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

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

Kurzfassung

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

Chapter 1

Introduction

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

1.1 Background and Motivation

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

The following expression $x_1 + 5 - \text{abs}(p_1) * \text{sqrt}(x_2)/10 + 2\hat{}x_3$ which contains simple mathematical operations as well as variables x_n and parameters p_n is one example that can be generated by the equation learning algorithm, Usually an equation learning algorithm generates multiple of such expressions per iteration. Out of these expressions all possibly relevant ones have to be evaluated. Additionally, multiple different values need to be inserted for all variables and parameters, drastically increasing the amount of evaluations that need to be performed.

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

compared to compute clusters. Enterprise GPUs like the B200[1] cost at least $30000, and are only available as a full system with 8 GPUs, power delivery etc (Bajwa, 2024). Data centres specialised for artificial intelligence workloads often cost billions of dollars (Bajwa, 2024). However, despite these costs for enterprise GPUs, cheaper consumer GPUs can also deliver a great performance uplift. Michalakes and Vachharajani (2008) have shown a noticeable speed-up when using GPUs for weather simulation. In addition to computer simulations, GPU acceleration also can be found in other places such as networking (S. Han et al., 2010) or structural analysis of buildings (Georgescu et al., 2013).

1.2 Research Question

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

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

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

1.3 Thesis Structure

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

Chapter 2: Fundamentals and Related Work

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

Chapter 3: Concept and Design

Within this chapter, the concepts of implementing the GPU interpreter and tran-

[1] https://www.nvidia.com/de-de/data-center/dgx-b200/

spiler 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. 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 aims at 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. Because of this, generating equations with a computer can more easily lead to discovering equations that describe the observed data. Brunton et al. (2016) describe an algorithm that leverages equation learning to discover equations for physical systems. A more literal interpretation of equation learning is demonstrated by Pfahler and Morik (2020). They use machine learning to learn the form of equations. Their aim was to simplify the discovery of relevant publications by the equations they use and not by technical terms, as they may differ by the field of research. However, this kind of equation learning is not relevant for this thesis.

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

2010, Koza (2010) provided an overview of results that were generated with the help of GP and were competitive with human solutions. Keijzer (2004) and Korns (2011) presented ways of improving the quality of symbolic regression algorithms, making symbolic regression more feasible for problem-solving. Bartlett et al. (2024) describe an exhaustive approach for symbolic regression which can find the true optimum for perfectly optimised parameters while retaining simple and interpretable results. Alternatives to GP for symbolic regression also exist with one proposed by Jin et al. (2020). Their approach increased the quality of the results noticeably compared to GP alternatives. Another alternative to heuristics like GP is the usage of neural networks. One such alternative has been introduced by Martius and Lampert (2016) where they used a neural network for their equation learner with mixed results. Later, an extension has been provided by Sahoo et al. (2018). They introduced the division operator, which led to much better results. Further improvements have been described by Werner et al. (2021) with their informed equation learner. By incorporating domain expert knowledge they could limit the search space and find better solutions for particular domains. As seen by these publications, increasing the quality of generated equations and also increasing the speed of finding these equations is a central part in symbolic regression and equation learning in general. This means research in improving the computational performance of these algorithms is desired.

The expressions generated by an equation learning algorithm can look like this $x_1 + 5 - \text{abs}(p_1) * \text{sqrt}(x_2)/10 + 2\hat{}x_3$. They consist of several unary and binary operators but also of constants, variables and parameters and expressions mostly differ in length and the kind of terms in the expressions. Per iteration many of these expressions are generated and in addition, matrices of values for the variables and parameters are also created. One row of the variable matrix corresponds to one instantiation of all expressions and this matrix contains multiple rows. This leads to a drastic increase of instantiated expressions that need to be evaluated. Parameters are a bit simpler, as they can be treated as constants for one iteration but can have a different value on another iteration. This means that parameters do not increase the number of expressions that need to be evaluated. However, the increase in evaluations introduced by the variables is still drastic and therefore increases the algorithm runtime significantly.

