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

Abstract

The objective of symbolic regression is to identify an expression that accurately models a system based on a set of inputs. For instance, one might determine the flow through pipes using inputs such as roughness, diameter, and length by conducting experiments with varying input configurations and observing the resulting flow and derive an expression from the experiments. This methodology, exemplified by Nikuradse (1950), can be applied to any system through symbolic regression. To find the best-fitting expression, millions of candidate expressions are generated, each requiring evaluation against every input configuration to assess how well they fit to the system. Consequently, millions of evaluations must be performed, a process that is computationally intensive and time-consuming. Thus, optimizing the evaluation phase of symbolic regression is crucial for discovering expressions that describe large and complex systems within a feasible timeframe.

This thesis presents the design and implementation of two evaluators that utilize the GPU to evaluate expressions generated at runtime by the symbolic regression algorithm. Performance benchmarks are conducted to compare the efficiency of the GPU evaluators against the current CPU evaluator.

The benchmark results indicate that the GPU can serve as a viable alternative to the CPU in certain scenarios. The determining factor for choosing between GPU and CPU evaluation is the number of input configurations. In a scenario with 10 000 expressions and 10 000 input configurations, the GPU outperformed the CPU by a significant margin.

This master thesis is associated with the FFG COMET project ProMetHeus (#904919). The developed software is used and further developed for modelling in the ProMetHeus project.

Kurzfassung

Das Ziel der symbolischen Regression ist es, einen Ausdruck zu finden, der ein System basierend auf einer Reihe von Variablen modelliert. Beispielsweise kann man den Durchfluss durch Rohre unter Verwendung von Variablen wie Rauheit, Durchmesser und Länge bestimmen, indem Experimente mit verschiedenen Werten für die Variablen durchgeführt werden. Für jedes Experiment wird der Durchfluss gemessen, wodurch man eine allgemeine Formel ableiten kann, welche die Beziehung der Variablen mit dem Durchfluss beschreibt. Diese Methodik, veranschaulicht durch die Arbeit von Nikuradse (1950), kann auf unterschiedliche Systeme mithilfe von symbolischer Regression angewendet werden. Um einen Ausdruck zu finden, welcher das System am besten beschreibt, werden Millionen von Kandidatenausdrücken generiert. Diese müssen, unter Verwendung der Variablenkonfiguration aller Experimente ausgewertet werden, um ihre Passgenauigkeit zum System zu beurteilen. Folglich müssen Millionen von Auswertungen durchgeführt werden, ein Prozess, der rechenintensiv und zeitaufwendig ist. Daher ist die Optimierung der Auswertungsphase der symbolischen Regression entscheidend. So wird es ermöglicht Ausdrücke in einem angemessenen Zeitrahmen zu finden, welche große und komplexe Systeme beschreiben.

Diese Arbeit präsentiert das Design und die Implementierung von zwei Evaluatoren, die die Grafikkarte (GPU) nutzen, um Ausdrücke zu bewerten, die zur Laufzeit der symbolischen Regression generiert werden. Leistungsbenchmarks werden durchgeführt, um die Performanz der GPU-Evaluatoren mit dem aktuellen CPU-Evaluator zu vergleichen.

Die Benchmark-Ergebnisse zeigen, dass die GPU in bestimmten Szenarien als eine tragfähige Alternative zur CPU dienen kann. Der entscheidende Faktor für die Wahl zwischen GPU- und CPU-Auswertung ist die Anzahl der Experimente und folglich die Menge an Variablenkonfigurationen. In einer Konfiguration mit 10 000 Ausdrücken und 10 000 Variablenkonfigurationen übertraf die GPU die CPU um ein bedeutendes Maß.

Diese Masterarbeit steht im Zusammenhang mit dem FFG COMET Projekt ProMetHeus (#904919). Die entwickelte Software wird für die Modellierung im ProMetHeus Projekt verwendet und weiterentwickelt.

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. Finally, the structure of this thesis is described, explaining how each part contributes to answering the research questions.

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 $5 - \text{abs}(x_1)\sqrt{p_1}/10 + 2^{x_2}$, 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 hundreds or even thousands of such expressions per iteration, all of which have to be evaluated. Additionally, multiple different values must be entered 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). These solutions were all developed using CUDA[1]. However, it is also possible to develop assembly like code for GPUs using Parallel Thread Execution (PTX)[2] to gain more control.

1.2 Research Question

Given the successful implementation of GPU acceleration, the aim of this thesis is 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 feasible. If it is feasible, it is important to determine if evaluating the expressions on the GPU 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, both 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 under which circumstances.

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.

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

[2] https://docs.nvidia.com/cuda/parallel-thread-execution/

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. In one instance Werner et al. (2021) described that they want to find an expression to predict the power loss of an electric machine based on known input values. 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. With the help of an equation learner, they were able to generate useful results.

A more literal interpretation of equation learning is demonstrated by Pfahler and Morik (2020). They use machine learning to learn the form of equations to simplify the discovery of relevant publications. Instead of searching for keywords which might differ from one field of research to another, they allow searching by the equations the publications use. This helps as the form of equations stay the same over different fields and are therefore not subject to specific terminology. However, this form 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 the evolutionary algorithm genetic programming (GP) for different problems, including symbolic regression, was first popularised 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 relevant 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. One of these scenarios is finding simpler but still accurate models for hydrodynamic simulations to speed up the design process of ship hulls. Another one is finding an expression to find the remaining capacity of a Lithium-ion battery by measuring its voltage. In total, they described ten scenarios from different domains to show the capabilities of symbolic regression.

Keijzer (2004), Gustafson et al. (2005), Korns (2011), Korns (2015), Bruneton (2025) and many more 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 for example Bayesian Symbolic Regression as proposed by Jin et al. (2020). Their approach increased the quality of the results noticeably compared to GP alternatives by for example incorporating prior knowledge. 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.

Another alternative to meta-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 Brunton et al. (2016). Their algorithm was specifically designed for nonlinear dynamics

often occurring in physical systems. An improvement to this approach was introduced by Sun et al. (2023) where they used Monte Carlo tree search. When compared to other implementations their equation learner was able to create better results but has the main drawback of high computational cost.

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 Koza (1994). Expressions also consist of constants as well as variables which represent the inputs. Assuming that a given problem has two variables and one parameter, the equation learner could generate an expression as seen in Equation 2.1 where x_n are the variables, p_1 is the parameter and O is the output which should correspond to the observed output for the given variables.

$$O = 5 - \text{abs}(x_1) + x_2 \sqrt{p_1}/10 \tag{2.1}$$

A typical equation learner generates multiple expressions at once. If for example the equation learner generates 300 expressions per GP generation, each of these expressions needs to be evaluated at least once to determine how well they can produce the desired output. Each expression lies in a different part of the search space and with only the variables, it would not easily be possible to explore the surrounding search space. To perform for example local search in this area, the parameter p_1 can be used. This local search phase helps to find the local or even global optimum. For example 50 local search steps can be used, meaning that each expression needs to be evaluated 50 times with the same variables, but different parameters. As a result, one GP generation consequently requires a total $300 * 50 = 15\,000$ evaluations of the expressions. However, typically more than one GP generation is needed to find a good local optimum. While the exact number of generations is problem specific, for this example a total of 100 generations can be assumed. Each generation again generates 300 expressions and needs to perform 50 local search steps. This results in a total of $300 * 50 * 100 = 1\,500\,000$ evaluations which need to be performed during the entire runtime of the GP algorithm. 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 $300 * 50 * 54 \approx 800\,000$ evaluations. 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 to compute 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 GPUs can yield noticeable better performance.

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 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

[1] https://developer.nvidia.com/cuda-toolkit
[2] https://julialang.org/
[3] https://www.amd.com/de/products/software/rocm.html
[4] https://www.khronos.org/opencl/

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.

2.2.1 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 how a modern CPU works Knuth (1999) introduced the MMIX architecture. It is a 64-bit CPU architecture containing many concepts and design decisions to compete with other CPUs on the market at that time. He provides the information in great detail and demonstrates the complexity of CPU architectures. Current CPUs are even more complex, and often contain features like sophisticated branch prediction among other things to achieve higher and higher performance. This makes a CPU perfect for handling complex control flows on a single program thread and even multiple threads 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 a single CPU core. As described by Nvidia (2025b), a GPU designates much more transistors towards floating-point computations. This, however, 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 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

[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/

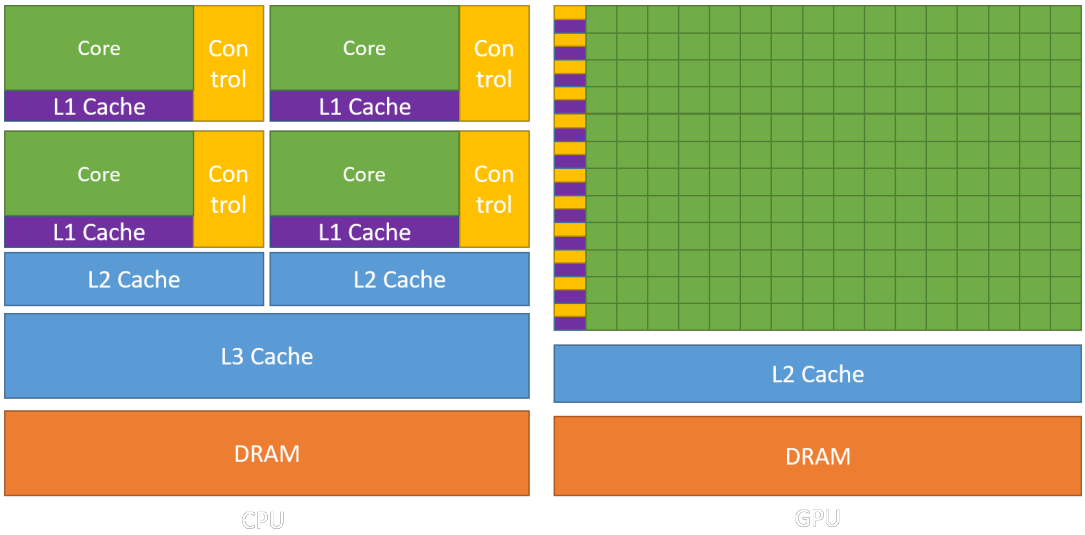


Figure 2.1: Overview of the architecture of a CPU (left) and a GPU (right). Note the higher number of simpler and smaller cores on the GPU (Nvidia, 2025b).

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, as well as the threads created by the developer, are grouped together in several categories. This is the so-called 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

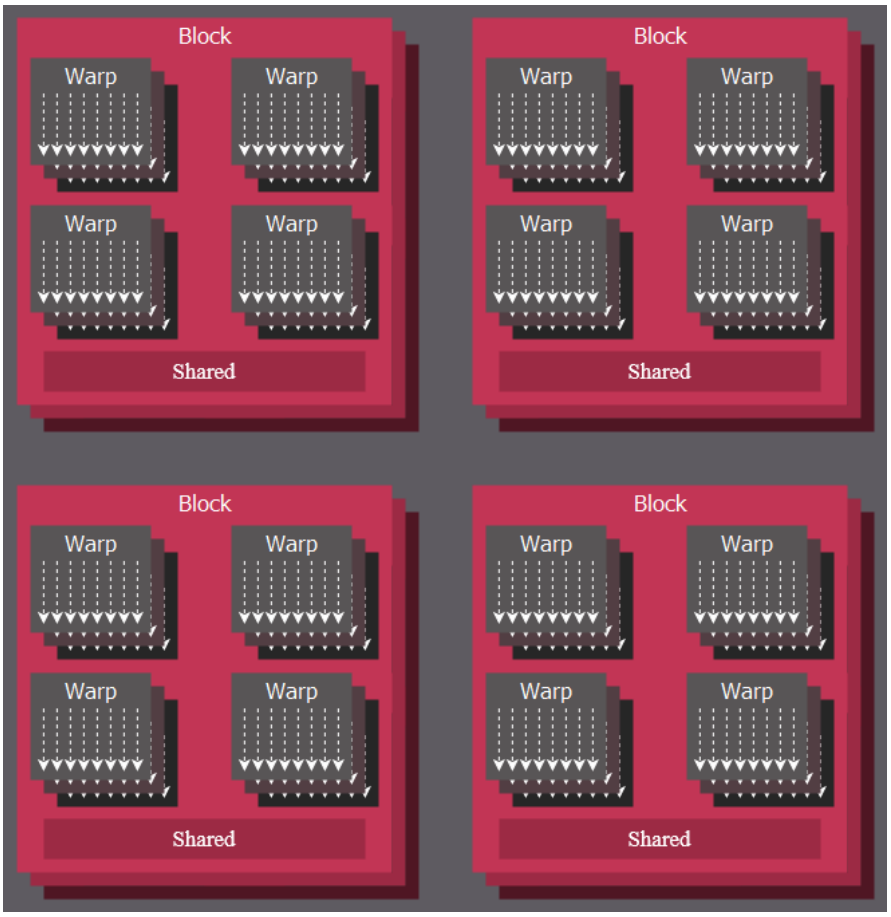


Figure 2.2: An overview of the thread hierarchy with blocks being split into multiple warps and their shared memory (AMD, 2025b).

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 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.

Once a kernel is dispatched, all threads start at the same point in a program. However, because a thread may encounter instructions, such as branches, where it can take a different path to the other threads, or in other words diverge, each thread has a unique instruction pointer. This allows threads to work independently, even if they are part of the same warp. However, because of the SIMD architecture, all threads in a warp must execute the same instructions and if threads start to diverge, 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 in the figure also shows, that the thread could re-converge after the divergence. On older hardware this does not happen and results in T2 being executed after T1 and T3 have finished. In situations where there is a lot of data dependent thread divergence, most of the benefits of using a GPU are likely to be lost. 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 poor resource utilisation, which in turn leads to poor performance. Another way in which threads can be paused (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 what is known as the Single-Instruction Multiple-Thread (SIMT) architecture. In many cases a developer does not need to know the details of SIMT and can design fast, correct and accurate 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 has occurred. SIMT can also help with increasing the occupancy of the GPU. Occupancy and its importance to performance is discussed in detail in Section 2.2.1.

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

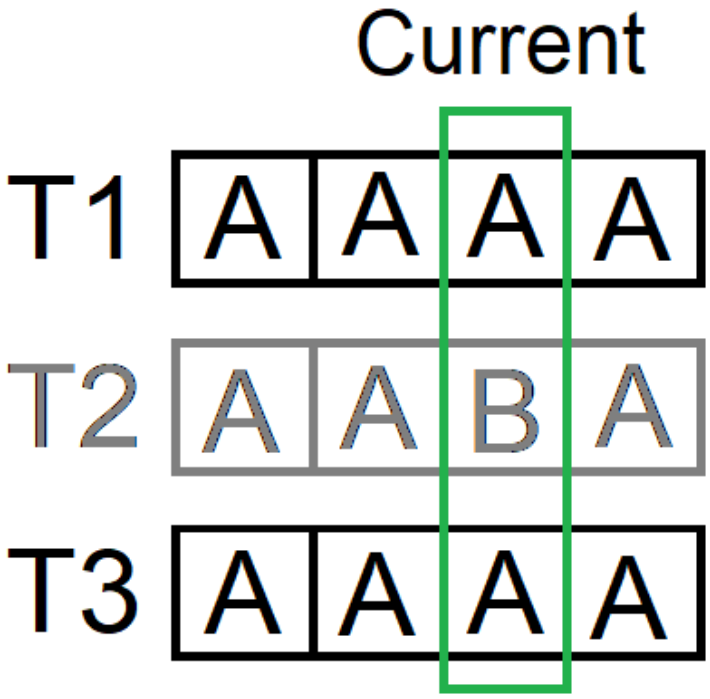


Figure 2.3: Thread T2 wants to execute instruction B while T1 and T3 want to execute instruction A. Therefore T2 will be an inactive thread this cycle and active once T1 and T3 are finished. This means that now the divergent threads are serialised.

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 reconvergence, 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

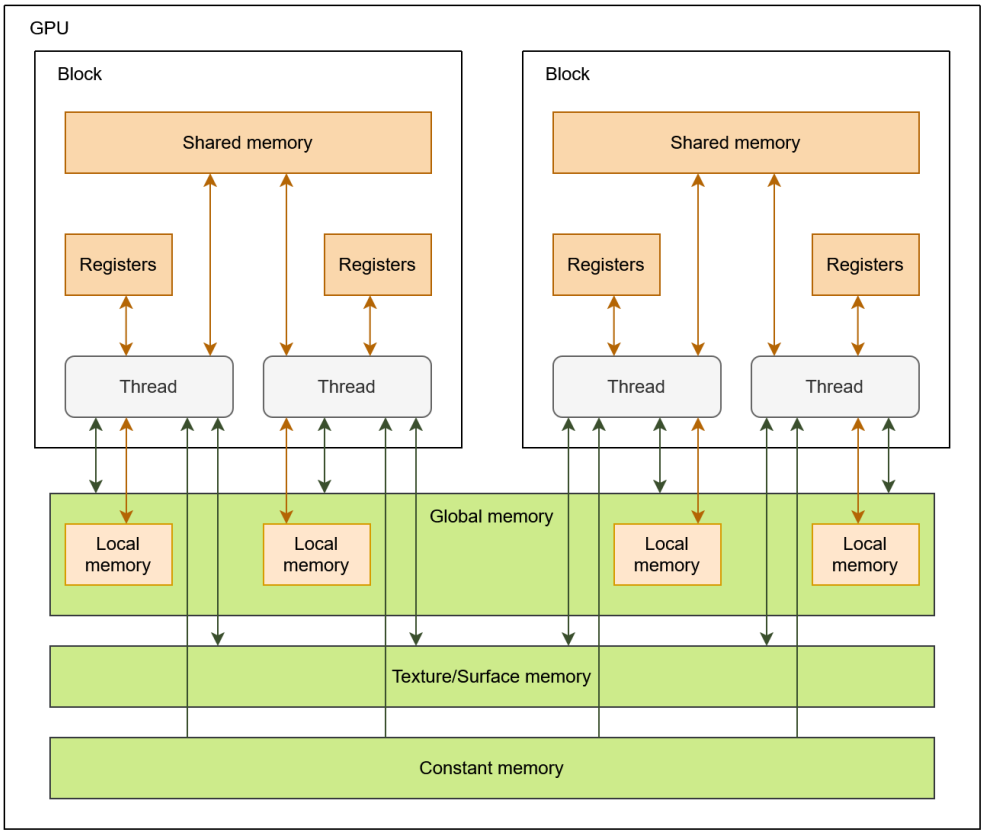


Figure 2.4: The layout of the memory in the GPU. The connections between the memory regions can be seen as well as the different kinds of memory available.

level, threads have registers and local memory available. Registers are the fastest way to access memory, but they are 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. In contrast to registers, local memory is significantly slower. This is due to the fact, that local memory is actually stored in global memory and therefore has the same limitations as explained later. This means that it is important to try and avoid local memory as much as possible. Local memory is usually only used when a thread is using too many registers. The compiler will then spill the remaining data into local memory and load it into registers once needed, slowing down the application drastically.

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 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 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 per-

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.

formance 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

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

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 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

[9] https://juliagpu.org/

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 is similar to assembler style code. Every PTX code must have a `.version` directive which indicates the PTX version and is immediately followed by the `.target` directive which indicates the compute capability. If the program needs 64-bit addresses instead of the default 32-bit addresses, the optional `.address_size` directive can be used to indicate this. Using 64-bit addresses enables the developer to access more than 4 GB of memory but also increases register usage, as a 64-bit address must be stored in two registers.

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:

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

Loops in the classical sense do not exist in PTX. Instead, 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

```
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.

also called backend. The frontend 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.

2.3.1 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

[10] https://learn.microsoft.com/en-us/dotnet/standard/clr

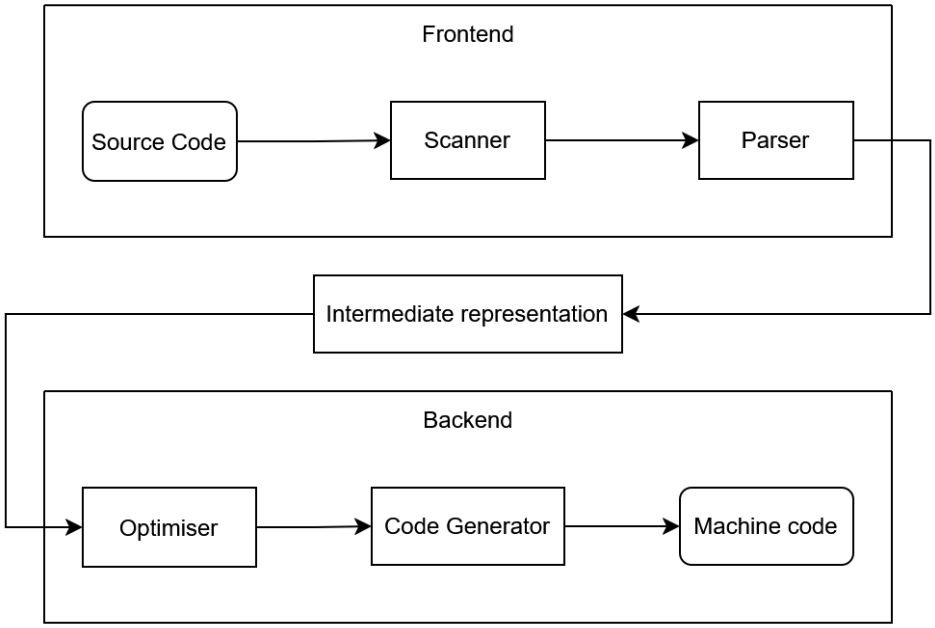


Figure 2.5: A simplified overview of how the architecture of a compiler looks, using Flex and Bison.

(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.

2.3.2 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 the TypeScript transpiler, 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 transpilers 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.

Chapter 3

Concept and Design

To be able to determine whether evaluating mathematical expressions on the GPU is better suited than on the CPU, two prototypes need to be implemented. More specifically, a prototype for interpreting these expressions on the GPU, as well as a prototype that transpiles expressions into PTX 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, $1/x$ and $\sqrt{x}$. Note that x and y can either stand for a constant, a variable, a parameter, 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.

With this, the required capabilities are outlined. However, for a better understand-

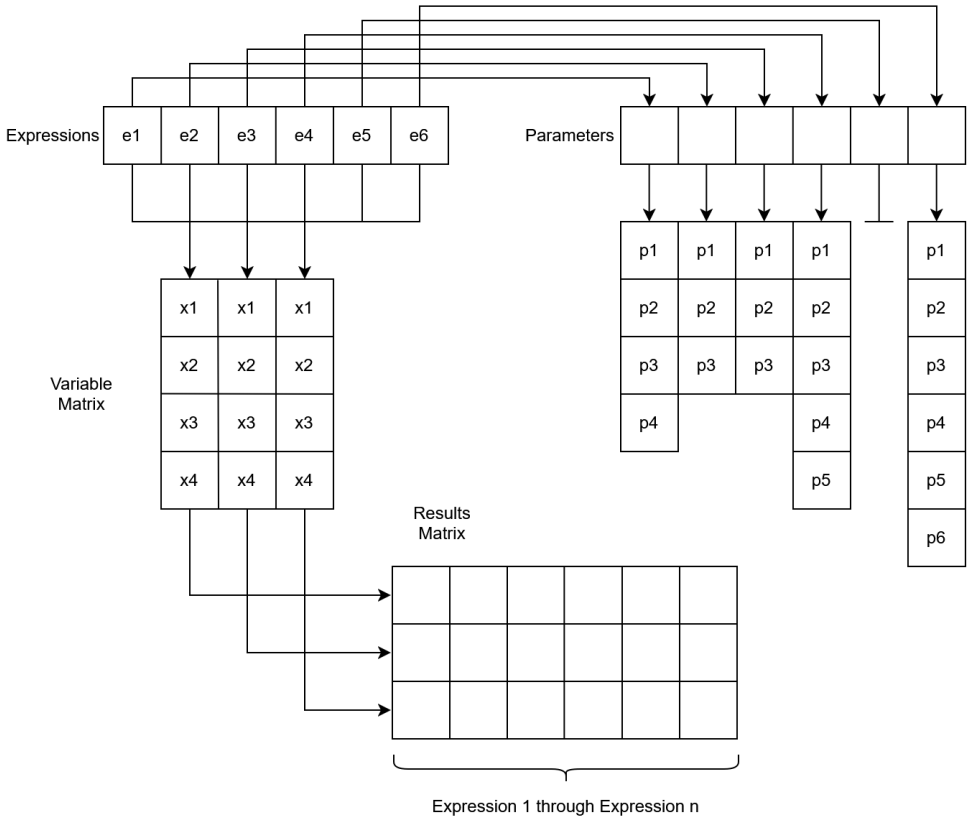


Figure 3.1: This diagram shows how the input and output looks like and how they interact with each other.

ing, the input and output data need to be explained further. The first input contains the expressions that need to be evaluated. These can be of any length and can contain constant values, variables and parameters, all of which are linked together with the supported operators. In the simplified example shown in Figure 3.1, there are six expressions e_1 to e_6.

Next is the variable matrix. An entry in this matrix corresponds to one variable in every expression. The row indicates which variable it holds the value for. For example the values in row three are used to parameterise the variable x_3. Each column holds a different set of variables. Each expression must be evaluated using each set of variables. In the provided example, there are three variable sets, each containing the values for four variables x_1 to x_4.

After all expressions have been evaluated using all variable sets, the results of these evaluations must be stored in the result matrix. Each entry in this matrix holds the result of the evaluation of one expression parameterised with one variable set. The row indicates the variable set and the column indicates the expression.

The prototypes developed in this thesis, are part of a GP algorithm for symbolic regression. This means that the expressions that are evaluated, represent parts of the

search space of all expressions being made up of any combination of allowed operators, the set of input variables, a set of parameters and constants. This means that the size of the search space grows exponentially. Exploring this search space by simply generating expressions, evaluating them once and then generating the next set of expressions leaves much of the search space unexplored. To combat this, parameters are introduced. These allow the algorithm to perform some kind of intensification. To enable this, the prototypes must support not only variables, but also parameters.

The parameters themselves are unique to each expression, meaning they have a one-to-one mapping to an expression. Furthermore, as can be seen in Figure 3.1, each expression can have a different number of parameters, or even no parameters at all. However, with no parameters, it wouldn't be possible to perform parameter optimisation. This is in contrast to variables, where each expression must have the same number of variables. Because parameters are unique to each expression and can vary in size, they are not structured as a matrix, but as a vector of vectors.

An important thing to consider, is the volume and volatility of the data itself. The example shown in Figure 3.1 has been drastically simplified. It is expected, that there are hundreds of expressions evaluate per GP generation. Each of these expressions may contain between ten and 50 tokens. A token is equivalent to either a variable, a parameter, a constant value or an operator.

It can be assumed that typically the number of variables per expression is around ten. However, the number of variable sets can increase drastically. It can be considered that 1 000 variable sets is the lower limit. On the other hand, 100 000 can be considered as the upper limit. Considering that one variable takes up 4 bytes of memory and 10 variables are needed per expression, at least $4 * 10 * 1\,000 = 40\,000$ bytes and at most $4 * 10 * 100\,000 = 400\,000$ bytes need to be transferred to the GPU for the variables.

These variables do not change during the runtime of the symbolic regression algorithm. As a result the data only needs to be sent to the GPU once. This means that the impact of this data transfer is minimal. On the other hand, the data for the parameters is much more volatile. As explained above, they are used for parameter optimisation and therefore vary from evaluation to evaluation and need to be sent to the GPU very frequently. The amount of data that needs to be sent depends on the number of expressions as well as on the number of parameters per expression. Considering 10 000 expressions that need to be evaluated and an average of two parameters per expression each requiring 4 bytes of memory, a total of $10\,000 * 2 * 4 = 80\,000$ bytes need to be transferred to the GPU on each parameter optimisation step.

3.2 Architecture

Based on the requirements and data structure 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 seen in Figure 3.2. 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

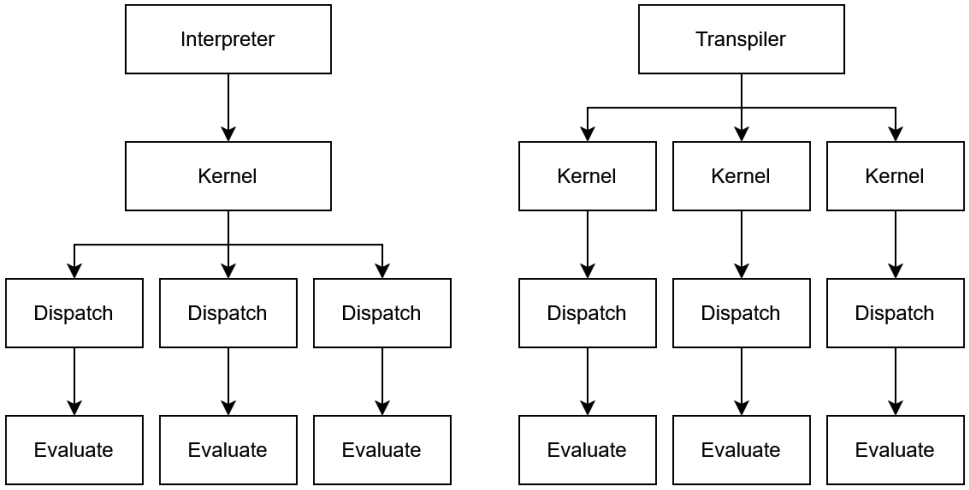


Figure 3.2: The interpreter has one kernel that is dispatched multiple times, while the transpiler, has multiple kernels that are dispatched once. This helps to eliminate thread divergence.

negative effects of thread divergence, it is still advisable 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, further increases GPU utilisation. 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 rather 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 expressions themselves. This also reduces the overhead on the GPU. One drawback of generating a kernel for each expression, is the generation itself. Especially for smaller variable sets, it is possible, that the time it takes to transpile an expression and compile the kernel into machine code is greater than the time it takes to evaluate it. However, for larger variable sets this should not be a concern.

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 expressions. This is comparable to the work a scanner like Flex[1] performs. Secondly, this step also converts the expression into an intermediate representation. In essence,

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

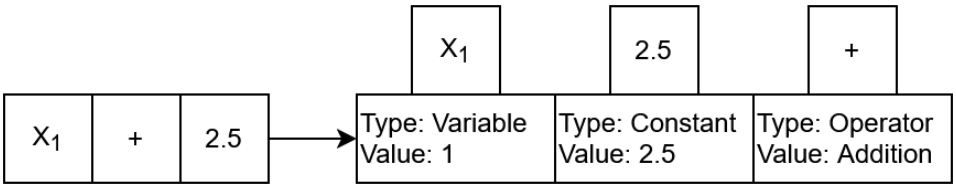


Figure 3.3: This diagram shows how an expression will be transformed in the pre-processing step.

the pre-processing step can be compared to the frontend of a compiler as described in Section 2.3. If new operators are required, the pre-processor must be extended as well. Otherwise, expressions containing these operators would be treated as invalid and never reach the evaluator.

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 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 3.3 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.

The already mentioned concept of processing one expression per thread can also be used on the CPU, which is better designed for this type of work. Concepts such as caching processed expressions, or caching parts of the processed expressions can also be employed on the CPU to speed up pre-processing. This would not be possible on the GPU, because a GPU can not save state between two kernel dispatches. This is a typical example of code that is better run on the CPU and shows how the CPU and GPU need to work together and exploit their respective strengths to achieve the best performance.

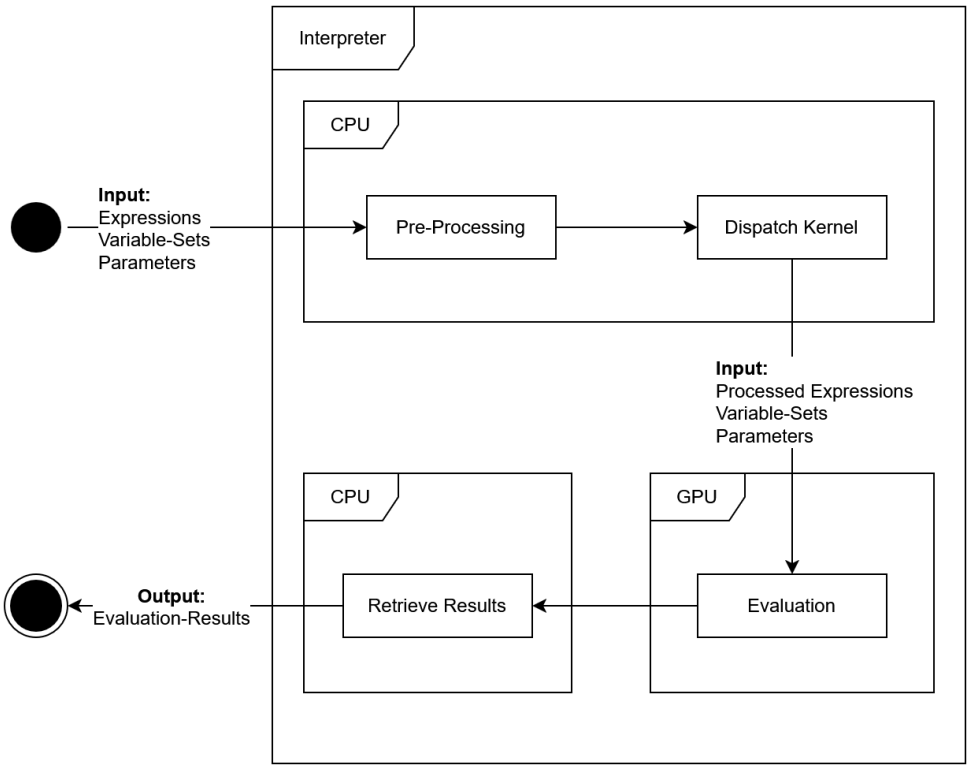


Figure 3.4: This diagram depicts the coarse-grained workflow of the interpreter. It shows how the parts interact with each other and with the system it will operate in.

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 of the components and the workflow of the interpreter is shown in Figure 3.4. Once the interpreter has received the expressions, they are pre-processed. This ensures that the expressions are valid, and that they are transformed into the intermediate representation needed to evaluate them. The result of this pre-processing step is then sent to the GPU, which performs the actual interpretation of the expressions. In addition to the expressions, the data for the variables and parameters must also be sent to the GPU.

Once all the necessary data is present on the GPU, the interpreter kernel can be dispatched. As previously mentioned, the kernel is dispatched for each expression to minimise thread divergence. In fact, dispatching the same kernel multiple times for each expression ensures that there will not occur any thread divergence, as will be explained later.

After the GPU has finished evaluating all expressions with all variable sets, the result is 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. Because the expressions are in postfix notation, the actual interpreter just needs to iterate over all the tokens

and perform the appropriate tasks. If the interpreter encounters a binary operator, it simply needs to read the previous two values and perform the operation specified by the operator. For unary operators, only the previous value needs to 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. This also means that each token is visited exactly once and no unnecessary or overhead work needs to be done. The Algorithm 3.1 shows how the interpreter works. Note that this is a simplified version, that only works with additions, multiplications, constants and variables.

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 = Variable then
8:              Push(stack, GetVariable(token.Value))
9:          else if token.Type = Operator then
10:             if token.Value = Addition then
11:                 right ← Pop(stack)
12:                 left ← Pop(stack)
13:                 Push(stack, left + right)
14:             else if token.Value = Multiplication then
15:                 right ← Pop(stack)
16:                 left ← Pop(stack)
17:                 Push(stack, left ∗ right)
18:     StoreResult(Pop(stack))
```

Handling constants, variables and parameters is very simple. Constants simply need to be stored on the stack for later use. Variables and parameters also only need to be stored on the stack. However, their value must first be loaded from the variable or parameter matrix according to the token value. Since the entire matrices are sent to the GPU, the index of the variable or parameter set is also needed to load the correct value. However, for simplicity, this has been omitted from the algorithm.

When an operator token is encountered, the handling becomes more complex. The value of the token indicates the type of operation to be applied. For binary operators, the top two values on the stack need to be used as input to the operator. For unary operators, only the top value of the stack needs to be used as an input. Once the result has been computed, it must be stored at the top of the stack to be used as an input for the next operation or the result for this expression.

At the end of the algorithm, the stack contains one last entry. This entry is the value computed by the expression with the designated variable set and parameters. In order to send this value back to the CPU, it must be stored in the result matrix. The last statement performs this action. It again has been simplified to omit the index

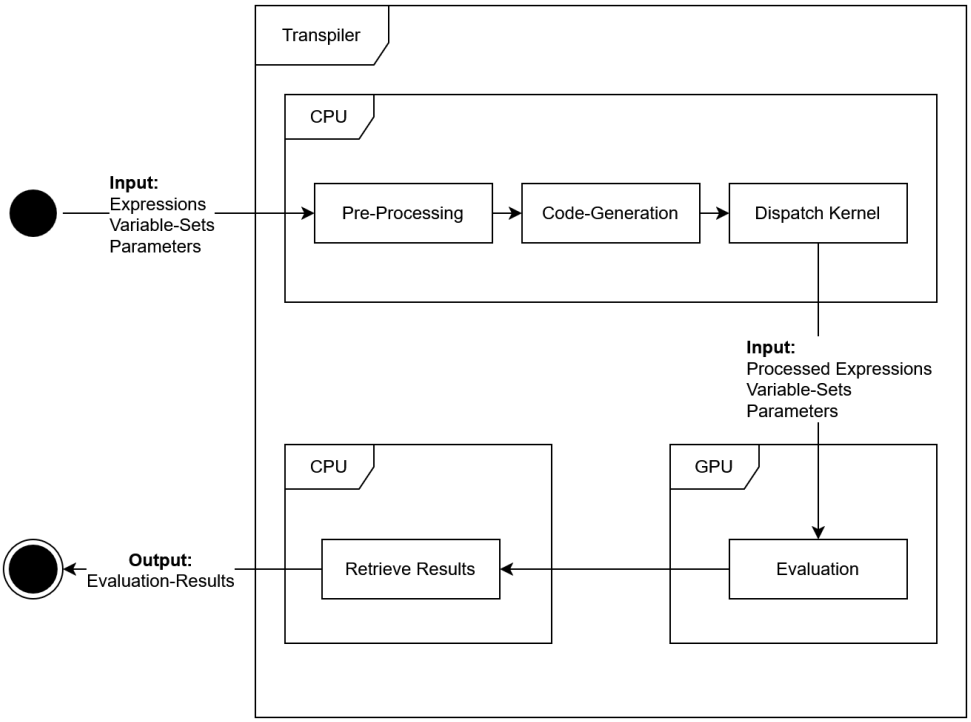


Figure 3.5: This diagram depicts the coarse-grained workflow of the transpiler. It shows how the parts interact with each other and with the system it will operate in.

calculation of the expression and variable set needed to store the result at the correct location.

The Algorithm 3.1 in this case resembles the kernel. This kernel will be dispatched for each expression that needs to be evaluated, to prevent thread divergence. Thread divergence can only occur 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, variables or parameters in the same order, and would therefore cause each thread to take a different path. However, one expression always has the same constants, operators, variables and parameter in the same locations, meaning that all threads will take the same path. This also means that although the interpreter contains many data-dependent branches, these branches only depend on the expression itself. Because of this, all threads will follow the same path and will therefore never diverge from one another.

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. Looking at the components and workflow of the transpiler, as shown in Figure 3.5, 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 kernels generated by the transpiler work very differently to the kernel for the interpreter. The difference between these evaluators will be explained later.

Before the expressions can be transpiled into PTX code, they have 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. By transforming the expressions into postfix notation, the code generation follows a similar pattern to the interpretation already described.

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 largely the same as the while loop of the interpreter. The main difference is in the operator branches, because now code needs to be generated instead of computing the result of the expression. Therefore, the branches themselves call their designated code generation function, such as *GetAddition*. This function returns the PTX code responsible for the addition. However, this function must return more than just the code that performs the addition. When executed, this addition 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 reference to the result.

This reference can then be put on the stack for later use, the same way the interpreter stores the value for later use. The code fragment must also be added to the already generated code 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 has been executed, this value is the reference to the result of the expression. This value then needs to be stored in the result matrix, so that it can be retrieved by the CPU after all expressions have been executed. Therefore, a final code fragment must be generated to handle the storage of this value in the result matrix. This fragment must then be added to the code already generated, and the transpilation process is complete.

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 focus on evaluating the postfix expression from left to right. There is no overhead work such as branching or managing a stack. However, this overhead is now shifted to the transpilation step on the CPU which can be seen in Algorithm 3.2. 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 would be transpiled multiple times as the transpiler is called multiple times with the same expressions.

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 involves branching or managing a stack, and therefore involves many instructions that are not used to evaluate the expression itself. However, this overhead can be mitigated by the fact, that all this work is performed in parallel rather than sequentially.

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 = Variable then
9:              (codeFragment, referenceToValue) ← GetVariable(token.Value)
10:             Push(stack, referenceToValue)
11:             Append(code, codeFragment)
12:         else if token.Type = Operator then
13:             if token.Value = Addition then
14:                 right ← Pop(stack)
15:                 left ← Pop(stack)
16:                 (referenceToValue, codeFragment) ← GetAddition(left, right)
17:                 Push(stack, referenceToValue)
18:                 Append(code, codeFragment)
19:             else if token.Value = Multiplication then
20:                 right ← Pop(stack)
21:                 left ← Pop(stack)
22:                 (referenceToValue, codeFragment) ← GetMultiplication(left, right)
23:                 Push(stack, referenceToValue)
24:                 Append(code, codeFragment)
25:     codeFragment ← GenerateResultStoring(Pop(stack))
26:     Append(code, codeFragment)
        return code
```

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. Furthermore, as explained in Section 2.2.2, any program running on the GPU, must be transpiled into PTX code before the driver can compile it into machine code. Therefore, the kernel written for the interpreter, must also be transpiled into PTX and then be compiled. However, this needs to be done only once, while for the transpiler this needs to be done for each expression. Since the generated code is tailored to evaluate expressions and not to generate generic code, this means the kernels are simpler and can be transpiled and compiled faster. The overhead of transpiling and compiling the expressions is further mitigated by re-using the compiled kernels during the parameter optimisation step.

Chapter 4

Implementation

This chapter focuses on the implementation phase of the thesis, building upon the concepts and designs previously discussed. It begins with an overview of the technologies employed for both the CPU and GPU parts of the prototypes. This is followed by a description of the pre-processing or frontend phase. The chapter concludes with a detailed overview of the core components, the interpreter and the transpiler.

4.1 Technologies

This section describes the technologies used for both the CPU side of the prototypes and the GPU side. The rationale behind these choices, including consideration of their performance implications, is presented. In addition, the hardware limitations imposed by the choice of GPU technology are outlined.

4.1.1 CPU side

Both prototypes were implemented using the Julia programming language. It was chosen mainly, because the current symbolic regression algorithm is also implemented in Julia. Being a high-level programming language, with modern features such as a garbage-collector (GC), support for meta-programming and dynamic typing, it also offers great convenience to the developer.

More interestingly however, is the high performance that can be achieved with this language. It is possible to achieve high performance despite the supported modern features, which are often deemed to be harmful to performance. Bezanson et al. (2017) have shown how Julia can provide C-like performance while supporting the developer with modern quality of life features. The ability of Julia to be used in high performance computing scenarios and to be competitive with C has been demonstrated by W.-C. Lin and McIntosh-Smith (2021). This shows how Julia is a good and valid choice for scenarios where developer comfort and C-like performance are needed.

4.1.2 GPU side

In addition to a programming language for the CPU, a method for programming the GPU is also required. For this purpose, the CUDA API was chosen. While CUDA offers

robust capabilities, it is important to note that it is exclusively compatible with Nvidia GPUs. An alternative would have been OpenCL, which provides broader compatibility by supporting GPUs from Nvidia, AMD and Intel. However, considering Nvidia's significant market share and the widespread adoption of CUDA in the industry, the decision was made to use CUDA.

A typical CUDA program is primarily written C++ and Nvidia also provides their CUDA compiler nvcc[1] for C and C++ and their official CUDA programming guide (Nvidia, 2025b) also uses C++ for code examples. It is also possible to call C++ code from within Julia. This would allow for writing the kernel and interaction with the GPU in C++, leveraging the knowledge built up in the industry over several years.

CUDA and Julia

Instead of writing the kernel in C++ and calling it from Julia, a much simpler and effective alternative is available. The Julia package CUDA.jl[2] enables a developer to write a kernel in Julia similar to how a kernel is written in C++ with CUDA. One drawback of using CUDA.jl however, is the fact that it is much newer compared to CUDA and therefore does not have years of testing and bug fixing in its history, which might be a concern for some applications. Apart from writing kernels with CUDA.jl, it also offers a method for interacting with the driver to compile PTX code into machine code. This is a must-have feature as otherwise, it wouldn't have been possible to fully develop the transpiler in Julia.

Additionally, the JuliaGPU initiative[3] offers a collection of additional packages to enable GPU development for AMD, Intel and Apple and not just for Nvidia. However, CUDA.jl is also the most mature of the available implementations, which is another reason why CUDA has been chosen instead of for example OpenCL.

Again, the question arises as to whether the performance of CUDA.jl is sufficient for it to be used as an alternative to C++ and CUDA. Studies by Besard et al. (2019a), Faingnaert et al. (2022), and W.-C. Lin and McIntosh-Smith (2021) have demonstrated, that CUDA.jl provides sufficient performance. They found that, in some cases, CUDA.jl performed better than the same algorithm implemented in C and C++, and that it is on par otherwise. These results provide the confidence, that Julia alongside CUDA.jl is a good choice for leveraging the performance of GPUs to speed up expression evaluation.

4.2 Pre-Processing

The pre-processing or frontend step is very important. As already explained in Chapter 3, it is responsible for ensuring that the given expressions are valid and that they are transformed into an intermediate representation. This section aims to explain how the intermediate representation is implemented, as well as how it is generated from a mathematical expression.

[1] https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/
[2] https://cuda.juliagpu.org/
[3] https://juliagpu.org/

4.2.1 Intermediate Representation

The intermediate representation is mainly designed to be lightweight and easily transferrable to the GPU. Since the interpreter runs on the GPU, this was a very important consideration. Because the transpilation process is done on the CPU, and is therefore very flexible in terms of the intermediate representation, the focus lied mainly on being efficient for the interpreter.

The intermediate representation cannot take any form. While it has already been defined that expressions are converted to postfix notation, there are several ways to store the data. The first logical choice is to create an array where each entry represents a token. On the CPU it would be possible to define each entry as a pointer to the token object. Each of these objects could be of a different type, for example one object that holds a constant value while another object holds an operator. In addition, each of these objects could contain its own logic about what to do when it is encountered during the evaluation process. However, on the GPU, this is not possible, as an array entry must hold a value and not a pointer to another memory location. Furthermore, even if it were possible, it would not be a feasible solution. As explained in Section 2.2.1, when loading data from global memory, larger chunks are retrieved at once. If the data is scattered across the GPU's global memory, a lot of unwanted data will be transferred. This can be seen in Figure 4.1, where if the data is stored sequentially, far fewer data operations and far less data in general needs to be transferred.

Due to this, and the fact that the GPU does not allow pointers, an alternative approach is required. Rather than storing pointers to objects of different types in an array, it is possible to store objects of a single type. As described in Section 3.2.1, the objects thus contain the type of the stored value and the value itself. The four types of values that need to be stored in this object differ significantly in terms of the value they represent. The following paragraphs explain how these values can be stored in objects of a single type.

Variables and parameters are very simple to store. Because they represent indices to the variable matrix or the parameter vector, this (integer) index can be stored as is in the value property of the object. The type can then be used to determine whether it is an index to a variable or a parameter access.

Constants are also very simple, as they represent a single 32-bit floating point value. However, due to the variables and parameters, the value property is already defined as an integer and not as a floating point number. Unlike in dynamically typed languages such as Python, where every number is a floating point number, in Julia they these have different types and therefore cannot be stored in the same property. Creating a second property for constants only is not feasible, as this would introduce four bytes per object that need to be sent to the GPU, which most of the time does not contain a defined value.

To avoid sending unnecessary bytes, Julia provides a mechanism called `reinterpret` that can be used. This allows the bits of a variable of one type, to be treated as the bits of a different type. For example, the bits used to represent a floating point number are then interpreted as an integer and can be stored in the same property. On the GPU, the same concept can be applied to reinterpret the integer value as a floating point value for further calculations. This is also the reason why the original type of the value needs

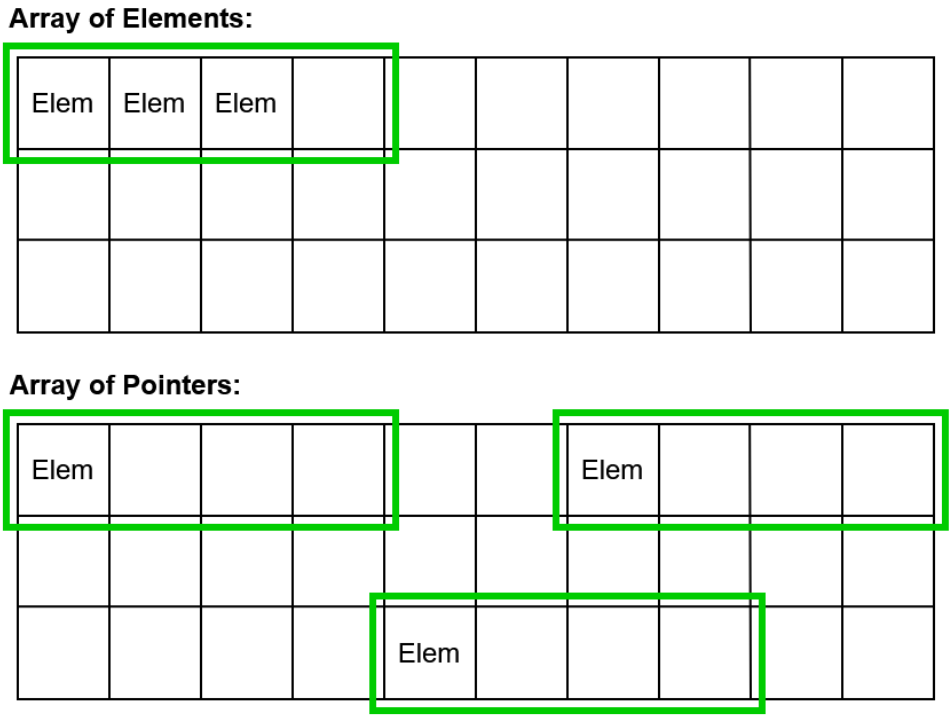


Figure 4.1: Loading data from global memory on the GPU always loads 32, 64 or 128 bytes (see Section 2.2.1). If pointers were supported and data would be scattered around global memory, many more data load operations would be required. Additionally, much more unwanted data would be loaded.

to be stored alongside the value in order for the stored value to be interpreted and the expressions to be evaluated correctly.

Operators are very different from variables, parameters and constants. Because they represent an operation rather than a value, a different way of storing them is required. An operator can be uniquely mapped to a number to identify the operation. For example, if the addition operator is mapped to the integer 1. Consequently, when the evaluator encounters an object of type operator and a value of 1, it can determine the corresponding operation to perform. This can be done for all operators which means it is possible to store them in the same object structure. The type must be specified to be an operator and the value can be stored without needing to reinterpret it. The mapping of an operator to a value is commonly referred to as an operation code, or opcode, ensuring that each operator is uniquely identifiable.

With this, the intermediate representation is defined. Figure 4.2 shows how a simple expression would look after the pre-processing step. Note that the bit representation of the value 2.5 has been reinterpreted as an integer, resulting in the seemingly random value.

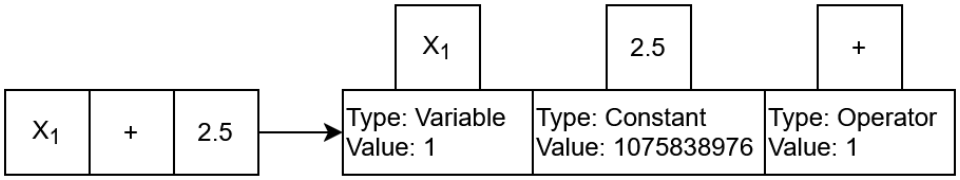


Figure 4.2: The expression $x_1 + 2.5$ after it has been converted to the intermediate representation. Note that the constant value 2.5 stores a seemingly random value due to the bits being reinterpreted as an integer.

4.2.2 Processing

Now that the intermediate representation has been defined, the processing step can be implemented. This section describes the structure of the expressions and how they are processed. It also explains the process of parsing the expressions to ensure their validity and converting them into the intermediate representation.

Expressions

With the pre-processing step, the first modern feature of Julia has been used. As already mentioned, Julia provides extensive support for meta-programming, which is important for this step. Julia represents its own code as a data structure, which allows a developer to manipulate the code at runtime. The code is stored in the so-called `Expr` object as an Abstract Syntax Tree (AST), which is the most minimal tree representation of a given expression. As a result, mathematical expressions can also be represented as such an `Expr` object instead of a simple string. This is a major benefit, because these expressions can then be easily manipulated by the symbolic regression algorithm. Because of this, the pre-processing step requires the expressions to be provided as an `Expr` object instead of a string.

Another major benefit of the expressions being stored in the `Expr` object and therefore as an AST, is the included operator precedence. Because it is a tree where the leaves are the constants, variables or parameters (also called terminal symbols) and the nodes are the operators, the correct result will be calculated when evaluating the tree from bottom to top. As can be seen in Figure 4.3, the expression $1 + x_1 \log(p_1)$, when parsed as an AST, contains the correct operator precedence. First the bottom most subtree $\log(p_1)$ must be evaluated before the multiplication, and after that, the addition can be evaluated.

It should be noted however, that Julia stores the tree as a list of arrays to allow a node to have as many children as necessary. For example the expression $1 + 2 + \cdots + n$ contains only additions, which is a commutative operation, meaning that the order of operations is irrelevant. The AST for this expression would contain the operator at the first position in the array and the values at the following positions. This ensures that the AST is as minimal as possible.

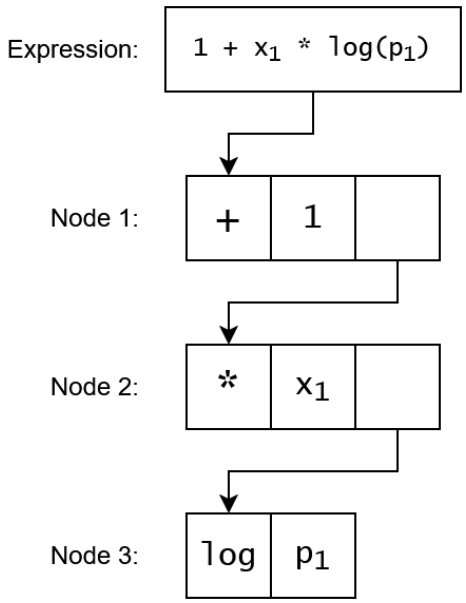


Figure 4.3: The AST for the expression $1 + x_1 \log(p_1)$ as generated by Julia. Some additional details Julia includes in its AST have been omitted as they are not relevant.

Parsing

To convert the AST of an expression into the intermediate representation, a top-down traversal of the tree is required. The steps for this are as follows:

1. Extract the operator and convert it to its opcode for later use.
2. Convert all constants, variables and parameters and operators to the object (expression element) described in Section 4.2.1.
3. Append the expression elements to the postfix expression.
4. If the operator is a binary operator and there are more than two expression elements, append the operator after the first two elements and then after each element.
5. If a subtree exists, apply all previous steps and append it to the existing postfix expression.
6. Append the operator
7. Return the generated postfix expression/intermediate representation.

The validation of the expression is performed throughout the parsing process. Validating that only correct operators are used is performed in step 1. To be able to convert the operator to its corresponding opcode, it must be validated that an opcode exists for it, and therefore whether it is valid or not. Similarly, converting the tokens into an expression element object ensures that only variables and parameters in the correct format are present in the expression. This is handled in step 2.

As explained above, a node of a binary operator can have n children. In these cases, additional handling is required to ensure correct conversion. This handling is

summarised in step 4. Essentially, the operator must be added after the first two elements, for each subsequent element, the operator must also be added. The expression $1+2+3+4$ is converted to the AST $+\,1\,2\,3\,4$ and without step 4 the postfix expression would be $1\,2\,3\,4+$. If the operator is added after the first two elements and then after each subsequent element, the correct postfix expression $1\,2\,+\,3\,+\,4+$ will be generated.

Each subtree of the AST is its own separate AST, which can be converted to postfix notation in the same way the whole AST can be converted. This means that the algorithm only needs to be able to handle leave nodes, and when it encounters a subtree, it recursively calls itself to parse the remaining AST. Step 5 indicates this recursive behaviour.

While the same expression usually occurs only once, sub-expressions can occur multiple times. In the example in Figure 4.3, the whole expression $1 + x_1 \log(p_1)$ is unlikely to be generated more than once by the symbolic regression algorithm. However, the sub-expression $\log(p_1)$ is much more likely to be generated multiple times. This means that the generation of the intermediate representation for this subtree only needs to be done once and can be reused later. Therefore, a cache can be used to store the intermediate representation for this sub-expression and access it again later to eliminate the parsing overhead.

4.3 Interpreter

The implementation of the interpreter is divided into two main components, the CPU-based control logic and the GPU-based interpreter as outlined in the Concept and Design chapter. This section aims to describe the technical details of these components. First the CPU-based control logic will be discussed. This component handles the communication with the GPU and is the entry point which is called by the symbolic regression algorithm. Following this, the GPU-based interpreter will be explored, highlighting the specifics of developing an interpreter on the GPU.

An overview of how these components interact with each other is outlined in Figure 4.4. The parts of this figure are explained in detail in the following sections.

4.3.1 CPU Side

The interpreter is given all the expressions it needs to interpret as an input. Additionally, it needs the variable matrix as well as the parameters for each expression. All expressions are passed to the interpreter as an array of `Expr` objects, as they are needed for the pre-processing step or the frontend. The first loop as shown in Figure 4.4, is responsible for sending the expressions to the frontend to be converted into the intermediate representation. After this step, the expressions are in the correct format to be sent to the GPU and the interpretation process can continue.

Data Transfer

Before the GPU can start with the interpretation, the data needs to be present on it. Because the variables are already in matrix form, transferring the data is fairly straightforward. Memory must be allocated in the global memory of the GPU and then

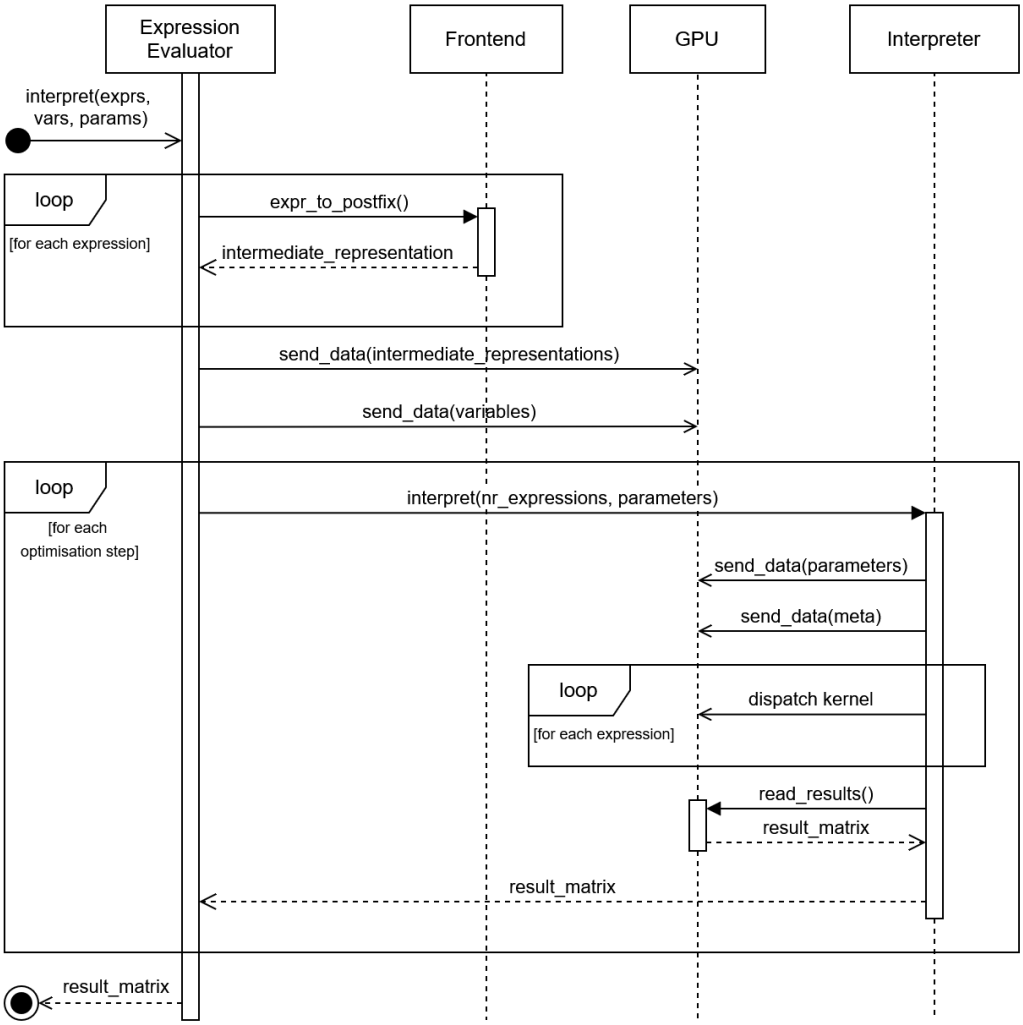


Figure 4.4: The sequence diagram of the interpreter.

be copied from RAM into the allocated memory. Allocating memory and transferring the data to the GPU is handled implicitly by the `CuArray` type provided by CUDA.jl.

To optimise the interpreter for parameter optimisation workloads, this step is performed before it is called. Although, the diagram includes this transmission for completeness, it is important to note that the variables never change, as they represent the observed inputs of the system that is being modelled by the symbolic regression algorithm. As a symbolic regression algorithm is usually implemented with GP, there are many generations that need to be evaluated. Therefore, re-transmitting the variables for each generation is inefficient. By transmitting the variables once before the symbolic regression algorithm begins, additional performance gains are very likely. However, this approach would require modifying the symbolic regression algorithm, which is the reason this optimisation has not been applied. Nonetheless, if needed it is still possible to modify the implementation at a later stage with minimal effort.

Once the variables are transmitted, the parameters must also be transferred to the

```
function convert_to_matrix(vecs::Vector{Vector{T}}, invalidElement::T)::Matrix{T}
    where T
  maxLength = get_max_inner_length(vecs)

  # Pad the shorter vectors with the invalidElement to make all equal length
  paddedVecs = [vcat(vec, fill(invalidElement, maxLength - length(vec))) for vec in
    vecs]
  vecMat = hcat(paddedVecs...) # transform vector of vectors into column-major
    matrix

  return vecMat
end

function get_max_inner_length(vecs::Vector{Vector{T}})::Int where T
  return maximum(length.(vecs))
end

```

Program 4.1: A Julia program fragment depicting the conversion from a vector of vectors into a matrix of the form $k \times N$.

GPU. Unlike the variables, the parameters are stored as a vector of vectors. In order to transmit the parameters efficiently, they also need to be put in a matrix form. The matrix needs to be of the form $k \times N$, where k is equal to the length of the longest inner vector and N is equal to the length of the outer vector. This ensures that all values can be stored in the matrix. It also means that if the inner vectors are of different lengths, some extra unnecessary values will be transmitted, but the overall benefit of treating them as a matrix outweighs this drawback. The Program 4.1 shows how this conversion can be implemented. Note that it is required to provide an invalid element. This ensures defined behaviour and helps with finding errors in the code. After the parameters have been brought into matrix form, they can be transferred to the GPU the same way the variables are transferred.

Similar to the parameters, the expressions are also stored as a vector of vectors. The outer vector contains each expression, while the inner vectors hold the expressions in their intermediate representation. Therefore, this vector of vectors also needs to be brought into matrix form following the same concept as the parameters. To simplify development, the special opcode *stop* has been introduced, which is used for the `invalidElement` in Program 4.1. As seen in Section 4.3.2, this element is used to determine if the end of an expression has been reached during the interpretation process. This removes the need for additional data to be sent which stores the length of each expression to determine if the entire expression has been interpreted or not. Therefore, a lot of overhead can be reduced.

Once the conversion into matrix form has been performed, the expressions are transferred to the GPU. Just like with the variables, the expressions remain the same over the course of the parameter optimisation part. Which is the reason they are transferred to the GPU before the interpreter is called, reducing the number of unnecessary data transfers.

Only raw data can be sent to the GPU, which means that meta information about

Figure 4.5: The expressions, variables and parameters as they are stored in the GPUs global memory. Note that while on the CPU they are stored as matrices, on the GPU, they are only three arrays of data. The thick lines represent, where a new column and therefore a new set of data begins.

the data layout is missing. The matrices are represented as flat arrays, which means they have lost their column and row information. This information must be sent separately to inform the kernel about the dimensions of the expressions, variables and parameters. Otherwise, the kernel does not know at which memory location the second variable set is stored for example, as it does not know how large a single set is. Figure 4.5 shows how the data is stored without any information about the rows or columns of the matrices. The thick lines help to identify where a new column, and therefore a new set of data begins. However, the GPU has no knowledge of this and therefore the meta information must be transferred separately to ensure that the data is accessed correctly.

In addition to the already described data that needs to be sent, one more step is required that has not been included in the Sequence Diagram 4.4. Global memory must be allocated, that allows the results of the evaluation to be stored. Without this, the kernel would not know where to store the interpretation results and the CPU would not know from which memory location to read the results from. Therefore, enough global memory needs to be allocated beforehand so that the results can be stored and retrieved after all kernel executions have finished.

Kernel Dispatch

Once all the data is present on the GPU, the CPU can dispatch the kernel for each expression. This dispatch requires parameters that specify the number of threads and their organisation into thread blocks. In total, one thread is required for each variable set and therefore the grouping into thread blocks is the primary variable. Taking into account the constraints explained in Section 2.2.1, this grouping needs to be tuned for optimal performance. The specific values alongside the methodology for determining these values will be explained in Chapter 5.

In addition, the dispatch parameters also include the pointers to the location of the data allocated and transferred above, as well as the index of the expression to be interpreted. Since all expressions and parameters are sent to the GPU at once, this index ensures that the kernel knows where in memory to find the expression it needs

to interpret and which parameter set it needs to use. After the kernel has finished, the result matrix needs to be read from the GPU and passed back to the symbolic regression algorithm.

Crucially, dispatching a kernel is an asynchronous operation, which means that the CPU does not wait for the kernel to finish before continuing. This allows the CPU to dispatch all kernels at once, rather than one at a time. As explained in Section 3.2, a GPU can have multiple resident grids, meaning that the dispatched kernels can run concurrently, reducing evaluation time. Only once the result matrix is read from the GPU does the CPU have to wait for all kernels to finish execution.

4.3.2 GPU Side

With the GPU's global memory containing all the necessary data and the kernel being dispatched, the interpretation process can begin. Before interpreting an expression, the global thread ID must be calculated. This step is crucial because each variable set is assigned to a unique thread. Therefore, the global thread ID determines which variable set should be used for the current interpretation instance.

Moreover, the global thread ID ensures that excess threads do not perform any work. As otherwise these threads would try to access a variable set that does not exist and therefore would lead to an illegal memory access. This is necessary because the number of required threads often does not align perfectly with the number of threads per block multiplied by the number of blocks. If for example 1031 threads are required, then at least two thread blocks are needed, as one thread block can hold at most 1024 threads. Because 1031 is a prime number, it can not be divided by any practical number of thread blocks. If two thread blocks are allocated, each holding 1024 threads, a total of 2048 threads is started. Therefore, the excess $2048 - 1031 = 1017$ threads must be prevented from executing. By using the global thread ID and the number of available variable sets, these excess threads can be easily identified and terminated early in the kernel execution.

Afterwards the stack for the interpretation can be created. It is possible to dynamically allocate memory on the GPU, which enables a similar programming model as on the CPU. Winter et al. (2021) have compared many dynamic memory managers and found, that the performance impact of them is rather small. However, if it is easily possible to use static allocations, it still offers better performance. In the case of this thesis, it is easily possible which is the reason why the stack has been chosen to have a static size. Because it is known that expressions do not exceed 50 tokens, including the operators, the stack size has been set to ten, which should be more than enough to hold the values and partial results, even in the worst case. It is very unlikely that ten values must be stored before a binary operator is encountered that reduces the number of values on the stack. Therefore, a stack size of ten should be sufficient, however it is possible to increase the stack size if needed.

Main Loop

Once everything is initialised, the main interpreter loop starts interpreting the expression. Because of the intermediate representation, the loop simply iterates through the expression from left to right. On each iteration the type of the current token is checked,

to decide which operation to perform.

If the current token type matches the *stop* opcode, the interpreter knows that it is finished. This simplicity is the reason why this opcode was introduced, as mentioned above.

More interestingly is the case, where the current token corresponds to an index to either the variable matrix, or the parameter matrix. In this case, the token's value is important. To access one of these matrices, the correct starting index of the set must first be calculated. As previously explained, information about the dimensions of the data is lost during transfer. At this stage, the kernel only knows the index of the first element of either matrix, which set to use for this evaluation, and the index of the value within the current set. However, the boundaries of these sets are unknown. Therefore, the additionally transferred data about the dimensions is used in this step to calculate the index of the first element in each set. With this calculated index and the index stored in the token, the correct value can be loaded by adding the token value to the index of the first element of the set. After the value has been loaded, it is pushed to the top of the stack for later use.

Constants work very similarly in that the token value is read and added to the top of the stack. However, the constants have been reinterpreted from floating-point values to integers for easy transfer to the GPU. This operation must be reversed before adding the value to the stack as otherwise the wrong values would be used for evaluation.

Evaluating the expression is happening if the current token is an operator. The token's value, which serves as the opcode, determines the operation that needs to be performed. If the opcode represents a unary operator, only the top value of the stack needs to be popped for the operation. The operation is then executed on this value and the result is pushed back to the stack. On the other hand, if the opcode represents a binary operator, the top two values of the stack are popped. These are then used for the operation, and the result is subsequently pushed back onto the stack.

Support for ternary operators could also be easily added. An example of a ternary operator that would help improve performance would be the GPU supported Fused Multiply-Add (FMA) operator. While this operator does not exist in Julia, the frontend can generate it when it encounters a sub-expression of the form $x * y + z$. Since this expression performs the multiplication and addition in a single clock cycle instead of two, it would be a feasible optimisation. However, detecting such sub-expressions is complicated, which why it is not supported in the current implementation.

Once the interpreter loop has finished, the result of the evaluation must be stored in the result matrix. By using the index of the current expression, as well as the index of the current variable set (the global thread ID) it is possible to calculate the index where the result must be stored. The last value on the stack is the result, which is stored in the result matrix at the calculated location.

4.4 Transpiler

Unlike the interpreter, the transpiler primarily operates on the CPU, with only a minor GPU-based component. This is because the transpiler must generate entire PTX kernels from Julia expressions, rather than simply executing a pre-written kernel like the interpreter. Similar to the interpreter, the CPU side of the transpiler manages communi-

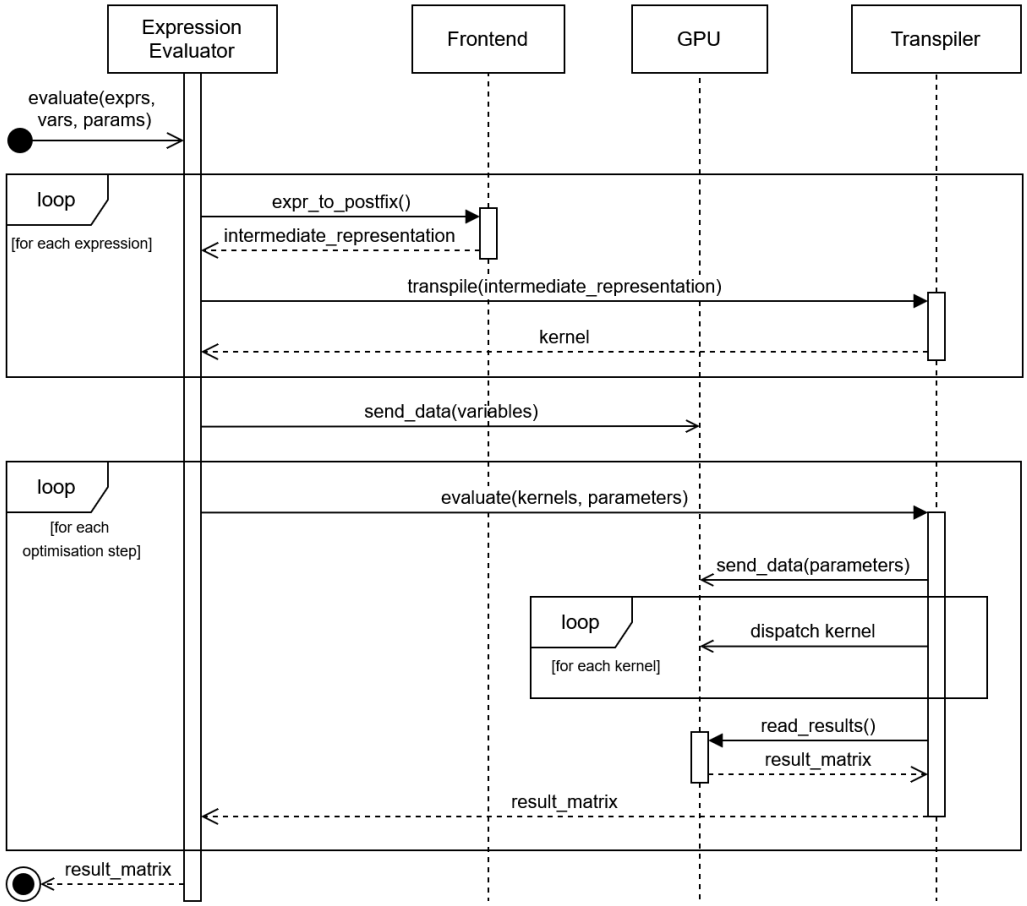


Figure 4.6: The sequence diagram of the transpiler.

cation with both the GPU and the symbolic regression algorithm. This section provides a detailed overview of the transpiler's functionality.

An overview of how the transpiler interacts with the frontend and GPU is outlined in Figure 4.6. The parts of this figure are explained in detail in the following sections.

4.4.1 CPU Side

After the transpiler has received the expressions to be transpiled, it first sends them to the frontend for processing. Once an expression has been processed, it is sent to the transpiler backend which is explained in more detail Section 4.4.2. The backend is responsible for generating the kernels. When finished, each expression is transpiled into its own kernel written in PTX code.

Data Transfer

Data is sent to the GPU in the same way it is sent in the interpreter. The variables are sent as they are, while the parameters are again brought into matrix form. Memory must also be allocated for the result matrix. Unlike the interpreter however, only the

```
1 # Dispatching the interpreter kernel
2 for i in eachindex(exprs)
3   numThreads = ...
4   numBlocks = ...
5
6   @cuda threads=numThreads blocks=numBlocks fastmath=true interpret(cudaExprs,
      cudaVars, cudaParams, cudaResults, cudaAdditional)
7 end
8
9 # Dispatching the transpiled kernels
10 for kernelPTX in kernelsPTX
11   # Create linker object, add the code and compile it
12   linker = CuLink()
13   add_data!(linker, "KernelName", kernelPTX)
14   image = complete(linker)
15
16   # Get callable function from compiled result
17   mod = CuModule(image)
18   kernel = CuFunction(mod, "KernelName")
19
20   numThreads = ...
21   numBlocks = ...
22
23   # Dispatching the kernel
24   cudacall(kernel, (CuPtr{Float32},CuPtr{Float32},CuPtr{Float32}), cudaVars,
      cudaParams, cudaResults; threads=numThreads, blocks=numBlocks)
25 end
```

Program 4.2: A Julia program fragment showing how the transpiled kernels need to be dispatched as compared to the interpreter kernel

variables and parameters need to be sent to the GPU. The variables are again sent before the parameter optimisation step to reduce the number of data transfers.

Because each expression is represented by its own kernel, there is no need to transfer the expressions themselves. Moreover, there is also no need to send information about the layout of the variables and parameters to the GPU. The reason for this is explained in the transpiler backend section below.

Kernel Dispatch

Once all the data is present on the GPU, the transpiled kernels can be dispatched. Dispatching the transpiled kernels is more involved than dispatching the interpreter kernel. Program 4.2 shows the difference between dispatching the interpreter kernel and the transpiled kernels. An important note, is that the transpiled kernels must be manually compiled into machine code. To achieve this, CUDA.jl provides functionality to instruct the driver to compile the PTX code. The same process of creating PTX code and compiling it must also be done for the interpreter kernel, however, this is done by CUDA.jl automatically when calling the @cuda macro in line 6.

Similar to the interpreter, the frontend and backend are executed before the parameter optimisation step to improve the runtime. Each kernel is compiled into machine code

after it has been generated to ensure, as little work as possible needs to be done during the parameter optimisation loop. However, as will be explained in Chapter 5, storing the compiled kernels is very memory intensive. This means that if many expressions need to be evaluated at once, a lot of memory is required.

After all kernels have been dispatched, the CPU waits for the kernels to complete their execution. Once the kernels have finished, the result matrix is read from global memory into system memory. The results can then be returned to the symbolic regression algorithm.

4.4.2 Transpiler Backend

The transpiler backend is responsible for creating a kernel from an expression in its intermediate representation. Transpiling an expression is divided into several parts, these parts are as follows:

- Register management
- Generating the header and kernel entry point
- Ensuring that only the requested amount of threads is performing work
- Generating the Code for evaluating the expression and storing the result

PTX assumes a register machine, which means that a developer has to work with a limited number of registers. This also means that the transpiler has to define a strategy for managing these registers. The second and third parts are rather simple and can be considered as overhead code. Finally, the last part is the main part of the generated kernel. It contains the code to load variables and parameters, evaluate the expression and store the result in the result matrix. All parts are explained in the following sections.

Register Management

Register management is a crucial part of the transpiler as it is important to balance register usage with occupancy and performance. Aho et al. (2006) and Cooper and Torczon (2022) describe techniques for efficient register management, especially for machines with few registers and register usage by convention on the CPU. On the GPU however, there are many more registers available, all of which can be used as needed without restrictions.

To allow for maximum occupancy and avoid spilling registers into local memory, the transpiler tries to reuse as many registers as possible. Furthermore, allocating and using a register in PTX is very similar to using variables in high level code, as they represent virtual registers. Therefore, much of the complexity of managing registers is handled by the PTX compiler of the driver.

Because much of the complexity of managing registers is hidden by the compiler, or does not apply in this scenario, it is implemented very simple. If a register is needed at any point in the transpilation process, it can be requested by the register manager. A register must be given a name and the manager uses this name to determine the type of this register. For example, if the name of the register is `f`, it is assumed to be an FP32 register. Several naming conventions exist to ensure that the register is of the correct data type. The manager then returns the identifying name of the register, which is used

to access it. The identifying name, is the name given as an input and a zero-based number that is incremented by one for each successive call.

PTX requires that the registers are defined before they are used. Therefore, after the transpiler has finished generating the code, the registers must be defined at the top of the kernel. As the manager has kept track of the registers used, it can generate the code to allocate and define the registers. If the kernel only uses five FP32 registers, the manager would generate the code `.reg .f32 %f<5>;`. This will allocate and define the registers `%f0` through `%f4`.

Header and Entry Point

Each PTX program must begin with certain directives in order to compile and use that program correctly. The first directive must be the `.version` directive. It indicates which PTX version the code was written for, to ensure that it is compiled with the correct tools in the correct version. Following the `.version` directive is the `.target` directive, which specifies the target hardware architecture.

Once these directives have been added to the generated code, the entry point to the kernel can be generated. It contains the name of the kernel, as well as all parameters that are passed to it, such as the pointers to the variable, parameter and result matrix. The kernel name is important as it is required by the CPU to dispatch it.

When the entry point is generated, the PTX code for loading the parameters into the kernel is also generated. This removes the need to iterate over the kernel parameters a second time. Loading the parameters into the kernel is necessary because it is not possible to address these values directly. Nvidia (2025d) states that addresses in the parameter state space can only be accessed using the `ld.param` instruction. Furthermore, since all three matrices are stored in global memory, the parameter address must be converted from parameter state space to global state space using the `cvta.to.global.datatype` instruction.

Guard Clause

As explained in Section 4.3.2, the guard clause ensures that any excess threads do not participate in the evaluation. The following code shows what this guard clause looks like when the kernel is written with Julia and CUDA.jl:

```
1 function my_kernel(nrOfVarSets::Int32)
2     threadId = (blockIdx().x - 1) * blockDim().x + threadIdx().x
3     if threadId > nrOfVarSets
4         return
5     end
6     # remaining kernel
7 end
```

This can be translated into the following PTX code fragment:

```
1   mov.u32         %r3, %ntid.x;   // r3 = blockIdx().x - 1
2   mov.u32         %r4, %ctaid.x;  // r4 = blockDim().x
3   mov.u32         %r5, %tid.x;    // r5 = threadIdx().x
4
5   mad.lo.s32      %r1, %r3, %r4, %r5; //r1 = r3 * r4 + r5
6   setp.ge.s32     %p1, %r1, %r2;  // p1 = r1 >= r2 (r2 = nrOfVarSets)
```

```
 7   @%p1 bra        End;
 8
 9   // remaining Kernel
10
11 End:
12   ret;
```

It needs to be noted, that the register `%r2` is not needed. Since the transpiler already knows the number of variable sets, it would be wasteful to transmit this information to the kernel. Instead, the transpiler inserts the number directly as a constant to save resources.

Main Loop

The main loop of the transpiler, which generates the kernel for evaluating a single expression, is analogous to the interpreter's main loop. Since the transpiler uses the same intermediate representation as the interpreter, both loops behave similarly. The transpiler loop also uses a stack to store the values and intermediate results. However, the transpiler does not require the special opcode *stop* which was necessary in the interpreter to handle expressions padded to fit into a matrix. The transpiler only needs to process a single expression, which is stored in an unpadded vector of known length. This means that all tokens within the vector are valid and therefore do not require this opcode.

When the loop encounters a token that represents an index to either the variable or the parameter matrix, the transpiler needs to generate code to load these values. In the general case, this works in exactly the same way as the interpreter, calculating the index and accessing the matrices at that location.

However, the first time a variable or parameter is accessed, it must be loaded from global memory. Although registers already exist that hold a pointer to the address of the matrices in global memory, the data is still not accessible. To make it accessible, the index to the value must first be calculated in the same way as it is calculated in the interpreter. Afterwards the value must be loaded into a register with the instruction `ld.global.f32 %reg1, [%reg2]`. Using the first register of the instruction, the data can be accessed. For example, if the variable x_1 is accessed several times, all subsequent calls only need to reference this register and do not need to load the data from global memory again.

In the case where the current token represents an operation, the code for this operation needs to be generated. Many operators have direct equivalents on the GPU. For example addition has the `add.f32 %reg1, %reg2, %reg3;` instruction. The instructions for division and square root operations have equivalent instruction, but these only support approximate calculations. Although the accuracy can be controlled with different options, the fastest option `.approx` has been selected. While a slightly slower but more accurate option `.full` exists, it is not fully IEEE 754 compliant and has therefore not been used.

However, not all supported operators have a single instruction GPU equivalent. For example, the x^y operation does not have an equivalent and must be generated differently. Compiling a kernel containing this operation using the Nvidia compiler and the *--use_fast_math* compiler flag will generate the following code:

```
lg2.approx.f32   %reg1, %reg2;
mul.f32          %reg4, %reg3, %reg1;
ex2.approx.f32   %reg5, %reg4;
```

While this compiler flag trades accuracy for performance, the more accurate version of this operation contains about 100 instructions instead of the three above. Therefore, the more performant version was chosen to be generated by the transpiler. Similarly, the operations $\log(x)$ and e^x have no equivalent instruction and are therefore generated using the same principle.

The final register of the generated code stores the result of the operation once it has been executed. As with the interpreter, this result is either the final value or an input to another operation. Therefore, this register must be stored on the stack for later use.

Once the main loop has finished, the last element on the stack holds the register with the result of the evaluation. The value of this register must be stored in the result matrix. As the result matrix is stored in global memory, the code for storing the data is similar to the code responsible for loading the data from global memory. First, the location where the result is to be stored must be calculated. Storing the result at this location is performed with the instruction `st.global.f32  [%reg1], %reg2;`.

4.4.3 GPU Side

On the GPU, the transpiled kernels are executed. Given that these kernels are relatively simple, containing minimal branching and overhead, the GPU does not need to perform a lot of operations. As illustrated in Program 4.3, the kernel for the expression $x_1 + p_1$ is quite straightforward. It involves only two load operations, the addition and the storing of the result in the result matrix. Essentially, the kernel mirrors the expression directly, with the already explained added overhead.

Note that Program 4.3 has been slightly simplified to omit the mandatory directives and the register allocation. From line five to line ten, the addresses stored in the parameters are converted from parameter state space into global state space so that they reference the correct portion of the GPU's memory. It needs to be noted, that this kernel uses 64-bit addresses, which is the reason why some 64-bit instructions are used throughout the kernel. However, the evaluation of the expression itself is performed entirely using the faster 32-bit instructions.

Lines 12 through 17 are responsible for calculating the global thread ID and ensuring that excessive threads are terminated early. Note that in line 16, if the global thread ID stored in register `%r3` is greater than one, it must terminate early. This is because only one variable set needs to be evaluated in this example.

The PTX code from line 22 to line 28 is the actual evaluation of the expression, with line 28 performing the calculation $x_1 + p_1$. All other lines are responsible for loading the values from global memory. The instructions in lines 22, 23, 25 and 26 are responsible for calculating the offset in bytes to the memory location where the value is stored with respect to the location of the first element.

The constants 4 and 0 are introduced for performance reasons. The number 4 is the size of a variable set in bytes. Since one variable set in this case stores only a single FP32 value, each variable set has a size of four bytes. Similarly, the number 0 represents the index of the value within the variable set. More precisely, this is the offset in bytes from

```
1  .visible .entry Evaluator(
2    .param .u64 param_1, .param .u64 param_2, .param .u64 param_3)
3    {
4      // Make parameters stored in global memory accessible
5      ld.param.u64   %rd0, [param_1];
6      cvta.to.global.u64   %parameter0, %rd0;
7      ld.param.u64   %rd1, [param_2];
8      cvta.to.global.u64   %parameter1, %rd1;
9      ld.param.u64   %rd2, [param_3];
10     cvta.to.global.u64   %parameter2, %rd2;
11
12     mov.u32    %r0, %ntid.x;
13     mov.u32    %r1, %ctaid.x;
14     mov.u32    %r2, %tid.x;
15     mad.lo.s32     %r3, %r0, %r1, %r2;
16     setp.gt.s32    %p0, %r3, 1;
17   @%p0 bra    L__BB0_2; // Jump to end of kernel if too many threads are started
18     cvt.u64.u32    %rd3, %r3;
19     mov.u64 %rd4, 0;
20
21     // Load variable and parameter from global memory and add them together
22     mad.lo.u64  %rd5, %rd3, 4, 0;
23     add.u64     %rd5, %parameter0, %rd5;
24     ld.global.f32 %var0, [%rd5];
25     mad.lo.u64  %rd6, %rd4, 4, 0;
26     add.u64     %rd6, %parameter1, %rd6;
27     ld.global.f32 %var1, [%rd6];
28     add.f32    %f0, %var0, %var1;
29
30     // Store the result in the result matrix
31     add.u64        %rd7, 0, %rd3;
32     mad.lo.u64     %rd7, %rd7, 4, %parameter2;
33     st.global.f32  [%rd7], %f0;
34
35   L__BB0_2: ret;
36   }
```

Program 4.3: The slightly simplified PTX kernel for the expression $x_1 + p_1$. For simplicity, the allocation of registers and the required directives `.version` and `.target` have been removed.

the index to the variable set, which is zero for the first element, four for the second, and so on. These two constants are calculated during the transpilation process to minimise the amount of data to be transferred to the GPU.

Storing the result in the result matrix is performed from line 31 to 33. The location where the value is to be stored is calculated in lines 31 and 32. Line 31 calculates the index inside the result matrix according to the current variable set stored in register `%rd3`. The constant 0 is the product of the index of the expression being evaluated and the number of variable sets, and represents the column of the result matrix. Converting this index into bytes and adding it as an offset to the first element of the result matrix gives the correct memory location to store the result at.

This kernel consists mostly of overhead code, as only lines 22 through 33 contribute to calculating the result of the expression with the designated variable and parameter set. However, for larger expressions, the percentage of overhead code shrinks drastically.

Chapter 5

Evaluation

This thesis aims to determine whether one of the two GPU evaluators is faster than the current CPU evaluator. This chapter describes the performance evaluation process. First, the environment in which the performance benchmarks are conducted is explained. Next the individual results for the GPU interpreter and transpiler are presented individually alongside the performance tuning process to achieve these results. Finally, the results of the GPU evaluators are compared to those of the CPU evaluator to answer the research questions of this thesis.

5.1 Benchmark Environment

In this section, the benchmark environment used to evaluate the performance is outlined. To ensure the validity and reliability of the results, it is necessary to specify the details of the environment. This includes a description of the hardware and software configuration as well as the performance evaluation process. With this, the variance between the results is minimised, which allows for better reproducibility and comparability between the implementations.

5.1.1 Hardware Configuration

The hardware configuration is the most important aspect of the benchmark environment. The capabilities of both the CPU and GPU can have a significant impact on the resulting performance. The following sections outline the importance of the individual components as well as the hardware used for the benchmarks and the performance tuning.

GPU

The GPU plays a crucial role, as different microarchitectures typically operate differently and therefore require different performance tuning. Although the evaluators can generally operate on any Nvidia GPU with a compute capability of at least 6.1, they are tuned for the Ampere microarchitecture which has a compute capability of 8.6. Despite the evaluators being tuned for this microarchitecture, more recent microarchitectures

can be used as well. However, additional tuning is required to ensure that the evaluators can utilise the hardware to its fullest potential.

Tuning must also be done on a per-problem basis. In particular, the number of variable sets impact how well the hardware is utilised. Therefore, it is crucial to determine which configuration yields the best performance. Section 5.2 outlines steps to tune the configuration for a specific problem.

CPU

Although the GPU plays a crucial role, work is also carried out on the CPU. The interpreter primarily utilises the CPU for the frontend and data transfer, making it more GPU-bound as most of the work is performed on the GPU. However, the transpiler additionally relies on the CPU to perform the transpilation step. This step involves generating a kernel for each expression and sending these kernels to the driver for compilation, a process also handled by the CPU. By contrast, the interpreter only required one kernel which needs to be converted into PTX and compiled by the driver only once. Consequently, the transpiler is significantly more CPU-bound and variations in the CPU used have a much greater impact. Therefore, using a more powerful CPU benefits the transpiler more than the interpreter.

System Memory

In addition to the hardware configuration of the GPU and CPU, system memory (RAM) also plays a crucial role. Although RAM does not directly contribute to the overall performance, it can have a noticeable indirect impact due to its role in caching and general data storage. Insufficient RAM forces the operating system to use the page file, which is stored on a considerably slower SSD. This leads to slower data access, thereby reducing the overall performance of the application.

As seen in the list below, only 16 GB of RAM were available during the benchmarking process. This amount is insufficient to utilise caching to the extent outlined in Chapter 4. Additional RAM was not available, meaning caching had to be disabled for all benchmarks as further explained in Section 5.2.

Hardware

With the requirements explained above in mind, the following hardware is used to perform the benchmarks for the CPU-based evaluator, as well as for the GPU-based evaluators:

- Intel i5 12500
- Nvidia RTX 3060 Ti
- 16 GB 4400 MT/s DDR5 RAM

5.1.2 Software Configuration

Apart from the hardware, the performance of the evaluators can also be significantly affected by the software. Primarily these three software components or libraries are involved in the performance of the evaluators:

- GPU Driver
- Julia
- CUDA.jl

Typically, newer versions of these components include, among other things, performance improvements. This is why it is important to specify the version which is used for benchmarking. The GPU driver has version *561.17*, Julia has version *1.11.5*, and CUDA.jl has version *5.8.1*. As with the hardware configuration, this ensures that the results are reproducible and comparable to each other.

5.1.3 Performance Evaluation Process

Now that the hardware and software configurations have been established, the benchmarking process can be defined. This process is designed to simulate the load and scenario in which these evaluators will be used. The Nikuradse dataset (Nikuradse, 1950) has been chosen as the data source. The dataset models the laws of flow in rough pipes and provides 362 variable sets, each set containing two variables. This dataset has first been used by Guimerà et al. (2020) to benchmark a symbolic regression algorithm.

Since only the evaluators are benchmarked, the expressions to be evaluated must already exist. These expressions are generated for the Nikuradse dataset using the exhaustive symbolic regression algorithm proposed by Bartlett et al. (2024). This ensures that the expressions are representative of what needs to be evaluated in a real-world application. In total, three benchmarks will be conducted, each having a different goal, which will be further explained in the following paragraphs.

The first benchmark involves a very large set of roughly 250 000 expressions with 362 variable sets. This means that when using GP all 250 000 expressions would be evaluated in a single generation. In a typical generation, significantly fewer expressions would be evaluated. However, this benchmark is designed to show how the evaluators can handle very large volumes of data. Because of memory constraints, it was not possible to conduct an additional benchmark with a higher number of variable sets.

Both the second and third benchmarks are conducted to demonstrate how the evaluators will perform in more realistic scenarios. For the second benchmark the number of expressions has been reduced to roughly 10 000, and the number of variable sets is again 362. The number of expressions is much more representative to a typical scenario, while the number of variable sets is still low. To determine if the GPU evaluators are a feasible alternative in scenarios with a realistic number of expressions but comparably few variable sets, this benchmark is conducted nonetheless.

Finally, a third benchmark will be conducted. Similar to the second benchmark, this benchmark evaluates the same roughly 10 000 expressions but now with 30 times more variable sets, which equates to roughly 10 000. This benchmark mimics the scenario where the evaluators will most likely be used. While the others simulate different conditions to determine if and where the GPU evaluators can be used efficiently, this benchmark is more focused on determining if the GPU evaluators are suitable for the specific scenario they are likely going to be used in.

All three benchmarks also simulate a parameter optimisation step, as this is the intended use-case for these evaluators. For parameter optimisation, 100 steps are used, meaning that all expressions are evaluated 100 times. During the benchmark, this pro-

cess is simulated by re-transmitting the parameters instead of generating new ones. Generating new parameters is not part of the evaluators and is therefore not implemented. However, because the parameters are re-transmitted each time, the overhead of sending the data is taken into account. This overhead is part of the evaluators and represents an additional burden that the CPU implementation does not have, making it important to be measured.

Measuring Performance

The performance measurements are taken, using the BenchmarkTools.jl[1] package. It is the standard for benchmarking applications in Julia, which makes it an obvious choice for measuring the performance of the evaluators.

It offers extensive support for measuring and comparing results of different implementations and versions of the same implementation. Benchmark groups allow to categorise the different implementations, take performance measurements and compare them. When taking performance measurements, it also supports setting a timeout and most importantly, set the number of samples to be taken. This is especially important, as it ensures to produce stable results by combating run-to-run variance. For this thesis, a sample size of 50 has been used. This means that each of the previously-mentioned benchmarks, gets executed 50 times.

5.2 Results

This section presents the results of the benchmarks described above. First the results for the GPU-based interpreter will be presented alongside the performance tuning process. This is followed by the results of the transpiler as well as the performance tuning process. Finally, both GPU-based evaluators will be compared with each other to determine which of them performs the best. Additionally, these evaluators will be compared against the CPU-based interpreter to answer the research questions of this thesis.

5.2.1 Interpreter

In this section, the results for the GPU-based interpreter are presented in detail. Following the benchmark results, the process of tuning the interpreter is described as well as how to adapt the tuning for the different benchmarks. This part not only contains the tuning of the GPU, but also performance improvements done on the CPU side.

Benchmark 1

The first benchmark consists of 250 000 expressions and 362 variable sets with 100 parameter optimisation steps. Because each expression needs to be evaluated with each variable set for each parameter optimisation step, a total of $250\,000 * 362 * 100 \approx 9.05\,billion$ evaluations have been performed per sample. In Figure 5.1 the result over all 50 samples is presented. The median value across all samples is 466.3 seconds with a standard deviation of 14.2 seconds.

[1] https://juliaci.github.io/BenchmarkTools.jl/stable/

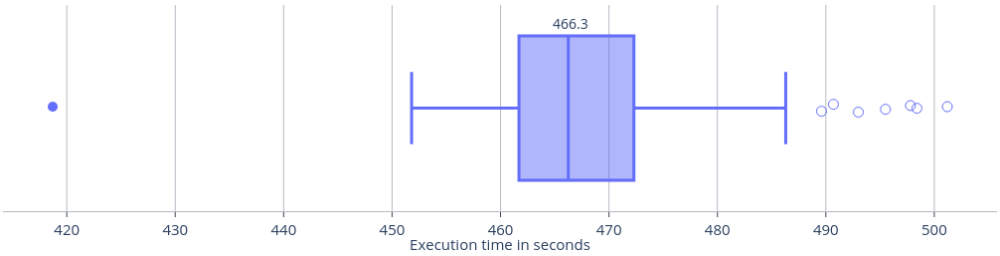


Figure 5.1: The results of the GPU-based interpreter for benchmark 1

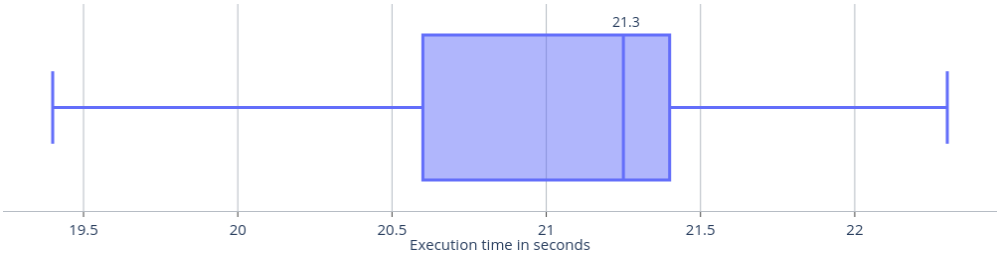


Figure 5.2: The results of the GPU-based interpreter for benchmark 2

For the kernel configuration, a block size of 128 threads has been used. As will be explained below, this has been found to be the configuration that results in the most performance. During the benchmark, the utilisation of both the CPU and GPU was roughly 100%.

Benchmark 2

With 10 000 expressions, 362 variable sets and 100 parameter optimisation steps, the total number of evaluations per sample was 362 *million*. The median across all samples is 21.3 seconds with a standard deviation of 0.75 seconds. Compared to the first benchmark, there were 25 times fewer evaluations which also resulted in a reduction of the median and standard deviation of roughly 25 times. This indicates a roughly linear correlation between the number of expressions and the runtime. Since the number of variable sets did not change, the block size for this benchmark remained at 128 threads. Again the utilisation of the CPU and GPU during the benchmark was roughly 100%.

Benchmark 3

The third benchmark used the same 10 000 expressions and 100 parameter optimisation steps. However, now there are 30 times more variable sets that need to be used for evaluation. This means, that the total number of evaluations per sample is now 10.86 *billion*. Compared to the first benchmark, an additional 1.8 *billion* evaluations were performed. However, as seen in Figure 5.3, the execution time was significantly faster. With a median of 30.3 seconds and a standard deviation of 0.45 seconds, this benchmark was only marginally slower than the second benchmark. This also indicates, that the GPU evaluators are much more suited for scenarios, where there is a high number of variable sets.

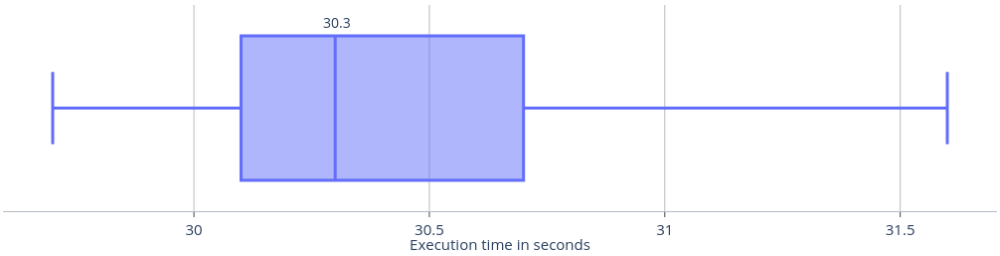


Figure 5.3: The results of the GPU-based interpreter for benchmark 3

Although the number of variable sets has been increased by 30 times, the block size remained at 128 threads. Unlike the previous benchmarks, the hardware utilisation was different. Now only the GPU was utilised to 100% while the CPU utilisation started at 100% and slowly dropped to 80%. The GPU needs to perform 30 times more evaluations per expression, meaning it takes longer for one kernel dispatch to be finished. At the same time, the CPU tries to dispatch the kernel at the same rate as before. Because only a certain number of kernels can be dispatched at once, the CPU needs to wait for the GPU to finish a kernel before another one can be dispatched. Therefore, in this scenario, the evaluator runs into a GPU-bottleneck and using a more performant GPU would consequently improve the runtime. In the previous benchmarks, both the CPU and GPU would need to be upgraded, to achieve better performance.

5.2.2 Performance Tuning Interpreter

Optimising and tuning the interpreter is crucial to achieve good performance. Especially tuning the kernel, as a wrongly configured kernel can drastically degrade performance. Before any performance tuning and optimisation has been performed, the kernel was configured with a block size of 256 threads since it is a good initial configuration as recommended by Nvidia (2025a). Additionally, on the CPU, the frontend was executed for each expression before every kernel dispatch, even in parameter optimisation scenarios, where the expressions did not change from one dispatch to the other. Moreover, the variables have also been transmitted to the GPU before ever dispatch. However, executing the frontend, as well as dispatching the kernel was multithreaded, utilising all 12 threads of the CPU and a cache for the frontend was utilised.

With this implementation, the initial performance measurements have been conducted for the first benchmark which served as the baseline for further performance optimisations. However, as already mentioned, during this benchmark, memory limitations where encountered, as too much RAM was being used. Therefore, the caching had to be disabled. Because the evaluator is multithreaded, this change resulted in significantly better performance. As the cache introduced critical sections where race conditions could occur, locking mechanisms were required. While locking ensures that no race conditions occur, it also means that parts of an otherwise entirely parallel implementation are now serialised, reducing the effect of parallelisation.

Without a cache and utilising all 12 threads, the frontend achieved very good performance. Processing 250 000 expressions takes roughly 88.5 milliseconds. On the other

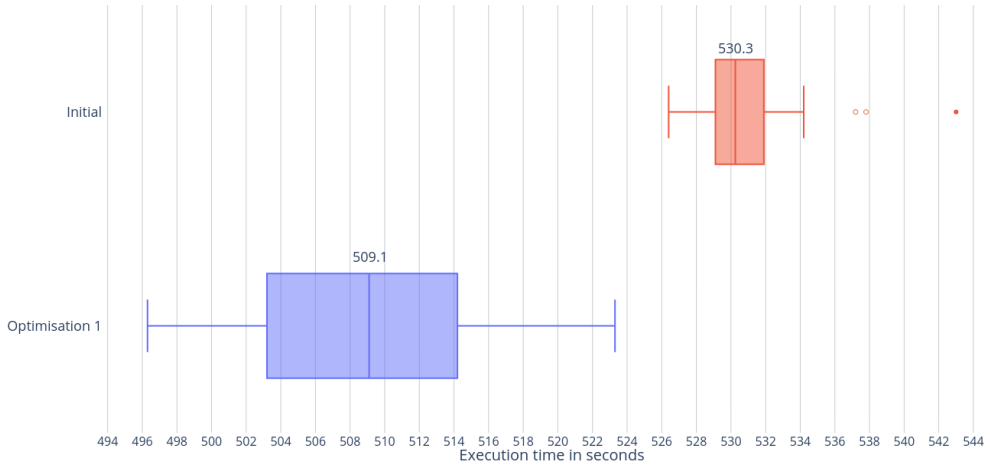


Figure 5.4: Comparison of the initial implementation with the first optimisation applied on benchmark one. Note that while the results of the optimisation fall within a much wider range, all samples performed better than the initial implementation.

hand, using a cache, resulted in the frontend running for 6.9 *seconds*. This equates to a speed-up of roughly 78 times when using no cache. Additionally, when looking at the benchmark results above, the time it takes to execute the frontend is negligible, meaning further optimising the frontend would not significantly improve the overall runtime.

During the tuning process 362 variable sets have been used, which is the number of variable sets used by benchmark one and two. Before conducting benchmark three, additional performance tuning has been performed to ensure that this benchmark also utilises the hardware as much as possible.

Optimisation 1

After caching has been disabled, the first performance improvement was to drastically reduce the number of calls to the frontend and the number of data transfers to the GPU. Because the expressions and variables never change during the parameter optimisation process, processing the expression and transmitting the data to the GPU on each step wastes resources. Therefore, the expressions are sent to the frontend once before the parameter optimisation process. Afterwards, the processed expressions as well as the variables are transferred to the GPU exactly once for this execution of the interpreter.

Figure 5.4 shows how this optimisation improved the overall performance as demonstrated with benchmark one. However, it can also be seen that the range the individual samples fall within is much greater now. While in all cases, this optimisation improved the performance, in some cases the difference between the initial and the optimised version is very low with roughly a two-second improvement.

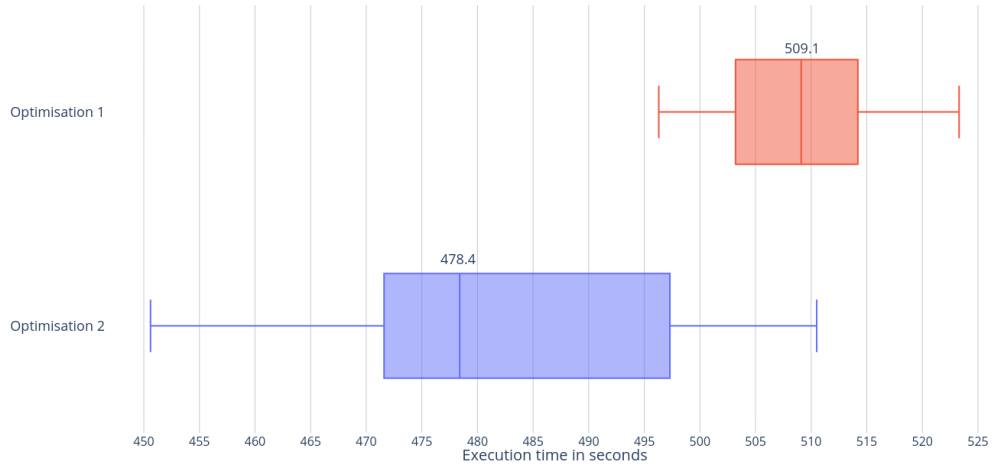


Figure 5.5: Comparison of the first optimisation with the second applied on benchmark one.

Optimisation 2

The second optimisation was concerned with tuning the kernel configuration. Using NSight Compute[2] it was possible to profile the kernel with different configurations. During the profiling a lot of metrics have been gathered that allowed to deeply analyse the kernel executions, with the application recommending different aspects that had potential for performance improvements.

Since the evaluator is designed to execute many kernel dispatches in parallel, it was important to reduce the kernel runtime. Reducing the runtime per kernel has a knock-on effect, as the following kernel dispatches can begin execution sooner reducing the overall runtime.

After the evaluator tuning has been concluded, it was found that a block size of 128 yielded the best results. With this kernel configuration, another performance measurement has been conducted with the results shown in Figure 5.5 using benchmark one. As can be seen, the overall runtime again was noticeably faster. However, the standard deviation also drastically increased, with the duration from the fastest to the slowest sample differing by roughly 60 seconds.

The found block size of 128 might seem strange. However, it makes sense, as in total at least 362 threads need to be started to evaluate one expression. If one block contains 128 threads a total of $362/128 \approx 3$ blocks need to be started, totalling 384 threads. As a result, only $384 - 362 = 22$ threads are excess threads. When choosing a block size of 121 three blocks could be started, totalling one excess thread. However, there is no performance difference between a block size of 121 and 128. Since all threads are executed inside a warp, which consists of exactly 32 threads, a block size that is not divisible by 32 has no benefit and only hides the true amount of excess threads started.

Benchmark three had a total of 10 860 variable sets, meaning at least this number of threads must be started. To ensure optimal hardware utilisation, the evaluator had to undergo another tuning process. As seen above, it is beneficial to start as little excess

[2]https://developer.nvidia.com/nsight-compute

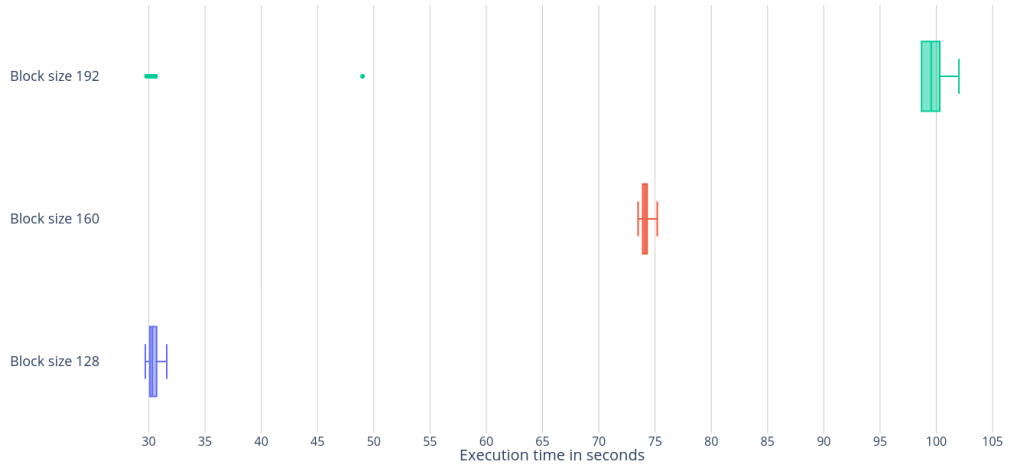


Figure 5.6: Comparison of the execution times of benchmark three with a block size of 128, 160 and 192.

threads as possible. By utilising NSight Compute, a performance measurement with a block size of 128 was used as the initial configuration. This already performed well as again very little excess threads are started. In total $10\,860/128 \approx 84.84$ blocks are needed, which must be round up to 85 blocks with the last block being filled by roughly 84% which equates to 20 excess threads being started.

This was repeated for two more configurations. Once for a block size of 160 and once for 192. With a block size of 160, the total number of blocks was reduced to 68, which again resulted in 20 excess threads being started. With the hypothesis behind increasing the block size was that using fewer blocks would result in better utilisation and therefore better performance. The same idea was also behind choosing a block size 192. However, While this only required 57 blocks, the number of excess threads increased to 84.

Using NSight Compute it was found, that a block size of 160 was the best performing followed by the block size of 192 and the worst performing configuration was with a block size of 128. However, this is not representative of how these configurations performed during the benchmarks. As seen in Figure 5.6 using a block size of 128 lead to significantly better performance than the other configurations. While a block size of 160 lead to worse results, it needs to be noted that it also improved the standard deviation by 25% when compared to the results with a block size of 128. These results also demonstrate that it is important to not only use NSight Compute but also conduct performance tests with real data to ensure the best possible configuration is chosen.

Optimisation 3

As seen in Figure 5.5, while the performance overall improved, the standard deviation also significantly increased. With the third optimisation the goal was to reduce the standard deviation. In order to achieve this, some minor optimisations where applied.

The first optimisation was to reduce the stack size of the interpreter from 25 to 10. As the stack is stored in local memory, it is beneficial to minimise the data transfer and allocation of memory. This change, however, means that the stack might not be

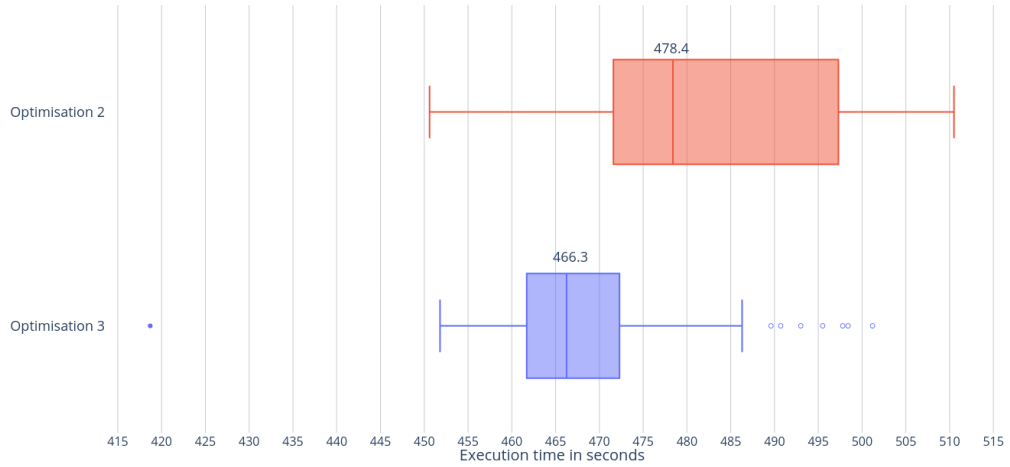


Figure 5.7: Comparison of the second optimisation with the third applied on benchmark one.

sufficient for larger expressions. Because with a stack size of 10 no problems were found during testing, it was assumed to be sufficient. In cases where this isn't sufficient, the stack size can be increased.

During the parameter optimisation step a lot of memory operations where performed. These are required as for each step new memory on the GPU must be allocated for both the parameters and the meta information. The documentation of CUDA.jl[3] mentioned that this can lead to higher garbage-collector (GC) pressure, increasing the time spent garbage-collecting. To reduce this, CUDA.jl provides the `CUDA.unsafe_free!(::CuArray)` function. This frees the memory on the GPU without requiring to run the Julia GC and therefore spending less resources on garbage-collecting and more on evaluating the expressions.

With these two changes the overall runtime has been improved as can be seen in Figure 5.7. Moreover, the standard deviation was also reduced which was the main goal of this optimisation.

5.2.3 Transpiler

In this section the results for the transpiler are presented in detail. First the results for all three benchmarks are shown. The benchmarks are the same as already explained in the previous sections. After the results, an overview of the steps taken to optimise the transpiler execution times is given.

Benchmark 1

This benchmark lead to very poor results for the transpiler. While the best performing kernel configuration of 128 threads per block was used, the above-mentioned RAM constraints meant that this benchmark performed poorly. After roughly 20 hours of

[3] https://cuda.juliagpu.org/stable/usage/memory/#Avoiding-GC-pressure

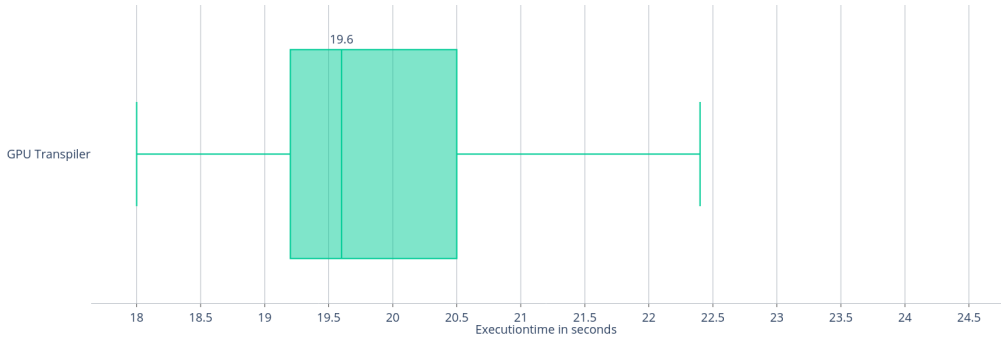


Figure 5.8: The results of the transpiler for benchmark 2.

execution only two samples have been taken at which point it was decided to not finish this benchmark and treat it as failed.

As described in Chapter 4 the expressions are transpiled into PTX code and then immediately compiled into machine code by the GPU driver before the compiled kernels are sent to the parameter optimisation step. This order of operations makes sense as the expressions remain the same during this process and otherwise would result in performing a lot of unnecessary transpilations and compilations.

However, only 16 GB of RAM where available with about half of that being used by the operating system. This meant that about eight GB of RAM where available to store 250 000 compiled kernels next to other required data for example the variable matrix. As a result, this was not enough memory and the benchmark failed. To combat this the step of compiling the kernels was moved into the parameter optimisation process, as this would free the memory taken up by the compiled kernel after it has been executed. As seen above consequently the performance was hurt dramatically and has shown that for these scenarios much more memory is required for the transpiler to work properly.

Benchmark 2

By reducing the number of expressions from 250 000 to roughly 10 000 the RAM constraint that hindered the first benchmark is not a concern any more. This can also be seen in Figure 5.8 where the benchmark could be completed in a much more reasonable time. The median of this benchmark was 19.6 seconds with a standard deviation of 1.16 seconds. Again for this benchmark a block size of 128 threads has been chosen.

During the benchmark it was observed that the CPU maintained a utilisation of 100%. However crucially the GPU rapidly oscillated between 0% and 100% utilisation. This pattern suggests that while the kernels can fully utilise the GPU, they complete the evaluations almost immediately. Consequently, although the evaluation is performed very quickly, the time spent evaluating is smaller than the time spent preparing the expressions for evaluation. To better leverage the GPU, more evaluations should be performed. This would increase the GPU's share of total execution time and therefore increase the efficiency drastically.

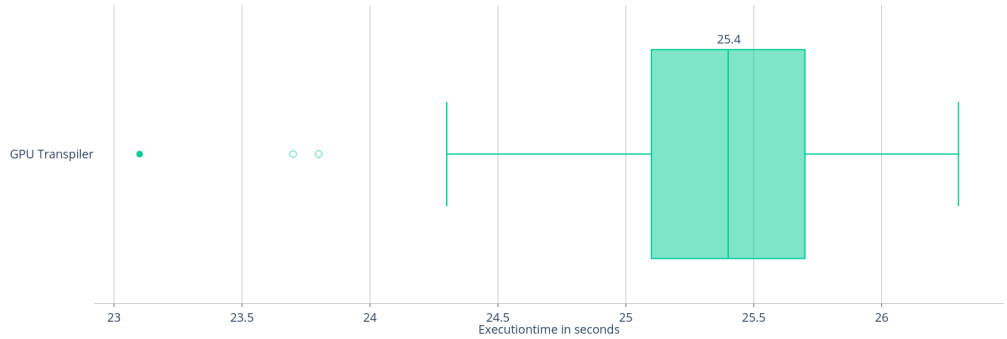


Figure 5.9: The results of the transpiler for benchmark 3.

Benchmark 3

This benchmark increased the amount of variable sets by 30 times and therefore also increases the total number of evaluations by 30 times. As observed in the second benchmark, the GPU was underutilised and thus had more resources available for evaluating the expressions. As shown in Figure 5.9 the available resources were better utilised. Although the number of evaluations increased by a factor of 30, the median execution time only increased by approximately six seconds, or 1.3 times, from 19.6 to 25.4. The standard deviation also decreased from 1.16 seconds to 0.65 seconds.

Given the change in the number of variable sets, additional performance tests with different block sizes were conducted. During this process it was found, that changing the block size from 128 to 160 threads resulted in the best performance. This is in contrast to the GPU interpreter where changing the block size to 160 resulted in degraded performance.

While conducting this benchmark, the CPU utilisation began at 100% during the frontend step as well as the transpilation and compilation steps. However, similar to the third benchmark of the GPU interpreter, the CPU utilisation dropped to 80% during the evaluation phase. This is very likely due to the same reason that the kernels are dispatched too quickly in succession, filling up the number of allowed resident grids on the GPU.

However, GPU utilisation also increased drastically. During the second benchmark, rapid oscillation was observed. With this benchmark the utilisation remained much more stable with the utilisation hovering around 60% to 70% most of the time. It should also be noted that there appeared frequent spikes to 100% and slightly less frequent drops to 20% utilisation. Overall the GPU utilisation was much higher compared to the second benchmark, which explains why the execution time only increased slightly despite the drastic increase in the number of evaluations.

5.2.4 Performance Tuning Transpiler

This section describes how the transpiler has been tuned to achieve good performance. Steps taken to improve the performance of the CPU-side of the transpiler are presented. Additionally, steps taken to improve the performance of the kernels are also shown.

Before any optimisations were applied, the block size was set to 256 threads. The

frontend as well as the transpilation and compilation were performed during the parameter optimisation step before the expression needed to be evaluated. Additionally, the variables have also been sent to the GPU on every parameter optimisation step. Multithreading has been used for the frontend, transpilation, compilation and kernel dispatch. Caching has also been used for the frontend and for the transpilation process in an effort to reduce the runtime.

As already mentioned in Section 5.2.2, using a cache in combination with multithreading for the frontend drastically slowed down the execution, which is the reason it has been disabled before conducting any benchmarks.

Caching has also been used for the transpilation step. The reason for this was to reduce the runtime during the parameter optimisation step. While this reduced the overhead of transpilation, the overhead of searching the cache if the expression has already been transpiled still existed. Because of the already mentioned RAM constraints this cache has been disabled and a better solution has been implemented in the first and second optimisation steps.

Most data of the tuning process has been gathered with the number of expressions and variable sets of the first benchmark, as this was the worst performing scenario. Therefore, it would show best where potential for performance improvements was. Before any optimisations were applied a single sample of the first benchmark took roughly 15 hours. However, it needs to be noted that only two samples were taken due to the duration of one sample.

Optimisation 1

Since all caching has been disabled, a better solution for reducing the number of calls to the frontend was needed. For this, the calls to the frontend were moved outside the parameter optimisation step and storing the result for later use. Furthermore, transmitting the variables to the GPU has also been performed before the parameter optimisation is started, further reducing the number and volume of data transfer to the GPU. These two optimisations were able to reduce the runtime of one sample to roughly 14 hours and are equivalent to the first optimisation step of the GPU interpreter.

Optimisation 2

With this optimisation step the number of calls to the transpiler and compiler have been drastically reduced. Both steps are now performed at the same time the frontend is called. The compiled kernels are then stored and only need to be executed during the parameter optimisation step. This meant that a cache was not needed any more. Because each time a new set of expressions needs to be evaluated, it is extremely unlikely that the same expression needs to be evaluated more than once. Consequently, the benefit of reducing the RAM consumption far outweighs the potential time savings of using a cache. Moreover, removing the cache also reduced the overhead of accessing it on every parameter optimisation step, further improving performance.

It also must be noted, that compiling the PTX kernels and storing the result before the parameter optimisation step lead to an out of memory error for the first benchmark. In order to get any results, this step had to be reverted for this benchmark. If much more RAM were available, the runtime would have been significantly better.

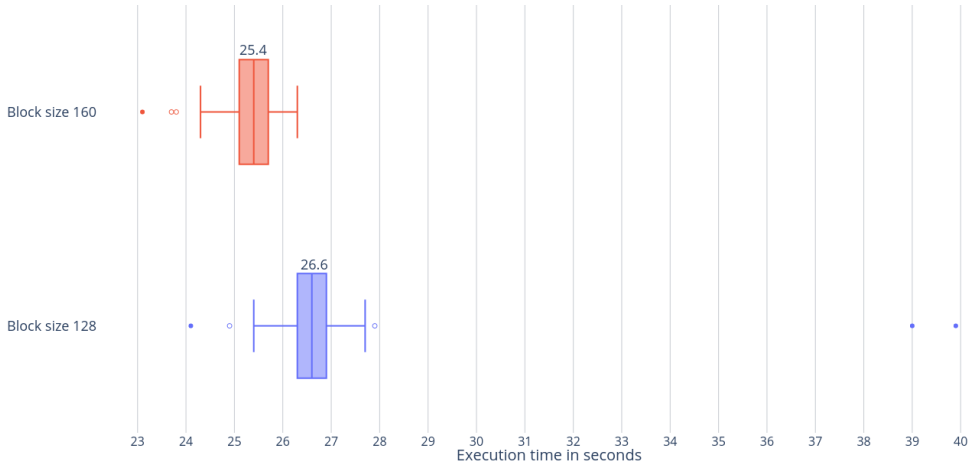


Figure 5.10: Runtime comparison of the third benchmark with block sizes of 128 and 160 threads.

These optimisations lead to a runtime of one sample of roughly ten hours for the first benchmark. Therefore, a substantial improvement of roughly four hours per sample was achieved. When 10 000 expressions are transpiled it takes on average 0.05 seconds over ten samples. Comparing this to the time spent compiling the resulting 10 000 kernels it takes on average 3.2 seconds over ten samples. This suggests that performing the compilation before the parameter optimisation step would yield drastically better results in the first benchmark.

Optimisation 3

The third optimisation step was more focused on improving the performance for the third benchmark as it has a higher number of variable sets than the first and second one. However, as with the interpreter, the function `CUDA.unsafe_free!(::CuArray)` has been used to reduce the standard deviation for all benchmarks.

Since the number of variable sets has changed in the third benchmark, it is important to re-do the performance tuning. This was done by measuring the kernel performance using NSight Compute. As with the interpreter, block sizes of 128 and 160 threads have been compared with each other. A block size of 192 threads has been omitted here since the number of excess threads is very high. In the case of the interpreter the performance of this configuration was the worst out of the three configurations, and it was assumed it will be similar in this scenario.

However, since the number of excess threads for 128 and 160 threads per block is the same, the latter using fewer blocks might lead to performance improvements in the case of the transpiler. As seen in Figure 5.10 this assumption was true and using a block size of 160 threads resulted in better performance for the third benchmark. This is in contrast to the interpreter, where this configuration performed much more poorly.

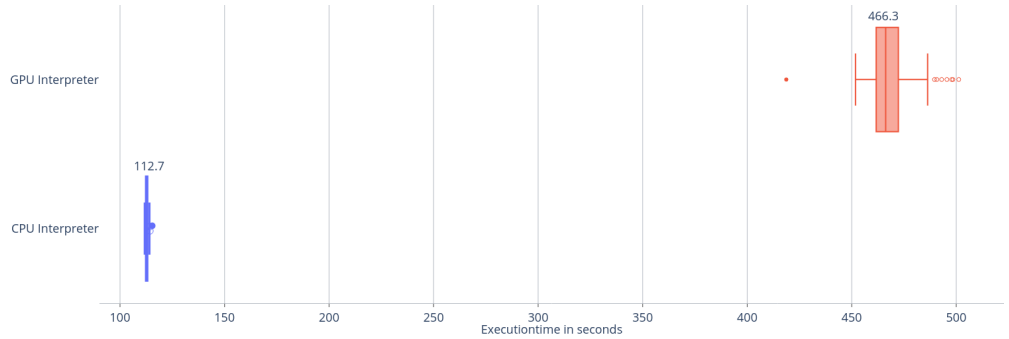


Figure 5.11: The results of the comparison of all three implementations for the first benchmark. Note that the transpiler is absent because it did not finish this benchmark.

5.2.5 Comparison

With the individual results of the GPU interpreter and transpiler presented, it is possible to compare them with the existing CPU interpreter. This section aims at outlining and comparing the performance of all three implementations across all three benchmarks to understand their strengths and weaknesses. Through this analysis the scenarios will be identified where it is best to leverage the GPU but also when using the CPU interpreter is the better choice, ultimately answering the research questions of this thesis.

Benchmark 1

The goal of the first benchmark was to determine how the evaluators are able to handle large amounts of expressions. While this benchmark is not representative of a typical scenario, it allows for demonstrating the impact the number of expressions has on the execution time. As already explained in Section 5.2.3 the transpiler failed to finish this benchmark due to RAM limitations. This required a slightly modified implementation to obtain results for at least two samples, each taking roughly ten hours to complete, which is the reason it has been omitted from this comparison.

Figure 5.11 shows the results of the first benchmark for the CPU and GPU interpreter. It can be seen that the GPU interpreter takes roughly four times as long on median than the CPU interpreter. Additionally, the standard deviation is much larger on the GPU interpreter. This shows that the CPU heavily benefits from scenarios where a lot of expressions need to be evaluated with very few variable sets. Therefore, it is not advisable to use the GPU to increase the performance in such scenarios.

Benchmark 2

Since the first benchmark has shown that with a large number of expressions the GPU is not a suitable alternative to the CPU. To further proof this statement a second benchmark with much fewer expressions was conducted. Now instead of 250 000 expressions, only 10 000 are evaluated. This reduction also meant that the transpiler can now be included in the comparison as it does not face any RAM limitations any more.

Reducing the number of expressions did not benefit the GPU evaluators at all in

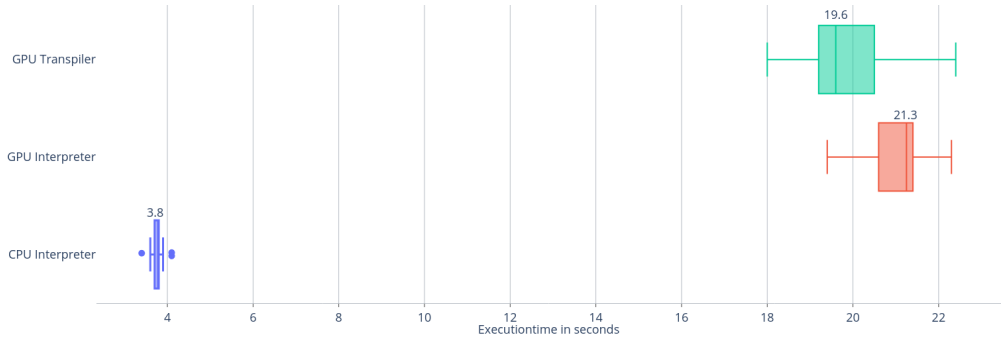


Figure 5.12: The results of the comparison of all three implementations for the second benchmark.

relation to the CPU interpreter. This can be seen in Figure 5.12. Furthermore, now the GPU evaluators are both roughly five times slower than the CPU interpreter instead of the previous performance reduction of roughly four times. Again the standard deviation is also much higher on both GPU evaluators when compared to the CPU interpreter. This means that a lower number of expressions does not necessarily mean that the GPU can outperform the CPU. Thus disproving the above statement that only a large number of expressions results in the GPU performing poorly.

On the other side, it can also be seen that the GPU transpiler tends to perform better than the GPU interpreter. While in the worst case both implementations are roughly equal, the GPU transpiler on median performs better. Additionally, the GPU transpiler can also outperform the GPU interpreter in the best case.

Benchmark 3

As found by the previous two benchmarks, varying the number of expressions only has a slight impact on the performance of the GPU in relation to the performance of the CPU. However, instead of varying the number of expressions, the number of variable sets can also be changed. For this benchmark, instead of 362 variable sets, a total of 10 860 variable sets were used, which translates to an increase by 30 times. It needs to be noted, that it was only possible to evaluate the performance with roughly 10 000 expressions with this number of variable sets. When using the same roughly 250 000 expressions of the first benchmark and the increase number of variable sets, none of the implementations managed to complete the benchmark, as there was too little RAM available.

Increasing the number of variable sets greatly benefited both GPU evaluators as seen in Figure 5.13. With this change, the CPU interpreter noticeably fell behind the GPU evaluators. Compared to the GPU transpiler, the CPU interpreter took roughly twice as long on median. The GPU transpiler continued its trend of performing better than the GPU interpreter. Furthermore, the standard deviation of all three evaluators is also very similar.

From this benchmark it can be concluded that the GPU heavily benefits from a larger number of variable sets. If the number of variable sets is increased even further, the

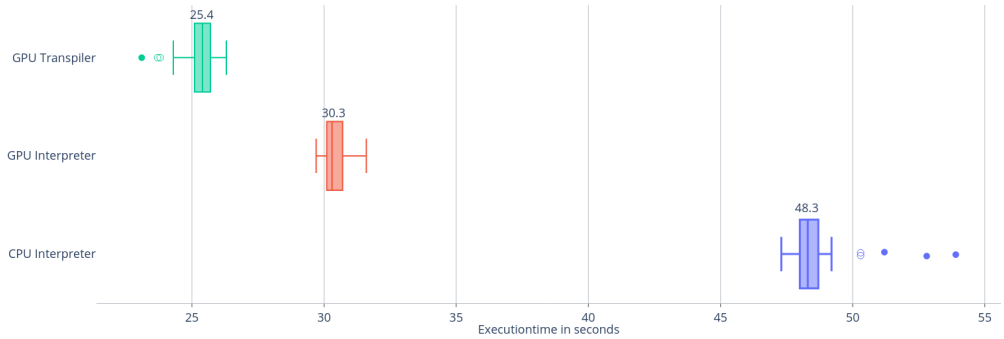


Figure 5.13: The results of the comparison of all three implementations for the third benchmark.

difference in performance between the GPU and CPU should be even more pronounced.

While the GPU is very limited in terms of concurrent kernel dispatches that can be evaluated, the number of threads and blocks can virtually be infinitely large. This means that a higher degree of parallelism is achievable with a higher number of variable sets. Increasing the number of expressions on the other hand does not influence the degree of parallelism to this extent. This is the reason no performance benefit was found by only decreasing the number of expressions with the same number of variable sets.

Chapter 6

Conclusion and Future Work

A typical system consists of a set of inputs with an observed output. For example when trying to model the flow in rough pipes as done by Nikuradse (1950) where the length, the diameter and the roughness of the pipes are the input. In this scenario the flow through the pipe is the output and a mathematical model is needed to describe the correlation between the inputs and outputs. Finding such a model or formula can be done by utilising a computer and symbolic regression. Symbolic regression typically is implemented using genetic programming. During the runtime thousands or even hundreds of thousands of formulas or expressions are generated which need to be evaluated to determine if they describe the observed system with sufficient accuracy. This process can take several hours to days to find a suitable formula on a single machine utilising the CPU only. Therefore, this thesis deals with the question of how the evaluation of the expressions generated at runtime can be sped up to minimise execution times.

Research has been conducted on how to best approach this problem statement. The GPU has been chosen to improve the performance as a cheap and powerful tool especially compared to compute clusters. Numerous instances exist were utilising the GPU lead to drastic performance improvements in many fields of research.

Two GPU evaluators were implemented which should determine if the GPU is more suitable for evaluating expressions generated at runtime as compared to the CPU. The two implementations are as follows:

GPU Interpreter
: A stack based interpreter that evaluates the expressions. The frontend converts these expressions into postfix notation to ensure the implementation can be as simple as possible. It consists of one kernel that is used to evaluate all expressions separately.

GPU Transpiler
: A transpiler that takes the expressions and transpiles them into PTX code. Each expression is represented in its own unique kernel. The kernels are simpler than the one GPU interpreter kernel, but more effort is needed to generate them.

In total three benchmarks were conducted to determine if and under which circumstances the GPU is a more suitable choice for evaluating the expressions. The current CPU implementation is the baseline against which the GPU evaluators are evaluated. To answer the research questions the benchmarks are structured as follows:

1. Roughly 250 000 expressions with 362 variable sets have been evaluated. The goal of this benchmark was determining how the evaluators can handle large volumes of expressions.
2. Roughly 10 000 expressions with 362 variable sets have been evaluated. This benchmark should demonstrate how a change in the number of expressions impacts the performance, especially compared with each other.
3. Roughly 10 000 expressions and roughly 10 000 variable sets have been evaluated. By increasing the number of variable sets a more realistic use-case is modelled with this benchmark. Additionally, by using more variable sets the strengths of the GPU should get more exploited.

After conducting the first and second benchmarks it was clear, that the CPU is the better choice in these scenarios. The first benchmark in particular demonstrated how the high RAM usage of the GPU transpiler lead to it not finishing this benchmark. Reducing the number of expressions demonstrated that the GPU transpiler can perform better than the GPU interpreter, however, in relation to the CPU implementation, no real change was observed between the first and second benchmark. However, in the third benchmark, both GPU evaluators managed to outperform the CPU, with the GPU transpiler performing the best.

To address the research questions, this thesis demonstrates that evaluating expressions generated at runtime can be more efficient on the GPU under specific conditions. Utilizing the GPU becomes feasible when dealing with a high number of variable sets, typically in the thousands and above. For scenarios with fewer variable sets, the CPU remains the better choice. Additionally, in scenarios where RAM is abundant, the GPU transpiler is the optimal choice. If too little RAM is available and the number of variable sets is sufficiently large, the GPU interpreter should be chosen, as it outperforms both the GPU transpiler and the CPU in such cases.

6.1 Future Work

This thesis demonstrated how the GPU can be used to accelerate the evaluation of expressions and therefore the symbolic regression algorithm as a whole. However, the boundaries at which it is more feasible to utilise the GPU are very coarse-grained. Therefore, conducting more research into how the number of expressions and variable sets impact performance is needed. Furthermore, only one dataset with only two variables per variable set was used. Varying the number of variables per set and their impact on performance could also be interesting. The impact of the parameters was omitted from this thesis entirely. Further research on how the number of parameters impact the performance is of interest. Since parameters need to be transferred to the GPU frequently, having too many parameters could impact the GPU more negatively than the CPU.

The current implementation also has flaws that can be improved in future work. Currently, no shared memory is utilised, meaning the threads need to always retrieve the data from global memory. This is a slow operation and efficiently utilising shared memory should further improve the performance of both GPU evaluators.

Additionally, neither of the implementations supports special GPU instructions. Es-

pecially the Fused Multiply-Add (FMA) instruction is of interest. Given that multiplying two values and adding a third is a common operation, this special instruction allows these operations to be performed in a single clock cycle. The frontend can be extended to detect and convert sub-expressions of this form into a special ternary opcode, enabling the backend to generate more efficient code. If the effort of detecting these sub-expressions is outweighed by the performance improvement needs to be determined in a future work.

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, 45).

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, 53)

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 pp. 7, 32)

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)

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671 (cit. on p. 31)

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)

Bruneton, J.-P. (2025, March 24). Enhancing symbolic regression with quality-diversity and physics-inspired constraints. https://doi.org/10.48550/arXiv.2503.19043. (Cit. on p. 5)

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. 5)

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. 19)

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. 20)

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, 45)

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. 19)

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. 11)

Faingnaert, T., Besard, T., & De Sutter, B. (2022). Flexible performant GEMM kernels on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, *33*(9), 2230–2248. https://doi.org/10.1109/TPDS.2021.3136457 (cit. on p. 32)

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. 8)

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. 19)

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, 8)

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. 19)

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)

Guimerà, R., Reichardt, I., Aguilar-Mogas, A., Massucci, F. A., Miranda, M., Pallarès, J., & Sales-Pardo, M. (2020). A bayesian machine scientist to aid in the solution of challenging scientific problems. *Science Advances*, *6*(5), eaav6971. https://doi.org/10.1126/sciadv.aav6971 (cit. on p. 53)

Gustafson, S., Burke, E., & Krasnogor, N. (2005). On improving genetic programming for symbolic regression. *2005 IEEE Congress on Evolutionary Computation*, *1*, 912–919 Vol.1. https://doi.org/10.1109/CEC.2005.1554780 (cit. on p. 5)

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, 8)

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. 7)

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. 7)

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)

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)

Knuth, D. E. (1999). MMIX. In D. E. Knuth (Ed.), *MMIXware: A RISC computer for the third millennium* (pp. 2–61). Springer. https://doi.org/10.1007/3-540-46611-8_2. (Cit. on p. 8)

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)

Korns, M. F. (2015). Extremely accurate symbolic regression for large feature problems. In R. Riolo, W. P. Worzel, & M. Kotanchek (Eds.), *Genetic programming theory*

and practice XII (pp. 109–131). Springer International Publishing. https://doi.org/10.1007/978-3-319-16030-6_7. (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, 19)

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 International Publishing. https://doi.org/10.1007/978-3-030-96772-7_35. (Cit. on p. 8)

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. 5, 6)

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. 19)

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. 8)

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. *Pro-*

ceedings of the 51st International Conference on Parallel Processing, 1–12. https://doi.org/10.1145/3545008.3545091 (cit. on p. 20)

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 pp. 7, 31, 32)

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. 20)

Marcelino, M., & Leitão, A. M. (2022). Transpiling python to julia using PyJL. https://doi.org/10.5281/ZENODO.6332890 (cit. on p. 20)

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. 20)

Nikuradse, J. (1950, November). *Laws of flow in rough pipes*. https://digital.library.unt.edu/ark:/67531/metadc63009/. (Cit. on pp. vii, viii, 53, 68)

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. 19)

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)

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. 6)

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. 8)

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. 20)

Wang, L.-T., Chang, Y.-W., & Cheng, K.-T. (2009, March 11). *Electronic design automation: Synthesis, verification, and test*. Morgan Kaufmann. (Cit. on p. 20).

Werner, M., Junginger, A., Hennig, P., & Martius, G. (2021, May 13). Informed equation learning. https://doi.org/10.48550/arXiv.2105.06331. (Cit. on pp. 4, 5)

Winter, M., Parger, M., Mlakar, D., & Steinberger, M. (2021). Are dynamic memory managers on GPUs slow? a survey and benchmarks. *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 219–233. https://doi.org/10.1145/3437801.3441612 (cit. on p. 41)

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. 20)

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. 13)

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. 18)

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. 20)

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, 19)

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. 20)

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 pp. 16, 56)

Nvidia. (2025b, March). *CUDA c++ programming guide*. Retrieved November 22, 2024, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/. (Cit. on pp. 7–9, 15, 24, 32)

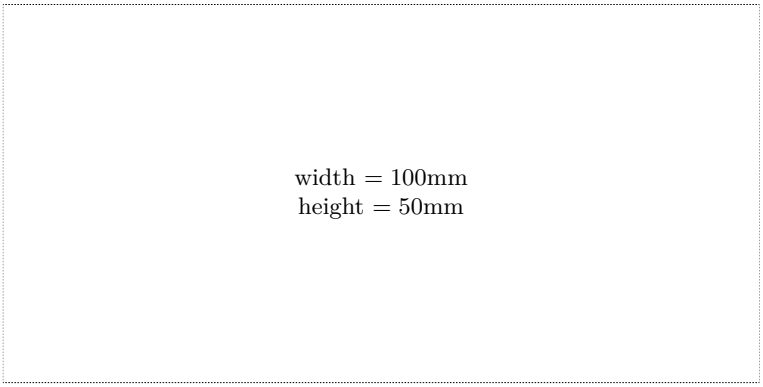
Nvidia. (2025c, March). *Nsight compute — NsightCompute 12.8 documentation*. Retrieved March 16, 2025, from https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator. (Cit. on p. 15)

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, 46)

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! —