Master's thesis

NTNU
Norwegian University of Science and Technology

Ingvild Brevik Høgstøyl

Exploring NVIDIA Ampere Tensor Cores for an Event Generator Code for High-Energy Physics

Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
Co-supervisor: Dr. Maria Girone
July 2021

NTNU
Norwegian University of
Science and Technology

Ingvild Brevik Høgstøyl

Exploring NVIDIA Ampere Tensor Cores for an Event Generator Code for High-Energy Physics

Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
Co-supervisor: Dr. Maria Girone
July 2021

Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Project Description

The demand and interest for more cost-effective solutions for the same amount of processing power is increasing. In addition, data centers are now a mix of HPC-systems that often include accelerators such as GPUs. To meet the computational demands of the future, CERN is working on porting CPU codes to GPU and CPU+GPU versions.

CERN is also working on developing a new benchmarking suite to evaluate the performance of the resources they have at their disposal through the Worldwide LHC Computing Grid (WLCG). The current benchmark suite HEP SPEC 2006, a subset of the SPEC CPU 2006 benchmark suite, has been found to show lack of correlations with today's HEP applications. A new approach for benchmarking is being investigated using experiment workloads. The benchmarks are put into containers to make them easier to execute in data centres.

Currently, the GPU version of Madgraph is a simple conversion of parts of the CPU version, so this thesis will also focus on ways to optimize the GPU code. The new NVIDIA A100 GPU offers several new features, such as 64-bit floating point tensor cores, so one of the main foci of the optimization is to investigate whether one or more of these new features can be used, and if so, profile and test them, and if time permits, integrate them into the Madgraph GPU code. Containerizing the application may also be considered.

Abstract

There is an increasing interest and demand for more cost-effective solutions for how to utilize the processing power of large HPC systems. Data centers are now progressively offering more heterogeneous computing, that include accelerators such as GPUs. To meet the computational demands of the future, CERN is working on utilizing modern hardware such as GPUs, both to speed up their workloads, but also to utilize all the hardware they have available through the Worldwide LHC Computing Grid (WLCG).

The work of this thesis includes finding a suitable workload to work with. We landed on that the focus should be on the MadGraph workload, a physics event generator planned to be used as part of one of the benchmarking suites at CERN. Event generators are large consumers of CPU resources, and their computational cost is expected to increase with future upgrades of the experiments on which they are built.

Our works expands on the current GPU version of the workload which is a simple conversion of some of the parts of the CPU version. Many of the calculations in the code involve double precision complex numbers, over-utilizing the FP64 pipeline in the GPU, while under-utilizing the others. The work of this thesis therefore focuses on investigating whether the code can be optimized by using the new NVIDIA A100 GPU, which offers 64-bit floating point Tensor Cores.

The thesis also includes profiling resulkts from using Valgrind and KCachegrind on the CPU as well as results from using Nsight Systems and Nsigth Compute for profiling the GPU kernels. Integration the usage of Tensor Cores into the MadGraph GPU code and analysing the effect of the changes, as well as some suggestions for future work, are also included.

Sammendrag

Det er en økende interesse og etterspørsel etter mer kosteffektive løsninger for hvordan en kan utnytte regneresursene til store HPC-systemer.

Datasentrene er tilbyr nå mer heterogen databehandling, som inkluderer systemer med akseleratorer, deriblant GPUer.

For åmøte fremtiden databeregningsbehov jobber nå CERN med å tilby fler og fler heterogene systemer med GPUer for å få 'workloads' (arbeidsmengden deres applikasjoner utgjør) til å gå raskere, men også for å kunne utnytte all maskinvaren de har tilgjengelig via nettverket Worldwide LHC Computing Grid (WLCG).

Arbeidet i denne masteroppgaven inkluderer å finne en egnet 'workload'/applikasjon å kunne jobbe med. Vi endte opp med å fokusere på en del av MadGraphm en fysikk event-generator som det er planer om skal bli en del av benchmarking suiten til CERN, Event-generatorer forbruker mange regneressurser, og dere krav om beregningskraft er forventet å øke ytterligere i fremtiden ettersom eksperimentene de bygger på oppgraderes.

Vårt arbeid videreutvikler den nåværende GPU versjonen av applikasjonene som for tiden er en enkel oversettelse av ekelte deler av CPU-versjonen. Mange av beregningene in koden involverer dobbel-presisjons komplekse tall, overlast på FP64 pipen av GPUen, mens then ikke fullt ut benytter andre deler av GPUen. Arebidet i denne masteroppgaven fokuserer derfor på å unersøke om koden kan bli opimert for de nye NVIDIA A100 GPUenen som nå tilbyr 64.bit flytalls Tensorkjerner.

Masteroppgaven includerer også profileringer via Valgrind og KCachegrind på CPU så vel som brukt av Nsight Systems og Nsigth Compute for profilering av GPU funksjonene (kernels). Integrering bruken av Tensor-kjernene med MadGraph CPU-koden og analyseringen av effekten av disse endrringene, så vels som forslag for videre arbeider, er også tatt med.

Acknowledgements

I would especially like to thank Dr. Maria Girone, CTO of CERN openlab, for co-advising and hosting me through the CERN technical student program, and the Research Council of Norway for providing support for my stay at CERN.

Unfortunately, due to the ongoing pandemic situation, part of my intended stay at CERN had to instead be done remotely from my home in Norway. I am therefore especially grateful for all the weekly Zoom meetings with Maria and her group and colleagues at CERN, including David Southwick, Victor Khristenko, Dr. Andrea Valassi, and Stefan Roiser and for all their help and support that made this work possible.

I would also like to thank the HPC-Lab and the Department of Computer Science at NTNU for providing access to the GPU systems I used for a related pre-project, and CERN and Ian Fisk who got me access to the NVIDIA A100s at the Flatiron Institute/San Diego Supercomputing Center (SDSC) in New York, where I developed and profiled the key piece of this work.

Additionally, I would like to thank the people at the HPC-Lab for their ideas, conversations and enjoyable Zoom meetings during the pandemic. I would especially like to thank Øystein Krogstie for his important help and discussions.

Last but not least, I would like to thank my main advisor, Prof. Anne C. Elster for her invaluable support and insights. I would not have gotten the technical studentship at CERN or finished this thesis without her.

List of Abbreviations and Selected Terms

Callgrind: CPU Profiling tool that is part of Valgrind.

CERN: The European Organization for Nuclear Research whose name stems from the French acronym for 'Conseil Européen pour la Recherche Nucléaire'.

Device: NVIDIA GPU, as in device kernel: a function running on a GPU.

GPU: Graphics Processing Unit.

GPGPU: General-purpose computing on graphics processing units: GPUs that can be programmed in CUDA, OpenCL, HIP and other higher-level programming environments.

HEP: High-Energy Physics.

ICT: Information and Communications Technology.

KCachegrind: Tool to visualize results from Callgrind.

Kernel: Function running on a GPU.

LHC: Large Hadron Collider.

MG5aMC: Madgraph5_aMC@NLO. The version of the Madgraph code considered for this thesis.

ML: Machine Learning.

Nsight Compute: NVIDIA Profiling tool at kernel-level. Provides detailed performance details and graphs for given kernels.

Nsight Systems: NVIDIA Profiling tool at system-level. Has lot higher overhead than CPU profilers.

SM: NVIDIA term for Streaming Multiprocessor. Each NVIDIA GPU has an array of SMs, each with several types of computational cores, register files etc.

Tensor Cores: Special GPU processing cores on available on Volta and Ampere architecture GPUs, among others, that target small matrix-operations common in Machine Learning.

Valgrind: Memory debugging tool and more.

Contents

Project Description . . . . . **iii**
Abstract . . . . . **iv**
Sammendrag . . . . . **v**
Acknowledgements . . . . . **vi**
List of Abbreviations and Selected Terms . . . . . **vii**
Contents . . . . . **ix**
Figures . . . . . **xi**
Code Listings . . . . . **xii**
1 Introduction **1**
1.1 The CERN Computing Challenge 2
1.2 Benchmarking of Hardware Resources 3
1.2.1 The MadGraph Workload 3
1.3 Goals and Contributions 4
1.4 Thesis Outline 4
2 Background **6**
2.1 CUDA and NVIDIA GPUs 6
2.1.1 CUDA Warps 7
2.1.2 Dynamic Parallelism in CUDA 8
2.1.3 Tensor Cores 8
2.2 Madgraph 11
2.2.1 The Madgraph Project 11
2.2.2 The GPU Port 12
3 Choosing the Workload and Profiling it on CPU and GPU **14**
3.1 Choosing the Workload to Optimize 14
3.2 Becoming Familiar with the Workload 15
3.3 CPU Profiling 15
3.4 GPU Profiling 16
3.4.1 Nsight Systems 17
3.4.2 Nsight Compute 18
4 Implementation and Results on the A100 GPU **20**
4.1 Planning and Testing 20
4.2 The First Implementation 21
4.3 The Final Implementation 22

5 Conclusion **25**
5.1 Future work 26
Bibliography **27**
A Benchmarking and Containers **30**
A.1 Containers 30
A.1.1 Docker versus Singularity 30
A.1.2 Usage and build 31
A.2 Containers and GPUs 31
B Code Implementations **32**
B.1 Code Used for Testing 32
B.1.1 Single-precision Tensor Core 32
B.1.2 Blocking Algorithm Testing 33
B.1.3 Complete Double-Precision Test Code using Tensor Cores . . 35
B.1.4 Makefile 39
B.2 The first implementation in Madgraph 39
B.3 The final implementation in Madgraph 41
C CERN openlab Technical Workshop Presentation **42**

Figures

2.1 Example of architecture differences of CPUs and GPUs. 6
2.2 GA100 7
2.3 A100-SM 9
2.4 Lagrangian of the Standard Model 12
2.5 Feynman diagram 12

3.1 Valgrind call from command line 16
3.2 Call graph of the CPU version Madgraph 16
3.3 Nsight Systems call from command line. 17
3.4 Complete result of profiling on Nsight Systems 17
3.5 Zoomed in result of profiling on Nsight Systems 18
3.6 Nsight Compute call from command line. 18
3.7 Profiling with Nsight Compute showing the SM vs. Memory usage . 18
3.8 Profiling with Nsight Compute showing the pipelines 19

4.1 The blocking of A and B 22
4.2 Profiling of the `sigmaKin_goodhel` kernel 23
4.3 Profiling of the `sigmaKin` kernel 24

Code Listings

4.1 Blocking algorithm used in the first implementation 21
B.1 Test code for a single-precision Tensor Core matrix multiplication using 16*16 size matrices. 32
B.2 Test code to validate the blocking algorithm for both one- and two dimensional matrices, using colour matrix from Madgraph. 33
B.3 Complete test code using colour matrix from Madgraph doing blocking, padding and Tensor Core matrix multiplication. 35
B.4 Makefile for running the code in B.1 and in B.3. 39
B.5 Set up code with blocking and padding 39
B.6 Tensor Core kernel from the naive implementation 41

Chapter 1

Introduction

This thesis is the product of a collaboration between the Norwegian University of Science and Technology (NTNU) and CERN Openlab. The European Organization for Nuclear Research, more commonly known as CERN, is an international organization whose main area of research is particle physics. CERN Openlab facilitates collaborations between CERN and both leading companies in information and communications technology (ICT) and research institutes, to tackle major challenges CERN faces in its mission. During the work of this thesis the author worked as a technical student at CERN Openlab.

With the limits of computer speed gained from frequency scaling hitting its limits back in 2005 (know as the power wall) it more important than ever that computational intensive applications, such as the ones used at CERN, to take advantage of the parallel computing platforms available.

Graphical Processing Units (GPUs) offer thousands of computational cores per chip. The GPUs are specialized hardware units designed to perform operations on data in parallel. The technology was originally used to offload the graphics processing from the CPU and early GPUs could perform the most common graphical computations. In recent years however, as they have become more programmable, GPUs have been found to be useful for other types of calculations beyond graphics that are highly data parallel. This was the birth of General-Purpose Graphics Processing Unit (GPGPU) computing and the GPUs are programmable instead of fixed-function. Several vendors offers GPGPUs and there are multiple frameworks, that are either cross-platform or vendor specific, to program them. One of these is the NVIDIA´s CUDA programming environment described in more details in the next chapter.

Students at the HPC-Lab (Heterogeneous and Parallel Computing Lab) at NTNU have been working on GPU computing since 2006 [1] , and continue to look at performance modeling and finding new and better ways to take advantage of the computational power of the GPUs as well as multi-core systems[2, 3].

We are particularly interested in exploring new GPU features for HPC workloads. As GPUs now are have become popular computing platforms for artificial intelligence, and in particular machine learning (ML), NVIDIA and other GPU vendors have added special compute cores, known as tensor cores, that target the many small matrix-vector operations associated with ML-based tasks. How to take advantage of these tensor cores for non-ML work loads is still a formidable challenge, and one that will be explored in this thesis work.

1.1 The CERN Computing Challenge

CERN operates the Large Hadron Collider (LHC), a 27-kilometre ring consisting of superconducting magnets, which is the largest and most powerful particle accelerator in the world. Inside the LHC, two particle beams travel in opposite directions at almost the speed of light and are then made to collide in a detector at an experiment. As many as one billion particle collisions can happen inside an experiment's detector in a second. Automatic triggers pick out potentially interesting events, but even with this reduction to a fraction of the data, the CERN data centre still record approximately 1 GB/s of data as of the second period of particle collision production, known as Run 2 [4].

However, experimental programs and data are only part of what CERN concerns itself with. Exploring High-Energy Physics (HEP) starts with theories and simulations and one of the steps is event generation. Event generators are programs that generates events as those produced by particle accelerator collider experiments and they help bridge the gap between theoretical calculations and the complex detector signatures and data.

After 2025, the upgraded version of the LHC, the High Luminosity LHC (HL-LHC), is planned to come online for Run 4 and it is expected that the compute requirements will be 50-100 times greater than today [5], as it requires more complicated and accurate analysis of both the experimental and generated data.

To satisfy the need for storage, distribution and analysis of all the data, the Worldwide LHC Computing grid (WLCG) was created in 2002. It consists of about 900 000 computer cores in 170 facilities in 42 countries [6]. The increasing interest and demand for more cost-effective solutions for the same amount of processing power, has caused data centres to gradually offer more heterogeneous computing resources, such as GPUs . Currently these resources are under-utilized by the LHC experiments.