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. 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.

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

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, Foket, et al. (2019) showed that it is possible to develop with CUDA in the high level programming language Julia[2] while performing similar to CUDA written in C. In a subsequent study 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 and has the additional benefit of developer comfort since it is a high level language with modern features such as garbage-collectors. Besard, Churavy, et al. (2019) 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].

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. Weather simulations began using GPUs very early for their models. In 2008 Michalakes and Vachharajani (2008) proposed a method for simulating weather with the WRF model on a GPU. With their approach, they reached a speed-up of the most compute intensive task of 5 to 20, with very little GPU optimisation effort. They also found that the GPU usages was very 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 to improve the performance of time step simulations, while retaining their precision and constraint correctness. Black hole simulations are crucial for science and education for a better understanding of our world. Verbraeck and Eisemann (2021) have shown that simulating complex Kerr (rotating) black holes can be done on consumer hardware in a few seconds. Schwarzschild black hole simulations can be performed in real-time with GPUs as described by Hissbach et al. (2022) which is especially helpful for educational scenarios. While both approaches do not have the same accuracy as detailed simulations on supercomputers, they show how single GPUs can yield similar accuracy at a fraction of the cost. Networking can also heavily benefit from GPU acceleration as shown by S. Han et al. (2010), where they achieved a significant increase in 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). However, it also needs to be noted, that GPUs are not always better performing than CPUs as illustrated by Lee et al. (2010), but they still can lead to performance improvements nonetheless.

[2] https://julialang.org/

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

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

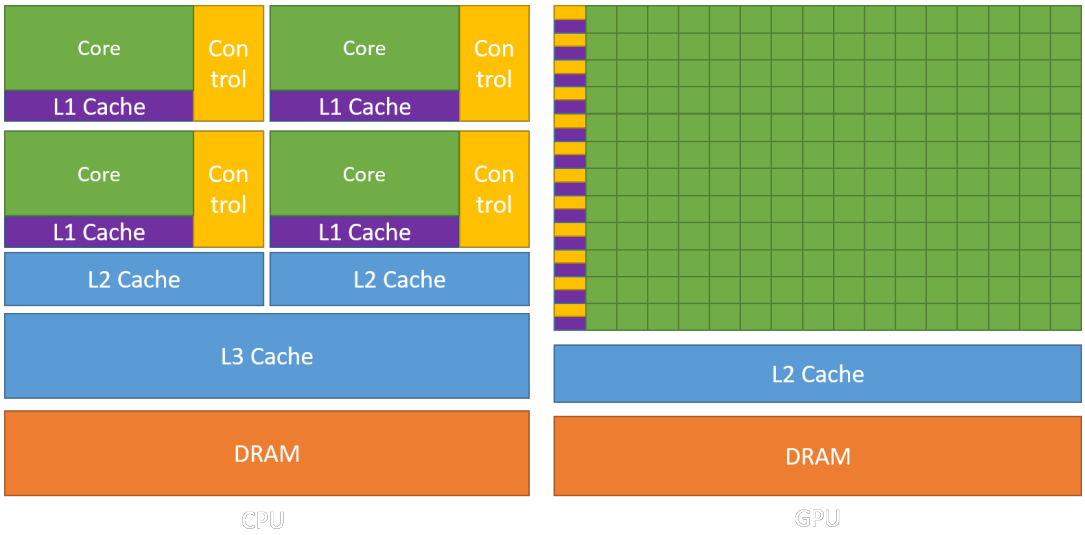


Figure 2.1: Overview of the architecture of a CPU (left) and a GPU (right). Note the higher number of simpler cores on the GPU (Nvidia, 2024).

