. (2016). Effectiveness of PID and DMC control algorithms automatic code generation for microcontrollers: Application to a thermal process. *2016 3rd Conference on Control and Fault-Tolerant Systems (SysTol)*, 618–623. https://doi.org/10.1109/SYSTOL.2016.7739817 (cit. on p. 19)

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, *2*(2), 137–167. https://doi.org/10.1016/S0019-9958(59)90362-6 (cit. on p. 18)

Collange, C. (2011, September). *Stack-less SIMT reconvergence at low cost* (Research Report). ENS Lyon. https://hal.science/hal-00622654. (Cit. on p. 11)

Cooper, K. D., & Torczon, L. (2022). *Engineering a compiler* (3rd ed.). Elsevier. http://dx.doi.org/10.1016/C2014-0-01395-0. (Cit. on pp. 17, 18)

Dabhi, V. K., & Chaudhary, S. (2012). A survey on techniques of improving generalization ability of genetic programming solutions. https://doi.org/10.48550/ARXIV.1211.1119 (cit. on p. 6)

Dietz, H. G., & Young, B. D. (2010). MIMD interpretation on a GPU. In G. R. Gao, L. L. Pollock, J. Cavazos, & X. Li (Eds.), *Languages and compilers for parallel computing* (pp. 65–79). Springer. https://doi.org/10.1007/978-3-642-13374-9_5. (Cit. on p. 20)

Dokken, T., Hagen, T. R., & Hjelmervik, J. M. (2005). The GPU as a high performance computational resource. *Proceedings of the 21st Spring Conference on Computer Graphics*, 21–26. https://doi.org/10.1145/1090122.1090126 (cit. on p. 6)

Dong, J., Zhong, J., Liu, W.-L., & Zhang, J. (2024). Evolving equation learner for symbolic regression. *IEEE Transactions on Evolutionary Computation*, 1–1. https://doi.org/10.1109/TEVC.2024.3404650 (cit. on p. 5)

ElTantawy, A., & Aamodt, T. M. (2016). MIMD synchronization on SIMT architectures. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1–14. https://doi.org/10.1109/MICRO.2016.7783714 (cit. on p. 12)

Franchetti, F., Kral, S., Lorenz, J., & Ueberhuber, C. (2005). Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, *93*(2), 409–425. https://doi.org/10.1109/JPROC.2004.840491 (cit. on p. 9)

Fua, P., & Lis, K. (2020, January 8). Comparing python, go, and c++ on the n-queens problem. https://doi.org/10.48550/arXiv.2001.02491. (Cit. on p. 20)

Fung, W. W. L., & Aamodt, T. M. (2011). Thread block compaction for efficient SIMT control flow. *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 25–36. https://doi.org/10.1109/HPCA.2011.5749714 (cit. on p. 11)

Georgescu, S., Chow, P., & Okuda, H. (2013). GPU acceleration for FEM-based structural analysis. *Archives of Computational Methods in Engineering*, *20*(2), 111–121. https://doi.org/10.1007/s11831-013-9082-8 (cit. on pp. 2, 7)

Gherardi, L., Brugali, D., & Comotti, D. (2012). A java vs. c++ performance evaluation: A 3d modeling benchmark. In I. Noda, N. Ando, D. Brugali, & J. J. Kuffner (Eds.), *Simulation, modeling, and programming for autonomous robots* (pp. 161–172). Springer. https://doi.org/10.1007/978-3-642-34327-8_17. (Cit. on p. 20)

Guillemot, H. (2022, December 31). Climate models. In K. De Pryck (Ed.), *A critical assessment of the intergovernmental panel on climate change* (1st ed., pp. 126–136). Cambridge University Press. http://dx.doi.org/10.1017/9781009082099.018. (Cit. on p. 4)

Han, S., Jang, K., Park, K., & Moon, S. (2010). PacketShader: A GPU-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, *40*(4), 195–206. https://doi.org/10.1145/1851275.1851207 (cit. on pp. 2, 7)

Han, T. D., & Abdelrahman, T. S. (2011). hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems*, *22*(1), 78–90. https://doi.org/10.1109/TPDS.2010.62 (cit. on p. 6)