In 2017, CERN openlab published a white paper detailing the major ICT challenges CERN and others faces in the coming years [7] and their possible solutions. In R&D Topic 2 they identify, among other things, utilizing heterogeneous hardware, such as GPUs, to close the resource gap: the difference between resources needed and resources available. By utilizing GPUs CERN may speed up

their workloads, but also utilize all the hardware they have available through the WLCG.

1.2 Benchmarking of Hardware Resources

Evaluation of hardware resources date back to the 1980's with the "CERN Unit". Workloads used in that benchmark were derived from applications used in HEP computing at that time. In the 1990's it was replaced by benchmarks based on the SPEC benchmarking suite, as it was shown that the benchmarks adequately represented the computing patterns of the HEP applications. However, in the last years the current benchmark HEP SPEC 2006, a subset of the SPEC CPU 2006 benchmarking suite, has been shown to lack correlation with today's HEP applications. This has been found to also apply to the new version, the SPEC CPU 2017 benchmarking suite after being analyzed by the HEPiX BWG [8]. The BWG has therefore been working on replacement benchmarking suite based on actual HEP workloads, and which is also representative of the evolving hardware landscape, such as the increasing heterogeneity of data centres. Of particular interest is therefore codes related to experiments at CERN that take advantage of modern GPUs.

1.2.1 The MadGraph Workload

The different workloads from the LHC experiments include several different steps, such as event generation, simulation, digitization and reconstruction. The work in this thesis concerns itself with the event generation(GEN) step. GEN workloads represent a major fraction of total CPU time for CERN workloads, which is expected to increase in the future with the HL-LHC, as more accurate theoretical predictions are required for physics analysis [9].

The work of this thesis will focus on the MadGraph5_aMC@NLO workload (hereafter referred to as MG5aMC), a physics event generator. It is currently being ported to GPU, as part of a project that aims to utilize modern hardware architectures to optimize the code for both CPU and GPU. It is considered well suited for a GPU port, as it consists of highly-parallelizable tasks that usually are computationally intensive depending on the particles involved.

So far, the GPU version of the workload is a simple conversion of some of the parts of the CPU version. Many of the calculations in the code involve double precision complex numbers, over-utilizing the FP64 pipeline in the GPU, while under-utilizing the others. The work of this thesis will therefore focus on investigating whether the code can be optimized by using the new NVIDIA A100 GPU, which offers 64-bit floating point Tensor Cores, and if so, integrate the use of them the into the Madgraph GPU code.

1.3 Goals and Contributions

The goals of this thesis are to:

- Evaluate whether 64-bit floating point NVIDIA Tensor Cores can be used to optimize the Madgraph4GPU code.
- Implement the usage of Tensor Cores in the Madgraph4GPU code
- Evaluate the effect of the usage of the Tensor Cores on the code

We also started to look at containerization of the application, but did not complete this task due to time constraints and the desire to do a good analysis on the A100 given that we were able to test them on the SDCS/Flatiron Institute resources. However, some notes on containerization by the author is included in Appendix A.

This main contributions from this thesis are:

- An example software implementation of matrix-vector multiplication using NVIDIA Tensor Cores, including blocking and padding of the matrix and vector.
- A software implementation using NVIDIA Tensor Cores to do the matrix-vector multiplication involving color flow physics in Madgraph4GPU, including blocking and padding of the matrix and vector.

1.4 Thesis Outline

The rest of the thesis consists of the following chapters:

- **Chapter 2: Background** – give some basic introdiction to CUDA, Tensor Cores and the main compute aspects of the Madgraph function analyzed

- **Chapter 3: Methodology** describing how we profiled and optimized the MadGraph kernal for on V100 and A100 GPUS, including utilizing Tensor Cores.

- **Chapter 4: Benchmarking and Results**

- **Chapter 5: Conclusions and Future Work**

- **Appendix A: Containers** – some background and thoughts containers.

- **Appendix B: Main Code Listings**

- **Appendix C: OpenLab Workshop talk by author** – Slides of the small

presentation given by the author on March 9 at the CERN openlab Technical Workshop.

Chapter 2

Background

This chapter gives an overview of GPUs, specifically NVIDIA GPUs and the computing unit called a Tensor Core, and an overview of the MG5aMC software, and current efforts to port it to GPU.

Figure 2.1illustrates the architectural differences between CPUs and GPUs, where the CPU has fewer more complete cores with their own caches whereas modern GPUs consist of several cores sharing caches and executed instructions in chunks of 32 threads using 32 cores, called warps, and are as such more similar to vector units.

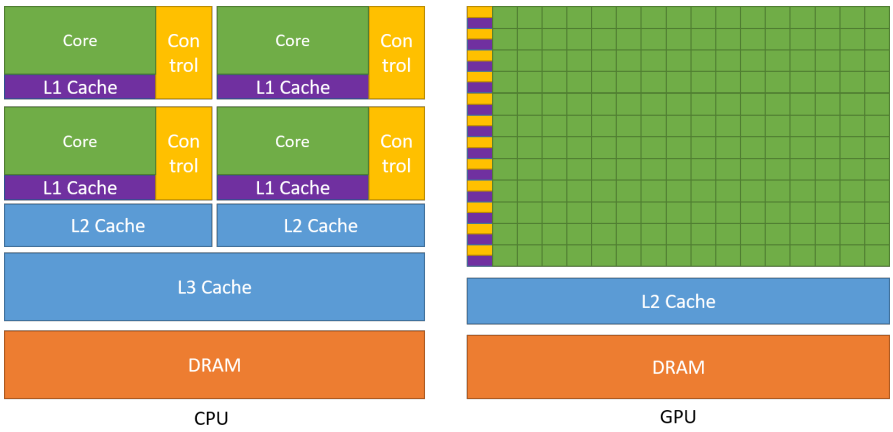


Figure 2.1: Example of architecture differences of CPUs and GPUs [**CUDA-guide**]. Figure used with permission from NVIDIA.

2.1 CUDA and NVIDIA GPUs

Compute Unified Device Architecture (CUDA) is NVIDIAs parallel computing platform and application programming interface (API). It makes it possible to use

CUDA-enabled GPUs for general purpose processing. CUDA comes with a software environment allowing it to be used with with high-level programming languages such as C ior C++ [10].

CUDA is a heterogeneous programming model that use both the CPU and the GPU. In CUDA, `host` refers to the CPU and its memory, while `device` refers to the GPU and its memory. The code that is running on the host can launch kernels, which are functions that are executed in parallel on the GPU by many threads. To determine how many device threads execute the kernel, the execution figuration is set when launching with two arguments. The first argument determines how many thread blocks in the grid, while the second specify how many threads in a thread block. Grid size multiplied with block size gives the total amount of threads. It is also possible to launch the grid and thread blocks in three dimensions.

Modern NVIDIA GPU consists of an array of Stream Multiprocessors (SMs), Figure shows Figure 2.2 shows an illustration including the 128 SMs of the NVIDIA GA100 Architecture. The A100s used in this thesis has 108 SMs, whereas the smaller GPUs used for embedded applications, such as the Jetson TX2 have only two Pascal Streaming Multiprocessors (SMs) with 128 CUDA cores each.

Figure 2.2: GA100 Full GPU with 128 SMs. The A100 Tensor Core GPU has 108 SMs From https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/ Figure is used with permission from NVIDIA.

2.1.1 CUDA Warps

CUDA GPUs seamlessly schedule groups of 32 threads, known as **w**arps that execute the same instructions at the time, following the SIMT (Single Instruction Multiple Treads) model. Each SM may schedule up to 4 warps at a given time. We will use this concept of warps throughout this thesis.

2.1.2 Dynamic Parallelism in CUDA

NVIDIA GPUs running CUDA date back to 2006. The early GPUs up through the NVIDIA Fermi GPU architectures limited all kernel launches from the CPU – that is, the GPU acted as a co-processor where its CPU host generated the work (all function calls).

CUDA 5.0 and the Kepler architecture introduced the concept of **d**ynamic parallelism where the GPUs can themselves generate more work. This avoids having to statically allocate worst-case grids and lets the user map the compute to the problem. By letting the GPU threads spawn more threads, and is thus general useful for problems that require nested parallelism, such as generating the Mandelbrot set.

See: https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/

and

http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0338-GTC2012-CUDA-Programming-Model.pdf

This dynamic parallel feature will also be used in connection with taking advantage of the tensor cores available on recent GPUs. Tensor cores will be described in more details in the next subsection

Further details on the CUDA programming environment can be found in the CUDA programming guide, currently at 11.4 available in PDF here:

https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

Other recent CUDA features provided by the CUDA 11 introduced in 2020 include the Compute-Sanitizer tool that replaces the `cuda-memcheck` tool and support for the A100 tensor cores. For details on the Compute-Sanitizer tool, see

https://docs.nvidia.com/cuda/sanitizer-docs/SanitizerApiGuide/index.html

2.1.3 Tensor Cores

As mentioned in the introduction, recent GPUs have Tensor Cores that can perform fast matrix tailored for Machine Learning (ML), but that also may provide useful for general HPC loads that can leverage matrix operations. Unlike regular CUDA cores that may execute one operation per cycle, the tensor core are executed at multiple operations per cycle, a bit like CPU vector processors.

The A100 SM architecture including the Tensor Cores, as well as the other compute units and memory structure is shown in Figure 2.3. Note that each SM has 4 warp schedulers so that each part of the SM can stay busy,

A summary of the main related features of the three most recent GPU architectures from NVIDIA is shown in table 2.1

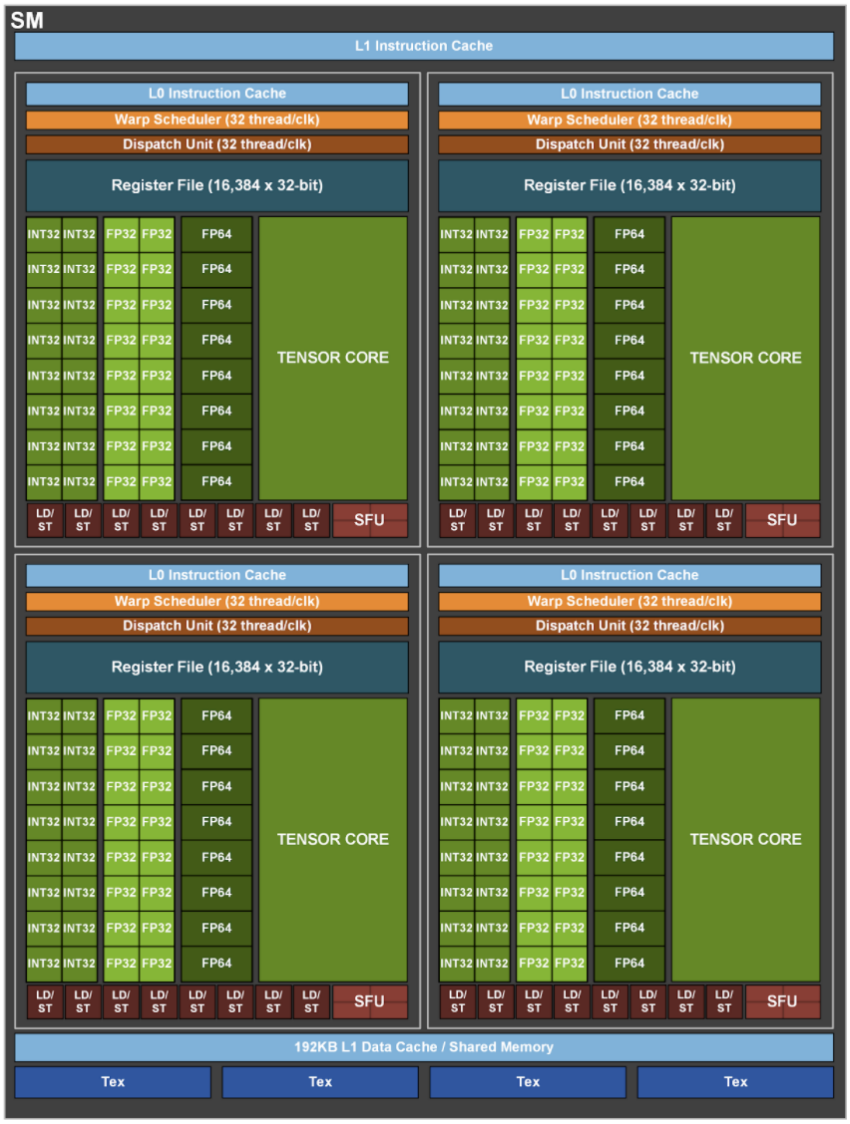


Figure 2.3: The A100 SM architecture
From https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
Figure is used with permission from NVIDIA.

Table 2.1: Main features and Compute Capabilities of recent NVIDIA GPUs (P100, V100 and A100) for HPC
Based on figures in
https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

Data Center GPU	**NVIDIA Tesla P100**	**NVIDIA Tesla V100**	**NVIDIA A100**
GPU Codename	GP100	GV100	GA100
GPU Architecture	Pascal	Volta	Ampere
Compute Capability	*6.0*	*7.0*	*8.0*
GPU Board Form Factor	SXM	SXM2	SXM4
SMs	56	80	108
FP32 Cores / SM	64	64	64
FP32 Cores / GPU	3584	5120	6912
FP64 Cores / SM	32	32	32
FP64 Cores / GPU	1792	2560	3456
Tensor Cores / SM	NA	8	42
Tensor Cores / GPU	NA	640	432
GPU Boost Clock	1480 MHz	1530 MHz	1410 MHz
Peak FP16 Tensor TFLO	NA	125	312/6243
Peak FP16 Tensor TFLO	NA	125	312/6243
Peak BF16 Tensor TFLO	NA	NA	312/6243
Peak TF32 Tensor TFLO	NA	NA	156/3123
Peak FP64 Tensor TFLO	NA	NA	19.5

In C++ CUDA, Warp matrix operations may use Tensor Cores to accelerate matrix calculations on the form $M = A * B + C$. The calculations requires the cooperation of all the threads in a warp and supports mixed precision floating point data for devices with compute capability 7.0 or above. The NVDIA Tesla V100 has compute capability 7.0, while the A100 has compute capability 8.0. By including the namespace `nvcuda:wmma` you get access to the useful functions and types.

2.2 Madgraph

This section gives a brief introduction to the MG5aMC software and the ongoing efforts to port it to GPU. The sections is based on the paper submitted to 25th International Conference on Computing in High-Energy and Nuclear Physics and more details about the project, physics background, design and implementation can be read in the paper [9].

2.2.1 The Madgraph Project

MG5aMC is a software framework for physics event generation used in the data processing workflow in HEP experiments, such as ATLAS[1] and CMS[2] at the LHC. Event generators are one of the large consumers of CPU time and their computational cost is predicted to increase with the High Luminosity upgrade of the LHC, as more accurate theoretical predictions are required. To mitigate this expected increase in resource usage, there is an ongoing work to re-engineer the Madgraph software to exploit modern hardware capabilities, such as CPUs with multiple cores and wide vector registers.

CERN relies on the WLCG and in recent years many HPC centers have deployed heterogeneous systems where the main computing power comes from GPUs. These resources are currently under-utilized by the LHC experiments and the current production version of MG5aMC is only developed to run on CPUs. There has been previous work to port components of Madgraph to CUDA, but the effort never reached production quality [11–15]. Therefore, together with improving the code on modern CPU architectures, the project also includes the effort to port MG5aMC to GPU. The project includes several different ports to run on NVIDIA, AMD and Intel in several different languages and abstract layers, such as HIP, Alpaka, Kokkos and SYCL. This thesis will focus on the C/C++ CUDA implementation.

MG5aMC acts as a unified framework to provide all the elements necessary for the study of HEP collision processes, both within the Standard Model and beyond[3]. It is designed as Python meta-code, that automatically generates the source code for the relevant process. The source code can be generated in C++ and Python, however the default is Fortran, which is used in production.

[1] https://atlas.cern/
[2] https://cms.cern/
[3] https://home.cern/science/physics/standard-model

$$\begin{aligned}\mathcal{L} = & -\frac{1}{4} F_{\mu\nu} F^{\mu\nu} \\ & + i\bar{\psi} \not{D} \psi \\ & + \psi_i y_{ij} \psi_j \phi + h.c. \\ & + |D_\mu \phi|^2 - V(\phi)\end{aligned}$$

Figure 2.4: A simplified Lagrangian representing the Standard Model of physics

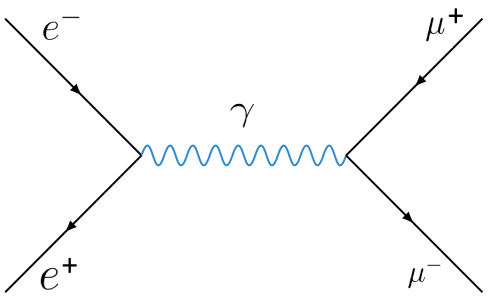


Figure 2.5: A Feynman diagram showing a photon exchange

The code-generation part takes a given process and the corresponding Lagrangian describing the physics model as input, determines the relevant Feynman diagrams, which describe the behaviour and interactions of subatomic particles, and generates the source code, which can then be compiled and run. An example of a Lagrangian can be seen in Figure 2.4 and an example of a Feynman diagram can be seen in Figure 2.5.

Some of the processes involve quarks. In quantum chromodynamics quarks have an attribute called color, that can have three values. The more particles in the final state of a process, the larger the pre-computed 'color matrix' involved, so speeding up this part of the code may not show up significantly in this test case, but can give a significant performance boost as this code develops. It has been shown that colour algebra can take up to 60% of the computational time [16].

2.2.2 The GPU Port

MG5aMC is an good candidate for a simple GPU port as almost all the numerical operations for transforming the input data to the output data are the same for generated events in the same sub-process. Because of the data parallelism, the code can be implemented using lockstep processing on GPU. Therefore the engineering effort so far has mostly focused on implementing every data processing step into a function that can operate on multiple events.

In the CUDA implementation, each event is processed by one thread. When launch-

ing the program, three command-line inputs are given: the number of blocks per grid, the number of threads per block and the number of iterations. The number of events generated in one iteration will therefor be the product of the numbers of blocks per grid and the number of threads per block and the product of this number and the number of iterations gives the total number of events generated.

The code can be run with double or single precision. Double precision is the default, as it follows the choice of the production Fortran version to achieve the required physics accuracy. Single precision was implemented to allow the code to run on consumer grade GPUs, but it also allows for a much higher computing performance when run on GPUs in data centres, as the GPUs usually have a lot more single precision cores than double precision cores.

The GPU version currently uses the NVIDIA cuRAND library for random number generation for the MC integration. It produces pseudo-random numbers, which ensures a deterministic output. It is useful for development, but will eventually be switched out for an algorithm that has been validated for LHC physics.

Chapter 3

Choosing the Workload and Profiling it on CPU and GPU

The basic idea for the thesis was to utilize some feature of the new NVIDIA A100 GPU in collaboration with CERN. So, the first step was to find a suitable workload that could benefit from features offered by the A100. This process will be described in the following section. The last two sections describes how we profiled the selected application kernel on CPU and GPU, respectively.

3.1 Choosing the Workload to Optimize

Most of the production code from CERN are huge programs, that has been development and maintained for many years and very few of those are programs utilizing GPUs. The challenge here was to find a GPU workload that didn't have an advanced setup or required a lot of knowledge to run it, but also not a too simple program, with no real practical application or that it could not benefit from the advanced features the A100 offers.

The first two candidates were Patatrack and Simpletrack. While they both have 'tracking' in their name, they deal with two different types of 'tracking' in the LHC. The Patatrack workload [17] is part of the reconstruction step. It reconstructs events chosen by a software trigger by 'tracking' or reconstructing the path of particles after a collision happens in the CMS experiment [18]. Simpletrack [19] is a simplified version of SixTrack and part of the simulation step. Simpletrack/SixTrack tracks the long-term path of a particle through a high-energy ring, such as the LHC [20].

After discussing both workloads with the programmers behind them, the conclusion was that Patatrack was possible to use, but rather complicated to set up and run. Meanwhile, Simpletrack was not very complicated, but because it was a simplified version of the real algorithm, it didn't have as much practical application

for CERN in its current version.

Another candidate was Madgraph4GPU [21]. After looking at the code and discussing with developers, it seemed to be the best choice as it was under active development, had a minimal set up and was easy to run, was computationally complex enough and had a real practical application. It did not come with a detailed documentation, but it was possible to have active communication with the developers which mitigated the problem a bit.

3.2 Becoming Familiar with the Workload

After deciding on the Madgraph workload, I needed to familiarize myself with the code, which was not trivial, as large parts of it was auto-generated and thus not as easy to read for humans. The developers explained their workflow of auto-generating code by using the MG5aMC version to generate it and then they modified this code to use CUDA and then back-ported this to the MG5aMC and generated a new version, with the new changes, which they could further work and improve on. So, to become familiar with the code, but also at the same time try to discover hot-spots in it that could be optimized, I started with profiling both the CPU and the GPU version of the auto-generated code.

My findings and experiences from this part of the work for this thesis became a presentation that I held at the CERN openlab Technical Workshop [22] and can be seen in appendix C, and my contribution to GPU part of the paper: HEPiX benchmarking solution for WLCG computing resources [23].

3.3 CPU Profiling

I started by profiling the generated C++ CPU code from epoch 0, before any changes were introduced to the code, with Valgrind. Valgrind was originally a memory debugging tool, but today also provides profiling tools such as Callgrind. I then used KCachegrind to visualize the results from Callgrind. When running the code, the number given as the argument decides how many events are generated. I started by creating only one event, then ran the code several times and increased the number of events by a factor of 100 until the call graph stabilised at 1 million events.

The command line calling sequence for Valgrind is shown in Figure 3.1 and the result of that run is showed in figure 3.2.

Figure 3.2 shows that there are two main parts of the program: `get_momenta` and the functions it calls and `sigmaKin` and the function it calls. Called functions are showed as an arrow from the function that calls it to the function. Both main functions are called 1 million times, the same amount as the number of events generated.

Figure 3.1: Valgrind call from command line

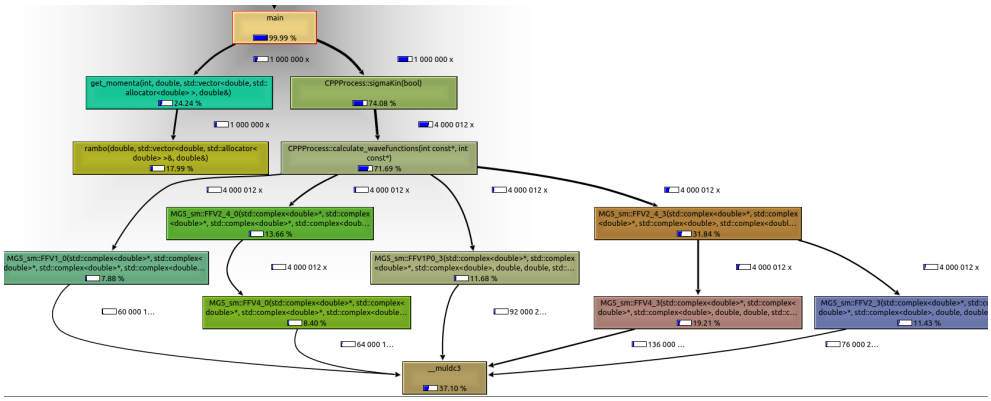


Figure 3.2: The call graph of the CPU version of Madgraph generating 1 million events, generated by the tool Callgrind in Valgrind visualized using Kcachegrind.

A number in percent is given for each of the functions: this shows the amount of time spent in that function and the functions it calls, in relation to the time of the entire program. For instance, `sigmaKin` uses 74% of the time, but calls `calculate_wavefunction` which uses 71% of the time. This means that only roughly 3% is actually spent in `sigmaKin`. All the functions pre-fixed by `MG5_sm` are auto-generated functions that depend on what process was chosen when generating the code. `_muldc3` is a C++ function that is used for multiplying floating point complex numbers.

By running and profiling the CPU code, I discovered the structure of GitHub repository, how to run the desired code version with different number of events, the structure of the code and certain characteristics to look for in the GPU code. The GPU version is similar in structure, but also includes the CPU version in the same files split by pre-processor variables.

3.4 GPU Profiling

I then moved on to profiling the GPU version of the code. Here I chose to base the profiling on epoch 1, as development had moved on to epoch 2, and epoch 1 was mostly stable. I used Nsight Systems for system-level profiling and Nsight Compute for kernel-level profiling. The profiling was done on a V100 connected

to the host over a PCI Express link.

3.4.1 Nsight Systems

Profiling with Nsight Systems incurred a much larger overhead than the profiling with Valgrind and Callgrind had done, and I could therefore only run with 12 iterations. However, this was representative enough. The command line call is shown in figure 3.3.

Figure 3.3: Nsight Systems call from command line.

The argument given to the CPU program both indicated the number of iterations and the number of events generated. Every CPU iteration, generated one event, however on GPU each thread will each generate one event times the number of iterations. I profiled the $e^+e^- \to \mu^+\mu^-$ processes from epoch 1. Figure 3.4 shows the complete result from profiling and it is possible to see the 12 iterations as the 12 vertical pink lines in the figure. Figure 3.5 shows a zoomed in version on one of the iterations.

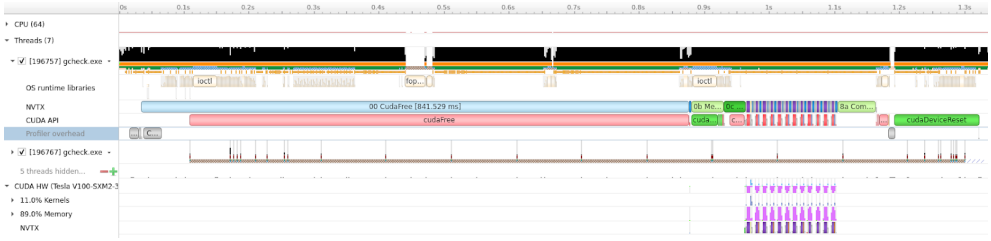


Figure 3.4: The complete result of profiling the $e^+e^- \to \mu^+\mu^-$ process on Nsight Systems.

The 3.5 figure shows that most of the time is spent on memory transfer, which is the long horizontal pink line. The short blue lines are the kernels. So for the $e^+e^- \to \mu^+\mu^-$ process, the calculations done are quite simple, compare to the time it takes to do the memory transfer. However, I know that the $gg \to t\bar{t}gg$ process is being developed in epoch 2, using the same kernels. So, I choose to look closer at the largest kernel, which is SigmaKin, which will also be relevant for the more advanced processes.

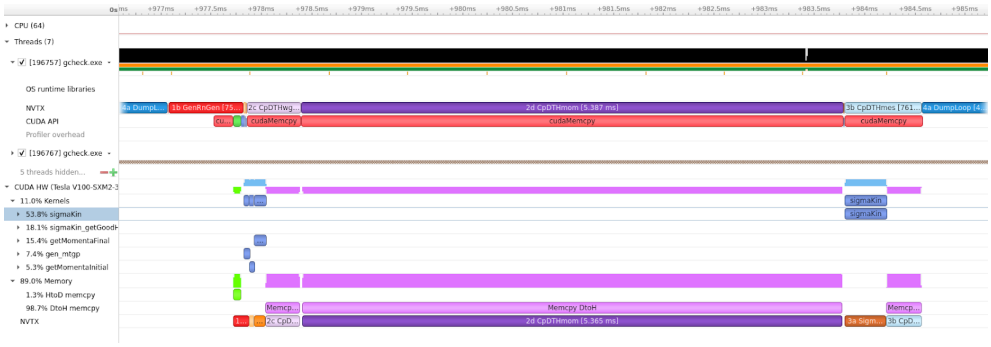


Figure 3.5: The result zooming in on one of the iterations of the $e^+e^- \rightarrow \mu^+\mu^-$ process on Nsight Systems.

3.4.2 Nsight Compute

With Nsight Compute I only ran 1 iteration, as the profiler itself runs the kernel enough times to get a stable result. The command line call is shown in 3.6.

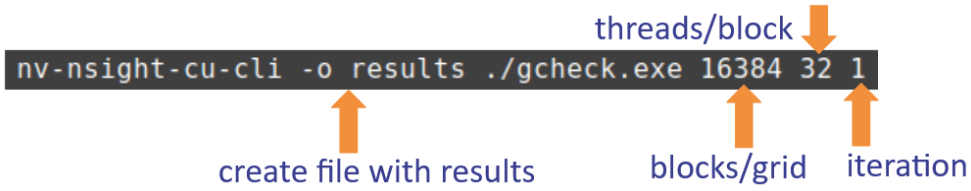


Figure 3.6: Nsight Compute call from command line.

From the profiling done with Nsight Systems, I could see that the `SigmaKin` kernel was a hot-spot. I therefore continued by profiling it in Nsight Compute to look at it closer. Figure 3.7 and 3.8 show part of the result from the profiling. Clearly the Fp64 pipline heavily used compared to the other pipelines and the application is compute-bound not memory-bound.

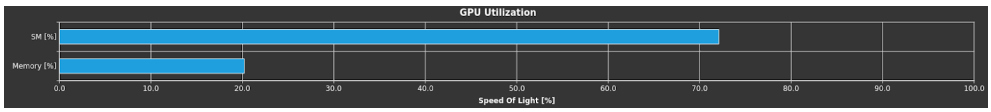


Figure 3.7: The result of the profiling with Nsight Compute showing the SM vs Memory usage.

After discussing my profiling of the code together with two of the developer of Madgraph4GPU, we identified a part of the code that is a potential computational hot-spot: a matrix-vector multiplication in the $gg \rightarrow t\bar{t}gg$ process. We came up with the idea to try to optimize this spot as it also takes up part of the FP64 pipeline. The idea was to use the Tensor Cores of the A100, as it has double precision floating cores in contrast to the V100 which only has single precision Tensor

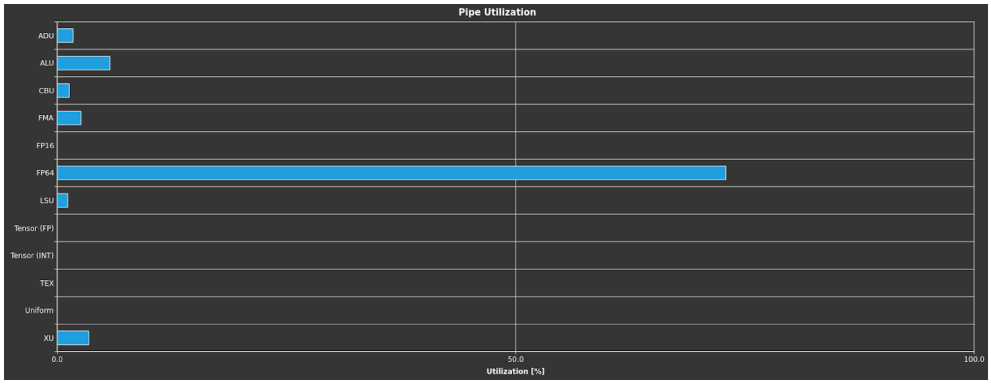


Figure 3.8: The result of the profiling with Nsight Compute showing the pipelines.

Cores. It is an ongoing discussion whether double precision is needed for accurate enough results, but to mitigate the restrictions of only doing calculations in double precision, the idea was to offload the calculations to the Tensor Cores as an alternative.

Chapter 4

Implementation and Results on the A100 GPU

This chapter details my implementation of the usage of Tensor Cores for the matrix-vector multiplication identified in the last chapter. The chapter includes details on my first, naive implementations, the problems with this and the improved and final implementation, as well as some details on the code I used to test features before implementing them in the Madgraph code. My implementations in their entirety can be found in appendix B.

4.1 Planning and Testing

The first step was to figure out if I could use a library that supported Tensor Cores for easier implementation. I looked into cuBLAS [24] and CUTLASS [25], however both need to be launched from host, while the current structure of the code required me to launch from within another kernel, called dynamic parallelism. It is a possible plan for the future to split the large kernels into smaller pieces and launch everything from host, however this was not in the scope of this thesis.

Since I could neither use cuBLAS or CUTLASS I used the `wmma` CUDA instructions. These provide memory structures, called fragments, that handle the alignment requirements for the matrices, as well as functions for executing the tensor cores. Using these are mostly just filling out C++ CUDA templates.

Tensor cores only perform one specific operation: $C = A*B+C$, where A, B and C are matrices and they only takes very specific matrix sizes as input. The available sizes depends on the precision. For double precision floating point there is only one choice. Matrix A must be 8*4, matrix B must be 4*8 and the accumulator C must be 8*8.

My initial implementation was a separate, standalone program using tensor cores,

to avoid complicating it with the matrix and vector from the Madgraph program. It was implemented in single precision, so that I would be able to test it on a V100 GPU, as I had easier access to V100s than A100s. The code can be seen in appendix B.1.

4.2 The First Implementation

The process $gg \to t\bar{t}gg$, has a fixed size colour matrix of size 24*24, this matrix is matrix A in my calculation. The Matrix B is not a matrix, but a column vector of size 24. To fit the matrix and the vector into the Tensor Cores, the vector needs to be padded and both need to be blocked. The vector is padded to 24*8 and the complex number is split up into the real and imaginary component. Real numbers were placed in the firs column and imaginary in the second, while the rest of the columns were filled up with zeros. The listing 4.1 shows the blocking algorithm and the figure 4.1 shows an illustration of how the Matrix A and B are blocked.

Code listing 4.1: Blocking algorithm used in the first implementation

```
int base_offset_a_row = 0;
int base_offset_a_col = 0;
int base_offset_a_block = 0;
int curr_offset_a_row = 0;
int curr_offset_a_col = 0;
int curr_offset_a_block = 0;
for (int blockrow = 0; blockrow < 3; blockrow++) {
for (int blockcol = 0; blockcol < 6; blockcol++) {
  base_offset_a_row = blockrow*8;
  base_offset_a_col = blockcol*4;
  base_offset_a_block = 32*(6*blockrow + blockcol);
  for (int row = 0; row < 8; row++) {
    for (int col = 0; col < 4; col++) {
      curr_offset_a_row = base_offset_a_row + row;
      curr_offset_a_col = base_offset_a_col + col;
      curr_offset_a_block = base_offset_a_block + row*4 + col;
      A_block[curr_offset_a_block] = cf[curr_offset_a_row][curr_offset_a_col];
    }
  }
}
```

After blocking into sub-matrices that fit into the tensor core requirements, the block matrices had dimensions of 3*6 and 6*1. The matrix blocks were stored row-major in memory, block by block. To calculate the matrix product of the two blocked matrices, 18 matrix multiplications were run on the tensor cores.

I elected to not use the accumulation function as that allowed me to run all 18 sub-matrix multiplications in parallel and store the results of the multiplications separately, sub-matrix by sub-matrix in row major order. All matrix multiplications were launched together, with one thread block with 32 threads each. This guarantees that every block can be executed as a warp, as required by the tensor cores.

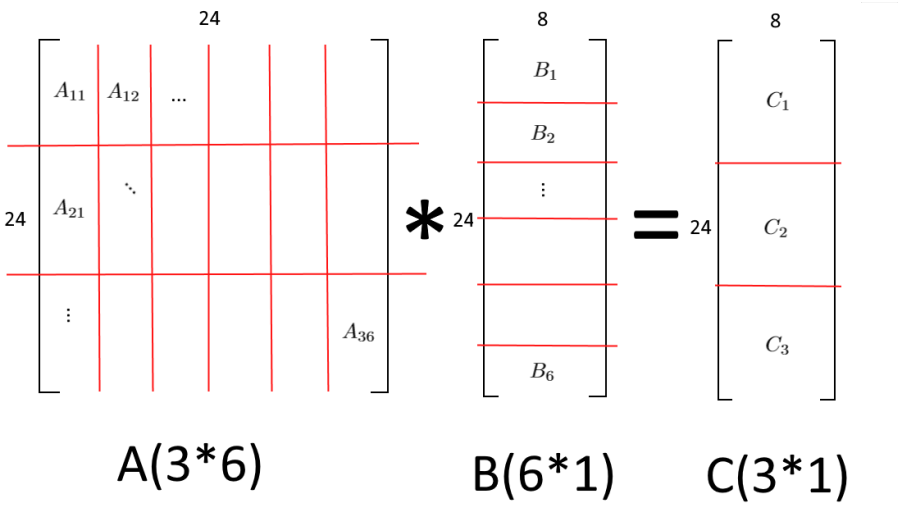


Figure 4.1: Illustration showing the blocking of Matrix A and B

After the tensor core kernel finishes, the sub-matrix multiplications are summed up into the final result block matrix of 3*1 blocks. Each of the blocks are the sum of six matrix multiplication results. From this result matrix, the real and imaginary numbers are extracted and put into a complex vector. This vector now contains the result of the full matrix-vector multiplication. Finally, the complex numbers are stored in the result vector are used to calculate `meHelSum`.

Since the Tensor Core kernel is called from another kernel, called dynamic parallelism, the files need to be compiled with the flags `-rdc=true` and `-lcudadevrt` to enable this.

To ensure that the code changes does not affect the calculations the output gives out an average value of the results that can be compared between the original and the modified code. Since the random number generator used in Madgraph is using a deterministic seed, equal runs should give equal results.

4.3 The Final Implementation

After developing and testing the first version, some issues materialized.

When allocating the new memory I needed in the GPU kernel, I allocated onto the heap, using `malloc`. The GPU only has 8MB of allocated heap memory by default [26] and could thus only support 11 blocks of 32 threads before memory was used up. The solution implemented was to stack allocate on the host before launching the kernel.

I also launched the Tensor Core kernels kernel by kernel in a for-loop. This was very inefficient, so I changed the launch parameter to launch all the kernel at the same time and removed the for-loop. Then I calculated the offsets inside the tensor Core kernel using `blockId`.

Another issue is that all thread blocks the same matrix. As they all write the same value, it is not a race condition. However, it is very inefficient and may also cause weird behaviour. The best solution would be to do the blocking in its own kernel launched from the host. Since restructuring the entire kernel was not in the scope of this thesis, this has not been implemented.

After implementing the improvements, the modified code was profiled using Nsight Compute and compared to the baseline Madgraph version. This is shown in figure 4.2 and 4.3. The code for the $gg \to t\bar{t}gg$ process is slightly different in structure than the code for the $e^+e^- \to \mu^+\mu^-$ process. The `sigmaKin` kernel is split into `SigmaKin_goodhel` and `SigmnKin`.

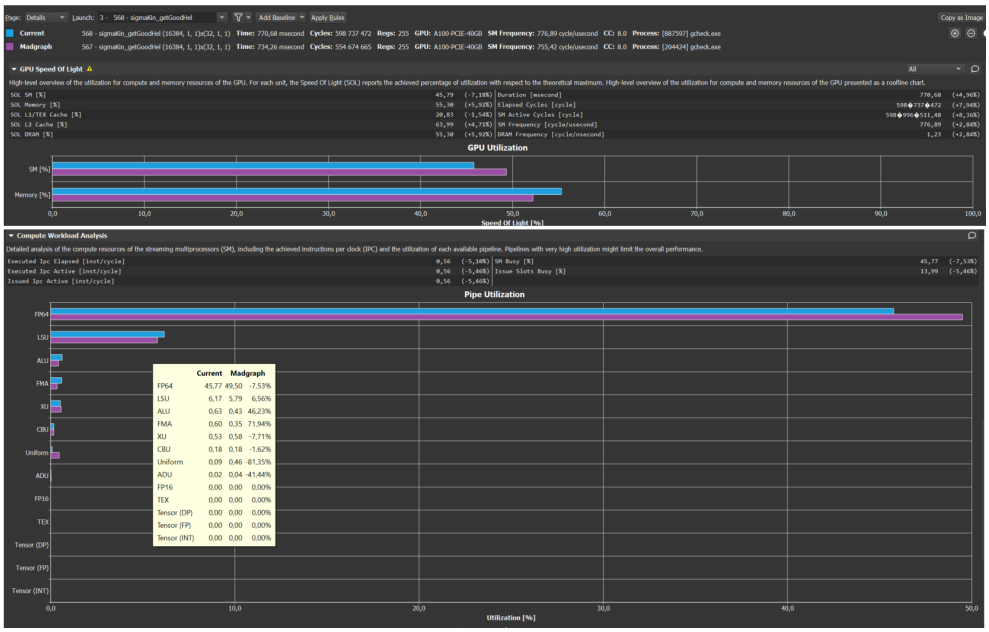


Figure 4.2: The profiling summary and the Compute Workload analysis of the `sigmaKin_goodhel` kernel of the original Madgraph code compared to the Tensor Core version of it

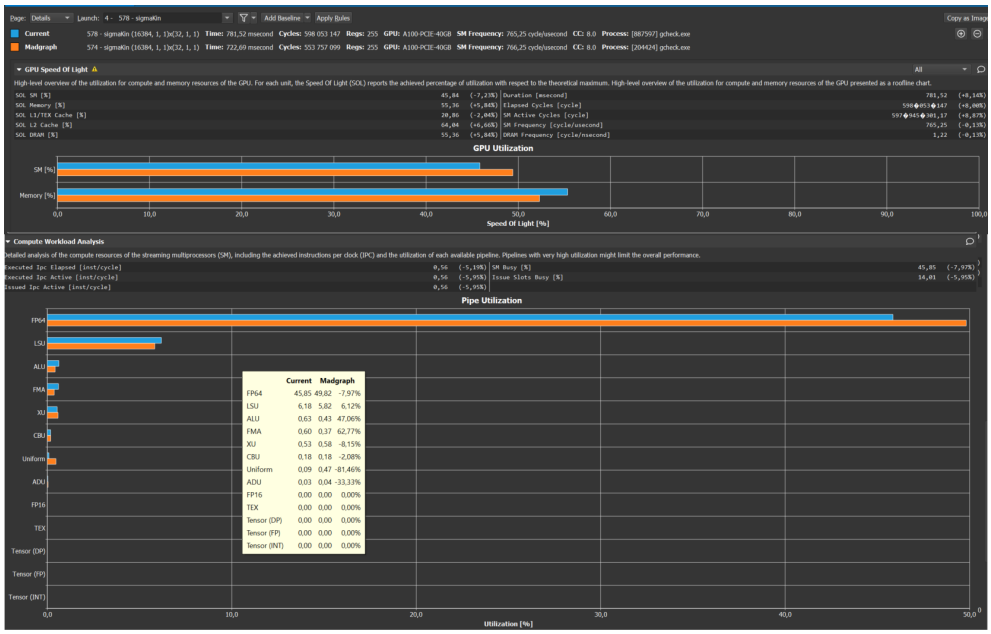


Figure 4.3: The profiling summary and the Compute Workload analysis of the `sigmaKin` kernel of the original Madgraph code compared to the Tensor Core version of it

Chapter 5

Conclusion

The computational demands of the future, is ever increasing, especially at CERN where they are working on utilizing modern hardware such as GPUs, both to speed up their workloads, but also to utilize all the hardware they have available through the Worldwide LHC Computing Grid (WLCG).

The work of this thesis sprung out from my technical student internship at CERNs openlab and focused on selecting a suitable code to work with with and to use new GPU features such as the A100's FP64 CUDA cores for speeding up the application.

After careful evaluations of several candidates, the choice landed on the MadGraph5_aMC@NLO workload, a physics event generator used at CERN. This application includes kernel that are expected to increase with future upgrades.

My work expanded on the the current GPU version of the workload which is a simple conversion of some of the parts of the CPU version. Many of the calculations in the code involve double precision complex numbers, over-utilizing the FP64 pipeline in the GPU, while under-utilizing the others. The work of this thesis therefore focused on investigating whether the code could be optimized by using the new NVIDIA A100 GPU, which offers 64-bit floating point Tensor Cores.

The thesis work also included extensive profiling using Valgrind and Kcachegrind on the CPU code as well as Nsight Systems and Nsight Compute on the GPU kernels.

The Tensor Cores does not show up in the profiling done with Nsight Compute. This is likely because the Tensor Core part is too small compared to all the FP64 calculations.

The Tensor Core code is likely slower, because it uses dynamic parallelism and because the setup introduces a bit of overhead. For more complicated processes with a larger pre-computed colour matrix, it might outweigh the overhead of using tensor Cores. In the future when the large kernels are split up, dynamic parallelism

can be removed and the code can then run faster, as more compiler optimizations are then available.

Ideas for future work are listed in the following chapter.

5.1 Future work

Following is a list of several items for future work:

- Generalise the Tensor Core kernel so it can be used for both single and double precision. Add pre-processors to check if the tensor core can be used on the architecture it is run on.

- Split up the large kernels, so that everything is launched from the host, to avoid the drawback of dynamic parallelism. Implement the blocking in its own kernel launched from host.

- Backport the Tensor Core Madgraph code to the MC5aMC generator. Are there any additional issues with the code, before this can be done?

- Share the blocked matrix between the Tensor Cores and FP64 cores. Find the ideal split on how much gets sent to the tensor Cores and how much gets sent to the FP64 cores This is possible with the way blocking done now.

Bibliography

[1] C. Larsen and C. A. Elster, *Utilizing gpus on cluster computers*, NTNU Master thesis pre-project, Dr. Anne C. Elster advisor, Tor Fevang, Schlumnberg, co-advisor, Dec. 2006.

[2] J. C. Meyer and A. C. Elster, 'Performance modeling of heterogeneous systems,' in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–4. DOI: 10.1109/IPDPSW.2010.5470682.

[3] L. Bjertnes, J. O. Tørring and C. A. Elster, 'Ls-cat: A large-scale cuda autotuning dataset,' *IEEE International Conference on Applied Artificial Intelligence (ICAPAI 2021)*, vol. 31, May 2021.

[4] *Processing: What to record?* https://home.cern/science/computing/processing-what-record, Accessed: 2021-02-27.

[5] *Computing*, https://home.cern/science/computing, Accessed: 2021-02-27.

[6] *The Worldwide LHC Computing Grid (WLCG)*, https://home.cern/science/computing/grid, Accessed: 2021-02-27.

[7] 'CERN Openlab white paper on future ICT challenges in scientific research,' CERN Openlab, Tech. Rep., 2017.

[8] Giordano, Domenico, Alef, Manfred and Michelotto, Michele, 'Next Generation of HEP CPU Benchmarks,' *EPJ Web Conf.*, vol. 214, p. 08 011, 2019. DOI: 10.1051/epjconf/201921408011. [Online]. Available: https://doi.org/10.1051/epjconf/201921408011.

[9] A. Valassi, S. Roiser, O. Mattelaer and S. Hageboeck, *Design and engineering of a simplified workflow execution for the mg5amc event generator on gpus and vector cpus*, 2021. arXiv: 2106.12631 [physics.comp-ph].

[10] *CUDA C++ Programming Guide, NVIDIA*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Accessed: 2021-07-05.

[11] K. Hagiwara, J. Kanzaki, N. Okamura, D. Rainwater and T. Stelzer, 'Fast calculation of helas amplitudes using graphics processing unit (gpu),' *The European Physical Journal C*, vol. 66, no. 3-4, pp. 477–492, Mar. 2010, ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-010-1276-8. [Online]. Available: http://dx.doi.org/10.1140/epjc/s10052-010-1276-8.

[12] K. Hagiwara, J. Kanzaki, N. Okamura, D. Rainwater and T. Stelzer, 'Calculation of helas amplitudes for qcd processes using graphics processing unit (gpu),' *The European Physical Journal C*, vol. 70, no. 1-2, pp. 513–524, Oct. 2010, ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-010-1465-5. [Online]. Available: http://dx.doi.org/10.1140/epjc/s10052-010-1465-5.

[13] K. Hagiwara, J. Kanzaki, Q. Li, N. Okamura and T. Stelzer, 'Fast computation of madgraph amplitudes on graphics processing unit (gpu),' *The European Physical Journal C*, vol. 73, no. 11, Nov. 2013, ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-013-2608-2. [Online]. Available: http://dx.doi.org/10.1140/epjc/s10052-013-2608-2.

[14] J. Kanzaki, 'Monte carlo integration on gpu,' *The European Physical Journal C*, vol. 71, no. 2, Feb. 2011, ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-011-1559-8. [Online]. Available: http://dx.doi.org/10.1140/epjc/s10052-011-1559-8.

[15] J. Kanzaki, 'Application of graphics processing unit (gpu) to software in elementary particle/high energy physics field,' *Procedia Computer Science*, vol. 4, pp. 869–877, 2011, Proceedings of the International Conference on Computational Science, ICCS 2011, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2011.04.092. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050911001505.

[16] K. Ostrolenk and O. Mattelaer, *Speeding up madgraph5_amc@nlo*, 2021. arXiv: 2102.00773 [hep-ph].

[17] *Patatrack – Git repository*, https://github.com/cms-patatrack, Accessed: 2021-07-05.

[18] A. Bocci, V. Innocente, M. Kortelainena and M. R. F. Pantaleo1, *Cms patatrack project*, https://indico.cern.ch/event/759388/contributions/3303050/attachments/1814366/2964775/slides_mk_patatrack_20190319.pdf, 2019.

[19] *Simpletrack – Git repository*, https://github.com/rdemaria/simpletrack, Accessed: 2021-07-05.

[20] *SixTrack - 6D tracking Code – Git repository*, http://sixtrack.web.cern.ch/SixTrack/, Accessed: 2021-07-05.

[21] *Madgraph4GPU – Git Repository*, https://github.com/madgraph5/madgraph4gpu, Accessed: 2021-07-05.

[22] I. Brevik Høgstøyl, *Profiling code on nvidia gpus*, Presentation at CERN OpenLab Workshop March 9, 2021. Attached as Appendix C.

[23] M. F. Medeiros, M. Alef, L. Atzori, J.-M. Barbet, I. B. Høgstøyl, O. Datskova, R. De Maria, D. Giordano, M. Girone, C. Hollowell *et al.*, 'HEPiX benchmarking solution for WLCG computing resources,' [Online]. Available: http://heprcdocs.phys.uvic.ca/papers/CPU_Benchmark_CHEP2021.pdf.

[24] *cuBLAS*, https://docs.nvidia.com/cuda/cublas/index.html, Accessed: 2021-07-05.

[25] *CUTLASS*, https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/, Accessed: 2021-07-05.

[26] *Dynamic Global Memory Allocation and Operations*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dynamic-global-memory-allocation-and-operations, Accessed: 2021-07-05.

[27] A. Valassi, M. Alef, J.-M. Barbet, O. Datskova, R. De Maria, M. Fontes Medeiros, D. Giordano, C. Grigoras, C. Hollowell, M. Javurkova and et al., 'Using hep experiment workflows for the benchmarking and accounting of wlcg computing resources,' *EPJ Web of Conferences*, vol. 245, C. Doglioni, D. Kim, G. Stewart, L. Silvestris, P. Jackson and W. Kamleh, Eds., p. 07 035, 2020, ISSN: 2100-014X. DOI: 10.1051/epjconf/202024507035. [Online]. Available: http://dx.doi.org/10.1051/epjconf/202024507035.

[28] *What is a Container?* https://www.docker.com/resources/what-container, Accessed: 2021-04-01.

Appendix A

Benchmarking and Containers

This appendix includes some of the early reflections the author did after reading up on containers for HPC work-loads.

A.1 Containers

As described in the previous section, CERN went from using the CERN Unit, which consisted of HEP workloads, to using the SPEC benchmarking suite. This was due to the fact that it was to deemed to complicated to capture all of the software and data dependencies of a HEP application in a benchmark. However, today CERN chooses again to turn to its own workloads to get representative benchmarks, as technologies available today can solve the problems that seemed unsolvable 30 years ago [27]. One of these technologies are OS-level virtualization, more commonly known as containers. A container is described as a lightweight and standalone software package that is ready to be executed as it includes everything one needs to run: code, runtime, system tools, system libraries and settings [28]. This enables packaging and distribution of HEP workloads with all dependencies and ensures reproducible of the results when running on different machines.

A.1.1 Docker versus Singularity

Docker was the first major container service and is today still one of the biggest. It has however not been embraced by the majority of the HPC community. HPC centers have strict security to ensure the protection of client data as several users jobs may be simultaneously executed on the same node and to ensure this all jobs must be executed unprivileged(cite hpc vchep paper). Docker containers execute with the help of docker deamon, which previously always has run as root. Docker has been working on a rootless mode, however in-between many have turned to Singularity instead.

Singularity is another container service, designed from the beginning to be used in

scientific computing. It runs with the same privileges as the host user, that means that the container will only have root privileges if the host is root outside the container. It is compliant with the Open Container Initiative (OCI) and can natively translate Docker Images. In addition, it manages seamless integration with a number of resource management and job scheduling systems, where Docker would require modifications.

A.1.2 Usage and build

For the current HEP workloads, both Docker and Singularity can be used. However, images are only built as docker images in the source code. This is because Docker images can be converted to the Singularity Image Format (SIF), but not the other way around, as Singularity from the beginning has been focusing on interoperability with Docker.

The docker image is built in layers. This is done to enable efficient caching. The first layer is the one least updated and the last layer is the most updated. Layers are useful for caching, since a layer that is not changed can be reused if non of the layers preceding it are changed as well. When an image is converted to a SIF, the layers are squashed into each other, but the SIF is cached as well, so as long as there are no changes it re-runs fast. Since Singularity images are only one layer, they require no building as Docker does.

There are also discussions of branching out to other container services in the future, such as Podman [1].

A.2 Containers and GPUs

While containment technology for CPU related tasks has been around for some time, support for host hardware, such as GPU accelerators, is a relatively new addition. In this nascent development, harnessing host peripherals requires additional configuration to a standard container call to inform the container of the resources outside the container runtime. For GPUs, this requires that a driver be present and loaded into the host kernel, and exposed to the container runtime and container being launched. Within the container, a library compatible with the exposed host hardware driver must be included, as there is no uniform way to expose host hardware libraries. This adds additional overhead to container development and image size. This becomes problematic when a host GPU architecture is unknown, resulting in many images maintained for a variety of accelerator architectures, or many libraries being included in a single image with the hopes of supporting more hardware.

[1] https://podman.io/

Appendix B

Code Implementations

B.1 Code Used for Testing

B.1.1 Single-precision Tensor Core

Code listing B.1: Test code for a single-precision Tensor Core matrix multiplication using 16*16 size matrices.

```
#include <mma.h>
#include <iostream>

using namespace nvcuda;

__global__ void wmma_ker(half *a, half *b, float *c) {
    // Declare the fragments
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

    // Initialize the output to zero
    wmma::fill_fragment(c_frag, 0.0f);

    // Load the inputs
    wmma::load_matrix_sync(a_frag, a, 16);
    wmma::load_matrix_sync(b_frag, b, 16);

    // Perform the matrix multiplication
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    // Store the output
    wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}

__global__ void setup(float* res) {
    half *d_a, *h_a, *d_b, *h_b;
    float *d_c;
    h_b = new half[16*16];
    h_a = new half[16*16];
    d_a = (half*) malloc(16*16*sizeof(half));
    d_b = (half*) malloc(16*16*sizeof(half));
```

```
    d_c = (float*) malloc(16*16*sizeof(float));
    for (int i = 0; i < 16*16; i++) {
    h_a[i] = 1.0f;
    h_b[i] = 1.0f;}
    memcpy(d_a, h_a, 16*16*sizeof(half));
    memcpy(d_b, h_b, 16*16*sizeof(half));
    wmma_ker<<<1,32>>>(d_a, d_b, d_c);
    for (int i = 0; i < 16*16; i++) {
        res[i] = d_c[i];
    }
}

int main(){
    float res[16*16] = {0};
    float *res_d;
    cudaMalloc(&res_d, 16*16*sizeof(float));
    cudaMemcpy(res_d, res, 16*16*sizeof(float), cudaMemcpyHostToDevice);
    setup<<<1,1>>>(res_d);
    cudaMemcpy(res, res_d, 16*16*sizeof(float), cudaMemcpyDeviceToHost);
    for (int i = 0; i < 16*16; i++) std::cout << res[i] << ",";
    std::cout << std::endl;
    cudaFree(res_d);
}
```

B.1.2 Blocking Algorithm Testing

Code listing B.2: Test code to validate the blocking algorithm for both one- and two dimensional matrices, using colour matrix from Madgraph.

```
#include <iostream>

void printMatrixDouble(double* A, int m, int n) {
  for(int row = 0; row < m; row++) {
    for(int col = 0; col < n; col++) {
      std::cout << A[row*n + col] << ", ";
    }
    std::cout << "\n";
  }
  std::cout << "\n";
}

int main(void) {

    const int a_rows = 24;
    const int a_cols = 24;

    double A_1[24][24] = {{512, -64, -64, 8, 8, 80, -64, 8,
      8, -1, -1, -10, 8, -1, 80, -10, 71, 62, -1, -10, -10, 62, 62, -28}, {-64,
      512, 8, 80, -64, 8, 8, -64, -1, -10, 8, -1, -1, -10, -10, 62, 62, -28, 8,
      -1, 80, -10, 71, 62}, {-64, 8, 512, -64, 80, 8, 8, -1, 80, -10, 71, 62,
      -64, 8, 8, -1, -1, -10, -10, -1, 62, -28, -10, 62}, {8, 80, -64, 512, 8,
      -64, -1, -10, -10, 62, 62, -28, 8, -64, -1, -10, 8, -1, -1, 8, 71, 62,
      80, -10}, {8, -64, 80, 8, 512, -64, -1, 8, 71, 62, 80, -10, -10, -1, 62,
      -28, -10, 62, -64, 8, 8, -1, -1, -10}, {80, 8, 8, -64, -64, 512, -10, -1,
      62, -28, -10, 62, -1, 8, 71, 62, 80, -10, 8, -64, -1, -10, 8, -1}, {-64,
      8, 8, -1, -1, -10, 512, -64, -64, 8, 8, 80, 80, -10, 8, -1, 62, 71, -10,
      62, -1, -10, -28, 62}, {8, -64, -1, -10, 8, -1, -64, 512, 8, 80, -64, 8,
      -10, 62, -1, -10, -28, 62, 80, -10, 8, -1, 62, 71}, {8, -1, 80, -10, 71,
```

```
    62, -64, 8, 512, -64, 80, 8, 8, -1, -64, 8, -10, -1, 62, -28, -10, -1,
    62, -10}, {-1, -10, -10, 62, 62, -28, 8, 80, -64, 512, 8, -64, -1, -10,
    8, -64, -1, 8, 71, 62, -1, 8, -10, 80}, {-1, 8, 71, 62, 80, -10, 8, -64,
    80, 8, 512, -64, 62, -28, -10, -1, 62, -10, 8, -1, -64, 8, -10, -1},
    {-10, -1, 62, -28, -10, 62, 80, 8, 8, -64, -64, 512, 71, 62, -1, 8, -10,
    80, -1, -10, 8, -64, -1, 8}, {8, -1, -64, 8, -10, -1, 80, -10, 8, -1, 62,
    71, 512, -64, -64, 8, 8, 80, 62, -10, -28, 62, -1, -10}, {-1, -10, 8,
    -64, -1, 8, -10, 62, -1, -10, -28, 62, -64, 512, 8, 80, -64, 8, -10, 80,
    62, 71, 8, -1}, {80, -10, 8, -1, 62, 71, 8, -1, -64, 8, -10, -1, -64, 8,
    512, -64, 80, 8, -28, 62, 62, -10, -10, -1}, {-10, 62, -1, -10, -28, 62,
    -1, -10, 8, -64, -1, 8, 8, 80, -64, 512, 8, -64, 62, 71, -10, 80, -1, 8},
    {71, 62, -1, 8, -10, 80, 62, -28, -10, -1, 62, -10, 8, -64, 80, 8, 512,
    -64, -1, 8, -10, -1, -64, 8}, {62, -28, -10, -1, 62, -10, 71, 62, -1, 8,
    -10, 80, 80, 8, 8, -64, -64, 512, -10, -1, -1, 8, 8, -64}, {-1, 8, -10,
    -1, -64, 8, -10, 80, 62, 71, 8, -1, 62, -10, -28, 62, -1, -10, 512, -64,
    -64, 8, 8, 80}, {-10, -1, -1, 8, 8, -64, 62, -10, -28, 62, -1, -10, -10,
    80, 62, 71, 8, -1, -64, 512, 8, 80, -64, 8}, {-10, 80, 62, 71, 8, -1, -1,
    8, -10, -1, -64, 8, -28, 62, 62, -10, -10, -1, -64, 8, 512, -64, 80, 8},
    {62, -10, -28, 62, -1, -10, -10, -1, -1, 8, 8, -64, 62, 71, -10, 80, -1,
    8, 8, 80, -64, 512, 8, -64}, {62, 71, -10, 80, -1, 8, -28, 62, 62, -10,
    -10, -1, -1, 8, -10, -1, -64, 8, 8, -64, 80, 8, 512, -64}, {-28, 62, 62,
    -10, -10, -1, 62, 71, -10, 80, -1, 8, -10, -1, -1, 8, 8, -64, 80, 8, 8,
    -64, -64, 512}};

    double A_2[24*24] = {512, -64, -64, 8, 8, 80, -64, 8,
    8, -1, -1, -10, 8, -1, 80, -10, 71, 62, -1, -10, -10, 62, 62, -28, -64,
    512, 8, 80, -64, 8, 8, -64, -1, -10, 8, -1, -1, -10, -10, 62, 62, -28, 8,
    -1, 80, -10, 71, 62, -64, 8, 512, -64, 80, 8, 8, -1, 80, -10, 71, 62,
    -64, 8, 8, -1, -1, -10, -10, -1, 62, -28, -10, 62, 8, 80, -64, 512, 8,
    -64, -1, -10, -10, 62, 62, -28, 8, -64, -1, -10, 8, -1, -1, 8, 71, 62,
    80, -10, 8, -64, 80, 8, 512, -64, -1, 8, 71, 62, 80, -10, -10, -1, 62,
    -28, -10, 62, -64, 8, 8, -1, -1, -10, 80, 8, 8, -64, -64, 512, -10, -1,
    62, -28, -10, 62, -1, 8, 71, 62, 80, -10, 8, -64, -1, -10, 8, -1, -64,
    8, 8, -1, -1, -10, 512, -64, -64, 8, 8, 80, 80, -10, 8, -1, 62, 71, -10,
    62, -1, -10, -28, 62, 8, -64, -1, -10, 8, -1, -64, 512, 8, 80, -64, 8,
    -10, 62, -1, -10, -28, 62, 80, -10, 8, -1, 62, 71, 8, -1, 80, -10, 71,
    62, -64, 8, 512, -64, 80, 8, 8, -1, -64, 8, -10, -1, 62, -28, -10, -1,
    62, -10, -1, -10, -10, 62, 62, -28, 8, 80, -64, 512, 8, -64, -1, -10,
    8, -64, -1, 8, 71, 62, -1, 8, -10, 80, -1, 8, 71, 62, 80, -10, 8, -64,
    80, 8, 512, -64, 62, -28, -10, -1, 62, -10, 8, -1, -64, 8, -10, -1,
    -10, -1, 62, -28, -10, 62, 80, 8, 8, -64, -64, 512, 71, 62, -1, 8, -10,
    80, -1, -10, 8, -64, -1, 8, 8, -1, -64, 8, -10, -1, 80, -10, 8, -1, 62,
    71, 512, -64, -64, 8, 8, 80, 62, -10, -28, 62, -1, -10, -1, -10, 8,
    -64, -1, 8, -10, 62, -1, -10, -28, 62, -64, 512, 8, 80, -64, 8, -10, 80,
    62, 71, 8, -1, 80, -10, 8, -1, 62, 71, 8, -1, -64, 8, -10, -1, -64, 8,
    512, -64, 80, 8, -28, 62, 62, -10, -10, -1, -10, 62, -1, -10, -28, 62,
    -1, -10, 8, -64, -1, 8, 8, 80, -64, 512, 8, -64, 62, 71, -10, 80, -1, 8,
    71, 62, -1, 8, -10, 80, 62, -28, -10, -1, 62, -10, 8, -64, 80, 8, 512,
    -64, -1, 8, -10, -1, -64, 8, 62, -28, -10, -1, 62, -10, 71, 62, -1, 8,
    -10, 80, 80, 8, 8, -64, -64, 512, -10, -1, -1, 8, 8, -64, -1, 8, -10,
    -1, -64, 8, -10, 80, 62, 71, 8, -1, 62, -10, -28, 62, -1, -10, 512, -64,
    -64, 8, 8, 80, -10, -1, -1, 8, 8, -64, 62, -10, -28, 62, -1, -10, -10,
    80, 62, 71, 8, -1, -64, 512, 8, 80, -64, 8, -10, 80, 62, 71, 8, -1, -1,
    8, -10, -1, -64, 8, -28, 62, 62, -10, -10, -1, -64, 8, 512, -64, 80, 8,
    62, -10, -28, 62, -1, -10, -10, -1, -1, 8, 8, -64, 62, 71, -10, 80, -1,
    8, 8, 80, -64, 512, 8, -64, 62, 71, -10, 80, -1, 8, -28, 62, 62, -10,
    -10, -1, -1, 8, -10, -1, -64, 8, 8, -64, 80, 8, 512, -64, -28, 62, 62,
    -10, -10, -1, 62, 71, -10, 80, -1, 8, -10, -1, -1, 8, 8, -64, 80, 8, 8,
    -64, -64, 512};
```

```
    double* A_block_1 = (double*) malloc(a_rows*a_cols*sizeof(double));
    for (int i=0;i<a_rows*a_cols; i++) A_block_1[i]=0;

    double* A_block_2 = (double*) malloc(a_rows*a_cols*sizeof(double));
    for (int i=0;i<a_rows*a_cols; i++) A_block_2[i]=0;

    int base_offset_a_row = 0;
    int base_offset_a_col = 0;
    int base_offset_a_block = 0;
    int curr_offset_a_row = 0;
    int curr_offset_a_col = 0;
    int curr_offset_a_block = 0;
    for (int blockrow=0; blockrow<3; blockrow++) {
      for (int blockcol=0; blockcol<6; blockcol++) {
        base_offset_a_row = blockrow*8;
        base_offset_a_col = blockcol*4;
        base_offset_a_block = 32*(6*blockrow + blockcol);
        for (int row=0; row<8; row++) {
          for (int col=0; col<4;col++) {
            curr_offset_a_row = base_offset_a_row + row;
            curr_offset_a_col = base_offset_a_col + col;
            curr_offset_a_block = base_offset_a_block + row*4 + col;
            A_block_1[curr_offset_a_block] = A_1[curr_offset_a_row][curr_offset_a_col];
          }
        }
      }
    }

    int curr_offset_a = 0;
    int base_offset_a = 0;
    for (int blockrow=0; blockrow<3; blockrow++) {
        for (int blockcol=0; blockcol<6; blockcol++) {
            base_offset_a = blockrow*8*a_cols + blockcol*4;
            base_offset_a_block = 32*(6*blockrow+blockcol);
            for (int row=0; row<8; row++) {
                for (int col=0; col<4;col++) {
                    curr_offset_a = base_offset_a + row*a_cols + col;
                    curr_offset_a_block = base_offset_a_block + row*4 + col;
                    A_block_2[curr_offset_a_block] = A_2[curr_offset_a];
                }
            }
        }
    }
    int acc = 0;
    for(int i=0; i<24*24; i++) {
        acc = A_block_1[i] - A_block_2[i];
    }

    std::cout << acc;

}
```

B.1.3 Complete Double-Precision Test Code using Tensor Cores

Code listing B.3: Complete test code using colour matrix from Madgraph doing blocking, padding and Tensor Core matrix multiplication.

```
#include <complex>
```

```
#include <iostream>
#include <cstring>

#include "mma.h"
#include <thrust/complex.h>

using namespace nvcuda;

typedef thrust::complex<double> cxtype;

__global__ void wmma_ker(double *a, double *b, double*c) {

  int a_offset = blockIdx.x*32;
  int b_offset = (blockIdx.x%6)*32;
  int c_offset = blockIdx.x*64;

  // Declare fragments
  wmma::fragment<wmma::matrix_a, 8, 8, 4, double, wmma::row_major> a_frag;
  wmma::fragment<wmma::matrix_b, 8, 8, 4, double, wmma::row_major> b_frag;
  wmma::fragment<wmma::accumulator, 8, 8, 4, double> c_frag;

  // Fill accumulator matrix with 0s
  wmma::fill_fragment(c_frag, 0.0);

  // Load inputs into factor fragments
  wmma::load_matrix_sync(a_frag, a+a_offset, 4);
  wmma::load_matrix_sync(b_frag, b+b_offset, 8);

  // Do matrix multiplication C = A*B+C
  wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

  // Store output
  wmma::store_matrix_sync(c+c_offset, c_frag, 8, wmma::mem_row_major);
}

void printMatrixCX(cxtype* A, int m, int n) {
  for(int row = 0; row < m; row++) {
    for(int col = 0; col < n; col++) {
      std::cout << A[row*n + col] << ",␣";
    }
    std::cout << "\n";
  }
  std::cout << "\n";
}

void printMatrixDouble(double* A, int m, int n) {
  for(int row = 0; row < m; row++) {
    for(int col = 0; col < n; col++) {
      std::cout << A[row*n + col] << ",␣";
    }
    std::cout << "\n";
  }
  std::cout << "\n";
}

__global__ void matrixVectorMultiply(cxtype* res, double* meHelSum, double* A_block, double* B, double* C) {

    const unsigned int a_rows = 24;
    const unsigned int a_cols = 24;
    const unsigned int block_matrix_rows = 3;
```

```
    const unsigned int block_matrix_cols = 6;
    const unsigned int block_rows = 8;
    const unsigned int block_cols = 4;

    double A[a_rows][a_cols] = {{512, -64, -64, 8, 8, 80, -64, 8,
      8, -1, -1, -10, 8, -1, 80, -10, 71, 62, -1, -10, -10, 62, 62, -28}, {-64,
      512, 8, 80, -64, 8, 8, -64, -1, -10, 8, -1, -1, -10, -10, 62, 62, -28, 8,
      -1, 80, -10, 71, 62}, {-64, 8, 512, -64, 80, 8, 8, -1, 80, -10, 71, 62,
      -64, 8, 8, -1, -1, -10, -10, -1, 62, -28, -10, 62}, {8, 80, -64, 512, 8,
      -64, -1, -10, -10, 62, 62, -28, 8, -64, -1, -10, 8, -1, -1, 8, 71, 62,
      80, -10}, {8, -64, 80, 8, 512, -64, -1, 8, 71, 62, 80, -10, -10, -1, 62,
      -28, -10, 62, -64, 8, 8, -1, -1, -10}, {80, 8, 8, -64, -64, 512, -10, -1,
      62, -28, -10, 62, -1, 8, 71, 62, 80, -10, 8, -64, -1, -10, 8, -1}, {-64,
      8, 8, -1, -1, -10, 512, -64, -64, 8, 8, 80, 80, -10, 8, -1, 62, 71, -10,
      62, -1, -10, -28, 62}, {8, -64, -1, -10, 8, -1, -64, 512, 8, 80, -64, 8,
      -10, 62, -1, -10, -28, 62, 80, -10, 8, -1, 62, 71}, {8, -1, 80, -10, 71,
      62, -64, 8, 512, -64, 80, 8, 8, -1, -64, 8, -10, -1, 62, -28, -10, -1,
      62, -10}, {-1, -10, -10, 62, 62, -28, 8, 80, -64, 512, 8, -64, -1, -10,
      8, -64, -1, 8, 71, 62, -1, 8, -10, 80}, {-1, 8, 71, 62, 80, -10, 8, -64,
      80, 8, 512, -64, 62, -28, -10, -1, 62, -10, 8, -1, -64, 8, -10, -1},
      {-10, -1, 62, -28, -10, 62, 80, 8, 8, -64, -64, 512, 71, 62, -1, 8, -10,
      80, -1, -10, 8, -64, -1, 8}, {8, -1, -64, 8, -10, -1, 80, -10, 8, -1, 62,
      71, 512, -64, -64, 8, 8, 80, 62, -10, -28, 62, -1, -10}, {-1, -10, 8,
      -64, -1, 8, -10, 62, -1, -10, -28, 62, -64, 512, 8, 80, -64, 8, -10, 80,
      62, 71, 8, -1}, {80, -10, 8, -1, 62, 71, 8, -1, -64, 8, -10, -1, -64, 8,
      512, -64, 80, 8, -28, 62, 62, -10, -10, -1}, {-10, 62, -1, -10, -28, 62,
      -1, -10, 8, -64, -1, 8, 8, 80, -64, 512, 8, -64, 62, 71, -10, 80, -1, 8},
      {71, 62, -1, 8, -10, 80, 62, -28, -10, -1, 62, -10, 8, -64, 80, 8, 512,
      -64, -1, 8, -10, -1, -64, 8}, {62, -28, -10, -1, 62, -10, 71, 62, -1, 8,
      -10, 80, 80, 8, 8, -64, -64, 512, -10, -1, -1, 8, 8, -64}, {-1, 8, -10,
      -1, -64, 8, -10, 80, 62, 71, 8, -1, 62, -10, -28, 62, -1, -10, 512, -64,
      -64, 8, 8, 80}, {-10, -1, -1, 8, 8, -64, 62, -10, -28, 62, -1, -10, -10,
      80, 62, 71, 8, -1, -64, 512, 8, 80, -64, 8}, {-10, 80, 62, 71, 8, -1, -1,
      8, -10, -1, -64, 8, -28, 62, 62, -10, -10, -1, -64, 8, 512, -64, 80, 8},
      {62, -10, -28, 62, -1, -10, -10, -1, -1, 8, 8, -64, 62, 71, -10, 80, -1,
      8, 8, 80, -64, 512, 8, -64}, {62, 71, -10, 80, -1, 8, -28, 62, 62, -10,
      -10, -1, -1, 8, -10, -1, -64, 8, 8, -64, 80, 8, 512, -64}, {-28, 62, 62,
      -10, -10, -1, 62, 71, -10, 80, -1, 8, -10, -1, -1, 8, 8, -64, 80, 8, 8,
      -64, -64, 512}};

    cxtype b[24];
    for(int i=0; i<24;i++) {
      b[i] = cxtype(i, i*2);
    }

    // STEP 1: PAD VECTOR B TO MATRIX
    const int paddedWidth = 8; // Width of padded matrix
    for (int i=0; i<a_cols*paddedWidth; i++) B[i]=0;
    for (int row=0; row < a_cols; row++) {
        B[paddedWidth*row+0] = b[row].real();
        B[paddedWidth*row+1] = b[row].imag();
    }

    // STEP 2: BLOCKING
    int base_offset_a_row = 0;
    int base_offset_a_col = 0;
    int base_offset_a_block = 0;
    int curr_offset_a_row = 0;
    int curr_offset_a_col = 0;
    int curr_offset_a_block = 0;
```

```
    for (int blockrow=0; blockrow<block_matrix_rows; blockrow++) {
      for (int blockcol=0; blockcol<block_matrix_cols; blockcol++) {
        base_offset_a_row = blockrow*block_rows;
        base_offset_a_col = blockcol*block_cols;
        base_offset_a_block = block_rows*block_cols*(block_matrix_cols*blockrow + blockcol);
        for (int row=0; row<block_rows; row++) {
          for (int col=0; col<block_cols;col++) {
            curr_offset_a_row = base_offset_a_row + row;
            curr_offset_a_col = base_offset_a_col + col;
            curr_offset_a_block = base_offset_a_block + row*block_cols + col;
            A_block[curr_offset_a_block] = A[curr_offset_a_row][curr_offset_a_col];
          }
        }
      }
    }

    // STEP 3: USE TENSOR CORES
    wmma_ker<<<18,32>>>(A_block, B, C);
    cudaDeviceSynchronize();

    // STEP 4: SUM TENSOR CORE RESULT INTO FINAL RESULT MATRIX
    double real_sum = 0;
    double imag_sum = 0;
    for (int output_row=0; output_row<a_cols; output_row++) {
      int input_block_row = output_row/8;
      for (int input_block_col=0; input_block_col<6; input_block_col++) {
        real_sum += C[input_block_row*6*8*8 + (output_row%8)*8 + input_block_col*64];
        imag_sum += C[input_block_row*6*8*8 + (output_row%8)*8 + input_block_col*64 + 1];
      }
            res[output_row].real(real_sum);
      res[output_row].imag(imag_sum);
      real_sum = 0; imag_sum = 0;
    }

    // STEP 5: COMPLETE MEHELSUM BY USING THE REAL COMPONENT FROM FINAL RESULT MATRIX
    for (int i=0; i<24; i++) {
      (*meHelSum) += ((res[i] * thrust::conj(b[i])).real())/54;
    }

}

int main(void) {
    int threads = 32;
    int blocks = 16384;

    cxtype res[24] = {0};
    double meHelSum = 0;
    double* A_block;
    double* B;
    double* C;
    cxtype* res_d;
    double* meHelSum_d;

    cudaMalloc(&res_d, 24*sizeof(cxtype));
    cudaMalloc(&meHelSum_d, sizeof(double));
    cudaMalloc(&A_block, 24*24*sizeof(double));
    cudaMalloc(&B, 24*8*threads*blocks*sizeof(double));
    cudaMalloc(&C, 8*8*18*threads*blocks*sizeof(double));

    cudaMemcpy(res_d, res, 24*sizeof(cxtype), cudaMemcpyHostToDevice);
```

```
182     cudaMemcpy(meHelSum_d, &meHelSum, sizeof(double), cudaMemcpyHostToDevice);
183 
184     matrixVectorMultiply<<<blocks, threads>>>(res_d, meHelSum_d, A_block, B, C);
185 
186     cudaMemcpy(res, res_d, 24*sizeof(cxtype), cudaMemcpyDeviceToHost);
187     cudaMemcpy(&meHelSum, meHelSum_d, sizeof(double), cudaMemcpyDeviceToHost);
188 
189     cudaFree(meHelSum_d);
190     cudaFree(res_d);
191     cudaFree(A_block);
192     cudaFree(B);
193     cudaFree(C);
194 
195     // Only guaranteed to be correct if run with 1 block and 1 thread for debugging purposes
196     std::cout << "RES:␣\n";
197     printMatrixCX(res, 24, 1);
198 
199     std::cout << "meHelSum:␣\n";
200     std::cout << meHelSum;
201 }
```

B.1.4 Makefile

Code listing B.4: Makefile for running the code in B.1 and in B.3.

```
1  EXECUTABLE_NAME = runTensor
2  EXECUTABLE_NAME_SINGLE = runSinglePres
3 
4  NVCC_LOCATION = /usr/local/cuda-11.0/bin/nvcc
5 
6  run: compile
7          ./$(EXECUTABLE_NAME)
8 
9  single: single-pres-test.cu
10         $(NVCC_LOCATION) -rdc=true -arch=sm_70 single_pres_test.cu -o $(EXECUTABLE_NAME_SINGLE) -lcudadevrt
11         ./$(EXECUTABLE_NAME_SINGLE)
12 
13 compile: tensor.cu
14         $(NVCC_LOCATION) -rdc=true -arch=sm_80 tensor.cu -o $(EXECUTABLE_NAME) -lcudadevrt
15 
16 .PHONY: clean
17 clean:
18         rm -rf *.o $(EXECUTABLE_NAME) $(EXECUTABLE_NAME_SINGLE)
```

B.2 The first implementation in Madgraph

Code listing B.5: Set up code with blocking and padding

```
1  #ifdef __CUDACC__
2 
3    const int a_rows = ncolor;
4    const int a_cols = ncolor;
5 
6    //STEP 1: PAD VECTOR B TO MATRIX
7    const int paddedWidth = 8; //Width of padded matrix
8    double* B = (double*) malloc(a_cols*paddedWidth*sizeof(double));
```

```
  for (int i = 0; i < a_cols*paddedWidth; i++) B[i] = 0;
  for (int row = 0; row < a_cols; row++) {
    B[paddedWidth*row + 0] = jamp[row].real();
    B[paddedWidth*row + 1] = jamp[row].imag();
  }

  //STEP 2: DO BLOCKING HERE!
  double* A_block = (double*) malloc(a_rows*a_cols*sizeof(double));
  for (int i = 0; i < a_rows*a_cols; i++) A_block[i] = 0;

  int base_offset_a_row = 0;
  int base_offset_a_col = 0;
  int base_offset_a_block = 0;
  int curr_offset_a_row = 0;
  int curr_offset_a_col = 0;
  int curr_offset_a_block = 0;
  for (int blockrow = 0; blockrow < 3; blockrow++) {
    for (int blockcol = 0; blockcol < 6; blockcol++) {
      base_offset_a_row = blockrow*8;
      base_offset_a_col = blockcol*4;
      base_offset_a_block = 32*(6*blockrow + blockcol);
      for (int row = 0; row < 8; row++) {
        for (int col = 0; col < 4; col++) {
          curr_offset_a_row = base_offset_a_row + row;
          curr_offset_a_col = base_offset_a_col + col;
          curr_offset_a_block = base_offset_a_block + row*4 + col;
          A_block[curr_offset_a_block] = cf[curr_offset_a_row][curr_offset_a_col];
        }
      }
    }
  }

  //STEP 3: USE TENSOR CORES
  double* C = (double*) malloc(8*8*18*sizeof(double));
  for (int i=0; i<8*8*18; i++) C[i] =0;
  for (int i=0; i<18;i++) {
    multiplyMatrixTensorCore<<<1,32>>>(A_block+32*i, B+32*(i%6), C+64*i);
  }
  cudaDeviceSynchronize();

  cxtype* resultVector = (cxtype*) malloc(24*sizeof(cxtype));

  double real_sum = 0;
  double imag_sum = 0;
  for (int output_row = 0; output_row < a_cols; output_row++) {
    int input_block_row = output_row/8;
    for (int input_block_col = 0; input_block_col < 6; input_block_col++) {
      real_sum += C[input_block_row*6*8*8 + (output_row%8)*8 + input_block_col*64];
      imag_sum += C[input_block_row*6*8*8 + (output_row%8)*8 + input_block_col*64 +1];
    }
    resultVector[output_row].real(real_sum);
    resultVector[output_row].imag(imag_sum);
    real_sum = 0; imag_sum = 0;
  }

  for (int row = 0; row < 24; row++) {
    meHelSum += ((resultVector[row] * thrust::conj(jamp[row])).real())/denom[row];
  }

  //STEP 4: FREE MEMORY
```

```
  free(B);
  free(A_block);
  free(C);
  free(resultVector);

  #else

  // Sum and square the color flows to get the matrix element
  for(int icol = 0; icol < ncolor; icol++ )
  {
    cxtype ztemp = cxmake(0, 0);
    for(int jcol = 0; jcol < ncolor; jcol++ )
      ztemp = ztemp + cf[icol][jcol] * jamp[jcol];
    meHelSum = meHelSum + cxreal(ztemp * conj(jamp[icol]))/denom[icol];
  }
  #endif

  // Store the leading color flows for choice of color
  // for(i=0;i < ncolor; i++)
  // jamp2[0][i] += real(jamp[i]*conj(jamp[i]));

  mgDebug(1, __FUNCTION__);
  return;
}
```

Code listing B.6: Tensor Core kernel from the naive implementation

```
#ifdef __CUDACC__

  __global__ void multiplyMatrixTensorCore(double *a, double *b, double*c) {

  // Declare fragments
  wmma::fragment<wmma::matrix_a, 8, 8, 4, double, wmma::row_major> a_frag;
  wmma::fragment<wmma::matrix_b, 8, 8, 4, double, wmma::row_major> b_frag;
  wmma::fragment<wmma::accumulator, 8, 8, 4, double> c_frag;

  // Fill summand/accumulator matrix with 0
  wmma::fill_fragment(c_frag, 0.0);

  // Load inputs into factor fragments
  wmma::load_matrix_sync(a_frag, a, 4);
  wmma::load_matrix_sync(b_frag, b, 8);

  // (you can see how c_frag is both summand and accumulator)
  wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

  // Store output
  wmma::store_matrix_sync(c, c_frag, 8, wmma::mem_row_major);
}
#endif
```

B.3 The final implementation in Madgraph

Updated version of the code can be found here: `https://github.com/ingvildh/madgraph4gpu`

Appendix C

CERN openlab Technical Workshop Presentation

Slides from a presentation the authro gave at the CERN openlab technical workshop in March 2021 based on my findings and experiences with profiling code to familiarize myself with it, while also trying to identify computational hot-spots.

Profiling code on NVIDIA GPUs

CERN openlab Technical Workshop 2021

Ingvild Brevik Høgstøyl (NTNU-Trondheim, CERN)

Anne C. Elster (NTNU-Trondheim) and Maria Girone (CERN)

Goal and Overview

- Profiling CPU code
  - Using Valgrind, Callgrind and KCacheGrind
- Profiling GPU code using Nsight Systems
  - Identify overall system bottlenecks
- Profiling GPU code using Nsight Compute
  - Identify bottlenecks in kernels

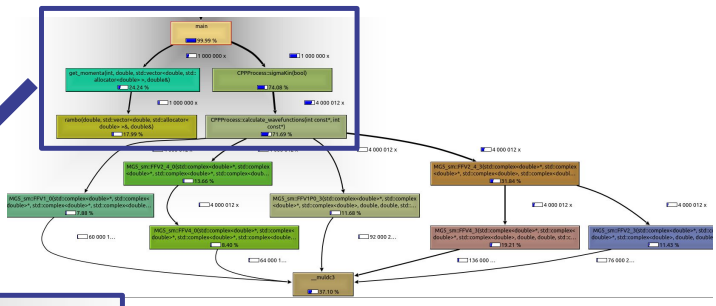
Profiling CPU code

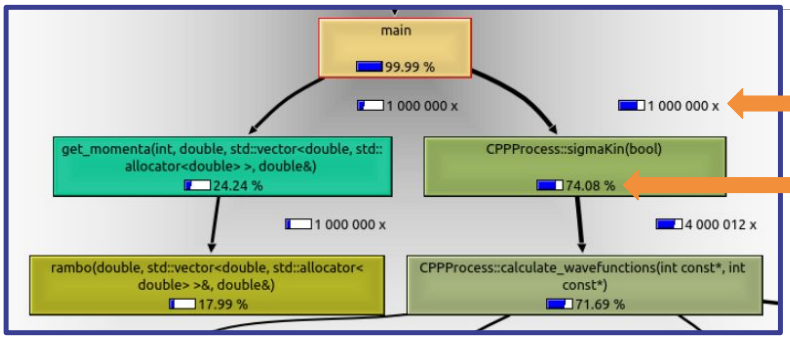
- Madgraph application

  https://github.com/madgraph5/madgraph4gpu
- Generate callgraph
  - Valgrind & Callgrind
- Hotspot visualization
  - KCacheGrind
  - Obtain overview

number of iterations

```
valgrind --tool=callgrind ./check.exe 1000000
```





number of times function is called

percentage of time spent in this function and the functions it calls

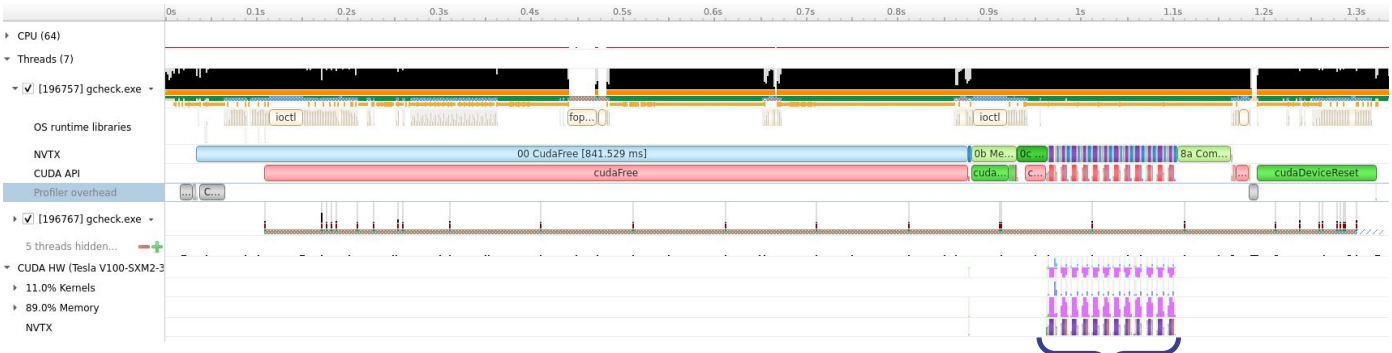
Profiling GPU code: Nsight Systems

- Profiling:
  - Nsight Systems CLI
  - Overhead on profiling GPUs vs CPUs
- Visualization:
  - Nsight Systems GUI
  - Overall behaviour of the application

threads/block

```
nsys profile ./gcheck.exe 16384 32 12
```

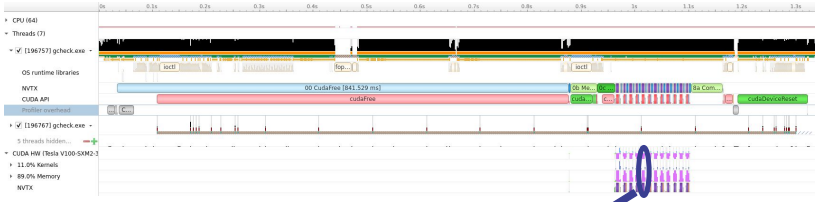
blocks/grid iterations

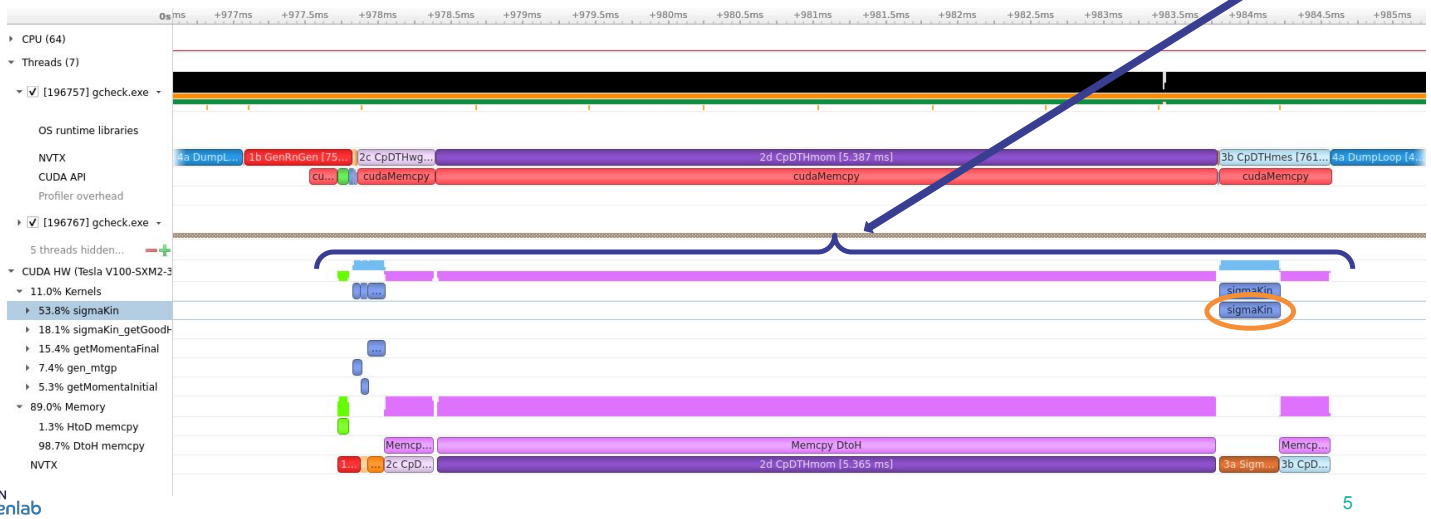


12 iterations

Profiling GPU code: Nsight Systems

- Performance bottlenecks:
  - Memory transfer
  - Kernel





Profiling GPU code: Nsight Compute

- Profiling:
  - Nsight Compute CLI
  - Kernel profiling
- Visualization
  - Nsight Compute GUI
  - SigmaKin kernel
- Performance bottlenecks
  - SM vs Memory utilization
  - Utilization of pipe

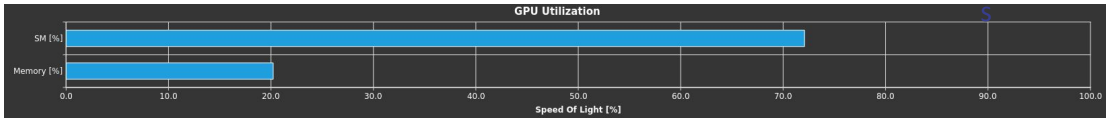
threads/block

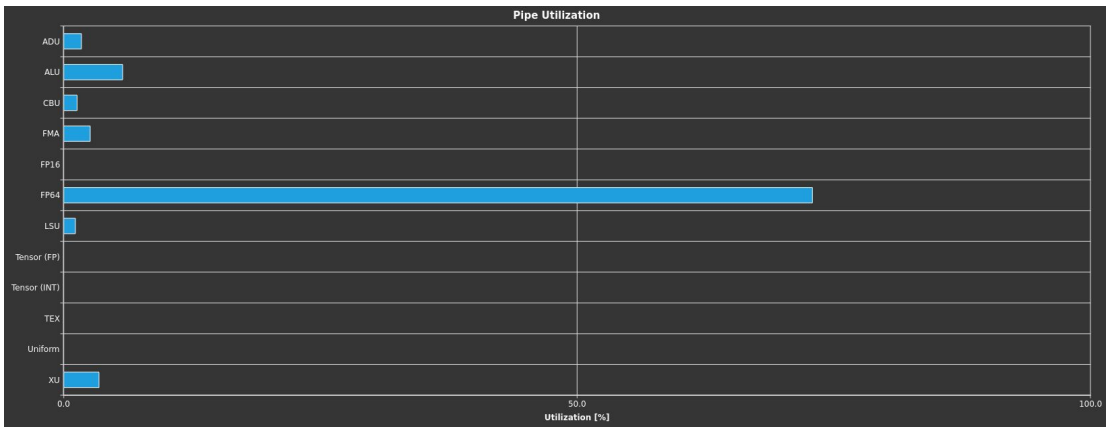
```
nv-nsight-cu-cli -o results ./gcheck.exe 16384 32 1
```

create file with results

blocks/grid

iteration





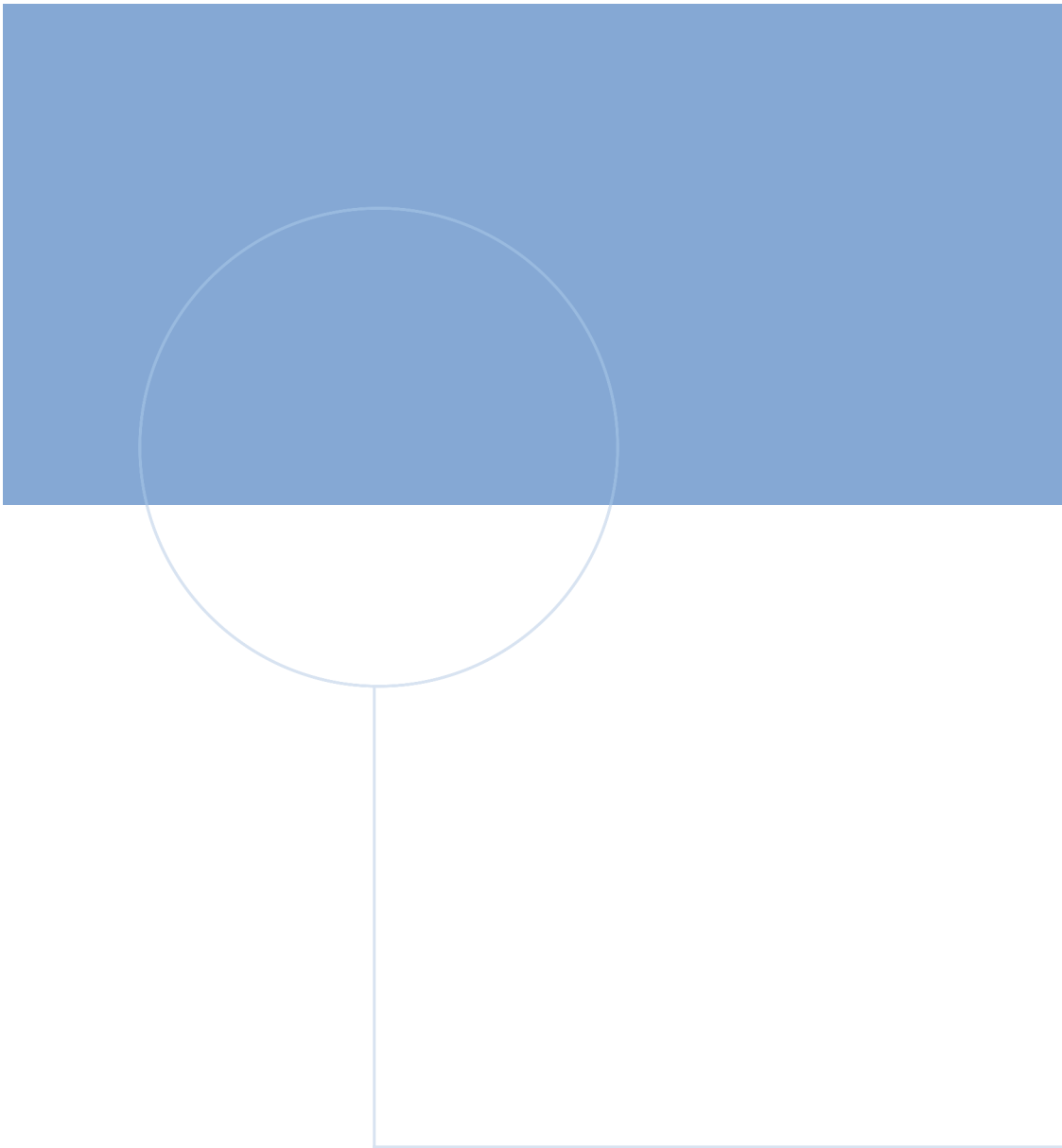
CERN openlab

Future work

- Ideas for GPU optimizations
  - Vary the number of threads/block
  - Change memory utilization
  - Reduce precision
- Use sampling information to identify hotspots

Thank you!

ingvild.brevik.hoegstoeyl@cern.ch



NTNU

Norwegian University of
Science and Technology