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 a staggering 192 of those complex cores and twice as many threads. A guide for a simple one core 8-bit CPU has been published by Schuurman (2013). He describes the many different and complex parts of a CPU core. Modern CPUs are even more complex, with dedicated fast integer and floating-point arithmetic gates as well as logic gates, sophisticated branch prediction and much more. This makes a CPU perfect for handling complex control flows on a single program strand and on modern CPUs even multiple strands simultaneously. However, as seen in section 2.2, this often isn't 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 21760 CUDA cores. To achieve this enormous core count a single GPU core has to be much simpler than one CPU core. As described by Nvidia (2024) a GPU designates much more transistors towards floating-point computations. This results in less efficient integer arithmetic and control flow handling. There is also less Cache available per core and clock speeds are usually also much lower than those on a CPU. An overview of the differences of a CPU and a GPU architecture can be seen in figure 2.1.

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

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

streams. Franchetti et al. (2005) and Tian et al. (2012) describe ways of using SIMD instructions on the CPU. Their approaches lead to noticeable speed-ups of 3.3 and 4.7 respectively by using SIMD instructions instead of serial computations. Extending this to GPUs which are specifically built for SIMD/data parallel calculations shows why they are so powerful despite having less complex and slower cores than a CPU.

Thread Hierarchy and Tuning

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

At the lowest level of a GPU exists a Streaming Multiprocessor (SM), which is a hardware unit responsible for scheduling and executing threads and also contains the registers used by these threads. An SM is always executing a group of 32 threads simultaneously, and this group is called a warp. The number of threads that can be started is virtually unlimited. However, threads must be grouped in a block, with one block typically containing a maximum of 2048 threads but is often configured to be less. Therefore, if more than 2048 threads are required, more blocks must be created. Blocks can also be grouped 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 the GPU, also called kernel (AMD, 2025). All threads in one block have access to some shared memory, which can be used for L1 caching or communication between threads. It is important that the blocks can be scheduled independently, with no dependencies between them. This allows the scheduler to schedule blocks and threads as efficiently as possible. All threads within a warp are guaranteed to be part of the same block, and are therefore executed simultaneously and can access the same memory addresses. Figure 2.2 depicts how threads in a block are grouped into warps for execution and how they share 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.

All threads in a warp start at the same point in a program, they have their own

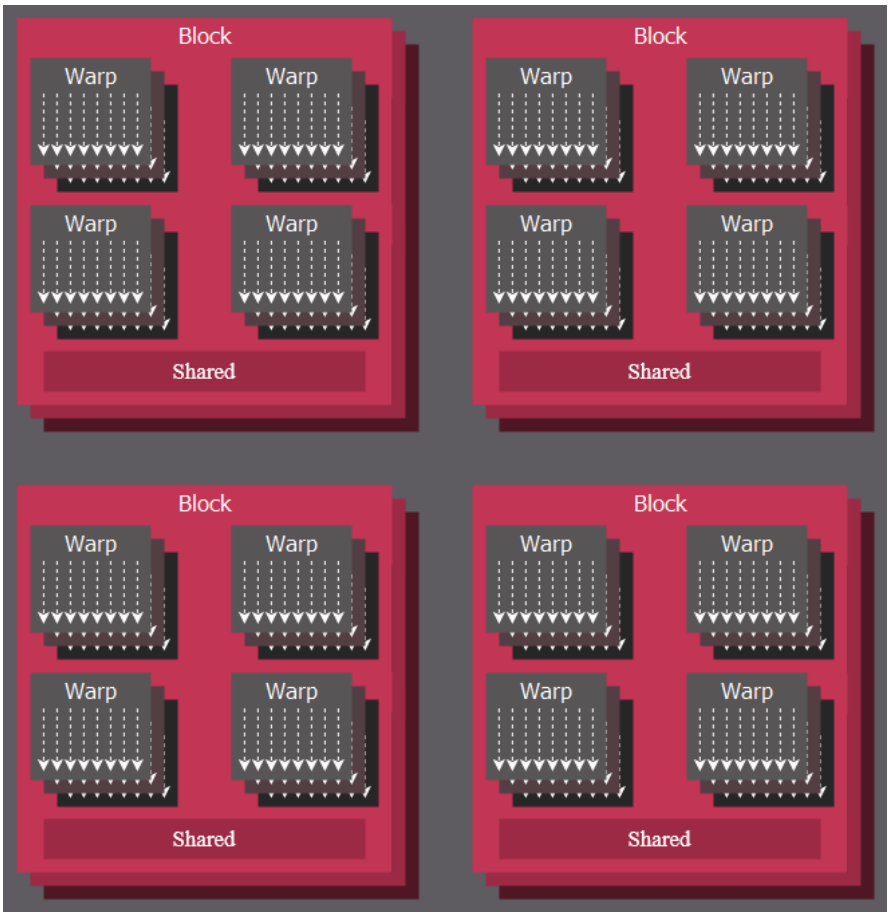


Figure 2.2: An overview of the thread hierarchy with blocks being split into multiple warps and their shared memory (AMD, 2025).

instruction address, allowing them to work independently. Because of the SIMD architecture, all threads in a warp must execute the same instructions and if threads start diverging, the SM must pause threads with different instructions and execute them later. Figure 2.3 shows how such divergences can impact performance. The situation described by the figure also shows, that after the divergent thread would reconverge, this does not happen and leads to T2 being executed after T1 and T3 are finished. In situations where a lot of data dependent thread divergence happens, most of the benefits of using a GPU have vanished.

Threads not executing the same instruction is against the SIMD principle but can happen in reality, due to data dependent branching. Consequently, this leads to bad resource utilisation, which in turn leads to worse performance. Another possibility of threads being paused (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 (Nvidia, 2024).

Modern GPUs implement the so called Single-Instruction Multiple-Thread (SIMT) architecture. In many cases a developer does not need to know the details of SIMT

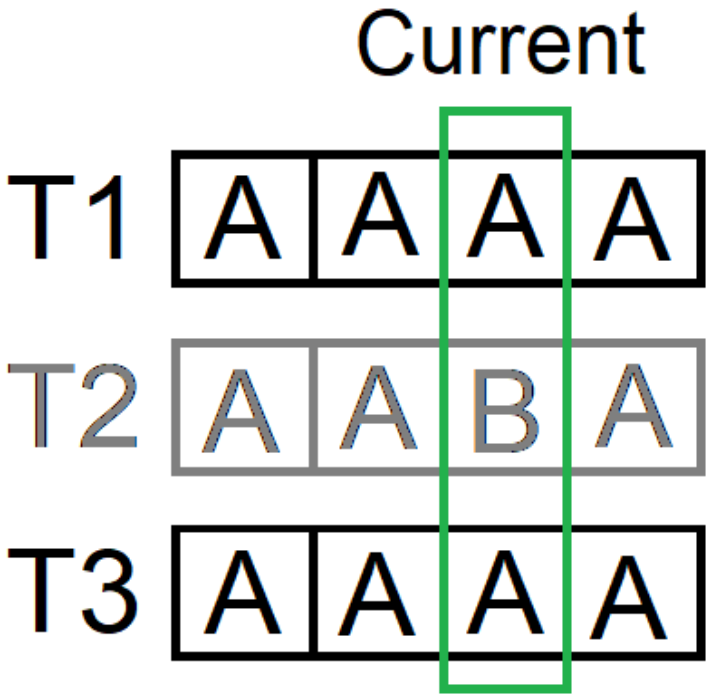


Figure 2.3: Thread T2 wants to execute instruction B while T1 and T3 want to execute instruction A. Therefore T2 will be an inactive thread this cycle and active once T1 and T3 are finished. This means that now the divergent threads are serialised.

and can develop fast and correct programs with just the SIMD architecture in mind. However, leveraging the power of SIMT can yield substantial performance gains by re-converging threads once data dependent divergence occurred. A proposal for a re-convergence algorithm was proposed by Collange (2011) where they have shown that these approaches help with hardware occupation, resulting in improved performance as threads are now no longer fully serialised. 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 reconverge. 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 deactivated 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.

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.

Talk about memory model and memory allocation (with the one paper diving into dynamic allocations) Memory transfer (with streams potentially)

2.2.2 Parallel Thread Execution

Describe what PTX is to get a common ground for the implementation chapter. Probably a short section

2.3 Compilers

Maybe even move this entire section to "Concept and Design"?

brief overview about compilers (just setting the stage for the subsections basically). Talk about register management and these things.

2.3.1 Interpreters

What are interpreters; how they work; should mostly contain/reference gpu interpreters

2.3.2 Transpilers

talk about what transpilers are and how to implement them. If possible also gpu specific transpilation.

Chapter 3

Concept and Design

introduction to what needs to be done. also clarify terms “Host” and “Device” here

3.1 Requirements and Data

short section. Multiple expressions; vars for all expressions; params unique to expression; operators that need to be supported

3.2 Interpreter

as introduction to this section talk about what “interpreter” means in this context. so “gpu parses expr and calculates”

3.2.1 Architecture

talk about the coarse grained architecture on how the interpreter will work. (.5 to 1 page probably)

3.2.2 Host

talk about the steps taken to prepare for GPU interpretation

3.2.3 Device

talk about how the actual interpreter will be implemented

3.3 Transpiler

as introduction to this section talk about what “transpiler” means in this context. so “cpu takes expressions and generates ptx for gpu execution”

3.3.1 Architecture

talk about the coarse grained architecture on how the transpiler will work. (.5 to 1 page probably)

3.3.2 Host

talk about how the transpiler is implemented

3.3.3 Device

talk about what the GPU does. short section since the gpu does not do much

Chapter 4

Implementation

4.1 Technologies

Short section; CUDA, PTX, Julia, CUDA.jl
Probably reference the performance evaluation papers for Julia and CUDA.jl

4.2 Interpreter

Talk about how the interpreter has been developed.

4.3 Transpiler

Talk about how the transpiler has been developed

Chapter 5

Evaluation

5.1 Test environment

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

5.2 Results

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

5.2.1 Interpreter

Results only for Interpreter

5.2.2 Transpiler

Results only for Transpiler

5.2.3 Comparison

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

Chapter 6

Conclusion and Future Work

Summarise the results

6.1 Future Work

talk about what can be improved

References

Literature

Bajwa, A. (2024). Microsoft, OpenAI plan $100 billion data-center project, media report says. *Reuters*. Retrieved March 13, 2025, from https://www.reuters.com/technology/microsoft-openai-planning-100-billion-data-center-project-information-reports-2024-03-29/ (cit. on p. 2)

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

Besard, T., Churavy, V., Edelman, A., & Sutter, B. D. (2019). Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software*, *132*, 29–46. https://doi.org/10.1016/j.advengsoft.2019.02.002 (cit. on p. 6)

Besard, T., Foket, C., & De Sutter, B. (2019). 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. 6)

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

Collange, C. (2011, September). *Stack-less SIMT reconvergence at low cost* (Research Report). ENS Lyon. Retrieved March 8, 2025, from https://hal.science/hal-00622654. (Cit. on p. 10)

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. 5)

Franchetti, F., Kral, S., Lorenz, J., & Ueberhuber, C. (2005). Efficient utilization of SIMD extensions [Conference Name: Proceedings of the IEEE]. *Proceedings of the IEEE*, *93*(2), 409–425. https://doi.org/10.1109/JPROC.2004.840491 (cit. on p. 8)

Fung, W. W. L., & Aamodt, T. M. (2011). Thread block compaction for efficient SIMT control flow [ISSN: 2378-203X]. *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 25–36. https://doi.org/10.1109/HPCA.2011.5749714 (cit. on p. 10)

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, 6)

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, 6)

Han, T. D., & Abdelrahman, T. S. (2011). hiCUDA: High-level GPGPU programming [Conference Name: IEEE Transactions on Parallel and Distributed Systems]. *IEEE Transactions on Parallel and Distributed Systems*, *22*(1), 78–90. Retrieved March 1, 2025, from https://ieeexplore.ieee.org/abstract/document/5445082 (cit. on p. 6)

Hissbach, A.-M., Dick, C., & Lawonn, K. (2022). *An overview of techniques for egocentric black hole visualization and their suitability for planetarium applications*. The Eurographics Association. Retrieved March 2, 2025, from https://doi.org/10.2312/vmv.20221207. (Cit. on p. 6)

Huang, Q., Huang, Z., Werstein, P., & Purvis, M. (2008). GPU as a general purpose computing resource [ISSN: 2379-5352]. *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. 5)

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)

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

Köster, M., Groß, J., & Krüger, A. (2020a). Massively parallel rule-based interpreter execution on GPUs using thread compaction. *International Journal of Parallel Programming*, *48*(4), 675–691. https://doi.org/10.1007/s10766-020-00670-2 (cit. on p. 10)

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. 6)

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 p. 4)

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 sym-*

posium on Computer architecture, 451–460. https://doi.org/10.1145/1815961.1816021 (cit. on p. 6)

Lin, W.-C., & McIntosh-Smith, S. (2021). Comparing julia to performance portable parallel programming models for HPC. *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 94–105. https://doi.org/10.1109/PMBS54543.2021.00016 (cit. on p. 6)

Martius, G., & Lampert, C. H. (2016). Extrapolation and learning equations [Version Number: 1]. https://doi.org/10.48550/ARXIV.1610.02995. (Cit. on p. 5)

Michalakes, J., & Vachharajani, M. (2008). GPU acceleration of numerical weather prediction [ISSN: 1530-2075]. *2008 IEEE International Symposium on Parallel and Distributed Processing*, 1–7. https://doi.org/10.1109/IPDPS.2008.4536351 (cit. on pp. 2, 6)

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)

Sahoo, S. S., Lampert, C. H., & Martius, G. (2018). Learning equations for extrapolation and control [Version Number: 1]. https://doi.org/10.48550/ARXIV.1806.07259. (Cit. on p. 5)

Schuurman, D. C. (2013). Step-by-step design and simulation of a simple CPU architecture. *Proceeding of the 44th ACM technical symposium on Computer science education*, 335–340. https://doi.org/10.1145/2445196.2445296 (cit. on p. 7)

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 [Conference Name: IEEE Transactions on Visualization and Computer Graphics]. *IEEE Transactions on Visualization and Computer Graphics*, *27*(2), 796–805. https://doi.org/10.1109/TVCG.2020.3030452 (cit. on p. 6)

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

Online sources

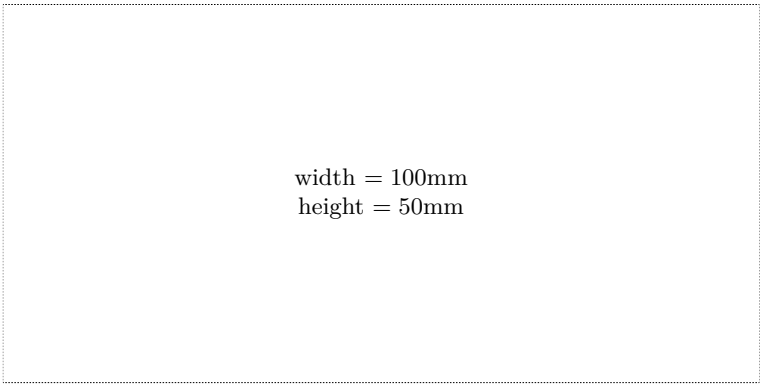
AMD. (2025, March). *HIP programming model — HIP 6.3.42134 documentation*. Retrieved March 9, 2025, from https://rocm.docs.amd.com/projects/HIP/en/latest/understand/programming_model.html#programming-model-simt. (Cit. on pp. 8, 9)

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

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! —