Hissbach, A.-M., Dick, C., & Lawonn, K. (2022). An overview of techniques for egocentric black hole visualization and their suitability for planetarium applications. In J. Bender, M. Botsch, & D. A. Keim (Eds.), *Vision, modeling, and visualization*. The Eurographics Association. https://doi.org/10.2312/vmv.20221207. (Cit. on p. 7)

Huang, Q., Huang, Z., Werstein, P., & Purvis, M. (2008). GPU as a general purpose computing resource. *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, 151–158. https://doi.org/10.1109/PDCAT.2008.38 (cit. on p. 6)

Jin, Y., Fu, W., Kang, J., Guo, J., & Guo, J. (2020, January 16). Bayesian symbolic regression. https://doi.org/10.48550/arXiv.1910.08892. (Cit. on p. 5)

Johnson, S. C. (1975). *Yacc: Yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ. (Cit. on p. 18).

Keijzer, M. (2004). Scaled symbolic regression. *Genetic Programming and Evolvable Machines*, *5*(3), 259–269. https://doi.org/10.1023/B:GENP.0000030195.77571.f9 (cit. on p. 5)

Khairy, M., Wassal, A. G., & Zahran, M. (2019). A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity. *Journal of Parallel and Distributed Computing*, *127*, 65–88. https://doi.org/10.1016/j.jpdc.2018.11.012 (cit. on p. 12)

Korns, M. F. (2011). Accuracy in symbolic regression. In R. Riolo, E. Vladislavleva, & J. H. Moore (Eds.), *Genetic programming theory and practice IX* (pp. 129–151). Springer. https://doi.org/10.1007/978-1-4614-1770-5_8. (Cit. on p. 5)

Köster, M., Groß, J., & Krüger, A. (2020a). Massively parallel rule-based interpreter execution on GPUs using thread compaction. *International Journal of Parallel Programming*, *48*(4), 675–691. https://doi.org/10.1007/s10766-020-00670-2 (cit. on pp. 11, 20)

Köster, M., Groß, J., & Krüger, A. (2020b). High-performance simulations on GPUs using adaptive time steps. In M. Qiu (Ed.), *Algorithms and architectures for parallel processing* (pp. 369–385). Springer International Publishing. https://doi.org/10.1007/978-3-030-60245-1_26. (Cit. on p. 7)

Köster, M., Groß, J., & Krüger, A. (2022). MACSQ: Massively accelerated DeepQ learning on GPUs using on-the-fly state construction. In H. Shen, Y. Sang, Y. Zhang, N. Xiao, H. R. Arabnia, G. Fox, A. Gupta, & M. Malek (Eds.), *Parallel and distributed computing, applications and technologies* (pp. 383–395). Springer Inter-

national Publishing. https://doi.org/10.1007/978-3-030-96772-7_35. (Cit. on p. 7)

Koza, J. R. (2010). Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, *11*(3), 251–284. https://doi.org/10.1007/s10710-010-9112-3 (cit. on p. 5)

Koza, J. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, *4*(2). https://doi.org/10.1007/BF00175355 (cit. on pp. 4, 5)

Kronberger, G., Burlacu, B., Kommenda, M., Winkler, S. M., & Affenzeller, M. (2024, July). *Symbolic regression*. Chapman; Hall/CRC. http://dx.doi.org/10.1201/9781315166407. (Cit. on pp. 5, 6)

Langdon, W. B., & Banzhaf, W. (2008). A SIMD interpreter for genetic programming on GPU graphics cards. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcázar, I. De Falco, A. Della Cioppa, & E. Tarantino (Eds.), *Genetic programming* (pp. 73–85). Springer. https://doi.org/10.1007/978-3-540-78671-9_7. (Cit. on p. 20)

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 75–86. https://doi.org/10.1109/CGO.2004.1281665 (cit. on p. 18)

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., & Dubey, P. (2010). Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. *Proceedings of the 37th annual international symposium on Computer architecture*, 451–460. https://doi.org/10.1145/1815961.1816021 (cit. on p. 7)

Lemos, P., Jeffrey, N., Cranmer, M., Ho, S., & Battaglia, P. (2022, February 4). Rediscovering orbital mechanics with machine learning. https://doi.org/10.48550/arXiv.2202.02306. (Cit. on p. 5)

Lin, D.-L., Ren, H., Zhang, Y., Khailany, B., & Huang, T.-W. (2023). From RTL to CUDA: A GPU acceleration flow for RTL simulation with batch stimulus. *Proceedings of the 51st International Conference on Parallel Processing*, 1–12. https://doi.org/10.1145/3545008.3545091 (cit. on p. 19)

Lin, W.-C., & McIntosh-Smith, S. (2021). Comparing julia to performance portable parallel programming models for HPC. *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 94–105. https://doi.org/10.1109/PMBS54543.2021.00016 (cit. on p. 7)

Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J. R., & Hassan, A. E. (2022). In rust we trust: A transpiler from unsafe c to safer rust. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 354–355. https://doi.org/10.1145/3510454.3528640 (cit. on p. 19)

Marcelino, M., & Leitão, A. M. (2022). Transpiling python to julia using PyJL. https://doi.org/10.5281/ZENODO.6332890 (cit. on p. 19)

Martius, G., & Lampert, C. H. (2016). Extrapolation and learning equations. https://doi.org/10.48550/ARXIV.1610.02995. (Cit. on p. 5)

Michalakes, J., & Vachharajani, M. (2008). GPU acceleration of numerical weather prediction. *2008 IEEE International Symposium on Parallel and Distributed Processing*, 1–7. https://doi.org/10.1109/IPDPS.2008.4536351 (cit. on pp. 2, 7)

Moses, W. S., Ivanov, I. R., Domke, J., Endo, T., Doerfert, J., & Zinenko, O. (2023). High-performance GPU-to-CPU transpilation and optimization via high-level parallel constructs. *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 119–134. https://doi.org/10.1145/3572848.3577475 (cit. on p. 19)

Palacios, J., & Triska, J. (2011). A comparison of modern GPU and CPU architectures: And the common convergence of both. https://api.semanticscholar.org/CorpusID:61428375 (cit. on p. 8)

Pfahler, L., & Morik, K. (2020). Semantic search in millions of equations. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 135–143. https://doi.org/10.1145/3394486.3403056 (cit. on p. 4)

Romer, T. H., Lee, D., Voelker, G. M., Wolman, A., Wong, W. A., Baer, J.-L., Bershad, B. N., & Levy, H. M. (1996). The structure and performance of interpreters. *SIGPLAN Not.*, *31*(9), 150–159. https://doi.org/10.1145/248209.237175 (cit. on p. 20)

Sahoo, S. S., Lampert, C. H., & Martius, G. (2018). Learning equations for extrapolation and control. https://doi.org/10.48550/ARXIV.1806.07259. (Cit. on p. 5)

Schuurman, D. C. (2013). Step-by-step design and simulation of a simple CPU architecture. *Proceeding of the 44th ACM technical symposium on Computer science education*, 335–340. https://doi.org/10.1145/2445196.2445296 (cit. on p. 8)

Sun, F., Liu, Y., Wang, J.-X., & Sun, H. (2023, February 2). Symbolic physics learner: Discovering governing equations via monte carlo tree search. https://doi.org/10.48550/arXiv.2205.13134. (Cit. on p. 5)

Tian, X., Saito, H., Girkar, M., Preis, S. V., Kozhukhov, S. S., Cherkasov, A. G., Nelson, C., Panchenko, N., & Geva, R. (2012). Compiling c/c++ SIMD extensions for function and loop vectorizaion on multicore-SIMD processors. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2349–2358. https://doi.org/10.1109/IPDPSW.2012.292 (cit. on p. 9)

Verbraeck, A., & Eisemann, E. (2021). Interactive black-hole visualization. *IEEE Transactions on Visualization and Computer Graphics*, *27*(2), 796–805. https://doi.org/10.1109/TVCG.2020.3030452 (cit. on p. 7)

Wang, C.-K., & Chen, P.-S. (2015). Automatic scoping of task clauses for the OpenMP tasking model. *The Journal of Supercomputing*, *71*(3), 808–823. https://doi.org/10.1007/s11227-014-1326-3 (cit. on p. 19)

Wang, L.-T., Chang, Y.-W., & Cheng, K.-T. (2009, March 11). *Electronic design automation: Synthesis, verification, and test*. Morgan Kaufmann. (Cit. on p. 19).

Werner, M., Junginger, A., Hennig, P., & Martius, G. (2021, May 13). Informed equation learning. https://doi.org/10.48550/arXiv.2105.06331. (Cit. on p. 5)

Zhang, Y., Ren, H., & Khailany, B. (2020). Opportunities for RTL and gate level simulation using GPUs. *Proceedings of the 39th International Conference on Computer-Aided Design*, 1–5. https://doi.org/10.1145/3400302.3415773 (cit. on p. 19)

Online sources

AMD. (2025a, February). *Hardware features — HIP 6.3.42134 documentation*. Retrieved March 15, 2025, from https://rocm.docs.amd.com/projects/HIP/en/latest/reference/hardware_features.html. (Cit. on p. 12)

AMD. (2025b, February). *HIP programming model — HIP 6.3.42134 documentation*. Retrieved March 9, 2025, from https://rocm.docs.amd.com/projects/HIP/en/latest/understand/programming_model.html. (Cit. on pp. 9, 10, 14)

GCC. (2025, January). *GCC online documentation*. Retrieved March 18, 2025, from https://gcc.gnu.org/onlinedocs/. (Cit. on p. 17)

Intel. (1978). *MCS · 86 assembly language converter operating instructions for ISIS · II users* [Technical Report]. Retrieved March 21, 2025, from http://www.bitsavers.org/pdf/intel/ISIS_II/9800642A_MCS-86_Assembly_Language_Converter_Operating_Instructions_for_ISIS-II_Users_Mar79.pdf. (Cit. on p. 19)

Lindholm, T., Yellin, F., Bracha, G., Buckley, A., & Smith, D. (2025, February). *The java® virtual machine specification*. Retrieved March 18, 2025, from https://docs.oracle.com/javase/specs/jvms/se24/html/. (Cit. on pp. 18, 20)

Microsoft. (2023, March). *Overview of .NET framework - .NET framework | microsoft learn*. Retrieved March 20, 2025, from https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview. (Cit. on p. 18)

Microsoft. (2025, March). *TypeScript: The starting point for learning TypeScript*. Retrieved March 21, 2025, from https://www.typescriptlang.org/docs/handbook/intro.html. (Cit. on p. 19)

Nvidia. (2025a, March). *CUDA c++ best practices guide 12.8 documentation*. Retrieved March 16, 2025, from https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. (Cit. on p. 16)

Nvidia. (2025b, March). *CUDA c++ programming guide*. Retrieved November 22, 2024, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/. (Cit. on pp. 7, 8, 14, 23)

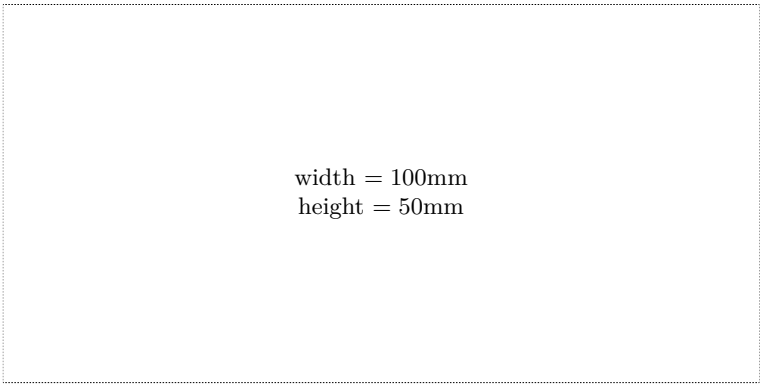
Nvidia. (2025c, March). *Nsight compute — NsightCompute 12.8 documentation*. Retrieved March 16, 2025, from https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator. (Cit. on p. 15)

Nvidia. (2025d, March). *Parallel thread execution ISA version 8.7*. Retrieved March 15, 2025, from https://docs.nvidia.com/cuda/parallel-thread-execution/. (Cit. on pp. 16, 17)

Sutter, H. (2004, December). *The free lunch is over: A fundamental turn toward concurrency in software*. Retrieved March 13, 2025, from http://www.gotw.ca/publications/concurrency-ddj.htm. (Cit. on p. 1)

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —