Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Knut Aasgaard Kirkhorn

BAT: A Benchmark Suite for Auto-Tuners

Development of BAT and Tuning on DGX-2 and More

Master's thesis in Computer Science
Supervisor: Anne C. Elster
November 2020

NTNU
Norwegian University of Science and Technology

Knut Aasgaard Kirkhorn

BAT: A Benchmark Suite for Auto-Tuners

Development of BAT and Tuning on DGX-2 and More

Master's thesis in Computer Science
Supervisor: Anne C. Elster
November 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

HPC (High Performance Computing) system have in the recent years become more and more heterogeneous, containing different architectures such as multicore CPUs and accelerators such as GPUs and FPGAs.

Because of the variety in architecture parameters, programs need to be optimized and performance-tweaked for the given architecture in order to get the best performance. Architectural features include special instructions to utilize the hardware parallelism of multi- and many-core processors, different types and sizes of memory, and other special architectural features, such as tensor cores on modern GPUs. This leads to a lot of performance tweaking for the given hardware in order to achieve the best performance.

Since there are a lot of tweakable parameters for performance tuning, such as varying block sizes, depths of loop unrolling and function inlining, one ends up with a huge search space that is possible, but very hard to do by hand. It is much more efficient to use an auto-tuner for this job, that automates the performance tuning of parameterized implementation. One problem is that there is no standard benchmark suite for measuring the performance of auto-tuners.

This thesis presents the creation of the benchmark suite for auto-tuners named BAT (**B**enchmark suite for **A**uto-**T**uners). BAT is a benchmark suite for auto-tuners for GPU and HPC oriented programs created in CUDA. This benchmark suite is proposed joint with Ingunn Sund as a solution to the problem of no standard benchmark suites for auto-tuners.

BAT includes a varied selection of benchmarks of different degree of complexity and scope. The main focus of this thesis are the parameterized codes and benchmarks done, including benchmarks for the multi-GPU systems NVIDIA DGX-2 and IBM Power System AC922. A summary of our findings and suggestions for future work is also included.

Sammendrag

HPC (High Performance Computing) systemer har i de siste årene blitt mer og mer heterogene, og inneholder forskjellige arkitekturer som flerkjernede CPUer og akseleratorer som GPUer og FPGAer.

På grunn av mangfoldet i arkitekturparametere, må programmene optimaliseres og ytelsesjusteres for den gitte arkitekturen for å få den beste ytelsen. Arkitektoniske funksjoner inkluderer spesielle instruksjoner for å bruke maskinvareparallellitet til fler- og mangekjerneprosessorer, forskjellige typer og størrelser på minnet og andre spesielle arkitektoniske funksjoner, for eksempel tensor cores på moderne GPUer. Dette fører til mye ytelsesjustering for den gitte maskinvaren for å oppnå best ytelse.

Siden det er mange justerbare parametre for ytelsesjustering, slik som varierende block size, dybde av loop unrolling og function inlining, ender man opp med enorme søkeområder som er mulig, men veldig vanskelig å gjøre for hånd. Det er mye mer effektivt å bruke en auto-tuner for denne jobben, som automatiserer ytelsesjusteringen av parametriserte implementasjoner. Et problem er at det ikke er noen standard benchmark suite for å måte ytelsen av auto-tunere.

Denne masteroppgaven presenterte hvordan en benchmark suite for auto-tuning kalt BAT (**B**enchmark suite for **A**uto-**T**uners), ble laget. BAT er en benchmark suite for auto-tunere for GPU and HPC-orienterte programmer laget i CUDA: Denne benchmark suiten er presentert og utviklet i samarbeid med Ingunn Sund som en løsning til mangelen på en standard benchmark suite for autotunere.

BAT inkluderer et variert utvalg av benchmarks av forskjellige grader av kompleksitet og omfang. Hovedfokuset i denne oppgaven har vært på parameteriseringen av koder og benchmarkene utviklet, inkludert benchmarker for multi-GPU systemene NVIDIA DGX-2 og IBM Power System AC922. En oppsummering av våre funn og forslag til videre arbeid er også inkludert.

Acknowledgments

First I would like to thank both my collaborator Ingunn Sund and my supervisor, Professor Anne C. Elster for all their support and help both with this thesis and the specialization project, that was the precursor to this work.

I would also like to thank Rolf Harald Dahl from IT support at our Dept (IDI) and HPC-Lab Admin Jacob O. Tørring for all their system support and suggestions.

Lastly, I want to thank NTNU and the HPC-lab at IDI for the providing access the HPC systems utilized and benchmarked in this thesis, including several workstations with high end graphics cards, and the IBM Power System AC922 with NVIDIA Tesla V100 cards as well as the NVIDIA DGX2.

Table of Contents

List of Figures **vii**

List of Tables **viii**

List of Listings **x**

List of Abbreviations **xii**

1 Introduction **1**
1.1 Motivation and Contribution . . . 1
1.2 Outline . . . 2

2 Background - GPUs and Docker **4**
2.1 GPU . . . 4
2.2 GPU Hardware . . . 4
2.2.1 NVIDIA GeForce GTX 980 . . . 4
2.2.2 NVIDIA Tesla V100 . . . 5
2.2.3 NVIDIA TITAN RTX . . . 5
2.2.4 NVIDIA Tesla T4 . . . 5
2.2.5 IBM Power System AC922 . . . 5
2.2.6 NVIDIA DGX-2 . . . 6
2.3 GPU- and Interconnect Communication . . . 7
2.3.1 Messaging Passing Interface . . . 7
2.3.2 PCI Express . . . 7
2.3.3 NVLink 2.0 and NVSwitch . . . 7
2.4 GPU Software . . . 8
2.5 Docker . . . 9
2.5.1 Docker Image . . . 9
2.5.2 Docker Container . . . 9
2.5.3 NVIDIA Docker . . . 10

3 Benchmarking GPUs and Auto-tuning **11**
3.1 SHOC Benchmark Suite . . . 11
3.1.1 Radix Sort . . . 11
3.1.2 Triad . . . 12
3.1.3 Reduction . . . 13
3.1.4 Molecular Dynamics . . . 13
3.1.5 Stencil 2D . . . 14
3.2 Auto-Tuning . . . 14
3.3 Auto-Tuning Frameworks . . . 14
3.3.1 Kernel Tuner . . . 14
3.3.2 CLTune . . . 15
3.3.3 Kernel Tuning Toolkit . . . 16
3.3.4 OpenTuner . . . 16

4 Related Work **18**

5 Planning of the Benchmark Suite **20**
5.1 Finding Benchmark Programs . . . 20
5.2 Finding Relevant Auto-Tuners . . . 20
5.3 Systems Selected for Testing . . . 21
5.4 Criteria for an Ideal Benchmark Suite . . . 21
5.5 Research Questions . . . 22

6 Creating the Benchmark Suite **23**
6.1 Parameterizing of Kernels . . . 23
6.1.1 Ensuring that Parameters are Applied . . . 23
6.1.2 Triad . . . 23
6.1.3 MD . . . 27
6.1.4 Reduction . . . 30
6.1.5 Sort . . . 35
6.1.6 Stencil 2D . . . 40
6.1.7 Parameter Search Space . . . 40
6.1.8 Original Parameter Values . . . 41
6.1.9 Summary of Implemented Parameters . . . 43
6.2 Making the Benchmark Suite User Friendly . . . 47
6.2.1 Command-Line Interface . . . 47
6.2.2 Documentation . . . 48
6.2.3 Project Structure . . . 48
6.2.4 Userbase . . . 49

7 Testing the Benchmark Suite **50**
7.1 Implementing Benchmarks with Auto-Tuners . . . 50
7.1.1 CLTune . . . 50
7.1.2 KTT . . . 54
7.1.3 Kernel Tuner . . . 56
7.1.4 OpenTuner . . . 57
7.2 Benchmark Parameter Availability . . . 59
7.2.1 Sort . . . 59
7.2.2 Triad . . . 60
7.2.3 Reduction . . . 60
7.2.4 MD . . . 61
7.2.5 Stencil 2D . . . 61
7.3 System Setup . . . 62
7.4 Testing Process . . . 66
7.4.1 What to Test . . . 66
7.4.2 How to Test . . . 66
7.4.3 Correctness Verification . . . 67

8 Results and Discussion **68**
8.1 Parameter Evaluation . . . 68
8.1.1 OpenTuner . . . 68
8.1.2 Kernel Tuner . . . 70
8.1.3 CLTune . . . 70
8.1.4 KTT . . . 70

8.2 Benchmark Suite Evaluation 72
8.3 Research Questions 74
8.4 Auto-Tuner Evaluation 75
8.4.1 CLTune 75
8.4.2 KTT 75
8.4.3 Kernel Tuner 76
8.4.4 OpenTuner 76

9 Conclusion and Future Work **77**
9.1 Future Work 77

References **78**

Appendix A Parameter Research **82**

Appendix B Repository Readme **89**

Appendix C System Information **92**

Appendix D Setup **100**

List of Figures

Figure 1 Comparison of GPU and CPU architecture. 4
Figure 2 Illustration of a IBM Power System AC922 with four GPUs and two CPUs. . 6
Figure 3 Interconnect diagram for NVIDIA DGX-2. 7
Figure 4 NVSwitch topology on NVIDIA DGX-2. 8
Figure 5 Virtual Machine vs. Docker comparison of needed components. 10
Figure 6 NVIDIA Docker architecture example. 10
Figure 7 Radix sort example. 12
Figure 8 Reduction sum operation example. 13
Figure 9 Illustration of how the GPUs are connected to the CPUs on Mini Summit. . 64
Figure 10 Stencil 2D with problem size 1 for both NVIDIA DGX-2 and IBM Power System AC922. 69
Figure 11 Stencil 2D with problem size 4 for both NVIDIA DGX-2 and IBM Power System AC922. 70
Figure 12 Work per thread parameter and block size parameter for GTX 980 system for the MD benchmark in KTT. 71
Figure 13 Work per thread parameter and block size parameter for all systems for the MD benchmark in KTT. 72

List of Tables

Table 1 Abbreviations and explanations. xii
Table 2 Search techniques implemented in Kernel Tuner. 15
Table 3 Search techniques implemented in CLTune. 15
Table 4 Search techniques implemented in KTT. 16
Table 5 Search techniques implemented in OpenTuner. 17
Table 6 Search space for each parameterized benchmark. 40
Table 7 Original parameter values in the Triad benchmark. 41
Table 8 Original parameter values in the MD benchmark. 41
Table 9 Original parameter values in the Reduction benchmark. 42
Table 10 Original parameter values in the Sort benchmark. 42
Table 11 Original parameter values in the Stencil 2D benchmark. 43
Table 12 Parameters, descriptions and values for the Triad benchmark. 43
Table 13 Parameters, descriptions and values for the MD benchmark. 44
Table 14 Parameters, descriptions and values for the Reduction benchmark. 45
Table 15 Parameters, descriptions and values for the Sort benchmark. 46
Table 16 Constraints with descriptions for the Sort benchmark. 46
Table 17 Parameters, descriptions and values for the Stencil 2D benchmark. 47
Table 18 Implemented parameters in each auto-tuner for the Sort benchmark. 59
Table 19 Implemented parameters in each auto-tuner for the Triad benchmark. 60
Table 20 Implemented parameters in each auto-tuner for the Reduction benchmark. . . 60
Table 21 Implemented parameters in each auto-tuner for the MD benchmark. 61
Table 22 Implemented parameters in each auto-tuner for the Stencil 2D benchmark. . . 61
Table 23 Hardware specifications for NVIDIA GeForce GTX 980 based computer. . . . 62
Table 24 Hardware specifications for NVIDIA TITAN RTX based computer. 63
Table 25 Hardware specifications for NVIDIA Tesla T4 based computer. 63
Table 26 IBM Power System AC922 hardware specifications. 64
Table 27 NVIDIA DGX-2 hardware specifications. 65
Table 28 Parameters used in Convolution example in Kernel Tuner. 82
Table 29 Parameters used in Convolution Streams example in Kernel Tuner. 82
Table 30 Parameters used in Expdist example in Kernel Tuner. 82
Table 31 Parameters used in Matrix Multiplication example in Kernel Tuner. 83
Table 32 Parameters used in Point-in-Polygon example in Kernel Tuner. 83
Table 33 Parameters used in Reduction example in Kernel Tuner. 83
Table 34 Parameters used in SpMV example in Kernel Tuner. 83
Table 35 Parameters used in Stencil example in Kernel Tuner. 83
Table 36 Parameters used in Texture example in Kernel Tuner. 84
Table 37 Parameters used in Vector Add example in Kernel Tuner. 84
Table 38 Parameters used in Zero Mean Filter example in Kernel Tuner. 84
Table 39 Parameters used in Simple example in CLTune. 84
Table 40 Parameters used in Convolution Simple example in CLTune. 84
Table 41 Parameters used in Convolution example in CLTune. 85
Table 42 Parameters used in GEMM example in CLTune. 85
Table 43 Parameters used in Conv 3D example in KTT. 86
Table 44 Parameters used in Coulomb Sum 2D example in KTT. 87
Table 45 Parameters used in BICG example in KTT. 87

Table 46 Parameters used in Transpose example in KTT. 87
Table 47 Important compiler flags for FFT and MM benchmarks in OpenTuner. 88
Table 48 Important compiler flags for RT and TSP GA benchmarks in OpenTuner. . . 88

List of Listings

Listing 1 Example of launching a CUDA kernel with 128 blocks and 32 threads. 9
Listing 2 Previous `Triad`-program for launching the kernel with 128 blocks. 23
Listing 3 Parameterized `Triad`-program for launching with different block sizes. 24
Listing 4 `Triad`-program including the `WORK_PER_THREAD` to process more elements. . . 25
Listing 5 `Triad`-program including the `LOOP_UNROLL_TRIAD` to unroll the loop. 25
Listing 6 `Triad`-program for creating the input data to the kernel in SHOC. 26
Listing 7 Relevant part of changed code for choosing precision for computations. . . . 26
Listing 8 Previous `MD`-program for launching the kernel with block size 256 in in SHOC. 27
Listing 9 Parameterized `MD`-program for launching the kernel with different block sizes. 27
Listing 10 Previous `MD`-program for calling function with different precisions in SHOC. . 28
Listing 11 Changed code for choosing precision for computations. 28
Listing 12 Whether to use texture memory or not for input to `compute_lj_force`. . . . 29
Listing 13 Implementation of parameter `WORK_PER_THREAD` for `MD` 30
Listing 14 Relevant part of previous `Reduction` code for launching `reduce` kernel. . . . 30
Listing 15 Relevant part of parameterized code for launching `reduce` kernel. 30
Listing 16 Relevant part of previous code for calling function with different precision. . 31
Listing 17 Parameterized code for calling function with different precisions. 31
Listing 18 Relevant part of parameterized compile command. 31
Listing 19 Relevant part for setting the max register count to the compile command. . . 32
Listing 20 Command for running Reduction on four nodes using MPI. 32
Listing 21 Command for running Reduction on parameterized devices using MPI. . . . 32
Listing 22 Relevant part of unrolled loops in Reduction code. 32
Listing 23 Parameterized unrolled loops for Reduction. 33
Listing 24 Reduction part of texture memory implementation. 34
Listing 25 Previous unrolled loops in Sort SHOC. 35
Listing 26 Parameterized loop in Sort. 35
Listing 27 Original Sort block sizes. 36
Listing 28 Previous code for data sizes in SHOC. 36
Listing 29 Parameterized Sort data sizes. 36
Listing 30 Previous data sizes in SHOC. 37
Listing 31 Parameterized data Sort sizes in SHOC. 37
Listing 32 Custom CUDA uint8 struct. 38
Listing 33 Parameterized data sizes in Sort SHOC. 38
Listing 34 Relevant shared memory objects for constraint. 38
Listing 35 Previously non-inlined functions in Sort SHOC. 39
Listing 36 Parameterized inlined functions in Sort. 39
Listing 37 Parameterized number of GPUs for Stencil 2D. 40
Listing 38 Common code for CLTune algorithms 50
Listing 39 Common JSON saver for CLTune results. 52
Listing 40 Triad helper function for CLTune. 53
Listing 41 Shared memory in Reduction CLTune. 53
Listing 42 Compiler directive for Reduction shared memory. 53
Listing 43 Sort data size constraint for CLTune. 54
Listing 44 Incorrect global size type for KTT 55
Listing 45 Tuning techniques for KTT algorithms. 55

Listing 46 Triad parameters for KTT. 55
Listing 47 Triad parameters for Kernel Tuner. 56
Listing 48 Sort constraint for Kernel Tuner. 57
Listing 49 Triad parameters for OpenTuner. 57
Listing 50 Reduction parameters for OpenTuner. 58
Listing 51 Sort constraint for OpenTuner. 58
Listing 52 Stencil 2D for OpenTuner. 59
Listing 53 Topology for GTX 980 system 92
Listing 54 NVLink status for GTX 980 system. No results because the GPU does not have the possibility for NVLink connections 92
Listing 55 Information about the GTX 980 GPU when running the nvidia-smi command 92
Listing 56 Information about the CPU in the GTX 980 based system when running the lscpu command 93
Listing 57 Topology for Titan RTX system 93
Listing 58 NVLink status for Titan RTX system. The system has a possibility for two NVLink connections, but they are not in use on this specific computer. 94
Listing 59 Information about the Titan RTX GPU when running the nvidia-smi command 94
Listing 60 Information about the CPU in the RTX Titan based system when running the lscpu command 94
Listing 61 NVLink status for part of NVIDIA Tesla T4 system. No results because the GPU does not have the possibility for NVLink connections 95
Listing 62 Information about the first GPU when running the nvidia-smi command on the NVIDIA Tesla T4 system 95
Listing 63 Topology for DGX-2 (G=GPU). First matrix is the direct communication matrix, the second is PCI only. 95
Listing 64 NVLink status on DGX-2. The listing shows only GPU 0 and 15 because the command will print the same for every GPU in the system. 96
Listing 65 Information about the first GPU when running the nvidia-smi command on the DGX-2 96
Listing 66 Information provided about the CPUs in the DGX-2. 97
Listing 67 Topology for IBM Power System AC922 (2 GPUs) 97
Listing 68 NVLink status on IBM Power System AC922 (2 GPUs) 98
Listing 69 Information about the first GPU when running the nvidia-smi command on IBM Power System AC922 (2 GPUs). 98
Listing 70 Information provided about the CPUs in IBM Power System AC922 (2 GPUs). 98
Listing 71 Dockerfile for CLTune. 100
Listing 72 Dockerfile for KTT. 100
Listing 73 Dockerfile for Kernel Tuner. 101
Listing 74 Dockerfile for OpenTuner. 102
Listing 75 Slurm reserving of a single NVIDIA T4 and 40 CPU cores. 102

List of Abbreviations

Table 1: Abbreviations and explanations.

Abbreviation	**Explanation**
AI	Artificial Intelligence
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
FLOPS	Floating Point Operations Per Second
GPU	Graphics Processing Unit
GB	Gigabyte
GT/s	Giga Transfers per second
HPC	High Performance Computing
KB	Kilobyte
KTT	Kernel Tuning Toolkit
MB	Megabyte
MD	Molecular Dynamics
MPI	Message Passing Interface
OS	Operating System
PCIe	Peripheral Component Interconnect Express
SHOC	Scalable HeterOgeneous Computing
VM	Virtual Machine

1 Introduction

HPC(High Performance Computing) system have in the recent years become more and more heterogeneous, containing different architectures such as multicore CPUs and accelerators such as GPUs and FPGAs.

Because of the variety in architecture parameters, programs need to be optimized and performance-tweaked for the given architecture in order to get the best performance. Architectural features include special instructions to utilize the hardware parallelism of multi- and many-core processors, different types and sizes of memory, and other special architectural features, such as tensor cores on modern GPUs. This leads to a lot of performance tweaking for the given hardware in order to achieve the best performance.

Since there are a lot of tweakable parameters for performance tuning, such as varying block sizes, depths of loop unrolling and function inlining, one ends up with a huge search space that is possible, but very hard to do by hand. It is much more efficient to use an auto-tuner for this job, that automates the performance tuning of paramerterzied implementation. One problem is that there are no standard benchmark suite for measuring the performance of auto-tuners.

Several different auto-tuners have thus arisen to compete in this challenge in the best way possible on different kinds of codes on either CPUs or GPUs, including KTT, CLTune, Kernel Tuner, OpenTuner and ATF.

1.1 Motivation and Contribution

While there are several different auto-tuners, there have, to our knowledge, yet to be created a standard benchmark framework. One of the problems related to this is that the majority of the auto-tuners create proprietary benchmarking codes for their own auto-tuner, but these codes are typically not compatible or optimal to benchmark other auto-tuners.

We think that one of the reasons for the lack of a standard for auto-tuner benchmarks, is that the benchmarking codes generally are complicated and very targeted, so they are not very user friendly for other developers to take advantage of.

Joint work and the BAT Framework

This thesis builds on the fall semester project (specialzaiton project) that the author did jointly with Ingunn Sund. The author has also collaborated with her during this work.

Jointly we propose a standardized benchmark suite for auto-tuners, named BAT (**B**enchmark suite for **A**uto-**T**uners). BAT contains a set of different benchmarks for already implemented auto-tuners to retrieve different parameter configuration. This thesis describes the author's contribution to BAT and the methodology and work related to it he contributed.

We split up the work between us by each focusing on one the multi-GPU machines we had access to between us as follows:

Ingunn Sund working on developing BAT on

- an IBM Power System AC922 with four Tesla V100-SXM2 32 GB GPUs and
- a Supermicro server with 20 Tesla T4 GPUs

The author developing the BAT suite for

- an IBM Power System AC922 with two Tesla V100-SXM2 16 GB GPUs and
- an NVIDIA DGX-2 with 16 Tesla V100-SXM3 32 GB GPUs.

As was described in Ingunn's thesis, also developed and tested code on a system with a GeForce GTX 980 graphics card and a system with a Titan RTX card. We also decided to test on one singular graphics card of each other's biggest multi-GPU system. Thus Ingunn did tests on one Tesla V100 GPU of DGX-2 and I tested on one GPU of Supermicro server with the Tesla T4s.

Resarch Questions

Some research questions that have come to light include:

- Is SHOC a good benchmark suite to base a benchmark suite for auto-tuners for?
- Will this benchmark suite have enough GPU focus?
- Will it work with different types of auto-tuners?
- Will the optimal values for the implemented parameters differ for different systems?

1.2 Outline

The structure of the rest of this thesis consist of the following chapters:

Chapter **2. Background – GPUs and Docker** describes information that this thesis builds upon and uses later. It starts with describing basic GPU hardware and interconnects, and continues with software, benchmarks and algorithms. An overview of Docker and how we use Docker on NVIDIA GPU systems is also included.

Chapter **3. Benchmarking GPUs and Auto-tuning** An description of the SHOC benchmarking suite we use as well as our definition of auto-tuning and and overview of related auto-tuning frameworks.

Chapter **4. Related Work** describes related work for this thesis.

Chapter **5. Planning of the Benchmark Suite** contains the planning, work and research previous to creating the benchmark suite. This includes finding of the benchmark programs and auto-tuners, presenting the systems used for testing, a criteria for a successful benchmark suite and some research questions.

Chapter **6. Creating the Benchmark Suite** describes the proceeded way of implementing the benchmark suite by first parameterizing the kernels, showing the final search space, original parameter values, implemented parameter values and how the benchmark suite are made user friendly.

Chapter **7. Testing the Benchmark Suite** describes the implementation of the benchmarks for different auto-tuners, shows the parameters implemented for each auto-tuner, presenting information about the different systems used in testing and the process of the testing.

Chapter **8. Results and Discussion** shows the results obtained in this thesis and an analysis of them. It contains an evaluation of the parameters implemented, the benchmark suite and of the auto-tuners used.

Chapter **9. Conclusion and Future Work** describes the final result of this thesis and shows the significance of the work. At the end of the chapter it describes what could be done in the future to improve this benchmark suite.

Appendices

This thesis also contains of the following appendices:

Appendix **A. Parameter Research** contains tables of parameters obtained by researching repositories and papers of auto-tuners.

Appendix **B. Repository Readme** shows the readme for the Git repository of BAT.

Appendix **C. System Information** contains information about the systems used for testing in this thesis.

Appendix **D. Setup** shows Dockerfiles and Slurm command used for setup of BAT.

Finally, this thesis also contains of the following attached files and directories:

- **BAT** includes the source code of the benchmark suite.
- **BAT-results** includes the results from the benchmarks from using BAT.

  Investigating New GPU Features for Performance which is the specialization project (fall project) the author wrote in collaboration with Ingunn Sund.

2 Background - GPUs and Docker

This chapter includes a description of modern GPUs and the recent GPUs that were used for our benchmarks as well as the various types of interconnection networks these systems use for CPU - GPU and between GPUs and a brief introduction to the CUDA programming environment used on Nvidia GPUs. A description of how to use Docker on GPUs is also included.

2.1 GPU

A Graphics Processing Unit (GPU) is a processor that originally was made for rendering images or graphics on a computer screen. However in later years it has been used for other computational tasks and general purpose computing due to it's high number of cores and threads. It can be used in parallel processing and High Performance Computing (HPC) and . [1] The difference between a GPU and a Central Processing Unit (CPU) is that a CPU is designed with fewer processing cores in mind for executing tens of parallel tasks rather than a GPU that is designed with thousands of cores capable of executing simultaneously. [2] In Figure 1 below, it is shown an example architecture comparison between GPU and CPU with the number of cores in mind.

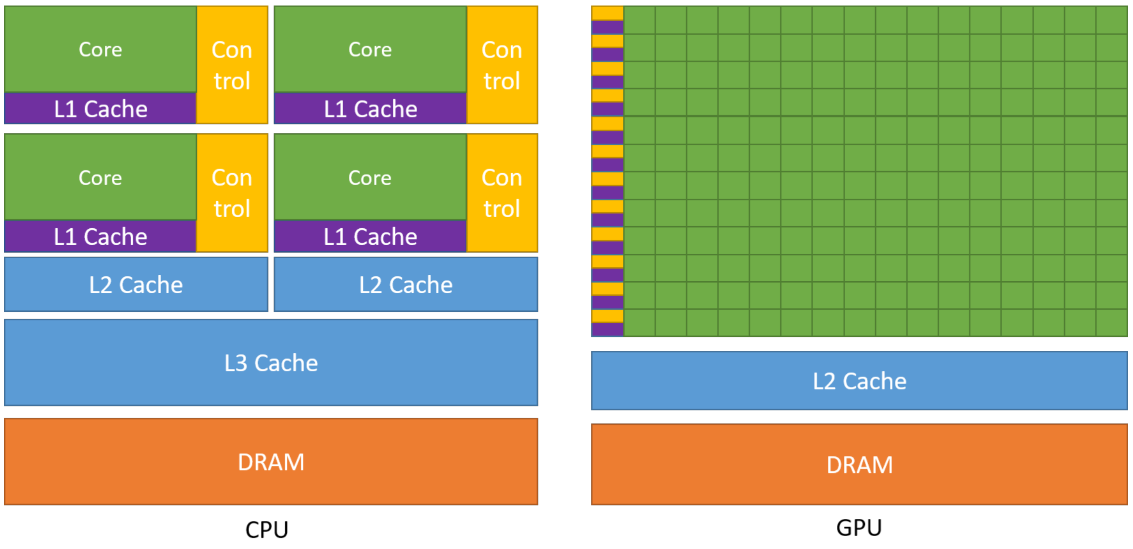

Figure 1: Comparison of GPU and CPU architecture. [2] Figure is used with permission from NVIDIA.

2.2 GPU Hardware

2.2.1 NVIDIA GeForce GTX 980

This section is from my specialization project, which is attached to this thesis.

The NVIDIA GeForce GTX 980 is a graphics card from 2014 with the Maxwell 2.0 architecture. It has 4 GB of GDDR5 memory with a bandwidth speed of 224 GB/s. It can achieve performances of 4.9 teraFLOPS for single precision and 155.6 gigaFLOPS for double precision. The GPU is equipped with 2048 CUDA cores. [3]

The Maxwell architecture introduced improved Streaming Multiprocessor (SM) architecture design. The architecture included more power efficient processors in numerous ways, for example by increasing the number of instructions per clock cycle. [4]

2.2.2 NVIDIA Tesla V100

This section is from my specialization project, which is attached to this thesis.

The NVIDIA Tesla V100 is a GPU based on the Volta architecture and there exists versions with 16 GB or 32 GB of the memory type HBM2 (High Bandwidth Memory) with a bandwidth speed of 900 GB/s. It can achieve performances of 125 teraFLOPS for deep learning (mixed precision), 15.7 teraFLOPS for single precision and 7.8 teraFLOPS for double precision. The GPU is equipped with 640 Tensor cores and 5120 CUDA cores. [5, p. 27]

Volta is the first architecture with specialized mixed-precision cores called NVIDIA Tensor Cores.

The Tensor Cores can perform one matrix multiply and accumulate operation in one clock cycle on a 4x4 matrix. Tensor Cores performs operations in mixed precision. The input data is half precision, multiplication is in half precision and accumulation is in single precision. This will lead to some precision loss, which deep neural networks can be tolerant to. HPC applications, on the other hand, cannot always handle the precision loss. [6]

2.2.3 NVIDIA TITAN RTX

This section is from my specialization project, which is attached to this thesis.

The NVIDIA TITAN RTX is a graphics card based on the Turing architecture. The GPU has 24 GB of GDDR6 GPU memory with a bandwidth of 672 GB/s. The card can achieve performance of 130 teraFLOPS with its 576 tensor cores made for mixed precision. The GPU also has 4608 CUDA cores. [7]

The Turing architecture provided new and improved Tensor cores. A part of the new design is the added INT8 and INT4 precision modes for inference operations. Another new feature on this graphics card is Ray Tracing cores. These cores came with the Turing architecture. [8, p. 4]

2.2.4 NVIDIA Tesla T4

The NVIDIA Tesla T4 is another GPU based on the Turing architecture and is in the NVIDIA's Tesla product lineup. [8, p. 17] It has 16 GB of GDDR6 GPU memory with a bandwidth of 320 GB/s. [9] It delivers almost double the memory and bandwidth of the previous Tesla P4 GPU and can achieve a performance of 65 teraFLOPS for mixed precision. The GPU has 2560 CUDA cores and 320 tensor cores.

2.2.5 IBM Power System AC922

This section is from my specialization project, which is attached to this thesis.

The IBM Power System AC922 is a system designed for giving great performance to data analytics, HPC applications and especially AI training. IBM Power System AC922 will be referred to as Power AC922 from now on. The system has two IBM POWER9 processors, the first chip with PCIe Gen4 which has twice the bandwidth of the previous PCIe generation. [10] [11]

The Power AC922 supports up to 4 or 6 NVIDIA Tesla V100 GPUs depending on the model, where the GPUs can have 16GB or 32GB memory. [5, p. 4-8] The GPUs are split evenly between two POWER9 CPUs. If there are a total of four GPUs, two will be directly connected to the first CPU and the other two will be connected to the second CPU, as can be seen in Figure 2. The GPUs are connected to their CPU and to any siblings with NVLink 2.0. The NVLink 2.0 channels are called NVLink Bricks, and each GPUs and CPUs has six of them. The NVLink Bricks are combined to achieve the highest bandwidth attainable. This means that if the Power AC922 has a total of four GPUs, there will be NVLink Brick groups of three (Figure 2), and with six GPUs there will be groups of two to ensure connection between a CPU and its connected GPUs and the connection between the GPUs connected to the same CPU. [5, p. 12-15]

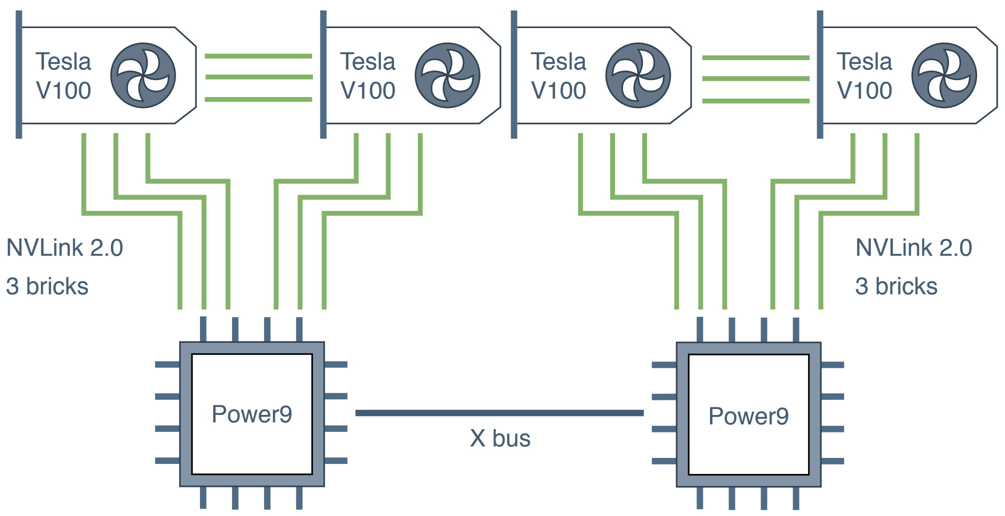

Figure 2: Illustration of a IBM Power System AC922 with four GPUs and two CPUs. Figure is made in collaboration with Ingunn Sund.

2.2.6 NVIDIA DGX-2

This section is from my specialization project, which is attached to this thesis.

The NVIDIA DGX is a series of systems created by NVIDIA for deep learning and complex AI applications. DGX-2 is version two of this system line and is approximately twice as fast as version one (DGX-1). It consists of 16 Tesla V100 GPUs with 32 GB of memory each, which is 512 GB in total. The system has in total 81 920 CUDA cores and 10 240 Tensor cores. [12] The system consists of two baseboards, with each having 8 GPUs. To increase the communication speed between the GPUs, they are connected with 12 NVSwitches, as can be seen in Figure 4. Six NVSwitches belongs to each baseboard, which means that the connection must traverse one NVSwitch if both GPUs are on the same baseboard, and through two NVSwitches if the GPUs are on different baseboards. All GPUs in this system have a bonded set of six NVLinks between each other as shown in Listing 63 in Appendix C.

The system has two Intel Xeon Platinum 8168 CPUs with 24 cores and a base clock frequency of 2.7 GHz. Between the two CPUs there is a QPI connection and each CPU has a PCIe connection with two PCIe switches to each GPU on their baseboard as can be seen in Figure 3. It can achieve the maximum performance for deep learning applications of 2 petaFLOPS which means that this system may be well suited for large workloads.

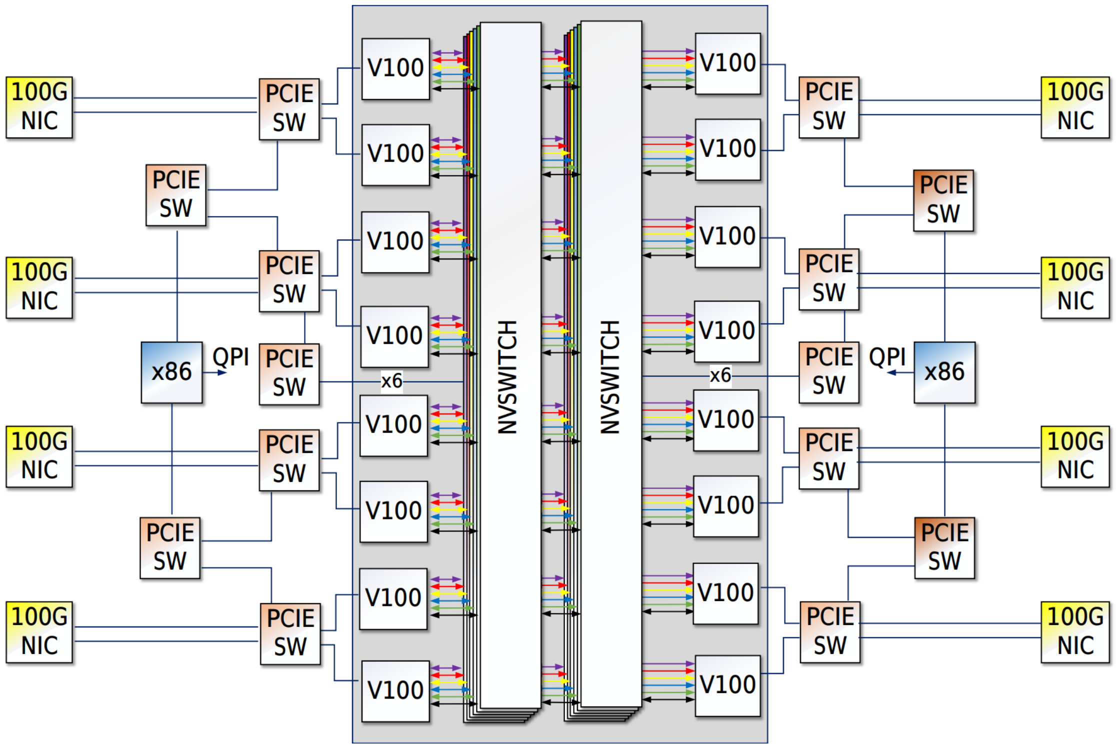

Figure 3: Interconnect diagram for NVIDIA DGX-2. [13, p. 19] Figure is used with permission from NVIDIA.

2.3 GPU- and Interconnect Communication

2.3.1 Messaging Passing Interface

This section is from my specialization project, which is attached to this thesis.

MPI (Message Passing Interface) is a standardized interface of protocols and functions for passing messages and communicating in a parallel environment with multiple computers. MPI provides a set of functions that are used in the implementations to communicate between the nodes. [14] There exist many different implementations, such as Open MPI [15], Spectrum MPI [16] and MPICH [17].

2.3.2 PCI Express

This section is from my specialization project, which is attached to this thesis.

PCI (Peripheral Component Interconnect) Express, or PCIe for short, is a bus standard that provides communication between connected components in a computer, such as hard drives and graphics cards. The normal connection between the GPU and CPU is done over PCIe. However, this can be a bottleneck due to its maximum transfer rate of 8 GT/s per lane for version 3 and 16 GT/s per lane for version 4. [18] [19]

2.3.3 NVLink 2.0 and NVSwitch

This section is from my specialization project, which is attached to this thesis.

NVIDIA NVLink is a GPU interconnect which offers much faster data transfer and is more scalable than using the PCIe. [20] NVLink can be used for both GPU to GPU and CPU to GPU connection. For each lane in the NVLink it has a transfer rate of 25 GT/s. [21, p. 115] This can reduce the bottleneck caused by transferring over the PCIe bus.

NVSwitch is a switch for connecting NVLinks together. It has 18 ports for connecting NVLinks and each NVLink connected can achieve simultaneously 25 GB/s bandwidth speed in both ways. In total the NVSwitch can therefore achieve a total bandwidth speed of 900 GB/s. [22, p. 3]

In Figure 4 below it is shown the connections consisting of NVLinks between the GPUs and NVSwitches. This is for the NVIDIA DGX-2 system.

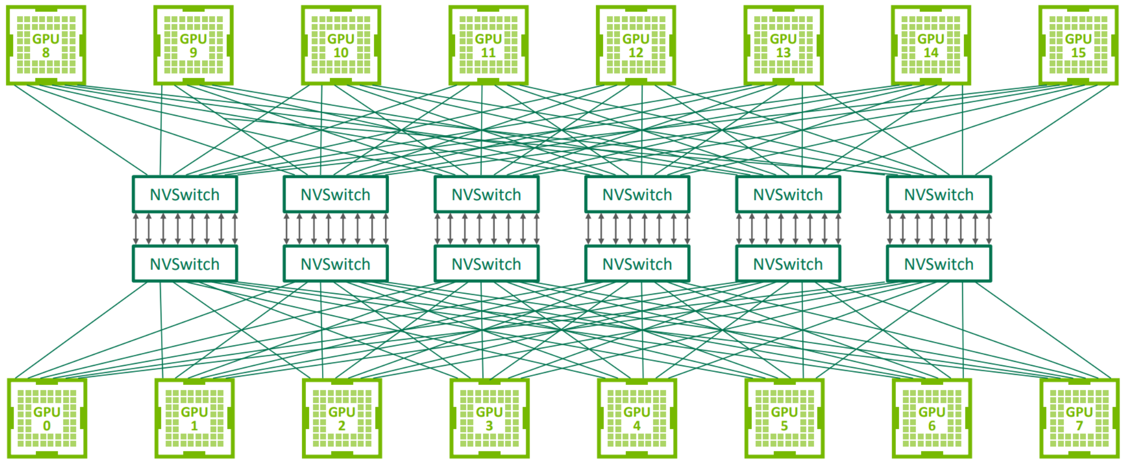

Figure 4: NVSwitch topology on NVIDIA DGX-2. [23, p. 8] Figure is used with permission from NVIDIA.

2.4 GPU Software

The standard programming model for NVIDIA GPUs is called CUDA (Compute Unified Device Architecture) and was introduced in 2006. CUDA is a general-purpose parallel computing platform and programming model and is designed for different programming languages such as C++, FORTRAN and Java.

A function that runs on the GPU is called a kernel. These functions can be executed in parallel on the device on a defined number of threads. Each of these threads are given a unique thread ID, which makes it trivial to compute elements in vectors, matrices or volumes. To make a kernel launchable from the host-code, one needs to define the called kernel with the `__global__` keyword. For kernels called from device kernels, `__device__` can be is used. [2]

An example of launching a CUDA kernel can be shown in Listing 1. The kernel is performing multiplication on two two input floating-point arrays and stores the results in the output array. The kernel is launched using 128 grids with 32 threads per block. Threads per block is also called block size. Total threads launched will therefore be $128 \times 32 = 4096$.

```
 1 __global__ void multiply(float* A, float B*, float* output) {
       int threadId = threadIdx.x + (blockIdx.x * blockDim.x);
 3     output[threadId] = A[threadId] * B[threadId];
   }
 5
   int main() {
 7     // ...
       // Set up A, B and output here
 9     // ...

11     multiply<<<128, 32>>>(A, B, output);
   }
```

Listing 1: Example of launching a CUDA kernel with 128 blocks and 32 threads per block. Line 1 to 4 is the kernel and 6 to 12 is the host-code function initiating the kernel.

2.5 Docker

Docker is a tool that allows users to run their application separately and isolated inside containers without the overhead of a full Virtual Machine. It lets the user specify versions of dependencies, environment settings and application files easily in a configurable Dockerfile. This ensures that the application is portable for different systems and guarantees that the different containers will run with the same files and dependencies. [24]

2.5.1 Docker Image

A Docker image is a standalone package of software that includes all files and configuration needed to run an application [25]. The image is a built based on a Dockerfile, and when built can be ran many times with the same environment settings the system and files. Examples of Dockerfiles can be seen in Appendix D.

2.5.2 Docker Container

A Docker container is a separate instance of a Docker Image. Docker containers are more lightweight than virtual machines (VM) and there are less required components. A VM needs to encapsulate the entire OS, but the containers only encapsulate the application and it's dependencies. For an illustration of the differences between virtual machines and docker containers lightweight than virtual machines see Figure 5 below.

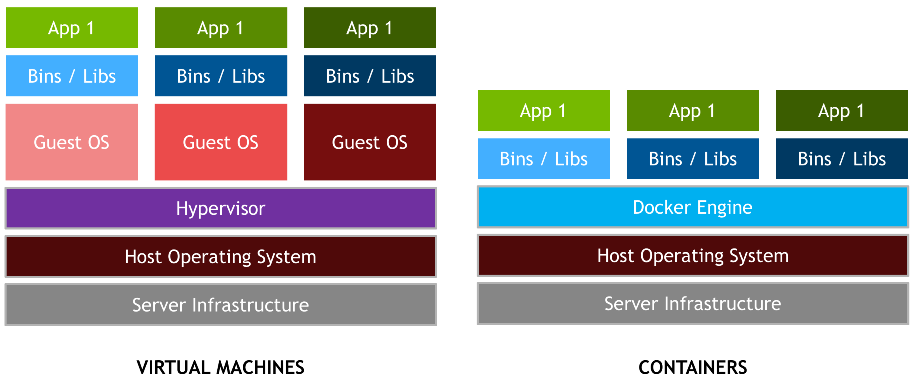

Figure 5: Virtual Machine vs. Docker comparison of needed components. This shows the overhead of using a VM instead of a Docker container. There is one host OS running three virtual machines and one running three Docker containers. [26] Figure is used with permission from NVIDIA.

2.5.3 NVIDIA Docker

NVIDIA Docker is an extension to Docker which lets users run containerized GPU accelerated applications. See Figure 6 below for an illustration of the Docker architecture and its connected components.

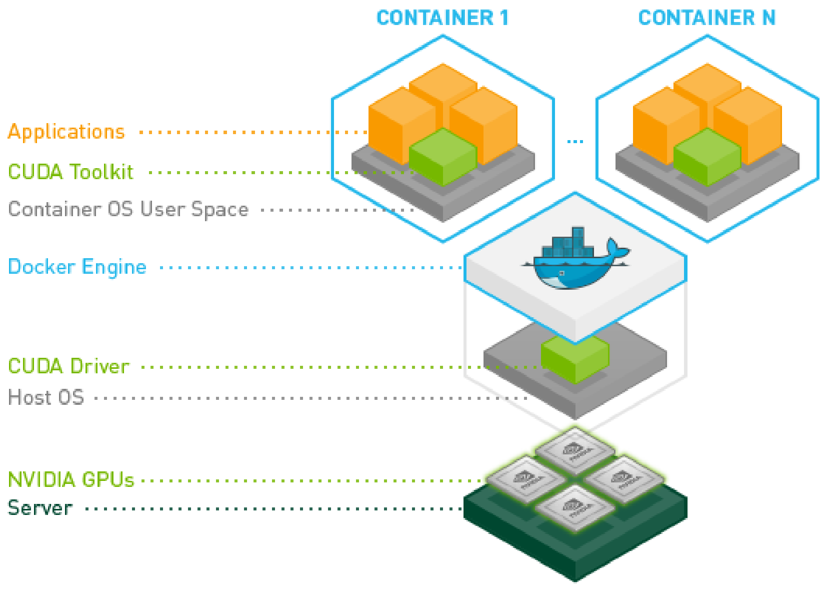

Figure 6: NVIDIA Docker architecture example containing components ranging from the NVIDIA GPUs to the applications ran inside Docker containers. The two Docker containers named *1* and *N*, represents that there can be N number of containers. [26] Figure is used with permission from NVIDIA.

3 Benchmarking GPUs and Auto-tuning

In this Chapter, we describe SHOC, the benchmarking suite we picked to build our BAT system. An overview of auto-tuning and auto-tuning frameworks related to our work, is also provided.

3.1 SHOC Benchmark Suite

SHOC (Scalable HeterOgeneous Computing) is a benchmark suite created by Anthony Danalis et al. [27] for measuring performance and stability of multi GPU and CPU systems. The benchmarks in SHOC is created for both CUDA and OpenCL programs and they are categorized into three different levels of benchmark applications, named 0, 1, 2 respectively. The first level measures low level architecture characteristics such as bandwidth and maximum FLOPS. The second measures performance for common parallel algorithms such as FFT, MD and SORT. The third level measures performance of real world applications.

The benchmarks also have three versions, named serial, embarrassingly parallel (EP) and true parallel (TP). Serial uses only one device to perform the tests, EP uses multiple devices to perform the tests, but does the same computation on all devices. TP uses multiple devices to perform the tests, and divide the workload between the devices. [28]

The sections below describes some of the different benchmarks in the SHOC benchmark suite.

3.1.1 Radix Sort

One of the benchmarks in SHOC is for measuring sorting performance on the device using an implementation of the radix sort algorithm. This is a sorting algorithm that groups the digits by its position and compares each digit one at a time in the selected positions. [29] SHOC's implementation sorts unsigned integer key-value pairs and supports the problem sizes 1, 8, 48 and 98 MB. The benchmark's performance of the sort kernel is measured in GB per seconds. [30]

In Figure 7 below, there is a step by step example of how the radix sort algorithm works for a smaller problem size than used in SHOC. Before the start, the initial order of the elements are in this example randomly selected. At the start it selects the rightmost digit and sorts the numbers with this digit only in mind. If two numbers are equal, the order they were in before are kept. This continues for the middle and the leftmost digits and is after that in the final sorted order.

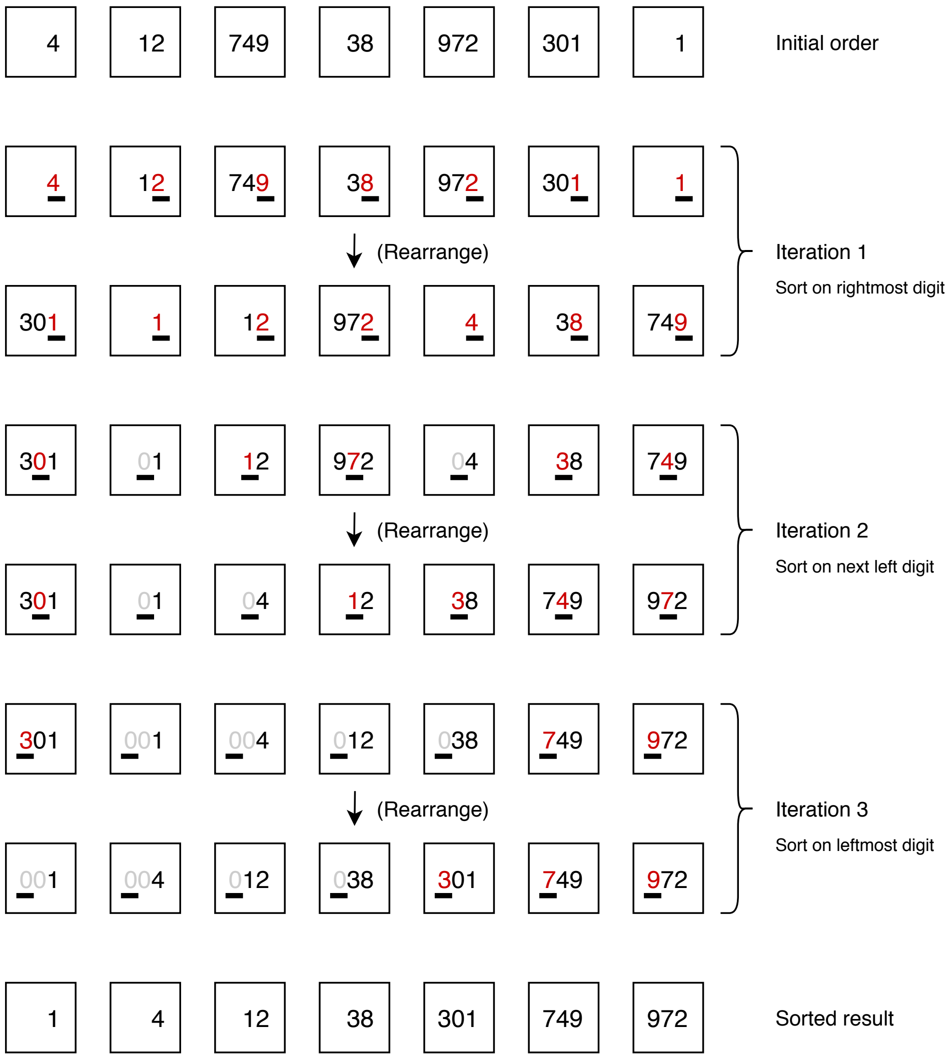

Figure 7: Radix sort example.

3.1.2 Triad

Triad is a benchmark in SHOC, that is based on the STREAM (Sustainable Memory Bandwidth in High Performance Computers) TRIAD benchmark [31]. STEAM is a benchmark set consisting of the benchmarks `COPY`, `SCALE`, `SUM` and `TRIAD`.

SHOC's implementation of the triad benchmark uses single precision computation with a problem size ranging from 16 KB to 64 MB and measures the sustainable memory performance of a series of dot product operations. In SHOC this benchmark does not have the possibility to select different problem size, and by default it tests all different sizes. [30] The algorithm is relatively simple and is shown in equation 1. In the figure A and B is input vectors, C is the output vector and s is the scalar which is used in a dot product with C.

$$C_i = A_i + s \cdot C_i \tag{1}$$

3.1.3 Reduction

Another benchmark in the SHOC benchmark suite is the reduction benchmark. Reduction is an algorithm that takes an input array of numbers and returns a single number using a operation on all the numbers. A type of reduction is a sum reduction, where the sum operation are applied to each element in the sequence. Other operators can be, minimum, maximum and count. [32, p. 546]. In SHOC it is made for measuring the performance of a large sum reduction operation. [28] This reduction benchmark is implemented for both single and double floating-point precision and have problem sizes ranging from 1 MB to 64 MB. There is also implemented a true parallel version for this benchmark. In this TP version, the data is communicated between the nodes using MPI. [27]

Figure 8 shows an example of a reduction algorithm with a sum operator. The input array shown at the top contains random generated numbers in the range 1 to 10. The algorithm in this example starts at the leftmost element and sums two and two numbers at a time. The previous sum is kept for the next iteration.

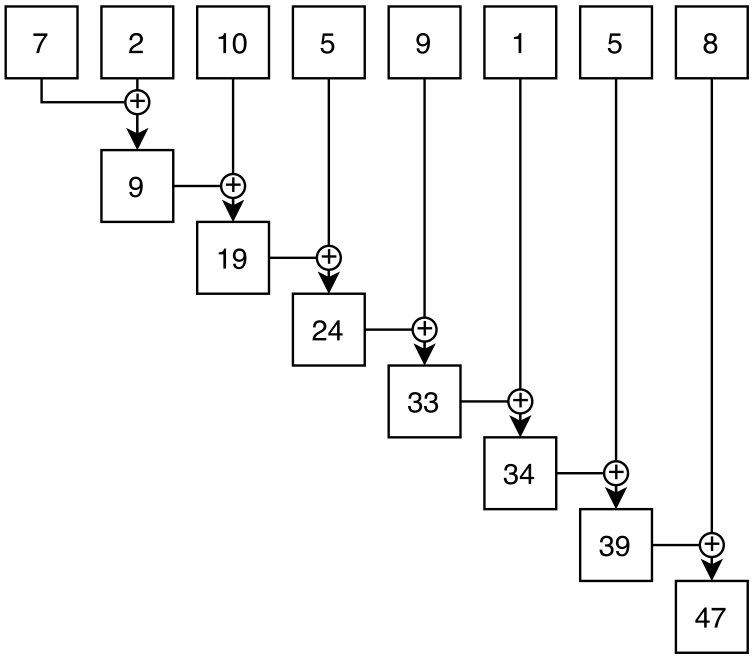

Figure 8: Reduction sum operation example.

3.1.4 Molecular Dynamics

The molecular dynamics (MD) benchmark in SHOC measures the performance of a MD problem named Lennard-Jones potential. This is a pair potential for calculating the potential energy between two atoms and is named after the mathematician Sir John Lennard-Jones. In SHOC, the Lennard-Jones force potential is computed in the MD benchmark and each thread on the GPU computes acceleration for a single atom with impact from the other atoms in the given space. This benchmark uses problem sizes based on number of atoms to compute, and it consists of problem sizes ranging from 12288 atoms to 73728 atoms. The input data for the kernel is both single and double floating-point precision. [28] [33]

Equation 2 shows the equation for the Lennard-Jones potential. Here r is the distance between the two atoms, ϵ is the energy strength between the two atoms and σ is the distance for the effective bond between the atoms.

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{2}$$

3.1.5 Stencil 2D

The Stencil 2D benchmark in SHOC measures the performance of 9-point stencil computations for a 2D array. The supported problem sizes for this benchmark supports are specified in MB and can be one of 512, 1024, 2048 and 4096. [28]

3.2 Auto-Tuning

Auto-tuning is tuning the performance of a program based on different defined parameters in the code to find the most optimal configuration of these parameters for a given system, such that these parameters can be used later to run with a good performance guarantee. Auto-tuning also allows for performance portability between different systems to ensure good performance not just for one system.

Different techniques for tuning for performance is available, where they change the way they iterate and select configurations of the whole search space, rather than brute forcing all configurations, which can be ineffective if at all possible for large search spaces. [34]

3.3 Auto-Tuning Frameworks

In the following sub sections, different auto-tuning frameworks are presented and the different tuning techniques are each of them are shown.

3.3.1 Kernel Tuner

Kernel Tuner is an auto-tuner made by Ben van Werkhoven that is created in Python and is described in his paper *Kernel Tuner: A search-optimizing GPU code auto-tuner* [35]. It can be installed as a Python package to be used to auto-tune GPU kernels in both CUDA and OpenCL. Kernel Tuner can also be used to auto-tune host code. All auto-tuning techniques that are implemented in Kernel Tuner are shown and described in Table 2 below. The default search strategy used is brute force.

Table 2: Search techniques implemented in Kernel Tuner.

Search Technique	**Description**
Brute force	Iterates over all different parameter configurations in the search space.
Random sample	Iterates over a random fraction sample of the search space.
Minimize	Selects a fraction of the search space using a minimizer.
Basin hopping	Selects a fraction of the search space using a minimizer.
Differential evolution	Uses differential evolution to select the parameter configurations.
Simulated annealing	Uses simulated annealing to select the parameter configurations.
Particle swarm optimization	Uses an implementation of particle swarm optimization that uses 20 particles for 100 iterations by default.
Genetic algorithms	Uses an implementation of genetic algorithms that uses a population of 20 for 100 generations by default.
Fire fly algorithm	Uses the fire fly algorithm with a setting with 20 fireflies for 100 iterations.
Bayesian Optimization	Uses the bayesian optimization to find configurations. This is implemented in the Kernel Tuner GitHub repository [36].

3.3.2 CLTune

CLTune is an auto-tuner made by Cedric Nugteren and Valeriu Codreanu mainly for tuning OpenCL kernels and is described in their paper *CLTune: A Generic Auto-Tuner for OpenCL Kernels* [37]. However it uses the abstraction layer CLCudaAPI [38] between the framework and the GPUs, which supports CUDA-code as well. It is created as a C++ framework and can compile and tune both CUDA and OpenCL kernels. All implemented auto-tuning techniques for CLTune can be found in Table 3 below. The default search strategy used is brute force.

Table 3: Search techniques implemented in CLTune.

Search Technique	**Description**
Full-search	Iterates over all different parameter configurations.
Random-search	Iterates randomly over a selected fraction of the search space.
Simulated annealing	Uses simulated annealing to select the parameter configurations with max temperature parameter as input. It finds parameter configuration in a selected fraction of the search space.
Particle swarm optimization	Uses an implementation of particle swarm optimization with swarm size input. It finds parameter configuration in a selected fraction of the search space.

3.3.3 Kernel Tuning Toolkit

Kernel Tuning Toolkit (KTT) is an auto-tuner by Filip Petrovič et al. described in *A Benchmark Set of Highly-efficient CUDA and OpenCL Kernels and its Dynamic Autotuning with Kernel Tuning Toolkit* [34]. The auto-tuner is based on CLTune, and does also work as a C++ framework for auto-tuning both CUDA and OpenCL kernels. It is different from CLTune in the internal backend of the tuner which was rewritten. [39] In Table 4 the different tuning techniques are described. The full search technique is the default technique.

Table 4: Search techniques implemented in KTT.

Search Technique	**Description**
Full Search	Iterates over all different parameter configurations.
Random Search	Iterates randomly over the full search space.
MCMC	Uses Markov chain Monte Carlo to select the parameter configurations.
Annealing	Uses simulated annealing to select the parameter configurations.

3.3.4 OpenTuner

OpenTuner is an auto-tuner made by Jason Ansel et al. and is described in the *OpenTuner: An Extensible Framework for Program Autotuning* paper [40]. OpenTuner is created in Python and can be installed as a Python package to auto-tune GPU kernels as well as host code and other programs defined by the user. The default technique is `AUC Bandit Meta Technique`, and the rest of the techniques can be seen in Table 5.

Table 5: Search techniques implemented in OpenTuner.

Search Technique	**Description**
AUC Bandit Meta Technique	Uses a combination of greedy mutation, differential evolution and two hill climber instances to find parameter configurations.
Pure Random	Finds parameter configurations randomly.
Nelder-Mead search	Uses Nelder-Mead to find parameter configurations. Variants: random, regular, right and multi.
Torczon	Uses torczon hillclimbers to find parameter configurations. Variants: random, regular, right and multi.
Greedy Mutation	Uses greedy mutation to find parameter configurations. Variants: uniform and normal.
Differential Evolution	Uses differential evolution to find parameter configurations.
Genetic algorithms	Uses genetic algorithms to find parameter configurations.
Particle swarm optimization	Uses particle swarm optimization to find parameter configurations.
Pattern search	Uses a type of pattern search to find parameter configurations.

4 Related Work

In this section some related work for auto-tuning and benchmarking is provided.

This part below is modified from my specialization project's abstract, which is attached to this thesis.

Investigating New GPU Features for Performance is the specialization project by myself and Ingunn Sund and it provides relevant work for this thesis for benchmarking and benchmark suites. It compares different GPUs and multi-GPU systems to evaluate the performance of hardware features such as Tensor Cores, NVLink and NVSwitch. Multi-GPU systems with special interconnect configurations were benchmarked and compared. The purpose of this evaluation is to find which of systems or GPUs that could be good for which tasks.

The systems and GPUs that were benchmarked are the NVIDIA DGX-2 and two versions of the IBM Power System AC922, a NVIDIA GeForce GTX 980 based system and a NVIDIA Titan RTX based system. The benchmarking was done with the benchmark suites SHOC, DeepBench, Tartan and Scope. The results from the benchmarks shows that DGX-2 was better at GPU-GPU communication than the Power AC922 systems, but the Power AC922 systems were better for CPU-GPU communication. Which system advisable to use will therefore depend on what kind of application that should run on it.

The Power AC922 systems seemed to have worse performance on the second NUMA node than the first. Choosing the right GPUs on this system can be essential for the best possible performance, depending on the application. An interesting result for the DGX-2 was that there were no significant difference in the performance for the GPU-GPU communication over NVSwitches for any GPU combination.

In the OpenTuner paper, they presents their tuning technique *AUC Bandit Meta Technique.* They include benchmarks such as *Poisson* from the PetaBricks project and the *Mario* benchmark, which has large search spaces of 10^{3657} and 10^{6328} respectively. These would not be possible to brute force in feasible time.

Kernel Tuner presents an auto-tuner that can tune OpenCL, CUDA and C kernels and showing tuning of GEMM kernel with 72.2 times speedup from brute force. For the implemented benchmarks, the search spaces are lower than in OpenTuner, and it may be able to brute force the optimal solutions within a shorter time frame. This is shown in the example in the paper, where the average runtime was lower than 2350 seconds.

In the paper for CLTune, they describe an auto-tuner that shows similar or better performance than the state-of-the-art 2D convolution auto-tuning. It was presented a search space for a matrix-multiplication benchmark with the search space of two-hundred thousand. This is however still much lower than in OpenTuner, and may be able to brute force within a short time frame.

KTT is an auto-tuner that in the paper show GPU implementations that outperform baseline CPU implementations for the Xeon Phis. They introduce a set of benchmarks for use in auto-tuner, however they are only available for their own auto-tuner. For the benchmarks in KTT the search space is also here lower than the ones used in OpenTuner, such as GEMM with 241600 different configurations.

TuneBench is a set of OpenCL kernels made for benchmarking tuning affects performance on various multi-core systems made by Alessio Sclocco. The source code is stored in it's GitHub page described as *Simple tunable OpenCL kernels for many-core accelerators* [41]. It includes benchmarks for MD, Reduction, Stencil and Triad.

ATF is an auto-tuner described in the paper *ATF: A Generic Auto-Tuning Framework* by [42] by Ari Rasch et al. ATF compares itself to OpenTuner and CLTune, and showing better tuning results for ATF. The comparison was done with comparing the runtime for the tuned programs. It is made to auto-tune programs made in all types of programming languages.

5 Planning of the Benchmark Suite

In this chapter, the planning, work and research previous to creating the benchmark suite are shown. This includes finding benchmark programs and auto-tuners suitable for the benchmark suite, selecting systems for testing, but also defining a list of criteria for the benchmark suite and some research questions.

While there are numerous different auto-tuners, there have yet to be created a benchmark for these to be classified as a standard. Therefore we see it as a necessity to create a standardized benchmark suite.

5.1 Finding Benchmark Programs

Before selecting benchmark programs and algorithms used as benchmarks, it was needed to find a benchmark set suitable for HPC and GPU based applications. GPU based benchmarks was selected as it is a central role in HPC and many auto-tuners tune GPU programs. These benchmark programs are selected both for this thesis and for the benchmark suite created in collaboration with Ingunn Sund. Different well known benchmark suites created for HPC was researched, and two benchmark suites became the finalists of selecting a benchmark suite. These was Rodinia and SHOC. Rodinia is described in the paper *Rodinia: A Benchmark Suite for Heterogeneous Computing* by Shuai Che et al. [43].

We chose in collaboration to use SHOC as a base for the benchmarks as we both had used that in our specialization project. Another reason to use SHOC was that it was made for multi-GPU benchmarking. We also both had some experience with CUDA, so the natural choice was to use the CUDA versions of the benchmarks in SHOC. In the KTT paper, they used a form of evaluating the benchmarks. We does not think this is necessary to use for the benchmarks from SHOC, as they are well used by others, and we therefore made an assumption that they decent to include.

5.2 Finding Relevant Auto-Tuners

To find relevant auto-tuners for testing the benchmarks, a thoroughly research was performed, with the mindset of collecting auto-tuners that: was well documented and easy to use. It is important to test with multiple auto-tuners to ensure that the benchmark suite works on many different systems. The auto-tuners would also need to work with tuning GPU code and be suitable for the benchmark programs selected. The auto-tuners should cover both tuning of GPU kernels and host code.

The selected auto-tuners was, OpenTuner, Kernel Tuner, CLTune and KTT. OpenTuner and Kernel Tuner because they they fit the need of tuning host code programs, and CLTune and KTT because they fit the need of tuning GPU kernels. Kernel Tuner also has the possibility to tune both host code and GPU kernels. This made the selection a broad variation of auto-tuners, making it a higher chance to be suitable for other auto-tuners.

As a part of this research, an analysis was performed of the different parameters used in these auto-tuners. The parameters found for most algorithms during this research can be seen in Appendix A. This was done in collaboration with Ingunn Sund to find commonly used parameters for auto-tuning, that could potentially be used for benchmarking auto-tuners.

5.3 Systems Selected for Testing

To ensure that the auto-tuner benchmarks worked as intended after implementation, tests should be completed on a varied selection of systems to ensure good coverage of auto-tuners, and that the benchmark suite would fit for other auto-tuners. Different systems was divided between me and Ingunn Sund. The systems chosen for this thesis was a GTX 980 based system, a RTX TITAN based system, a multi-GPU IBM AC922 system, a multi-GPU NVIDIA DGX-2 system and a single GPU from a multi-GPU system containing NVIDIA T4s.

5.4 Criteria for an Ideal Benchmark Suite

Before creating a benchmark suite, some points are defined as a check list to provide a guidance to what the benchmark suite should include to be ideal. For this thesis, it's not meant to compare auto-tuners, but to find out what a auto-tuner benchmark should include. Future work could include auto-tuner comparisons. Note that all points are not required to be fulfilled to succeed in creating the benchmark suite, but as many as possible is better. These points is created in collaboration with Ingunn Sund based on the findings of relevant benchmark programs and relevant auto-tuners. The criteria for a successful benchmark suite for auto-tuners, can be defined as:

- The benchmark suite should have:
  - HPC based benchmarks.
  - Parameterized algorithms as benchmarks.
  - Varied selection of benchmarks with different degree of complexity and scope.
  - Benchmarks that utilizes frameworks to enable running code on GPUs. There should be support for both CUDA and OpenCL to make it possible to run the code on both NVIDIA and AMD GPUs.
  - Benchmarks that can run on multi-GPU systems and distribute work on multiple nodes.
  - Support for different types of auto-tuners. If the auto-tuner does not support certain parameters or the auto-tuner only supports tuning of kernels, there should still not be a problem using the benchmarks.
  - Benchmarks that have been well tested with different auto-tuners and on different machines.
  - Examples of how to use the benchmarks with auto-tuners.
  - A way to compare auto-tuners with other auto-tuners.
- The parameterized algorithms should contain:
  - Both full programs and single GPU kernels.
  - Some algorithms with enough parameters that brute force is not efficient. There should be a variation of the search space size for the different algorithms.
  - Parameters that potentially could have different values on different machines or architectures.
  - Some benchmarks with possibility for restrictions or constraints on the parameters possible values.

- The benchmark suite should be user friendly by being:
  - A well structured project.
  - Easy to use.
  - A benchmark suite with good documentation. It should be clear what the project is and who could benefit from using it. There should be a guide for using the benchmark suite.

5.5 Research Questions

After planning the different aspects of the benchmark suite creation, a hypothesis was created: *A benchmark suite based on SHOC will fulfill the criteria made for a benchmark suite for auto-tuners.* A set of different research questions also occurred that we wanted to be answered. These are:

- Is SHOC a good benchmark suite to base a benchmark suite for auto-tuners for?
- Will this benchmark suite have enough GPU focus?
- Will it work with different types of auto-tuners?
- Will the optimal values for the implemented parameters differ for different systems?

6 Creating the Benchmark Suite

This section describes how the creating and implementation of the benchmark suite was done and how to make it user friendly. It starts with parameterizing of the different algorithms, then continues with how the constructing the benchmark suite was performed and making it user friendly.

6.1 Parameterizing of Kernels

The process of parameterizing an algorithm was to first extracted the benchmark code and build commands from SHOC into it's own isolated directory. Then locating code that could be parameterized and testing the algorithm with different values of the parameter. To automatically test different values of parameters, the auto-tuner OpenTuner was used for the isolated benchmark.

In the following sections, the parameters that were selected for each program-code are presented with an implementation.

6.1.1 Ensuring that Parameters are Applied

Some parameters are not guaranteed to have an effect on the actual executed program and needs to be tested if they are applied correctly. An example for a such parameter is loop unrolling. This is because not all loops can be unrolled and it is up to the compiler to do it. Therefore to ensure these types of parameters, the CUDA binary files were checked and the disassembled CUDA assembly information was extracted using the command `cuobjdump -sass <file_name>`. The assembly could then be compared with a CUDA assembly where loop unroll is disabled. However one can not guarantee that the loop is unrolled even if the contents are different, but if they are exactly the same, it will at least not unroll.

6.1.2 Triad

Block Size

`BLOCK_SIZE` is a parameter that was found to be quite commonly used in the investigated examples. The previous version of the Triad kernel and the launching from the host code in SHOC can be seen in Listing 2.

```
__global__ void triad(float* A, float* B, float* C, float s) {
    int gid = threadIdx.x + (blockIdx.x * blockDim.x);
    C[gid] = A[gid] + s*B[gid];
}

void RunBenchmark(...) {
    // ...
    const size_t blockSize = 128;
    // ...
    // ... for (int i ...
    size_t globalWorkSize = elemsInBlock / blockSize;
    triad<<<globalWorkSize, blockSize, 0, streams[0]>>>
            (d_memA0, d_memB0, d_memC0, scalar);
    // ...
}
```

Listing 2: Relevant part of previous Triad-program for launching the kernel with 128 as block size in SHOC. Kernel code is on line 1 to 5 and line 7 to 17 shows the host code.

The parameter was implemented by changing the launching arguments for the triad kernel, specified by the variable blockSize. A new integer input argument numberOfElements was also added to the kernel to let it know the length of the total elements, and to ensure the thread-id was not out of range of the arrays. The grid-size kernel launch argument globalWorkSize was also changed to adapt the block size change so that the kernel is launched with enough threads. This was done by using the ceil-function on the grid-size.

The possible values for this block size parameter is set to all integers in the range 1 to 1024. Even though it should be best to choose a block size that is a multiple of 32, due to the warp size being 32, the possible values for this parameter was chosen to be all integers in the range 1 to 1024. This was done to increase the search space for this program, to check if it was actual best to choose a multiple of 32 and if the auto-tuners could find this due to the larger search space. This implementation with the BLOCK_SIZE parameter can be seen in Listing 3.

```
 1 __global__ void triad(float* A, float* B, float* C, float s, int numberOfElements)
   {
 3     int gid = threadIdx.x + (blockIdx.x * blockDim.x);

 5     // Ensure that the current thread id is less than total number of elements
       if (gid < numberOfElements) {
 7         C[gid] = A[gid] + s*B[gid];
       }
 9 }

11 void RunBenchmark(...) {
       // ...
13     const size_t blockSize = BLOCK_SIZE;
       // ...
15     // ... for (int i ...
       size_t globalWorkSize = ceil((double)elemsInBlock / (double)blockSize);
17     triad<<<globalWorkSize, blockSize, 0, streams[0]>>>
               (d_memA0, d_memB0, d_memC0, scalar, elemsInBlock);
19     // ...
   }
```

Listing 3: Relevant part of parameterized Triad-program for launching the kernel with different block sizes. Kernel code is on line 1 to 5 and line 11 to 22 shows the host code.

Work Per Thread

Another parameter WORK_PER_THREAD was found to match the Triad-program. The code in the CUDA-kernel was changed to adapt this new parameter by creating a loop over the items processed. The thread id was therefore needed to be based on the WORK_PER_THREAD and the loop-iteration. A change in the grid size variable globalWorkSize was also needed to ensure less threads launched when increasing the number of elements processed in each thread.

The possible values for this parameter was chosen to be integers in the range 1 to 10. This was chosen to increase the search space and since this kernel is relatively small it could be possible that there was an improvement by computing more elements per thread. The implemented change is based on the previously parameterized Listing 3 with the BLOCK_SIZE parameter, and the new implementation including WORK_PER_THREAD can be seen in Listing 4.

```
   __global__ void triad(float* A, float* B, float* C, float s, int numberOfElements)
 2 {
       int gid = (threadIdx.x + (blockIdx.x * blockDim.x)) * WORK_PER_THREAD;
 4
       for (int i = 0; i < WORK_PER_THREAD; i++) {
 6         int threadId = gid + i;

 8         // Ensure that the current thread id is less than total number of elements
           if (threadId < numberOfElements) {
10             C[threadId] = A[threadId] + s*B[threadId];
           }
12     }
   }
14
   void RunBenchmark(...) {
16     // ...
       // ... for (int i ...
18     size_t globalWorkSize =
           ceil((double)elemsInBlock / (double)blockSize / (double) WORK_PER_THREAD);
20     // ...
   }
```

Listing 4: Relevant part of parameterized Triad-program for including the WORK_PER_THREAD to process more elements per thread in the kernel. Kernel code is on line 1 to 13 and from line 15 to 22 shows the host code.

Loop Unrolling

The parameter LOOP_UNROLL_TRIAD was implemented for the triad-kernel-loop that was added with the WORK_PER_THREAD-loop. This added a possibility to select if the loop was unrolled or not. The loop unrolling was added by using the compiler-directives #if and #pragma unroll. It was ensured that the compiler did not unroll by providing #pragma unroll(1) and letting the compiler choose unroll factor otherwise. This was not set to include a factor x such as #pragma unroll(x) due to issues with pragma unroll having higher priority by the compilers used in this thesis.

Possible values for this parameter is True (1) and False (0). The implemented change is based on the previously parameterized Listing 4, and the new implementation including LOOP_UNROLL_TRIAD can be seen in Listing 5.

```
__global__ void triad(float* A, float* B, float* C, float s, int numberOfElements)
{
    // ...
    #if LOOP_UNROLL_TRIAD
    #pragma unroll
    #else
    #pragma unroll(1)
    #endif
    for (int i = 0; i < WORK_PER_THREAD; i++) {
        // ...
    }
}
```

Listing 5: Relevant part of parameterized `Triad`-program for including the `LOOP_UNROLL_TRIAD` to choose whether to unroll the loop in the `triad` function or not. Kernel code is on line 1 to 13 and from line 5 to 9 is the loop-unroll compiler-directives shown.

Precision

`PRECISION` was another parameter found that could be implemented in the `Triad` benchmark. This was done by adding input data as either single-precision (`float`) or double-precision (`double`) floating-points arrays. This could be interesting to see if the precision of the data had any impact on performance of the benchmark. The implementation set the compiler-directives `#if` and `#define` to select which precision (`float` or `double`) to choose. In the host code, all input data types was changed from the previously set `float` input to match the compile-time defined `DATA_TYPE`. A selection of relevant parts of the `DATA_TYPE` variable in the host code in previous SHOC version can be seen in Listing 6.

```
void RunBenchmark(...) {
    float *h_mem;
    cudaMallocHost((void**) &h_mem, sizeof(float) * numMaxFloats);
    // ...
    float* d_memA0, *d_memB0, *d_memC0;
    // ...
    float scalar = 1.75f;

    // ... for (int i ...
    int elemsInBlock = blockSizes[i] * 1024 / sizeof(float);
    for (int j = 0; j < halfNumFloats; ++j) {
        h_mem[j] = h_mem[halfNumFloats + j] = (float) (drand48() * 10.0);
    }
}
```

Listing 6: Relevant part of previous `Triad`-program for creating the input data to the kernel in SHOC. This is a part of the host code that launches the kernel.

Possible values for this parameter are `float` (32) and `double` (64). The new implemented parameter are also based on the previously parameterized Listing 5, and the new one can be shown in Listing 7.

```
// Select which precision that are used in the calculations
#if PRECISION == 32
    #define DATA_TYPE float
#elif PRECISION == 64
    #define DATA_TYPE double
#endif

void RunBenchmark(...) {
    DATA_TYPE *h_mem;
    cudaMallocHost((void**) &h_mem, sizeof(DATA_TYPE) * numMaxFloats);
    // ...
    DATA_TYPE* d_memA0, *d_memB0, *d_memC0;
    // ...
    DATA_TYPE scalar = 1.75f;
```

```
16     // ... for (int i ...
       int elemsInBlock = blockSizes[i] * 1024 / sizeof(DATA_TYPE);
18     for (int j = 0; j < halfNumFloats; ++j) {
           h_mem[j] = h_mem[halfNumFloats + j] = (DATA_TYPE) (drand48() * 10.0);
20     }
   }
```

Listing 7: Relevant part of changed code for choosing whether to use single-precision floating-point or double-precision floating-point for computations. The compiler-directives are on line 2 to 6 and from line 8 to 22 shows the host code.

6.1.3 MD

Block Size

The `BLOCK_SIZE` parameter can be used in the `MD`-program as well, and can be implemented similarly as the `Triad` version seen in Listing 3. The part of the previous SHOC code that is relevant for this parameter can be seen in Listing 8.

```
 1 __global__ void compute_lj_force (...) {
       int idx = blockIdx.x*blockDim.x + threadIdx.x;
 3     posVecType ipos = position[idx];
       // ...
 5 }

 7 void runTest (...) {
       // ...
 9     int blockSize = 256;
       int gridSize  = nAtom / blockSize;
11     // ...
   }
```

Listing 8: Relevant part of previous `MD`-program for launching the kernel with block size 256 in SHOC. Kernel code is on line 1 to 6 and line 8 to 14 shows the host code.

The possible values for this block size parameter is set to all integers in the range 1 to 1024. It was added a `ceil`-function around the `gridSize` launching parameter in the host code and in the kernel the element computation was wrapped in an if-check to ensure not computing too many elements. This implementation with the `BLOCK_SIZE` parameter can be seen in Listing 9.

```
   __global__ void compute_lj_force (...) {
 2     int idx = blockIdx.x*blockDim.x + threadIdx.x;
       // ...
 4     // Ensure that the current thread id is less than total number of elements
       if (idx < inum) {
 6         // Position of this thread's atom
           posVecType ipos = position[idx];
 8     // ...
   }
10
   void runTest (...) {
12     // ...
       int blockSize = BLOCK_SIZE;
14     int gridSize  = ceil((double)nAtom / (double)blockSize);
16      compute_lj_force<T, forceVecType, posVecType, useTexture, texReader>
            <<<gridSize, blockSize>>>
18          (...);
        // ...
20  }
```

Listing 9: Relevant part of parameterized MD-program for launching the kernel with different block sizes. Kernel code is on line 1 to 10 and from line 12 to 22 shows the host code.

Precision

SHOC's benchmark for MD measures the performance in both single- and double precision. This was split up with the use of the parameter `PRECISION` for this program by wrapping the `runTest` function calls into compiler-directives inside the `RunBenchmark` function. This was done similarly to how `PRECISION` was implemented for the `Triad`-program seen in Listing 7. Related MD code from SHOC benchmark can be seen in Listing 10.

```
    template <class T, class forceVecType, class posVecType, bool useTexture,
2             typename texReader>
    __global__ void compute_lj_force(...) { ... }
4
    // ...
6
    void RunBenchmark(...) {
8       // ...
        cout << "Running single precision test" << endl;
10      runTest<float, float3, float4, true, texReader_sp>("MD-LJ", op);
        cout << "Running double precision test" << endl;
12      runTest<double, double3, double4, true, texReader_dp>("MD-LJ-DP", op);
        // ...
14  }

16  template <class T, class forceVecType, class posVecType, bool useTexture,
              typename texReader>
18  void runTest(...) { ... }
```

Listing 10: Relevant part of previous MD-program for calling the benchmark function with different precisions in SHOC. Kernel code is on line 1 to 2 and from line 7 to 19 shows the host code.

The SHOC code used class template types for the input to the kernel. This was in this implementation replaced with compiler-directives to be suitable for the selected auto-tuners. This was done due to the auto-tuners not being able to use templated code because of the auto-tuners compiling the code with `extern "C"` which is not compatible with templates due to `extern "C"` disabling name mangling and templates depend on them. [44]

The possible values for this parameter are `float` (32) and `double` (64) and the new implemented parameter are based on the previously MD parameterized Listing 9. The types `T`, `forceVecType`, `posVecType`, `useTexture` and `texReader`, was therefore replaced with compiler-directives as shown in Listing 11.

```
    // Select which precision that are used in the calculations
2   // And define the replacements for the template inputs
    #if PRECISION == 32
```

```
 4      #define T float
        #define forceVecType float3
 6      #define posVecType float4
        #define texReader texReader_sp
 8  #elif PRECISION == 64
        #define T double
10      #define forceVecType double3
        #define posVecType double4
12      #define texReader texReader_dp
    #endif
14
    // ...
16
    void RunBenchmark(...) {
18  #if PRECISION == 32
        cout << "Running single precision test" << endl;
20      runTest<float, float3, float4, true, texReader_sp>("MD-LJ", op);
    #elif PRECISION == 64
22      cout << "Running double precision test" << endl;
        runTest<double, double3, double4, true, texReader_dp>("MD-LJ-DP", op);
24  #endif
    }
```

Listing 11: Relevant part of changed code for choosing whether to use single-precision floating-point or double-precision floating-point for computations. Kernel relevant code is on line 1 to 13 and from line 17 to 26 shows the host code.

Texture Memory

The parameter `TEXTURE_MEMORY` was implemented to choose whether to use texture memory or not for the input data to the `compute_lj_force` kernel. Texture memory is a type of read-only memory access pattern. SHOC's benchmark for `MD` already contained the use of texture memory, so there was no needed changes regarding implementing that. This was implemented based on the previous parameter `PRECISION`, which can be seen in Listing 11. The change needed was to set the `#define useTexture TEXTURE_MEMORY` compiler directive instead of the input to the `compute_lj_force` kernel. The possible values for this parameter is `True` (1) and `False` (0). This implementation is based on the previous Listing 11 and can be seen in Listing 12.

```
 1  #define useTexture TEXTURE_MEMORY

 3  // ...

 5  void RunBenchmark(...) {
    #if PRECISION == 32
 7      cout << "Running single precision test" << endl;
        runTest<float, float3, float4, TEXTURE_MEMORY, texReader_sp>(...);
 9  #elif PRECISION == 64
        cout << "Running double precision test" << endl;
11      runTest<double, double3, double4, TEXTURE_MEMORY, texReader_dp>(...);
    #endif
13  }
```

Listing 12: Relevant part of changed code for choosing whether to use texture memory or not for the input data to the `compute_lj_force` kernel. Kernel code is on line 1 and from line 5 to 14 shows the host code.

Work Per Thread

The parameter `WORK_PER_THREAD` was implemented similarly for `MD` as for `Triad` shown in Listing 4. It was done by creating a loop over the items processed in the kernel and divide the grid-size parameter on the work per thread value.

The possible values for this parameter was integers in the range 1 to 5. It was chosen to be this rather than the 1 to 10 used in `Triad`, as this is a more time consuming benchmark, so the total time used for running the program with the auto-tuners would be larger for this. This is based on the previous `TEXTURE_MEMORY` as shown in Listing 12 and the relevant new implemented code can be seen in Listing 13.

```
// Global ID - "WORK_PER_THREAD" atoms per thread
    int idx = (blockIdx.x*blockDim.x + threadIdx.x) * WORK_PER_THREAD;

    for (int i = 0; i < WORK_PER_THREAD; i++) {
        int threadId = idx + i;

        // Ensure that the current thread id is less than total number of elements
        if (threadId < inum) {
            // Position of this thread's atom
            posVecType ipos = position[threadId];
            // ...
```

Listing 13: Relevant part of parameterized `MD`-program for including the `WORK_PER_THREAD` to process more elements per thread in the kernel. Kernel code is on line 1 to 5 and from line 11 to 22 shows the host code.

6.1.4 Reduction

Grid- and Block Size

In the reduction implementation in SHOC, we can see that the variables `num_blocks` and `num_threads` are used to specify the grid size and block size respectively. See Listing 14 for the SHOC code for this.

```
int num_threads = 256;
int num_blocks = 64;
// ...
reduce<T,256><<<num_blocks,num_threads, smem_size>>>(d_idata, d_odata, size);
```

Listing 14: Relevant part of previous `Reduction` code for launching `reduce` kernel. Kernel is launched with 64 blocks and 256 threads per block.

I decided to parameterize these constants as `GRID_SIZE` and `BLOCK_SIZE`. The values for both `GRID_SIZE` and `BLOCK_SIZE` was chosen to be 2^i where i is in the range $[0, 11]$, but with the exception that `BLOCK_SIZE` needed to be below max threads per block for the selected GPU. Then the parameterized part of the code can be seen in Listing 15.

```
int num_threads = BLOCK_SIZE;
int num_blocks = GRID_SIZE;
// ...
reduce<T, BLOCK_SIZE><<<num_blocks,num_threads, smem_size>>>
```

```
    (d_idata, idataTextureObject, d_odata, size);
```

Listing 15: Relevant part of parameterized code for launching `reduce` kernel.

Precision

The reduction implementation does run both single and double precision by default. This is similarly to the MD implementation and I wanted to add this parameter in this implementation as well. SHOC's implementation can be found in Listing 16.

```
void RunBenchmark(...) {
    // ...
    RunTest<float>("Reduction", resultDB, op);
    // ...
    RunTest<double>("Reduction-DP", resultDB, op);
    // ...
}
```

Listing 16: Relevant part of previous code for calling function with different precision.

I chose to implement this like in the MD benchmark, where the `runTest` function calls was wrapped into compiler-directives inside the `RunBenchmark` function. See Listing 17 for code with implemented precision parameter.

```
void RunBenchmark(...) {
#if PRECISION == 32
    cout << "Running single precision test" << endl;
    RunTest<float>("Reduction", op);
#elif PRECISION == 64
    cout << "Running double precision test" << endl;
    RunTest<double>("Reduction-DP", op);
#endif
}
```

Listing 17: Parameterized code for calling function with different precisions.

Compiler Optimizations

During the parameterization of the reduction benchmark I noticed that the compiler could be presented different optimizations. These compiler optimizations would also work with auto-tuners. By default SHOC compiled with the compiler optimization `O2`. So this could change the performance somewhat of the benchmark, if changed. The different compiler optimizations that I found to match the benchmark, was compiler optimization for the host- and device-code, the `fast-math` compiler flag and the flag for setting `max-register`. This was done by providing the different parameters in the compile commands as shown in Listing 18.

```
nvcc -I ./cuda-common -I ./common -use_fast_math -maxrregcount={MAX_REGISTERS} \
-O{COMPILER_OPTIMIZATION_HOST} -Xptxas -O{COMPILER_OPTIMIZATION_DEVICE} -c \
./reduction/reduction.cu
```

Listing 18: Relevant part of parameterized compile command.

The values that was found to match the parameters was for COMPILER_OPTIMIZATION_HOST and COMPILER_OPTIMIZATION_DEVICE integers in the range 1 to 4, for USE_FAST_MATH True (1) and False (0) and for MAX_REGISTERS these values -1, 20, 40, 60, 80, 100, 120. For the MAX_REGISTERS parameter, the value -1 is meant to specify if the parameter is disabled. This can be done before calling the compile command like shown in Listing 19.

```
if MAX_REGISTERS != -1:
    compile_command += f'-maxrregcount={MAX_REGISTERS} '
```

Listing 19: Relevant part for setting the max register count to the compile command.

Number of GPUs

Since this benchmark is possible to run on multiple GPUs in SHOC, it was discovered that a parameter for selecting the number of GPUs to perform the computations on was possible. This is however a more special parameter as it is not accessible in the code, but from the command-line to start the benchmark. This means that for auto-tuners to tune this parameter, needs to be able to specify which command to run for the test. The SHOC benchmark uses MPI for communicating of the data between the GPU nodes, and the following command in Listing 20 is the one used to run the benchmark on four GPUs with indexes ranging from 0 to 3.

```
mpirun -np 4 --allow-run-as-root ./reduction -s 1 -d 0,1,2,3
```

Listing 20: Command for running Reduction on four nodes using MPI.

Using a parameter for selecting a number of connected GPUs to perform the benchmark on, can simply be implemented as in the following Listing 21. Possible values for this parameter was selected as all legal values ranging from 1 to number of connected GPUs.

```
devices = ','.join([str(i) for i in range(0, GPUS)])
mpirun -np {GPUS} --allow-run-as-root ./reduction -d {devices}
```

Listing 21: Command for running Reduction on parameterized devices using MPI.

Loop Unrolling

In the SHOC reduction benchmark, there was found two already unrolled loops. These loops can be seen in Listing 22.

```
if (blockSize >= 512) {
    if (tid < 256) {
        sdata[tid] += sdata[tid + 256];
    }
    __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) {
        sdata[tid] += sdata[tid + 128];
    }
    __syncthreads();
}
```

```
if (blockSize >= 128) {
    if (tid < 64)  { sdata[tid] += sdata[tid + 64]; }  __syncthreads();
}
if (tid < warpSize) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8)  sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4)  sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2)  sdata[tid] += sdata[tid + 1];
}
```

Listing 22: Relevant part of unrolled loops in Reduction code.

This meant that it would be possible to reroll the loops and provide a parameter to the compiler to tell if it should try to unroll the loops. The result of the rerolled loops was two separate loops and the possible values for these parameters is `True` (1) and `False` (0). These loops combined with the compiler-directives for the parameters can be seen in Listing 23.

```
#if LOOP_UNROLL_REDUCE_1
#pragma unroll
#else
#pragma unroll(1)
#endif
for (int i = BLOCK_SIZE; i > 64; i /= 2) {
    if (blockSize >= i) {
        if (tid < (i / 2)) {
            sdata[tid] += sdata[tid + (i / 2)];
        }
        __syncthreads();
    }
}

#if LOOP_UNROLL_REDUCE_2
#pragma unroll
#else
#pragma unroll(1)
#endif
for (int i = 64; i > 1; i /= 2) {
    if (tid < warpSize) {
        if (blockSize >= i) {
            sdata[tid] += sdata[tid + (i / 2)];
        }
    }
}
```

Listing 23: Parameterized unrolled loops for Reduction.

Texture Memory

This benchmark did not contain the use of texture memory in the SHOC version, but I found it suitable to implement it for this benchmark. This was done by creating a `cudaTextureObject_t` to store the information about the texture data and inserting data to it if the compiler directive is true. Inside the GPU kernel it retrieves the data neededed using the function `tex1Dfetch` or convertTextureObjectToDouble, depending on the precision used. Possible values for this parameter True (1) and False (0). The implementation with both the host code and kernel can be seen in Listing 24.

```
// ...
__inline__ __device__ double convertTextureObjectToDouble(cudaTextureObject_t
    textureObject, const int &position) {
    uint2 values = tex1Dfetch<uint2>(textureObject, position);
    return __hiloint2double(values.y, values.x);
}

__global__ void
reduce(..., cudaTextureObject_t idataTextureObject, ...) {
    // ...
    #if TEXTURE_MEMORY
        #if PRECISION == 32
            sdata[tid] += tex1Dfetch<T>(idataTextureObject, i) + tex1Dfetch<T>(
    idataTextureObject, i+blockSize);
        #elif PRECISION == 64
            sdata[tid] += convertTextureObjectToDouble(idataTextureObject, i) +
    convertTextureObjectToDouble(idataTextureObject, i+blockSize);
        #endif
    #else
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    #endif
    // ...
}

// ...

    cudaTextureObject_t idataTextureObject = 0;

#if TEXTURE_MEMORY
    // Setup the texture memory
    // Create the texture resource descriptor
    cudaResourceDesc resourceDescriptor;
    memset(&resourceDescriptor, 0, sizeof(resourceDescriptor));
    resourceDescriptor.resType = cudaResourceTypeLinear;
    resourceDescriptor.res.linear.devPtr = d_idata;
#if PRECISION == 32
    resourceDescriptor.res.linear.desc.f = cudaChannelFormatKindFloat;
#elif PRECISION == 64
    resourceDescriptor.res.linear.desc.f = cudaChannelFormatKindUnsigned;
#endif
    resourceDescriptor.res.linear.desc.x = 32;
#if PRECISION == 64
    resourceDescriptor.res.linear.desc.y = 32;
#endif
    resourceDescriptor.res.linear.sizeInBytes = size * sizeof(T);

    // Create the texture resource descriptor
    cudaTextureDesc textureDescriptor;
    memset(&textureDescriptor, 0, sizeof(textureDescriptor));
    textureDescriptor.readMode = cudaReadModeElementType;

    // Create the texture object
    cudaCreateTextureObject(&idataTextureObject, &resourceDescriptor,
        &textureDescriptor, NULL);
#endif
    // ...
    reduce<T, BLOCK_SIZE><<<num_blocks,num_threads, smem_size>>>
        (d_idata, idataTextureObject, d_odata, size);
```

Listing 24: Reduction part of texture memory implementation.

6.1.5 Sort

The sort benchmark was inspired by one of KTT's auto-tuner example implementations found on it's GitHub page, but parameters was extended even more. The parameters that was inspired was the block size and data size parameters.

Loop Unrolling

SHOC's sort benchmark contained two loops that was already unrolled. These are in the kernels `scanLSB` and `scanLocalMem` and the parameters was named `LOOP_UNROLL_LSB` and `LOOP_UNROLL_LOCAL_MEMORY`. These could be rerolled and added a parameter to specify if the compiler should try to unroll or not. Possible values for both these parameters is `True` (1) and `False` (0). The two loops are very similar so only one of them will be shown here. The previous code in SHOC for the loop in the `scanLocalMem` kernel can be seen in Listing 25.

```
uint t;
s_data[idx] = val;      __syncthreads();
t = s_data[idx -   1];  __syncthreads();
s_data[idx] += t;       __syncthreads();
t = s_data[idx -   2];  __syncthreads();
s_data[idx] += t;       __syncthreads();
t = s_data[idx -   4];  __syncthreads();
s_data[idx] += t;       __syncthreads();
t = s_data[idx -   8];  __syncthreads();
s_data[idx] += t;       __syncthreads();
t = s_data[idx -  16];  __syncthreads();
s_data[idx] += t;       __syncthreads();
t = s_data[idx -  32];  __syncthreads();
s_data[idx] += t;       __syncthreads();
t = s_data[idx -  64];  __syncthreads();
s_data[idx] += t;       __syncthreads();
t = s_data[idx - 128];  __syncthreads();
s_data[idx] += t;       __syncthreads();
```

Listing 25: Previous unrolled loops in Sort SHOC.

These parameters was implemented similarly to as the unroll parameters in the Reduction benchmark. The parameterized loop for the `LOOP_UNROLL_LOCAL_MEMORY` parameter can be seen in Listing 26.

```
uint t;
s_data[idx] = val;
__syncthreads();

#if LOOP_UNROLL_LOCAL_MEMORY
```

```
#pragma unroll
#else
#pragma unroll(1)
#endif
for (uint i = 0; (SCAN_BLOCK_SIZE >> i) > 1; i++) {
    t = s_data[idx - (1 << i)]; // (1 << i) = pow(2, i)
    __syncthreads();
    s_data[idx] += t;
    __syncthreads();
}
```

Listing 26: Parameterized loop in Sort.

Block Size

In the SHOC benchmark, variables for scan- and sort block sizes was already present. These can be seen in Listing 27.

```
static const int SORT_BLOCK_SIZE = 128;
static const int SCAN_BLOCK_SIZE = 256;
```

Listing 27: Original Sort block sizes.

This meant that the constants `SCAN_BLOCK_SIZE` and `SORT_BLOCK_SIZE` could be removed from the code and used as parameters instead. The search space for both these parameters was selected as values in {16, 32, 64, 128, 256, 512, 1024}.

Data Size

Related to the block size parameters, the code in SHOC contained code for the size of the input data to the kernels. This was 2 (`uint2`) for the scan data size and 4 (`uint4`) for the sort data size. Relevant code from SHOC can be seen in Listing 28.

```
const size_t radixGlobalWorkSize   = numElements / 4;
const size_t findGlobalWorkSize    = numElements / 2;
const size_t reorderGlobalWorkSize = numElements / 2;

// ...

radixSortBlocks
    <<<radixBlocks, SORT_BLOCK_SIZE,
       4 * sizeof(uint)*SORT_BLOCK_SIZE>>>(...);

findRadixOffsets
    <<<findBlocks, SCAN_BLOCK_SIZE,
       2 * SCAN_BLOCK_SIZE*sizeof(uint)>>>(...);
```

Listing 28: Previous code for data sizes in SHOC.

This could be replaced with parameters and the parameterized code then became:

```
const size_t radixGlobalWorkSize   = numElements / SORT_DATA_SIZE;
const size_t findGlobalWorkSize    = numElements / SCAN_DATA_SIZE;
const size_t reorderGlobalWorkSize = numElements / SCAN_DATA_SIZE;
```

```
// ...

radixSortBlocks
    <<<radixBlocks, SORT_BLOCK_SIZE,
       SORT_DATA_SIZE * sizeof(uint)*SORT_BLOCK_SIZE>>>(...);

findRadixOffsets
    <<<findBlocks, SCAN_BLOCK_SIZE,
       SCAN_DATA_SIZE * SCAN_BLOCK_SIZE*sizeof(uint)>>>(...);
```

Listing 29: Parameterized Sort data sizes.

Relevant example of data size in the SHOC for the kernel `scan4` can be seen here:

```
__device__ uint4 scan4(uint4 idata, uint* ptr)
{
    uint4 val4 = idata;
    uint4 sum;

    // Scan the 4 elements in idata within this thread
    sum.x = val4.x;
    sum.y = val4.y + sum.x;
    sum.z = val4.z + sum.y;
    uint val = val4.w + sum.z;

    // ...
}
```

Listing 30: Previous data sizes in SHOC.

After the parameterization of the data sizes and addition of the data types, this kernel can be seen in Listing 31. The data types that acts as a template for which `uint` type to use.

```
SORT_DATA_TYPE scan4(SORT_DATA_TYPE idata, uint* ptr)
{
    SORT_DATA_TYPE val4 = idata;
    SORT_DATA_TYPE sum;

    // Scan the "SORT_DATA_SIZE" elements in idata within this thread
#if SORT_DATA_SIZE == 2
    sum.x = val4.x;
    uint val = val4.y + sum.x;
#elif SORT_DATA_SIZE == 4
    sum.x = val4.x;
    sum.y = val4.y + sum.x;
    sum.z = val4.z + sum.y;
    uint val = val4.w + sum.z;
#elif SORT_DATA_SIZE == 8
    sum.a.x = val4.a.x;
    sum.a.y = val4.a.y + sum.a.x;
    sum.a.z = val4.a.z + sum.a.y;
    sum.a.w = val4.a.w + sum.a.z;
    sum.b.x = val4.b.x + sum.a.w;
    sum.b.y = val4.b.y + sum.b.x;
    sum.b.z = val4.b.z + sum.b.y;
    uint val = val4.b.w + sum.b.z;
#endif

    // ...
}
```

Listing 31: Parameterized data Sort sizes in SHOC.

These parameters could also be extended to include a `uint8` as well. This was implemented using a custom `uint8` data type, consisting of two `uint4` data types. This is implemented as simple as shown in Listing 32.

```
// Custom struct of two uint4s combined
typedef struct __builtin_align__(16) {
    uint4 a;
    uint4 b;
} uint8;
```

Listing 32: Custom CUDA uint8 struct.

Possible values for these parameters are therefore 2, 4 or 8. The related part for defining the data sizes in the code can be seen in Listing 33.

```
// Select data type based on data type size
// Scan data type size
#if SCAN_DATA_SIZE == 2
    #define SCAN_DATA_TYPE uint2
#elif SCAN_DATA_SIZE == 4
    #define SCAN_DATA_TYPE uint4
#elif SCAN_DATA_SIZE == 8
    #define SCAN_DATA_TYPE uint8
#endif

// Sort data type size
#if SORT_DATA_SIZE == 2
    #define SORT_DATA_TYPE uint2
#elif SORT_DATA_SIZE == 4
    #define SORT_DATA_TYPE uint4
#elif SORT_DATA_SIZE == 8
    #define SORT_DATA_TYPE uint8
#endif
```

Listing 33: Parameterized data sizes in Sort SHOC.

While implementing the data size parameters, a constraint was discovered between scan and sort parameters. Equation 3 shows the discovered constraint.

$$\texttt{SCAN\_DATA\_SIZE} \times \texttt{SCAN\_BLOCK\_SIZE} = \texttt{SORT\_DATA\_SIZE} \times \texttt{SORT\_BLOCK\_SIZE} \quad (3)$$

Another constraint was found between the available shared memory for the selected GPU and for the shared memory in the `reorderData` kernel. The relevant code from the kernel can be shown here:

```
__shared__ SCAN_DATA_TYPE sKeys2[SCAN_BLOCK_SIZE];
__shared__ SCAN_DATA_TYPE sValues2[SCAN_BLOCK_SIZE];
__shared__ uint  sOffsets[16];
__shared__ uint  sBlockOffsets[16];
```

Listing 34: Relevant shared memory objects for constraint.

we can see that the total shared memory usage is:

$$\begin{aligned}\texttt{sizeof(sKeys2)} + \texttt{sizeof(sValues2)} + \texttt{sizeof(sOffsets)} + \texttt{sizeof(sBlockOffsets)}\\ = \texttt{SCAN\_DATA\_SIZE} \times \texttt{SCAN\_BLOCK\_SIZE} \times 2 + \texttt{sizeof(uint)} \times 16 \times 2\end{aligned}$$

which can be rewritten as:

$$(8 \times \texttt{SCAN\_DATA\_SIZE} \times \texttt{SCAN\_BLOCK\_SIZE} + 128) \leq \mathit{available\_shared\_memory} \tag{4}$$

Function Inlining

In the sort benchmark in SHOC there was some functions that was small and looked like they could be inlined. This was the functions `scanLSB`, `scan4` and `scanLocalMem`. The content of `scanLSB` was only a loop, in `scan4` there was only updates to a variable and in `scanLocalMem` was also only a loop. These functions in SHOC can be seen in Listing 35.

```
__device__ uint scanLSB(const uint val, uint* s_data) {...}

// ...

__device__ SORT_DATA_TYPE scan4(SORT_DATA_TYPE idata, uint* ptr) {...}

// ...

__device__  uint scanLocalMem(const uint val, uint* s_data) {...}
```

Listing 35: Previously non-inlined functions in Sort SHOC.

To use a parameters for these function inlines, I used the CUDA compiler directives `__forceinline__` and `__noinline__` to force inline or prevent inline respectively for each function. Possible values for each of these parameters are `True` (1) and `False` (0). This implementation can be seen in Listing 36.

```
__device__
#if INLINE_LSB
__forceinline__
#else
__noinline__
#endif
uint scanLSB(const uint val, uint* s_data) {...}

// ...

__device__
#if INLINE_SCAN
__forceinline__
#else
__noinline__
#endif
SORT_DATA_TYPE scan4(SORT_DATA_TYPE idata, uint* ptr) {...}
```

```
// ...

__device__
#if INLINE_LOCAL_MEMORY
__forceinline__
#else
__noinline__
#endif
uint scanLocalMem(const uint val, uint* s_data) {...}
```

Listing 36: Parameterized inlined functions in Sort.

6.1.6 Stencil 2D

Number of GPUs

As the reason for including the Stencil 2D benchmark from SHOC was that it had a true parallel (TP) version of the benchmark and in the specialization project it was discovered that it could have different performance depending on the number of GPUs used. The only wanted parameter for us to include for this benchmark was therefore for checking performance for using different GPUs. This implementation was also done in collaboration with Ingunn Sund. The parameter for number for GPUs, `GPUS` was implemented by using the input device number to run the benchmark on. The implemented `GPUS` parameter can be seen in Listing 37.

```
devices = ','.join([str(i) for i in range(0, GPUS)])
mpirun -np {GPUS} --allow-run-as-root ./Stencil2D -d {devices}
```

Listing 37: Parameterized number of GPUs for Stencil 2D.

6.1.7 Parameter Search Space

After implementing the different parameters for each algorithm, we can calculate the total parameter search space for each benchmark. This is presented in Table 6.

Table 6: Search space for each parameterized benchmark.

Benchmark	**Search Space**
Triad	$1024 \times 10 \times 2 \times 2 = 40960$
MD	$1024 \times 2 \times 2 \times 5 = 20480$
Reduction	$10 \times 11 \times 2 \times 4 \times 4 \times 2 \times 7 \times GPUS \times 2 \times 2 \times 2 = 394240 \times GPUS$
Sort	$2 \times 2 \times 3 \times 3 \times 7 \times 7 \times 2 \times 2 \times 2 = 14112$
Stencil 2D	$GPUS$

Note, for all GPUs used in this thesis, 1024 was the maximum threads per GPU block and 1024 used in the table above represents that. This might be different for different GPUs. $GPUS$ is the number of connected GPUs.

6.1.8 Original Parameter Values

In this section, tables of what the values were for the different benchmarks before the parameterization of them are shown. These values are useful to different auto-tuners that can specify reference values for correctness verification of the kernels to ensure correct computation.

Triad

Table 7: Original parameter values in the Triad benchmark.

Parameter	**Original Value**
`BLOCK_SIZE`	128
`WORK_PER_THREAD`	1
`LOOP_UNROLL_TRIAD`	`False` (0)
`PRECISION`	`float` (32)

MD

Table 8: Original parameter values for the MD benchmark. Both precisions `float` (32) and `double` (64) was used in SHOC's benchmark, but not as a parameter.

Parameter	**Original Value**
`BLOCK_SIZE`	256
`PRECISION`	Both `float` (32) and `double` (64)
`TEXTURE_MEMORY`	`True` (1)
`WORK_PER_THREAD`	1

Reduction

Table 9: Original parameter values in the Reduction benchmark. Both precisions `float` (32) and `double` (64) was used and all combinations of `GPUS` connected was available in SHOC's implementation. The compiler optimizations for device, fast-math and max registers was not used in SHOC's benchmark.

Parameter	**Original Value**
`BLOCK_SIZE`	256
`GRID_SIZE`	64
`PRECISION`	Both `float` (32) and `double` (64)
`COMPILER_OPTIMIZATION_HOST`	2
`COMPILER_OPTIMIZATION_DEVICE`	Not available
`USE_FAST_MATH`	Not available
`MAX_REGISTERS`	Not available
`GPUS`	All combinations of connected GPUs
`LOOP_UNROLL_REDUCE_1`	`False` (0)
`LOOP_UNROLL_REDUCE_2`	`False` (0)
`TEXTURE_MEMORY`	`False` (0)

Sort

Table 10: Original parameter values in the Sort benchmark.

Parameter	**Original Value**
`LOOP_UNROLL_LSB`	`True` (1)
`LOOP_UNROLL_LOCAL_MEMORY`	`True` (1)
`SCAN_DATA_SIZE`	2
`SORT_DATA_SIZE`	4
`SCAN_BLOCK_SIZE`	256
`SORT_BLOCK_SIZE`	128
`INLINE_LSB`	`False` (0)
`INLINE_SCAN`	`False` (0)
`INLINE_LOCAL_MEMORY`	`False` (0)

Stencil 2D

Table 11: Original parameter values in the Stencil 2D benchmark. All combinations of `GPUS` connected was available in SHOC's implementation, but not as a parameter.

Parameter	**Original Value**
`GPUS`	All combinations of connected GPUs

6.1.9 Summary of Implemented Parameters

In this section a summary of the implemented parameters for each of the benchmark are presented. The parameters are provided a description as well as the search space values. For the sort benchmark, the constraints between the parameters are also shown.

Triad

Table 12: Parameters, descriptions and values for the Triad benchmark.

Parameter	**Description**	**Values** v where $v \in$
`BLOCK_SIZE`	Block size to launch the `triad` kernel with.	{1, ..., *Max GPU threads per block*}
`WORK_PER_THREAD`	The amount of work performed by each GPU thread.	{1, ..., 10}
`LOOP_UNROLL_TRIAD`	Whether to unroll the loop in the `triad` function or not.	{False (0), True (1)}
`PRECISION`	Whether to use single-precision floating-point or double-precision floating-point for the computations.	{`float` (32), `double` (64)}

MD

Table 13: Parameters, descriptions and values for the MD benchmark.

Parameter	**Description**	**Values** v where $v \in$
`BLOCK_SIZE`	Block size to launch the `compute_lj_force` kernel with.	{1, ..., *Max GPU threads per block*}
`PRECISION`	Whether to use single-precision floating-point or double-precision floating-point for the computations.	{`float` (32), `double` (64)}
`TEXTURE_MEMORY`	Whether to use texture memory or not for the input data to the `compute_lj_force` kernel.	{False (0), True (1)}
`WORK_PER_THREAD`	The amount of work performed by each GPU thread.	{1, ..., 5}

Reduction

Table 14: Parameters, descriptions and values for the Reduction benchmark.

Parameter	**Description**	**Values** v where $v \in$
`BLOCK_SIZE`	Block size to launch the `reduce` kernel with.	2^i where i is in the range [0, 11] with a max of threads per block and excluding 32.
`GRID_SIZE`	Grid size to launch the `reduce` kernel with.	{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}
`PRECISION`	Whether to use single-precision floating-point or double-precision floating-point for the computations.	{`float` (32), `double` (64)}
`COMPILER_OPTIMIZATION_HOST`	The level of optimization that are applied by the compiler for the host code.	{1, ..., 4}
`COMPILER_OPTIMIZATION_DEVICE`	The level of optimization that are applied by the compiler for the device code.	{1, ..., 4}
`USE_FAST_MATH`	Whether or not to use the compiler option to use the fast math library.	{False (0), True (1)}
`MAX_REGISTERS`	Maximum number of registers that GPU functions can use.	{-1, 20, 40, 60, 80, 100, 120}
`GPUS`	Number of GPUs to perform the computation of `redcution` on.	{1, ..., *Connected GPUs*}
`LOOP_UNROLL_REDUCE_1`	Whether to unroll the first loop in the `reduce` function or not.	{False (0), True (1)}
`LOOP_UNROLL_REDUCE_2`	Whether to unroll the second loop in the `reduce` function or not.	{False (0), True (1)}
`TEXTURE_MEMORY`	Whether to use texture memory or not for the input data to the `reduce` kernel.	{False (0), True (1)}

Sort

Table 15: Parameters, descriptions and values for the Sort benchmark.

Parameter	**Description**	**Values** v where $v \in$
`LOOP_UNROLL_LSB`	Whether to unroll the loop in the `scanLSB` function or not.	{False (0), True (1)}
`LOOP_UNROLL_LOCAL_MEMORY`	Whether to unroll the loop in the `scanLocalMem` function or not.	{False (0), True (1)}
`SCAN_DATA_SIZE`	Size of the data type that is used for the `scan` functions. It chooses either `uint2`, `uint4` or a custom `uint8` data type.	$\{2, 4, 8\}$
`SORT_DATA_SIZE`	Size of the data type that is used for the `sort` functions. It chooses either `uint2`, `uint4` or a custom `uint8` data type.	$\{2, 4, 8\}$
`SCAN_BLOCK_SIZE`	Block size to launch the `scan` kernels with.	{16, 32, 64, 128, 256, 512, 1024}
`SORT_BLOCK_SIZE`	Block size to launch the `sort` kernels with.	{16, 32, 64, 128, 256, 512, 1024}
`INLINE_LSB`	Whether to inline the `scanLSB` function or not.	{False (0), True (1)}
`INLINE_SCAN`	Whether to inline the `scan4` function or not.	{False (0), True (1)}
`INLINE_LOCAL_MEMORY`	Whether to inline the `scanLocalMem` function or not.	{False (0), True (1)}

Table 16: Constraints with descriptions for the Sort benchmark.

Constraint	**Description**
`SCAN_DATA_SIZE` × `SCAN_BLOCK_SIZE` = `SORT_DATA_SIZE` × `SORT_BLOCK_SIZE`	The ratio between sort and scan data- and block sizes needs to be equal.
(8 × `SCAN_DATA_SIZE` × `SCAN_BLOCK_SIZE` + 128) $\leq$ *available_shared_memory*	Available shared memory for the selected GPU can not be less than needed shared memory in the `reorderData` kernel.

Stencil 2D

Table 17: Parameters, descriptions and values for the Stencil 2D benchmark.

Parameter	**Description**	**Values** v where $v \in$
`GPUS`	Number of GPUs to perform the computation of `StencilKernel` on.	$\{1, ..., \textit{Connected GPUs}\}$

6.2 Making the Benchmark Suite User Friendly

In the process of creating the benchmark suite, we wanted to make it user friendly. This was because of our own experience with other benchmark suites that did not prioritize this part, making it much harder for user to use. We also had some positive experiences with some benchmarks, that we wanted to include in this benchmark suite. The parts that we wanted to include for this benchmark suite was a good documentation, an easy to use command-line interface, an organized project structure and covering different users for the benchmark suite.

To start off with making it user friendly and for it to potentially become a standard, we named the benchmark suite *BAT*: a **B**enchmark suite for **A**uto-**T**uners. To make it even more friendly we created a logo for it, that can be seen in Appendix B.

6.2.1 Command-Line Interface

The benchmark suite was created to contain a command-line interface (CLI), which is located in the `main.py` attached to this thesis. We believe that this should be an easy-to-use CLI, and for a more advanced user, more functionality should be possible. The CLI has the purpose of compiling (if specified), running the auto-tuner implementations and parsing the results after tuning. The CLI takes different input arguments. These are:

- `--benchmark`: The benchmark to run. If no benchmark is selected, all benchmarks will be ran for given auto-tuner(s). Example: `sort`.
- `--auto-tuner`: The auto-tuner to run benchmarks for. If no auto-tuner is selected, all auto-tuners will be benchmarked. Example: `ktt`
- `--verbose`: If all `stdout` and `stderr` should be printed out during the building of the benchmark(s).
- `--size`: The problem size to provide for input data for the benchmark(s). This is up to the specific auto-tuner to handle.
- `--technique`: The tuning technique that the auto-tuners use for benchmarking. This is up to the specific auto-tuner to handle.

BAT will first start by looking for auto-tuner directories below the `tuning_examples` directory and then look for benchmarks inside that directory. For each benchmark, it is required to have a `config.json` file in the benchmark directory. This is for BAT to know which command to run on benchmarking. After the benchmarking is completed, BAT will copy all `JSON` and `CSV` files in the benchmark directory to an own results directory. For each benchmark run the results directory is named after the UNIX timestamp, to prevent clashes of results. BAT will after benchmarking also read the results files specified in the configuration file and print out the best parameters found.

6.2.2 Documentation

The documentation of the benchmark suite was made in a readme markdown format in the Git repository along with the source code. This was to display an easy to understand guide of how to use the benchmarks and the benchmark suite, how to use the different command-line arguments, how to add an own auto-tuner to the benchmarks and how to use Docker for running BAT. The documentation also provided a good explanation of what the repository is and who it applies to. There is also provided a list of all parameter values for each parameter in the documentation. The documentation can be seen in Appendix B.

6.2.3 Project Structure

The project structure was created with a mindset of not making it too complicated, but also make it easy if someone were to extract only a single benchmark from the source code. The structure had to be made in such a way that it would be easy to add new auto-tuner benchmarks as well.

We chose to create a source (`src`) directory for storing the benchmark algorithms. The algorithms were split into a programs directory for host code and GPU kernels, and kernels directory for GPU kernels. Inside the programs directories, we stored both host code and GPU kernels to make it easier for the user to extract the benchmarks if copied outside of the BAT directory.

Some parts of the project structure can be seen in:

```
BAT
├── docker
│   ├── cltune.Dockerfile
│   └── ...
├── src
│   ├── kernels
│   │   ├── triad
│   │   │   ├── triad_kernel_helper.cu
│   │   │   └── triad_kernel.cu
│   │   └── ...
│   └── programs
│       ├── triad
│       │   ├── triad_kernel.cu
│       │   ├── triad_kernel.h
│       │   └── triad.cu
│       └── ...
└── tuning_examples
    └── cltune
        └── triad
            ├── config.json
            ├── Makefile
            ├── reference_kernel.cu
            └── triadtuner.cpp
```

```
│    │    │
│    │    │
│    │    │
│    │    └── ...
│    └── ...
├── main.py
└── README.md
```

6.2.4 Userbase

This benchmark suite applies to different users and the defined users were defined with Ingunn Sund into three different groups. This group represents the users that BAT should be ensured to work for. These are users that:

- are making their own auto-tuner and want to use the benchmarks for testing it.
- would like to compare an auto-tuner to other known auto-tuners.
- wants to check how a parameter's value changes for different architectures.

7 Testing the Benchmark Suite

In this section the testing of the benchmark suite and it's benchmarks was performed. First, the implementation of each benchmark in each auto-tuner is shown, then a summary of the parameters implemented are presented. After that, all the systems that BAT are being tested on are presented and the testing process is described.

The responsibility of implementing benchmarks for the auto-tuners are divided between me and Ingunn Sund. In this thesis the focus is implementing and testing benchmarks for CLTune and KTT, and in Ingunn Sund's thesis the focus is on OpenTuner and Kernel Tuner.

7.1 Implementing Benchmarks with Auto-Tuners

For all implementations using CLTune and KTT, the GPU kernels was changed to include the `extern "C"`, because of the kernels needing it by the auto-tuners. Kernel Tuner also needed this due to the compiler that it used.

7.1.1 CLTune

Before the start of implementing algorithms in CLTune, it was discovered that the official repository of CLTune contained some issues for tuning CUDA code. This was solved in a Pull Request to it's open-source GitHub repository [1].

For all the implemented algorithms in CLTune some common code was created for selecting problem size and tuning technique form the input arguments to the program code, and then later select it before the tuning and also saving the results using a common CLTune JSON results saver. The relevant code for these parts is shown in Listing 38 below.

```
// If only one extra argument and the flag is set for size
if (argc == 2 && (string(argv[1]) == "--size" || string(argv[1]) == "-s")) {
    cerr << "Error: You need to specify an integer for the problem size." << endl;
    exit(1);
}

// If only one extra argument and the flag is set for tuning technique
if (argc == 2 && (string(argv[1]) == "--technique" || string(argv[1]) == "-t")) {
    cerr << "Error: You need to specify a tuning technique." << endl;
    exit(1);
}

// Check if the provided arguments does not match in size
if ((argc - 1) % 2 != 0) {
    cerr << "Error: You need to specify correct number of input arguments."
          << endl;
    exit(1);
}

// Loop arguments and add if found
for (int i = 1; i < argc; i++) {
    // Skip the argument value iterations
    if (i % 2 == 0) {
        continue;
```

[1]https://github.com/CNugteren/CLTune/pull/56

```
    }

    // Check for problem size
    if (string(argv[i]) == "--size" || string(argv[i]) == "-s") {
        try {
            inputProblemSize = stoi(argv[i + 1]);

            // Ensure the input problem size is between 1 and 9
            if (inputProblemSize < 1 || inputProblemSize > 9) {
                cerr << "Error: The problem size needs to be an integer in the
    range 1 to 9." << endl;
                exit(1);
            }
        } catch (const invalid_argument &error) {
            cerr << "Error: You need to specify an integer for the problem size."
                 << endl;
            exit(1);
        }
    // Check for tuning technique
    } else if (string(argv[i]) == "--technique" || string(argv[i]) == "-t") {
        tuningTechnique = argv[i + 1];
    } else {
        cerr << "Error: Unsupported argument " << "`" << argv[i] << "`" << endl;
        exit(1);
    }
}

// ...

// Select the tuning technique for this benchmark
if (tuningTechnique == "annealing") {
    double maxTemperature = MAX_TEMPERATURE;
    auto_tuner.UseAnnealing(searchFraction, {maxTemperature});
} else if (tuningTechnique == "pso") {
    double swarmSize = SWARM_SIZE;
    auto_tuner.UsePSO(searchFraction, swarmSize, 0.4, 0.0, 0.4);
} else if (tuningTechnique == "random") {
    auto_tuner.UseRandomSearch(searchFraction);
} else if (tuningTechnique == "brute_force") {
    auto_tuner.UseFullSearch();
} else {
    cerr << "Error: Unsupported tuning technique: `" << tuningTechnique
         << "`." << endl;
    exit(1);
}

auto_tuner.Tune();

// Get the best computed result and save it as a JSON to file
saveJSONFileFromCLTuneResults(auto_tuner.GetBestResult(), "best-" + kernelName +
    "-results.json", inputProblemSize, tuningTechnique);
```

Listing 38: Common code for CLTune algorithms

The common JSON results saver (`saveJSONFileFromCLTuneResults`) for CLTune was implemented to take the map returned by CLTune as input, the file name where to create the JSON file, the problem size and the tuning technique used in the implementation and creating a JSON file based on these inputs. This function can be seen in Listing 39.

```cpp
void saveJSONFileFromCLTuneResults(
    const unordered_map<string, size_t> &computationResult,
    const string &fileName, const int &problemSize,
    const string &tuningTechnique) {
    string jsonOutput = "{\n\t\"PROBLEM_SIZE\": " + to_string(problemSize) +
        ",\n\t\"TUNING_TECHNIQUE\": \"" + tuningTechnique + "\"";

    // Counters to check if the last item in the map
    int counter = 0;
    int maxCount = computationResult.size();

    // Add comma if parameters
    if (maxCount > 0) {
        jsonOutput += ",";
    }

    // Loop all parameters and add one by one
    for (auto const& parameter: computationResult) {
        jsonOutput += "\n\t\"" + parameter.first + "\": " +
            to_string(parameter.second);

        counter++;

        // If not the last item in the vector add a comma (",")
        if (counter != maxCount) {
            jsonOutput += ",";
        }
    }

    jsonOutput += "\n}";

    // Save the JSON to file
    fstream fs;
    fs.open(fileName, fstream::out);
    fs << jsonOutput;
    fs.close();
}
```

Listing 39: Common JSON saver for CLTune results.

Triad

The relevant parts in the Triad implementation for CLTune, can be seen in Listing 7.1.1. Here we can see the different parameters and thread modifications for local and global sizes.

```cpp
// Add parameters to tune
auto_tuner.AddParameter(kernel_id, "BLOCK_SIZE", block_sizes);
// To set the different block sizes (local size) multiplied by the base (1)
auto_tuner.MulLocalSize(kernel_id, {"BLOCK_SIZE"});

auto_tuner.AddParameter(kernel_id, "WORK_PER_THREAD",
    {1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
// To set the different grid sizes (global size) divided by the amount of work per
     thread
auto_tuner.DivGlobalSize(kernel_id, {"WORK_PER_THREAD"});
auto_tuner.AddParameter(kernel_id, "LOOP_UNROLL_TRIAD", {0, 1});
auto_tuner.AddParameter(kernel_id, "PRECISION", {32, 64});
```

Due to CLTune not being able to change the input parameters for each run to the kernel, a helper function was created with both single and double precisions. Relevant parts for this kernel can be seen in Listing 40. This was related to the reference kernel being created only once before the tuning, making it not possible to include reference checking when tuning both precisions, and only on development of them.

```
extern "C" __global__ void triad_helper(float* Af, float* Bf, float* Cf, float sf,
     double* Ad, double* Bd, double* Cd, double sd, int numberOfElements) {
    #if PRECISION == 32
        triad(Af, Bf, Cf, sf, numberOfElements);
    #elif PRECISION == 64
        triad(Ad, Bd, Cd, sd, numberOfElements);
    #endif
}
```

Listing 40: Triad helper function for CLTune.

MD

Parameters was implemented the same way as for the Triad benchmark including a helper function for the precision parameter. CLTune does not support texture memory for kernels, meaning there had to be a change for always disabling texture memory. This was applied using the compiler directive `#define useTexture false`.

Reduction

For Reduction, the implementation was also similar to the previous ones. One minor difference was for kernel and host code due to CLTune not having a possibility to set the shared memory to kernels. The implementation needed was:

```
// It sets shared memory size equal to the block size
auto_tuner.AddParameter(kernel_id, "KERNEL_SHARED_MEMORY_SIZE", {1});
```

Listing 41: Shared memory in Reduction CLTune.

In the kernel code, this was implemented as a compiler directive to set the shared memory size in the kernel rather than in the host code. The relevant part of the kernel code can be seen in Listing 42.

```
#if KERNEL_SHARED_MEMORY_SIZE
    extern volatile __shared__ float sdata[BLOCK_SIZE*T_SIZE];
#else
    // ...
    extern volatile __shared__ float sdata[];
    // ...
#endif
```

Listing 42: Compiler directive for Reduction shared memory.

Sort

CLTune does not support a way of combining multiple kernels and tuning them in a composite way, therefore each of the kernels are tuned individually. This may lead to different results than tuning the full program.

The separate functions tuned was `tuneRadixSortBlocks`, `tuneScan` and `tuneVectorAddUniform4`. They were tuned similarly as to the previous example, except that constraints was needed for some of the parameters. These constraints was for shared memory size and block- and data sizes. The constraints can be seen in Listing 43.

```cpp
// Constraint for block sizes and data sizes
// Expects input in order:
// {"SCAN_DATA_SIZE", "SORT_DATA_SIZE", "SCAN_BLOCK_SIZE", "SORT_BLOCK_SIZE"}
auto dataSizeBlockSizeConstraint = [](const vector<size_t> &parameters) {
    return parameters.at(2) / parameters.at(3) ==
        parameters.at(1) / parameters.at(0);
};

// ...

// Constraint for shared memory used
auto sharedMemoryConstraint = [&](const std::vector<size_t>& vector) {
    return ((vector.at(0) * vector.at(1) * 4 * 2) + (4 * 16 * 2))
        <= available_shared_memory;
};
auto_tuner.AddConstraint(kernel_id, sharedMemoryConstraint,
    {"SCAN_BLOCK_SIZE", "SCAN_DATA_SIZE"});

// Constraint for block sizes and data sizes
auto_tuner.AddConstraint(
    kernel_id,
    dataSizeBlockSizeConstraint,
    {"SCAN_DATA_SIZE", "SORT_DATA_SIZE", "SCAN_BLOCK_SIZE", "SORT_BLOCK_SIZE"}
);
```

Listing 43: Sort data size constraint for CLTune.

Stencil 2D

This benchmark was not able to be implement in CLTune, due to the only parameter implemented for it was the `GPUS`, which is not possible without the possibility to run the program on multiple GPUs.

7.1.2 KTT

Also for KTT at the start of implementing the kernels, it was discovered that it did not work with CUDA code. This was also solved in a a Pull Request to it's open-source GitHub repository [2].

It was also discovered that the grid size input to the kernels was not grid size, as they mentioned in their documentation, but rather the total number of threads. This lead to adjustments in the kernels grid argument. The printed out launching parameters to the kernel during launching was also wrong and it was not using `ceil` in the backend, leading to too few threads launched.

[2]https://github.com/Fillo7/KTT/pull/41

It was used a common JSON saver for the KTT implementations (`saveJSONFileFromKTTResults`) as in CLTune. This was the same code as for the implementations using CLTune, as both these two auto-tuners use C++ code. With a slight modification in that KTT used its own type `ComputationResult`, instead of the results map as used in the CLTune JSON saver.

One inconvenience when implementing KTT, was that the global size type that was set for the auto-tuner, did not work with CUDA, but was required to be OpenCL, as shown in Listing 44.

```
auto_tuner.setGlobalSizeType(ktt::GlobalSizeType::OpenCL);
// Rather than
auto_tuner.setGlobalSizeType(ktt::GlobalSizeType::CUDA);
```

Listing 44: Incorrect global size type for KTT

KTT included different tuning techniques and setup of them than for CLTune, which can be seen here in Listing 45 below.

```
// Select the tuning technique for this benchmark
if (tuningTechnique == "annealing") {
    double maxTemperature = 4.0f;
    auto_tuner.setSearchMethod(ktt::SearchMethod::Annealing, {maxTemperature});
} else if (tuningTechnique == "mcmc") {
    auto_tuner.setSearchMethod(ktt::SearchMethod::MCMC, {});
} else if (tuningTechnique == "random") {
    auto_tuner.setSearchMethod(ktt::SearchMethod::RandomSearch, {});
} else if (tuningTechnique == "brute_force") {
    auto_tuner.setSearchMethod(ktt::SearchMethod::FullSearch, {});
} else {
    cerr << "Error: Unsupported tuning technique: ‘" << tuningTechnique << "‘."
         << endl;
    exit(1);
}
```

Listing 45: Tuning techniques for KTT algorithms.

Triad

Relevant parts of the KTT implementation specifically for Triad is the parameter inserting and the thread modifications to the launching of the kernel. This can be seen in Listing 46 below.

```
auto_tuner.addParameter(kernelId, "BLOCK_SIZE", block_sizes);
auto_tuner.addParameter(kernelId, "WORK_PER_THREAD",
    {1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
auto_tuner.addParameter(kernelId, "LOOP_UNROLL_TRIAD", {0, 1});
auto_tuner.addParameter(kernelId, "PRECISION", {32, 64});

// To set the different block sizes (local size) multiplied by the base (1)
auto_tuner.setThreadModifier(kernelId, ktt::ModifierType::Local,
    ktt::ModifierDimension::X, "BLOCK_SIZE", ktt::ModifierAction::Multiply);
// To set the different grid sizes (global size) divided by the amount of work per
     thread
// Divide on block size and multiply by block size after ceiling to ensure enough
    threads used
// Also use precision to get the grid size
// Using ceil because KTT does not ceil the divided grid size
auto globalModifier = [](const size_t size, const vector<size_t>& vector) {
    return int(ceil(double(size) / double(vector.at(0)) / double(vector.at(1)))) *
     vector.at(0) * (vector.at(2)/32*4);
};
auto_tuner.setThreadModifier(kernelId, ktt::ModifierType::Global, ktt::
    ModifierDimension::X, {"BLOCK_SIZE", "WORK_PER_THREAD", "PRECISION"},
    globalModifier);
```

Listing 46: Triad parameters for KTT.

MD

Parameters was implemented the same way as for the Triad benchmark including a helper function for the precision parameter. As for CLTune, KTT did not support texture memory either, leading to the same compiler directive `#define useTexture false` being applied here as well.

Reduction

For Reduction, the implementation was also similar to the previous ones. One minor difference was the compiler directive `KERNEL_SHARED_MEMORY_SIZE`, as previously described for the Reduction implementation in CLTune.

Sort

The Sort implementation was different from the other KTT implementations. This was because KTT has the support for using a composite of multiple kernels and tune them together. This was using KTT's `TuningManipulator` class. For reference checking the class `ReferenceClass` was used. This implementation is also inspired by one of KTT's own examples of sort, named `sort-new`. The implementation also included the constraints that were described in the Sort implementation for CLTune.

Stencil 2D

As for CLTune, this benchmark was not able to be implement in KTT either. This was for the same reason, where `GPUS` depends on running the program on multiple GPUs.

7.1.3 Kernel Tuner

Kernel Tuner was different from CLTune and KTT in that it could tune both full programs and kernel code. For the implemented algorithms described in this thesis, all are implemented using full program.

Triad

The relevant parts in the Triad implementation for Kernel Tuner, can be seen in Listing 47. Here we can see the different parameters used.

```
tune_params = dict()
tune_params["BLOCK_SIZE"] = [i for i in range(1, max_block_size + 1)]
tune_params["WORK_PER_THREAD"] = [i for i in range(1, 11)] # Range: [1, ..., 10]
tune_params["LOOP_UNROLL_TRIAD"] = [0, 1]
tune_params["PRECISION"] = [32, 64]
```

Listing 47: Triad parameters for Kernel Tuner.

MD

Parameters was implemented the same way as for the Triad benchmark using host code for the tuning. There was one difference for this implementation in contrast to the CLTune and KTT implementations, texture memory was possible to use since Kernel Tuner can run host code.

Reduction

For Reduction, the implementation was also similar to the previous ones. One minor difference was like for MD, it could use the parameter for texture memory.

Sort

The Sort implementation for Kernel Tuner was implemented using the full program to tune and the rest of the implementation is similar to the previously described for Kernel Tuner. The constraint for block sizes and data sizes implemented in Kernel Tuner are shown in the listing below.

```
constraint = ["(SCAN_BLOCK_SIZE / SORT_BLOCK_SIZE) == (SORT_DATA_SIZE / 
    SCAN_DATA_SIZE)",
          f"((SCAN_BLOCK_SIZE * SCAN_DATA_SIZE * 4 * 2) + (4 * 16 * 2)) <= {
    available_shared_memory}"]
```

Listing 48: Sort constraint for Kernel Tuner.

Stencil 2D

As for CLTune and KTT, this benchmark was not able to be implement in Kernel Tuner either. This was for the same reason, where `GPUS` depends on running the program on multiple GPUs.

7.1.4 OpenTuner

OpenTuner was similar to Kernel Tuner in a way that they both could tune the full programs, but OpenTuner could not tune just the kernel as Kernel Tuner could. Therefore all implemented algorithms described in this thesis are implemented using the full programs. In OpenTuner, the compile commands can be created to include the parameters.

Triad

The relevant parts in the Triad implementation for OpenTuner, can be seen in Listing 49. Here we can see the different parameters used and a compile command at the end.

```
manipulator.add_parameter(IntegerParameter('BLOCK_SIZE', 1, max_block_size))
manipulator.add_parameter(IntegerParameter('WORK_PER_THREAD', 1, 10))
manipulator.add_parameter(IntegerParameter('LOOP_UNROLL_TRIAD', 0, 1))
manipulator.add_parameter(EnumParameter('PRECISION', [32, 64]))
```

```
make_program = f'nvcc -gencode=arch=compute_{cc},code=sm_{cc} -I {start_path}/cuda
    -common -I {start_path}/common -g -O2 -c {start_path}/triad/triad.cu'
make_program += ' -D{0}={1}'.format('BLOCK_SIZE', cfg['BLOCK_SIZE'])
```

Listing 49: Triad parameters for OpenTuner.

MD

Parameters was implemented the same way as for the Triad benchmark using host code for the tuning. Texture memory was possible in this implementation as well as in Kernel Tuner, which run host code.

Reduction

For this implementation of Reduction it was possible to use the compiler optimization parameters. See Listing 50 below for implemented parameters and the make command to include the parameters to the program.

```
manipulator.add_parameter(IntegerParameter('COMPILER_OPTIMIZATION_HOST', 0, 3))
manipulator.add_parameter(IntegerParameter('COMPILER_OPTIMIZATION_DEVICE', 0, 3))
manipulator.add_parameter(IntegerParameter('USE_FAST_MATH', 0, 1))
manipulator.add_parameter(EnumParameter('MAX_REGISTERS', [-1, 20, 40, 60, 80, 100,
     120]))

# ...

make_program = f'nvcc -gencode=arch=compute_{cc},code=sm_{cc} -I {start_path}/cuda
    -common -I {start_path}/common {use_fast_math}{max_registers} -O{cfg["
    COMPILER_OPTIMIZATION_HOST"]} -Xptxas -O{cfg["COMPILER_OPTIMIZATION_DEVICE"]} -
    c {start_path}/reduction/reduction.cu'
```

Listing 50: Reduction parameters for OpenTuner.

Sort

The Sort implementation for OpenTuner was implemented similarly to Kernel Tuner, with using the full program to tune. The rest of the implementation is similar to the previously described for OpenTuner. The constraint for block sizes and data sizes implemented in Kernel Tuner are shown in the listing below.

```
# Check constraint for block sizes and data sizes
if cfg['SCAN_BLOCK_SIZE'] / cfg['SORT_BLOCK_SIZE'] != cfg['SORT_DATA_SIZE'] / cfg[
    'SCAN_DATA_SIZE']:
    return Result(time=float("inf"), state="ERROR", accuracy=float("-inf"))

# Constraint to ensure not attempting to use too much shared memory
# 4 is the size of uints and 2 is because shared memory is used for both keys and
    values in the "reorderData" function
# 16 * 2 is also added due to two other shared memory uint arrays used for offsets
shared_memory_needed = (cfg['SCAN_BLOCK_SIZE'] * cfg['SCAN_DATA_SIZE'] * 4 * 2) +
    (4 * 16 * 2)
gpu = cuda.get_current_device()
available_shared_memory = gpu.MAX_SHARED_MEMORY_PER_BLOCK
```

```
if shared_memory_needed > available_shared_memory:
    return Result(time=float("inf"), state="ERROR", accuracy=float("-inf"))
```

Listing 51: Sort constraint for OpenTuner.

Stencil 2D

The Stencil 2D implementation for OpenTuner is the only auto-tuner the program is implemented in due to the restriction of using multiple GPUs for the parameter. In this implementation it uses MPI to run the program on multiple devices with the parameter `GPUS`:

```
manipulator.add_parameter(IntegerParameter('GPUS', 1, len(cuda.gpus)))

# ...

devices = ','.join([str(i) for i in range(0, chosen_gpu_number)])
run_cmd = f'mpirun -np {chosen_gpu_number} --allow-run-as-root {program_command} -
    d {devices}'
```

Listing 52: Stencil 2D for OpenTuner.

7.2 Benchmark Parameter Availability

Not all parameters was possible to implement for all auto-tuners. This was due to different reasons, such as that we could not specify a parameter for compiler optimization using CLTune, KTT and Kernel Tuner. Therefore it is presented tables of different parameters available for benchmarking for the different implementations.

7.2.1 Sort

Table 18: Implemented parameters in each auto-tuner for the Sort benchmark.

Auto-Tuner / **Parameter**	OpenTuner	Kernel Tuner	CLTune	KTT
LOOP_UNROLL_LSB	Yes	Yes	Yes	Yes
LOOP_UNROLL_LOCAL_MEMORY	Yes	Yes	Yes	Yes
SCAN_DATA_SIZE	Yes	Yes	Yes	Yes
SORT_DATA_SIZE	Yes	Yes	Yes	Yes
SCAN_BLOCK_SIZE	Yes	Yes	Yes	Yes
SORT_BLOCK_SIZE	Yes	Yes	Yes	Yes
INLINE_LSB	Yes	Yes	Yes	Yes
INLINE_SCAN	Yes	Yes	Yes	Yes
INLINE_LOCAL_MEMORY	Yes	Yes	Yes	Yes

7.2.2 Triad

Table 19: Implemented parameters in each auto-tuner for the Triad benchmark.

Auto-Tuner / **Parameter**	OpenTuner	Kernel Tuner	CLTune	KTT
`BLOCK_SIZE`	Yes	Yes	Yes	Yes
`WORK_PER_THREAD`	Yes	Yes	Yes	Yes
`LOOP_UNROLL_TRIAD`	Yes	Yes	Yes	Yes
`PRECISION`	Yes	Yes	Yes	Yes

7.2.3 Reduction

Table 20: Implemented parameters in each auto-tuner for the Reduction benchmark.

Auto-Tuner / **Parameter**	OpenTuner	Kernel Tuner	CLTune	KTT
`BLOCK_SIZE`	Yes	Yes	Yes	Yes
`GRID_SIZE`	Yes	Yes	Yes	Yes
`PRECISION`	Yes	Yes	Yes	Yes
`COMPILER_OPTIMIZATION_HOST`	Yes	No	No	No
`COMPILER_OPTIMIZATION_DEVICE`	Yes	No	No	No
`USE_FAST_MATH`	Yes	No	No	No
`MAX_REGISTERS`	Yes	No	No	No
`GPUS`	Yes	No	No	No
`LOOP_UNROLL_REDUCE_1`	Yes	Yes	Yes	Yes
`LOOP_UNROLL_REDUCE_1`	Yes	Yes	Yes	Yes
`TEXTURE_MEMORY`	Yes	Yes	No	No

7.2.4 MD

Table 21: Implemented parameters in each auto-tuner for the MD benchmark.

Auto-Tuner / **Parameter**	OpenTuner	Kernel Tuner	CLTune	KTT
`BLOCK_SIZE`	Yes	Yes	Yes	Yes
`PRECISION`	Yes	Yes	Yes	Yes
`TEXTURE_MEMORY`	Yes	Yes	No	No
`WORK_PER_THREAD`	Yes	Yes	Yes	Yes

7.2.5 Stencil 2D

Table 22: Implemented parameters in each auto-tuner for the Stencil 2D benchmark.

Auto-Tuner / **Parameter**	OpenTuner	Kernel Tuner	CLTune	KTT
`GPUS`	Yes	No	No	No

7.3 System Setup

The systems used for running the auto-tuners were computer with a NVIDIA GeForce GTX 980 GPU, a computer with a NVIDIA TITAN RTX GPU, a system with 20 NVIDIA T4's (only one were used), an IBM Power System AC922 (Mini Summit) and the NVIDIA DGX-2 system (Heid).

NVIDIA GeForce GTX 980 Based System

This section is a modified version from my specialization project, which is attached to this thesis.

This system has a NVIDIA GeForce GTX 980 GPU with 4 GB memory and an Intel Core i7-6700K CPU. There is a PCIe interconnect between the GPU and the CPU. Additional hardware specifications are shown in Table 23. Appendix C has additional information about the GPU in some listings. Listing 53 shows the topology, Listing 54 shows that the system can not use NVLinks, and Listing 55 shows extra GPU information.

Table 23: Hardware specifications for NVIDIA GeForce GTX 980 based computer.

Component	**Information**
CPU	Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz with 4 cores. Maximum clock frequency: 4.2 GHz. 2 threads per core, total 8 threads.
RAM	16 GB main memory 4 GB GPU memory
GPU	NVIDIA GeForce GTX 980, 4 GB.
CPU-GPU Interconnect	PCIe
OS	Ubuntu 18.04.3

NVIDIA TITAN RTX Based System

This section is a modified version from my specialization project, which is attached to this thesis.

This computer has a NVIDIA TITAN RTX graphics card with 24 GB GPU memory. PCIe connects the GPU to the CPU, which is an Intel Core i9-9900K processor. The TITAN RTX GPU has 576 Tensor Cores, and more specifications can be seen in Table 24. More information about the GPU can be found in listings in Appendix C. Listing 57 shows the topology, Listing 58 shows that the graphics card can have two NVLinks and Listing 59 displays extra GPU information.

Table 24: Hardware specifications for NVIDIA TITAN RTX based computer.

Component	**Information**
CPU	Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz with 8 cores. Maximum clock frequency: 5 GHz. 2 threads per core, total 16 threads.
RAM	16 GB main memory 24 GB GPU memory
GPU	NVIDIA TITAN RTX, 24 GB. 576 Tensor Cores.
CPU-GPU Interconnect	PCIe
OS	Ubuntu 18.04.3

NVIDIA Tesla T4 Based System

The system with NVIDIA Tesla T4 GPUs contains two CPUs and 20 T4's, but in this thesis for this system the focus is only on a single GPU. A single of these cards has 16 GB of memory. The CPU for this system is a Intel(R) Xeon(R) Gold 6230. Between the GPUs and GPU there are a PCIe connection. Additional hardware specifications are shown in Table 25. Appendix C has additional information about the GPU in some listings. Listing 61 shows that the system can not use NVLinks, and Listing 62 shows extra GPU information.

Table 25: Hardware specifications for NVIDIA Tesla T4 based computer.

Component	**Information**
CPU	2 × Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz with 20 cores each. Maximum clock frequency: 3.9 GHz. 2 threads per core, total 80 threads in the system.
RAM	378 GB main memory 320 GB GPU memory
GPU	20 × NVIDIA Tesla T4, 16 GB 320 Tensor Cores.
GPU-GPU Interconnect	PCIe
CPU-GPU Interconnect	PCIe
OS	Ubuntu 18.04.5

IBM Power System AC922

This section is a modified version from my specialization project, which is attached to this thesis.

The Power AC922 system are of the model 8335-GTH, which among other things means that it has air cooling instead of water cooling [5]. The system has two POWER9 processors and two NVIDIA

Tesla V100-SXM2 GPUs with 16 GB memory each. The hardware specifications for Mini Summit are shown in Table 26 and see Listing 69 in Appendix C for GPU information about the system. The listings show that the SXM2 model has a power limit of 300W. This can be seen when running the command `nvidia-smi -q -i 0`.

Table 26: IBM Power System AC922 hardware specifications.

Component	**Information**
CPU	2 × POWER9 CPUs with 16 cores each. Maximal clock frequency: 3.8 GHz. 4 SMT threads per core, total 128 threads in the system.
RAM	512 GB main memory 32 GB GPU memory
GPU	2 × NVIDIA Tesla V100-SXM2, 16 GB. 640 Tensor Cores.
GPU-GPU Interconnect	NVLink 2.0
CPU-GPU Interconnect	NVLink 2.0
OS	Red Hat Enterprise Linux 7.6

Mini Summit has one GPU connected to each processor and also three bricks of NVLink 2.0 connecting the GPUs to their CPUs. Mini Summit's architecture can be seen in Figure 9. See Listing 67 in Appendix C for the outprinted topology. Both the POWER9 CPUs and the Tesla V100 GPUs have possibility for six NVLink 2.0 bricks to another device, but this system does not use the other three bricks. See Listing 68 in the same appendix for the NVLink active status for the GPUs.

Mini Summit is divided into two NUMA nodes, where each node has one CPU and one GPU.

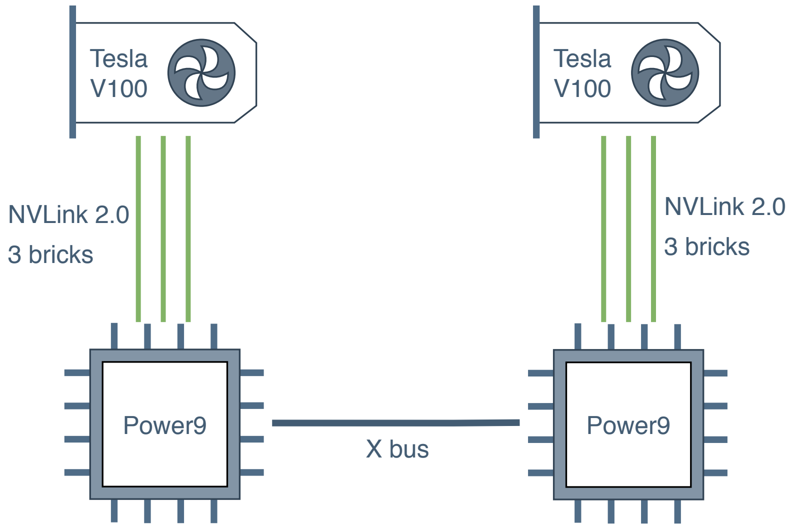

Figure 9: Illustration of how the GPUs are connected to the CPUs on Mini Summit. The illustration is made in collaboration with Ingunn Sund.

NVIDIA DGX-2

This section is a modified version from my specialization project, which is attached to this thesis.

The NVIDIA DGX-2 has two Intel Xeon Platinum 8186 processors and 16 NVIDIA Tesla V100-SXM3 GPUs, each with 32 GB memory. These Tesla V100 graphic cards differs from the SXM2 version in that they have different power consumption limits and there are some architectural differences [45]. In Listing 65 from Appendix C there is shown that Tesla V100-SXM3 has a power limit of 350W, 50W more than the SXM2 models. This extra power is dedicated to increasing the clock rate [46] which is about 60-80 MHz higher than for the SXM2 models, depending on the usage. The clock rates can be seen by running the `nvidia-smi -q -i 0` command. This command also shows that the GPUs in the DGX-2 have a minimum power limit of 100W.

This system has NVSwithes between the GPUs and PCIe connection between CPU and GPU. The connection between the GPUs traverses through a bounded set of six NVLinks, the NVLink status can be seen in 64. The system is divided into two NUMA nodes with 8 GPUs per node. This topology can be seen in Listing 63 in Appendix C. The graphic cards has 640 Tensor Cores and more specifications can be seen in Table 27.

Table 27: NVIDIA DGX-2 hardware specifications.

Component	**Information**
CPU	2 × Intel(R) Xeon(R) Platinum 8168 CPUs @ 2.70GHz with 24 cores each. Maximum clock frequency: 3.7 GHz. 2 threads per core, total 96 threads in the system.
RAM	1510 GB main memory 512 GB GPU memory
GPU	16 × NVIDIA Tesla V100-SXM3, 32GB. 640 Tensor Cores.
GPU-GPU Interconnect	NVSwitch
CPU-GPU Interconnect	PCIe
OS	Ubuntu 18.04.3

7.4 Testing Process

The process for the testing was made by first defining what to test, then defining how to test. After these steps, the actual testing of the implementations were performed on the described systems.

7.4.1 What to Test

In the planning section previously described, it was defined that the systems to test these implementations on was a GTX 980 based system, a RTX TITAN based system, a multi-GPU IBM AC922 system, a multi-GPU NVIDIA DGX-2 system and a single GPU from a multi-GPU system containing NVIDIA T4s.

The different tuning techniques used for testing the auto-tuner was decided in collaboration with Ingunn Sund, leading to similar tests performed for our theses.

We wanted to test that the parameterized benchmarks for the auto-tuners worked correctly for both single- and multi-GPU auto-tuning. This was done by using single-GPU on all machines tested, and I would test multi-GPU benchmarks for DGX-2 and the IBM Power System AC922. The implementations, Reduction and Stencil 2D was possible to use with OpenTuner for this test.

We wanted to test that the algorithms worked for different types of systems and that they worked for different auto-tuners. First we tested that the auto-tuner implementations just worked at all, then that the different tuning techniques worked as expected.

We also wanted to test that the values for the parameters was suitable for the search spaces and that they varied for different tested systems. This can be tested with brute force technique to get all the different parameter combinations.

We wanted to test the brute force tuning technique with OpenTuner, due to it containing all parameters, making it a good baseline for most performant configuration of parameters. This includes the multi-GPU and the compiler optimization parameters. However OpenTuner did not have a brute force tuning technique, and Kernel Tuner was used instead because of it being the one containing the second most implemented parameters. KTT was also used for brute forcing due to Kernel Tuner using too long time, due to the host code version being slower than a version that only tunes kernels. Since we wanted to be sure that the results was retrieved in time, KTT was used for these benchmarks.

To evaluate KTT, we used both `MCMC` and brute force as CLTune used a better tuning of multiple kernels. We wanted to evaluate CLTune by using the particle swarm optimization technique with the swarm sizes 1, 5 and 20. We also wanted to test CLTune with the simulated annealing technique with max temperatures 0.1, 2 and 10. These techniques we wanted to test for the reduction and triad implementations.

We wanted to evaluate Kernel Tuner by using the brute force technique and genetic algorithm with a max iteration of 50 and population size of 10.

We wanted to test the OpenTuner implementations by using the `AUCBanditMetaTechnique` tuning technique.

7.4.2 How to Test

The benchmark implementations was mainly tested using Docker. This was a good way to keep the reproducibility of the benchmarks in the different systems and to have an easy installation of the dependencies. The NVIDIA Docker plugin was used for Docker to enable GPU accelerated applications. A single Dockerfile was created for each of the separate auto-tuners, which all can be seen in Appendix D. The Dockerfiles are also available in the source code in the `docker` directory.

There is also a guide for using the Dockerfiles in the readme for BAT. This can be seen in Appendix B.

For one of the machines, the job scheduler `Slurm` was required to reserve resources for the system, to then run the auto-tuning benchmarks. This was for using a queue system on the machine. This was the NVIDIA Tesla T4 based system. Example command for reserving resources on the machine using Slurm can be seen Listing 75 in Appendix D. After the allocated resources were reserved, it had installed NVIDIA Docker so it was similar to the DGX-2, GTX 980 and RTX TITAN systems for running the implementations inside Docker containers.

Where Docker was not possible to use, the code had to be ran bare metal. The system that did not have Docker was the IBM Power machine. This lead to problems with both setting up the benchmarks and running the code, due to the machine having issues with it's already installed packages and environment. Therefore a lot of the dependencies was needed to be install manually from source. Which again lead to the Kernel Tuner implementations not being ran on this machine.

The commands used for running BAT's `main.py` file, are described in the readme in Appendix B. This script conveniently saved the results to it's own directory after completed each of the auto-tuning benchmarks.

7.4.3 Correctness Verification

To ensure the correctness of the implemented programs, different steps were applied. These included using reference kernels for the auto-tuners that supported this and for full program auto-tuning, it was possible to include already created result checkers. These steps were mainly applied during the development to assure corrected computed kernels.

8 Results and Discussion

All the parameterized benchmarks for the auto-tuners worked correctly when tuning both the single- and multi-GPU versions.

8.1 Parameter Evaluation

8.1.1 OpenTuner

Stencil 2D

OpenTuner results for Stencil 2D on the multi-GPU systems are shown in Figure 10 with problem size 1, and in Figure 11 with problem size 4. Since this benchmark only has GPUs as a parameter, this is the only parameter shown in the graphs. In Figure 11, there are some data points missing for some of the numbers of GPUs, this is because OpenTuner used quite a long time on the tuning with size 4 and not every combination of the search space were run.

By looking at the figure we can see that even if the work is divided between multiple GPUs, there is no speedup. The most efficient number of GPUs used is 1. In my specialization project, we saw that Stencil 2D showed improved performance, in GFLOPS, when being run on multiple GPUs. The tuning results might have been different if we used performance (GFLOPS) as the unit when tuning instead of time.

In the future, I would consider trying to change how the algorithm uses multiple GPUs, to see if it is possible to get a runtime speedup when running multi-GPU benchmarks.

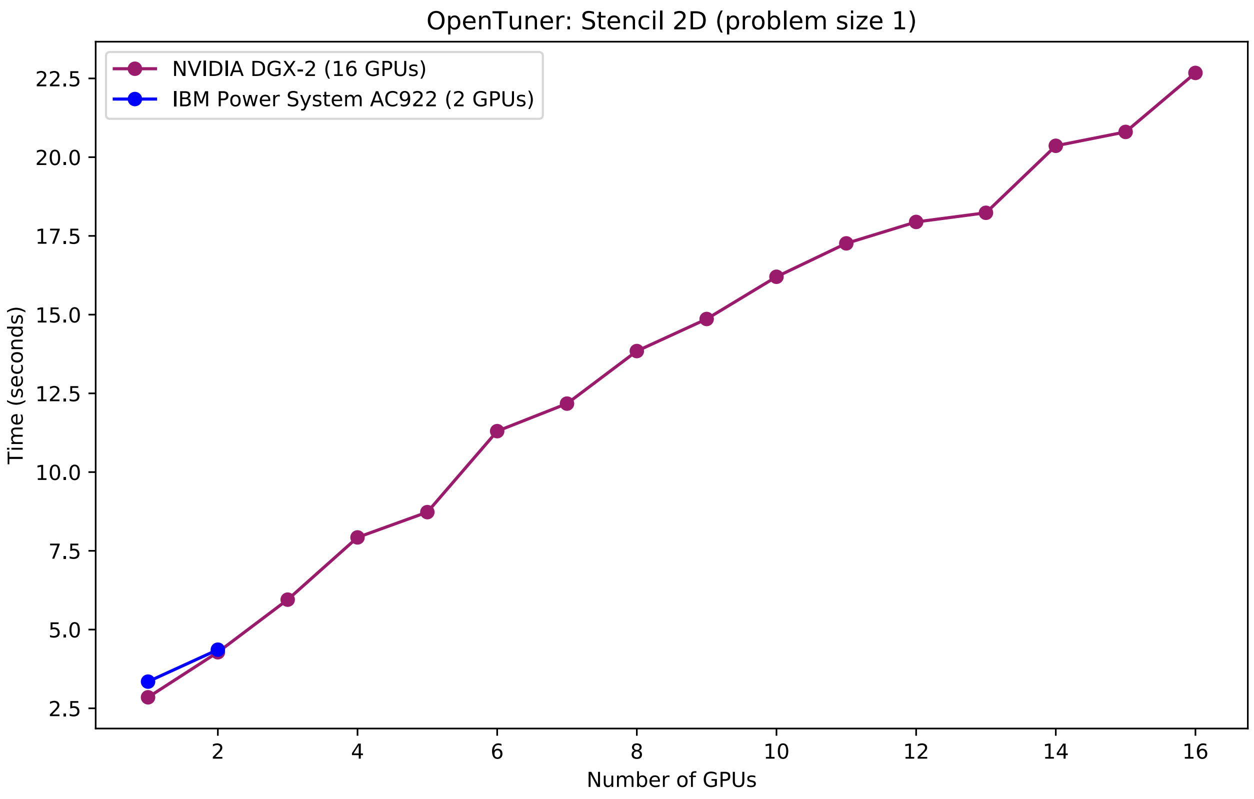

Figure 10: Stencil 2D with problem size 1 for both NVIDIA DGX-2 and IBM Power System AC922.

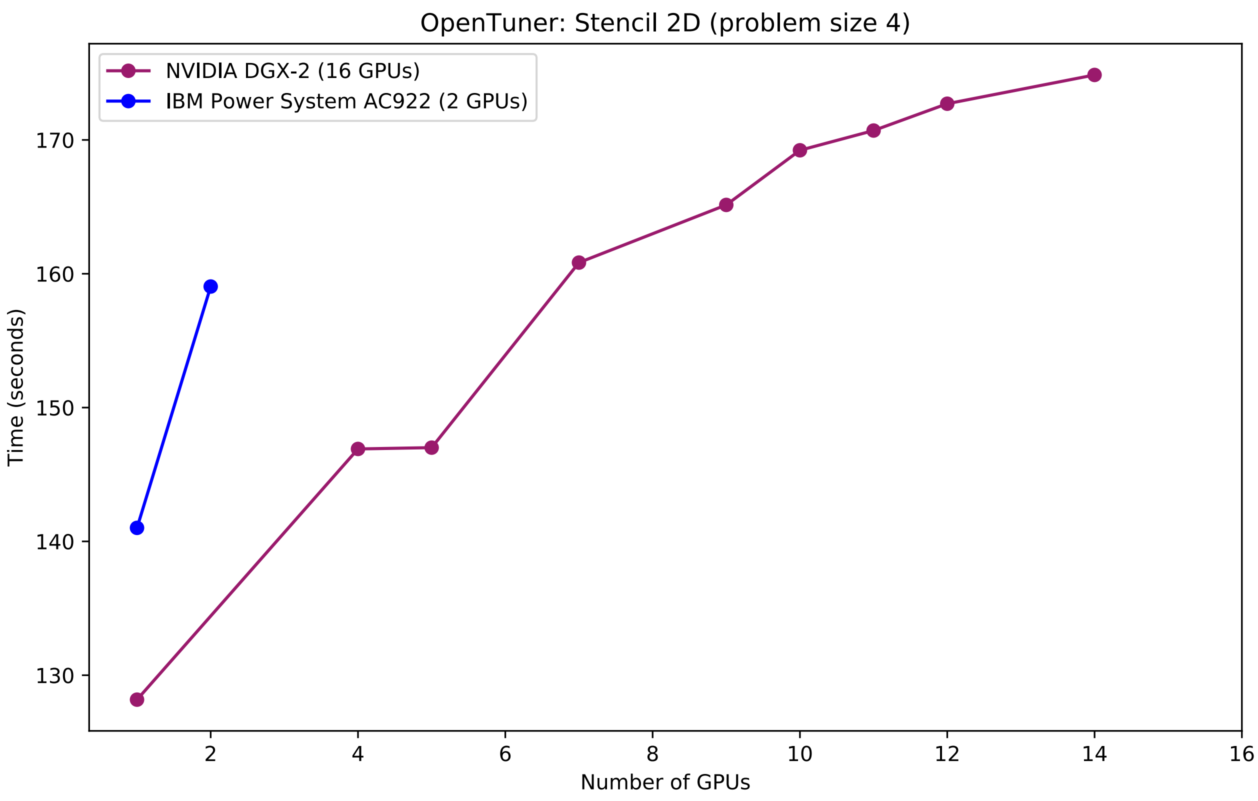

Figure 11: Stencil 2D with problem size 4 for both NVIDIA DGX-2 and IBM Power System AC922.

8.1.2 Kernel Tuner

Sort

Using the sort benchmark implementation in Kernel Tuner, we can see that the parameter for sort data size was 8 in all results and 2 for scan data size, 512 for scan block size and 128 for sort block size for most results. This might indicate that these values are good, but needs more analysis to confirm this. The rest of the parameters, loop unroll and inline functions, varied for all results and more analysis is needed for these parameters.

8.1.3 CLTune

A more extensive analysis is needed for these results to conclude which maximum temperature for simulated annealing and which swarm size for particle swarm optimization is best.

8.1.4 KTT

After tuning with KTT, it was discovered that both the MCMC and brute force techniques iterated over the full search space, without the possibility to limit it. This means that both these techniques are versions of brute force, just that they iterate over the search space in a different order.

By looking at the reduction results for different architectures, one can see that it is not clear if one value of a parameter is best. Therefore more analysis of these results needs to be done.

For the triad results for KTT, one can see that precision is 32 for all results and that it generally has a low value for the work per thread parameter. It is also noticed that 2 might be the best value for this parameter.

Reduction

The systems IBM AC922 and DGX-2 found the best values for the precision to be 64 was best, but on other machines found 32 to be the best. Both the IBM AC922 and DGX-2 systems contains the same GPU, so this could be the reason for that.

For the grid size parameter, one can see that all the best values found, are high numbers. And all the values are 1024 except one, which is 256.

MD

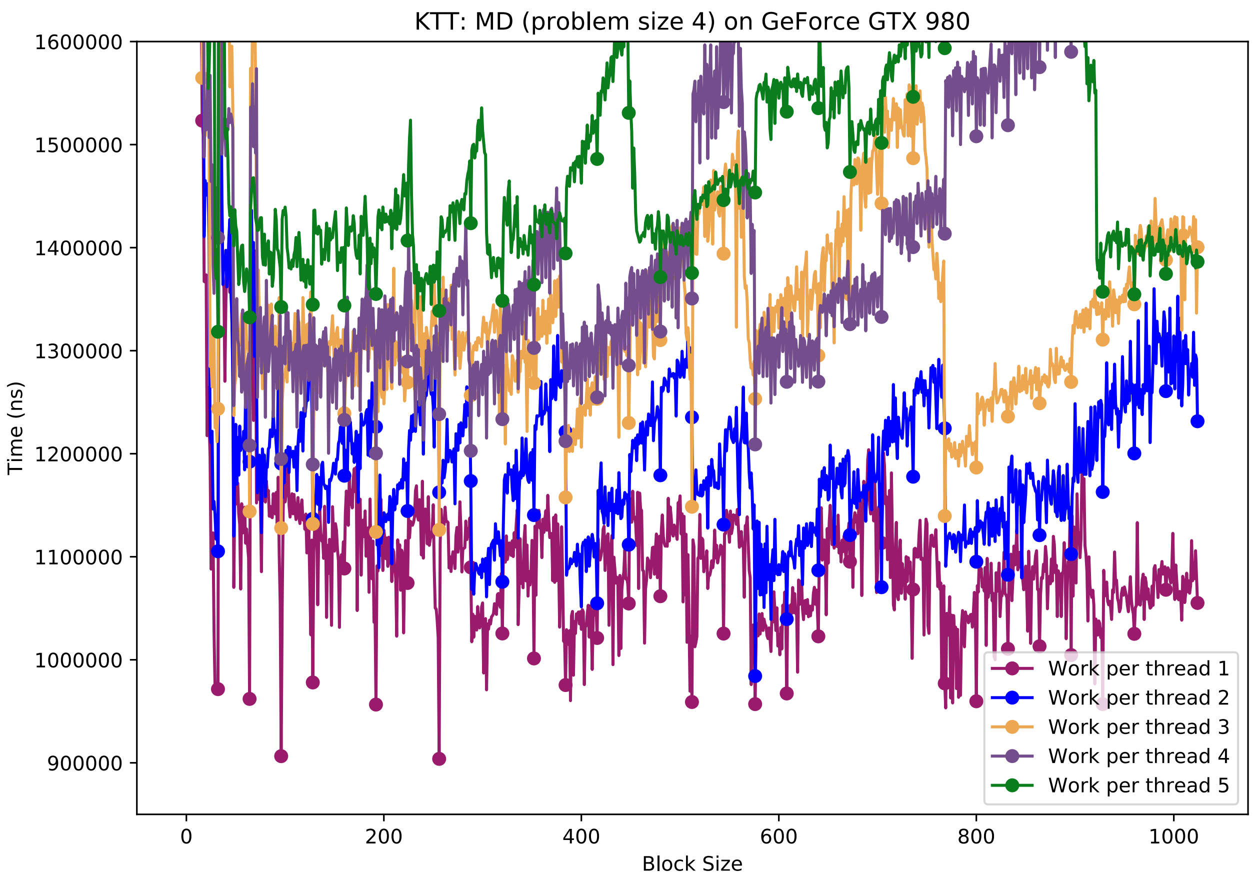

Figure 12: Work per thread parameter and block size parameter for GTX 980 system for the MD benchmark in KTT.

We can see in this figure that the lower the work per thread is, the faster this benchmark is. This is valid for all block sizes. It is also noticed that the time used for the different configurations varies for the different block sizes, but also varies differently depending on the work per threads used.

In the figure, block sizes dividable by 32 and the block size equal to 16 is marked with a circle. We can easily notice that the configurations where block size has these values, is faster. This is perhaps due to the warp size being 32, which is often said to be more optimal when launching kernels.

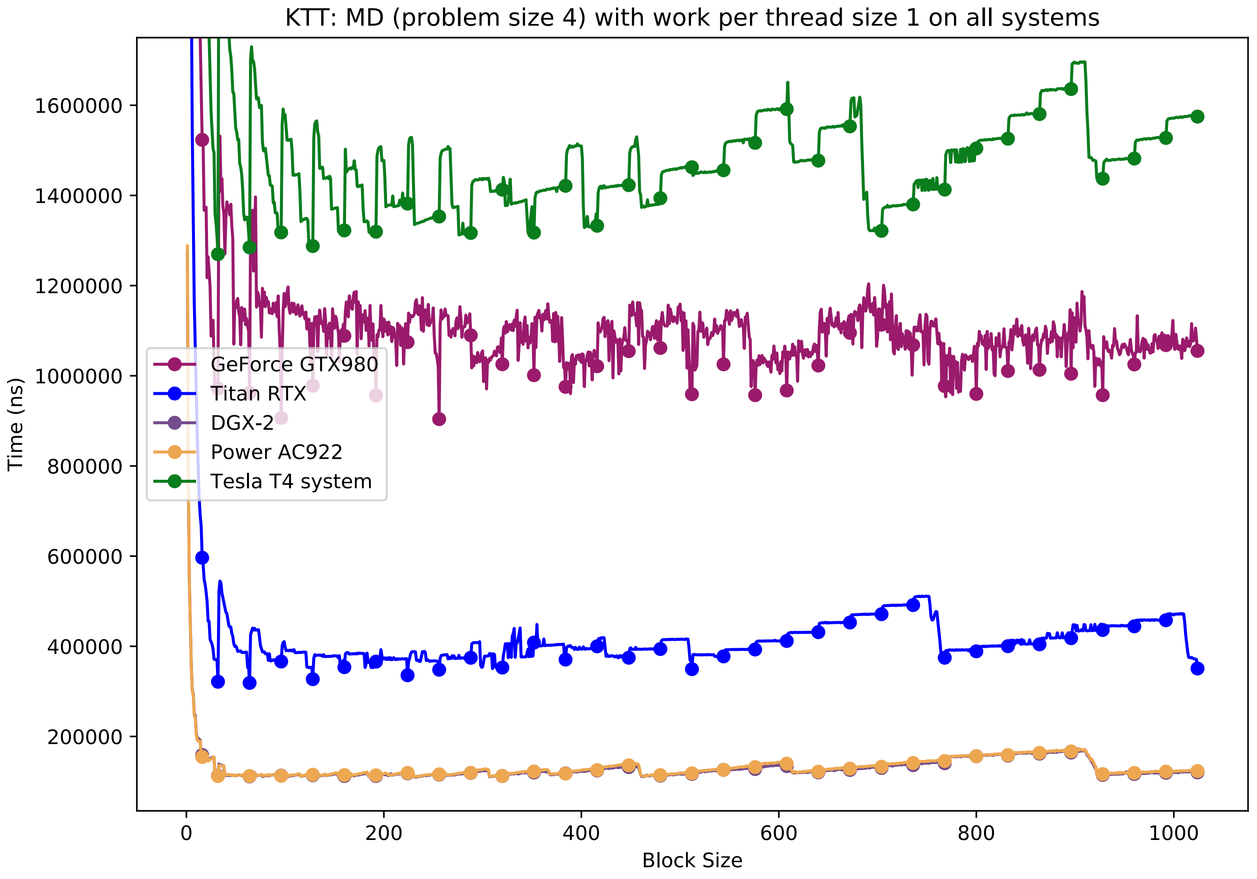

Figure 13: Work per thread parameter and block size parameter for all systems for the MD benchmark in KTT.

In this figure we can see that the parameter for block size is a good parameter because it changes for different architectures. Note, that behind the Power AC922 system plots, there are plots for a DGX-2 system. This is most likely due to the GPUs being the same model for these systems, leading to very similar runtime. For all the different GPU results in the figure, a very low block size results into very bad runtime, and just by increasing this slightly we can see that it improves a lot. However the runtime of block size does not change any drastically for the rest of these sizes.

In this figure as well, the block sizes are dividable by 32 and the block size equal to 16 is marked with a circle. From this we can notice better runtime for each block size value matching these values. At least very easily for the Tesla T4 system.

One observation in Figure 13, was that despite NVIDIA Tesla T4 GPUs being a much newer generation graphics card and having generally better performance than the NVIDIA GeForce GTX 980, it performed worse. This is an interesting result and might be due to the difference in the interconnects and how the GPUs are connected to the CPUs in the two systems.

8.2 Benchmark Suite Evaluation

Below is an evaluation of each of the criteria that were defined prior to the creating of the benchmark suite in collaboration with Ingunn Sund. The criteria are evaluated based on the benchmarks implemented in this thesis.

- The benchmark suite should have:
  - *HPC based benchmarks.*
    This criteria was was fulfilled by that common HPC benchmarks such as `Sort`, `Reduction`, `Triad`, `Molecular Dynamics` and `Stencil 2D` was implemented.
  - *Parameterized algorithms as benchmarks.*
    All the implemented algorithms was parameterized and added multiple parameters each.
  - *Varied selection of benchmarks with different degree of complexity and scope.*
    The implemented benchmarks range from simple computations as in the `Triad` benchmark to the more advanced `Stencil 2D` benchmark.
  - *Benchmarks that utilizes frameworks to enable running code on GPUs. There should be support for both CUDA and OpenCL to make it possible to run the code on both NVIDIA and AMD GPUs.*
    All the benchmarks implemented are running on GPUs and are implemented using CUDA. However none of them are using OpenCL and this should be considered implemented in a future version of BAT to make the benchmark suite available on AMD GPUs as well as NVIDIA GPUs.
  - *Benchmarks that can run on multi-GPU systems and distribute work on multiple nodes.*
    The benchmark `Reduction` is implemented to work with the true parallel version, that is using MPI to communicate the data between multiple GPUs. The benchmark `Stencil 2D` does work for multiple GPUs as well.
  - *Support for different types of auto-tuners. If the auto-tuner does not support certain parameters or the auto-tuner only supports tuning of kernels, there should still not be a problem using the benchmarks.*
    This criteria is fulfilled by that the auto-tuners OpenTuner, Kernel Tuner, CLTune and KTT was implemented for all benchmarks. This shows that it is possible for different types of auto-tuners because OpenTuner and Kernel Tuner can tune a full program and CLTune and KTT can tune kernels.
  - *Benchmarks that have been well tested with different auto-tuners and on different machines.*
    The benchmarks was tested for the different implemented auto-tuners and correction verification of the results computed was added to the implementations. The different auto-tuners was tested on a number of different systems, ranging from single GPU system to multi-GPU systems.
  - *Examples of how to use the benchmarks with auto-tuners.*
    BAT contains a series of the different implemented auto-tuner benchmarks in a separate directory. Users of the benchmark suite can get inspiration about how to implement other auto-tuners by looking at these sources.
  - *A way to compare auto-tuners with other auto-tuners.*
    This was not a goal for this thesis due to this being a large task to implement with the need of researching a scoring system for the auto-tuners and finding the best way to do this. This should be included in a future version of BAT.
- The parameterized algorithms should contain:
  - *Both full programs and single GPU kernels.*
    The benchmark suite's source code contains two different directories, where one of them

is for programs and one for the kernels. This is so different auto-tuners can choose which to tune.

  - *Some algorithms with enough parameters that brute force is not efficient. There should be a variation of the search space size for the different algorithms.*
    Benchmarks such as `Triad` and `Reduction` have search spaces of 40960 and 394240 $\times$ $GPUS$ respectively. This will increase the tuning time, but for our tested systems a good enough value. The benchmarks range from `GPUS` to 394240 $\times$ $GPUS$, which shows a variation in the search spaces. But even larger search spaces could be obtained by implementing even more parameters for the algorithms to ensure that brute force is not possible at all in reasonable time for the benchmarks.
  - *Parameters that potentially could have different values on different machines or architectures.*
    As shown in the previously discussed results, the values can have different optimal values depending on the system architecture.
  - *Some benchmarks with possibility for restrictions or constraints on the parameters possible values.*
    The benchmark `Sort` has two constraints, one between the data size parameters and the block size parameters, and another between the available and needed shared memory. The benchmark `Reduction` has a restriction for the block size parameter that can not be 32.

- The benchmark suite should be user friendly by being:
  - *A well structured project.*
    After making BAT user friendly, the project was well structured in my opinion. This because one can easily copy out a single benchmark from the source directory or implement ones own new auto-tuner for BAT.
  - *Easy to use.*
    The benchmark suite was made easy to use by including Dockerfiles for setup and a `main.py` file for running the benchmarks. The `main.py` command-line interface was made easy to use and customizable for running different types of benchmarks.
  - *A benchmark suite with good documentation. It should be clear what the project is and who could benefit from using it. There should be a guide for using the benchmark suite.*
    The BAT repository contains a readme file documentation guide including a description of what this project is. This readme specifies who the target audience of the benchmark suite is and who could benefit from using it. In the readme file it also specifies how to use the Dockerfiles for setup and benchmarking and for using the command-line interface to run the benchmarks. The documentation also includes how others can use BAT with their own auto-tuners.

8.3 Research Questions

The research questions are shown and answered here:

- *Is SHOC a good benchmark suite to base a benchmark suite for auto-tuners for?*
  As shown in the results, SHOC seems to be a good fit for basing a benchmark suite for auto-tuners of. This is shown by that SHOC is based on a variation of HPC benchmarks, some

of the benchmarks are created for multi-GPU benchmarking and that the benchmark suite is known meaning the quality might be good.

- *Will this benchmark suite have enough GPU focus?*
  All the benchmarks implemented to be tuned with BAT are having a GPU focus for the benchmarks.
- *Will it work with different types of auto-tuners?*
  The benchmarks in BAT was implemented to work in the auto-tuners OpenTuner, Kernel Tuner, CLTune and KTT. These auto-tuners are different in the way that they represent both full program and host-code tuning.
- *Will the optimal values for the implemented parameters differ for different systems?*
  As discussed in the previous section, the optimal values differ for different systems.

8.4 Auto-Tuner Evaluation

In this subsection, I will provide an evaluation of each of the auto-tuners used to implement the benchmarks for BAT. The evaluation will contain both positive and negative feedback for the auto-tuners, what tuning techniques that are available to use in the auto-tuners and how easy it is to set it up.

8.4.1 CLTune

At the start CLTune did not work out of the box, when tuning CUDA kernels. This was fixed in it's open-source Git repository. CLTune does not contain a way of using correctness verification of the kernels for both single- and double precision at the same time. This lead to only having correctness verification when developing the benchmarks. The tuned kernels was not able to include templates, which lead to some work required to convert the benchmarks from SHOC. CLTune does not include the possibility to tune multiple kernels at the same time. It is not possible to use multi-GPU while tuning either, which is something that could have been practical for testing scaling of benchmarks.

There was no possibility to use texture memory with a CLTune kernel tuning due to that the host-code needed to create the texture memory mapped to the input data. This could be implemented into KTT in the future, but might require some work. Therefore the parameter `TEXTURE_MEMORY` was not able to tune in CLTune.

CLTune contained different tuning techniques that could be used, but also worked well by adjusting the search space when changing tuning technique. The user friendly command-line interface indicates to the user how far the auto-tuner is in the tuning process, which can be good and nice to have. The tuning of the kernels also worked relatively fast.

8.4.2 KTT

As KTT is based on CLTune, it lacks some of the same functionality. However KTT contained a tuning manipulator class to be able to tune kernels for them selves. This turned out to be a good feature, especially for the sort benchmark implementation where there are multiple kernels that depends on the results of each other. At the start KTT did not work out of the box either, when tuning CUDA kernels. This was also fixed in it's open-source Git repository.

Even though there exists a possibility to use different tuning techniques, there seems to be an issue about using them. This is because KTT will just run the complete set of different parameter configurations.

As for CLTune, when a reference kernel is used for a correctness verification, the reference checking cannot use different precisions for the input data. This makes it tedious to test implementations with multiple precisions. The way used in this thesis was to hardcode the precision and test it manually for both floating point precisions.

There is no way to limit some of the auto tuner techniques such as MCMC and random before iterating over all different possibilities. This makes the different techniques for KTT not that usable, and just a brute force technique, but with a different order of the tested parameter configurations.

As for CLTune, it was not possible to use texture memory with a KTT kernel. Therefore parameter `TEXTURE_MEMORY` was not able to tune in KTT. Shared memory had the same issue as texture memory and was therefore also disabled for the tuning.

8.4.3 Kernel Tuner

For Kernel Tuner both full program and GPU kernel was possible to use for tuning the kernels. However for the full program, it took longer time to perform the tuning. The documentation for Kernel Tuner was good and explanatory for the different parts of the auto-tuner. As for KTT and CLTune, there was no possibility to perform the tuning on multiple GPUs for Kernel Tuner either.

8.4.4 OpenTuner

OpenTuner had the possibility to tune kernels or other programs created in different programming languages. However the documentation of OpenTuner was not good, for example to retrieve a list of the different tuning techniques implemented, it had to be printed out from the terminal. OpenTuner does support for different commands executed for compiling and for running, leading to parameters used in the commands itself. This also included the way to use multiple GPUs. Despite OpenTuner including many different tuning techniques, there was no technique for using brute force.

9 Conclusion and Future Work

This thesis presented the creation of the benchmark suite for auto-tuners named BAT (**B**enchmark suite for **A**uto-**T**uners). BAT is a benchmark suite for auto-tuners for GPU and HPC oriented programs created in CUDA developed jointly with Ingunn Sund where I worked on developing code for the multi-GPU side for the DGX2 and an IBM Power9 system with two GPUs, whereas Sund worked on the 20-GPU Supermicro system and a 4-GPU IBM Power9 System. This benchmark suite was proposed as a solution since there are currently no other standard benchmark suites for auto-tuners that we are aware of.

A varied selection of benchmarks of different degree of complexity and scope were included in our BAT framework. Some of the benchmarks were able to run on multi-GPU systems and distribute the the work among multiple nodes. The benchmark suite has support for benchmarking different types of auto-tuners. If the benchmarked auto-tuner does not support a specific type of parameter, or only support tuning of kernels, the benchmark was provided with different versions so comprehend this issue. BAT provides a well structured project containing a good user friendly documentation for how to setup and use the benchmark suite. In the documentation it show how to easily add a new auto-tuner for benchmarking or extracting benchmarks out of the BAT source code. There are also a command-line interface for running the benchmarks which comes with different input arguments both for advanced and novice users.

The benchmarks included as examples in the benchmark suite were well tested for different types of systems using different types of auto-tuners. BAT thus provided an answer to the research question, *Is SHOC a good benchmark suite to base a benchmark suite for auto-tuners for?* by presenting a both working and good benchmark suite for auto-tuners based on the SHOC benchmarks.

9.1 Future Work

In the future BAT could be extended to compare different auto-tuners by assigning a scoring for each of them while benchmarking. This was not a goal for this thesis due to being a large task to implement with the need of researching a scoring system for the auto-tuners and finding the best way to do this. However it is a good way of continuing the work started in this project and with the current command-line interface this could be added very nicely to the already existing benchmarking.

The results obtained by running the benchmarks by BAT should be further analyzed. And other results could be collected as well. It was not time for this in the time frame of this thesis.

In a future version of BAT, OpenCL kernels should be included in the benchmarks, making the benchmarks available to many more types of architectures. Such as AMD GPUs and Intel Core processors.

A tip for a feature for auto-tuners that could be implemented are distributing the auto-tuning process on multiple GPUs for parallel tuning of programs.

References

[1] I. M. Liseter and A. C. Elster, “Grafikkprosessor.” https://snl.no/grafikkprosessor, September 2019. [Accessed Dec. 10, 2019].

[2] NVIDIA, “Programming Guide :: CUDA Toolkit Documentation.” https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, June 2020. [Accessed Jun. 13, 2020].

[3] TechPowerUp, “NVIDIA GeForce GTX 980.” https://www.techpowerup.com/gpu-specs/geforce-gtx-980.c2621. [Accessed Dec. 11, 2019].

[4] N. Corporation, “Maxwell Architecture — NVIDIA Developer.” https://developer.nvidia.com/maxwell-compute-architecture. [Accessed Dec. 16, 2019].

[5] R. Nohria, G. Santos (IBM Corporation), “IBM Power System AC922: Technical Overview and Introduction.” https://www.redbooks.ibm.com/redpapers/pdfs/redp5494.pdf, July 2018. [Accessed Nov. 5, 2019].

[6] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, “NVIDIA Tensor Core Programmability, Performance & Precision.” https://arxiv.org/pdf/1803.04014.pdf, March 2018. [Accessed Nov. 5, 2019].

[7] NVIDIA Corporation, “TITAN RTX Ultimate PC Graphics Card with Turing — NVIDIA.” https://www.nvidia.com/en-us/deep-learning-ai/products/titan-rtx/, 2019. [Accessed Dec. 10, 2019].

[8] NVIDIA Corporation, “NVIDIA TURING GPU ARCHITECTURE.” https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018. [Accessed Dec. 20, 2019].

[9] NVIDIA Corporation, “T4 Tensor Core Datasheet.” https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf, 2019. [Accessed Apr. 14, 2020].

[10] IBM Corporation, “IBM Power System AC922.” https://www.ibm.com/downloads/cas/6PRDKRJ0, 2019. [Accessed Nov. 5, 2019].

[11] IBM Corporation, “IBM Power System AC922 - Details.” https://www.ibm.com/us-en/marketplace/power-systems-ac922/details, 2018. [Accessed Nov. 5, 2019].

[12] NVIDIA Corporation, “NVIDIA DGX-2 Datasheet.” https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-2-datasheet-us-nvidia-955420-r2-web-new.pdf, July 2019. [Accessed Nov. 5, 2019].

[13] A. Ishii, D. Foley, E. Anderson, B. Dally, G. Dearth, L. Dennison, M. Hummel, and J. Schafer, “NVSWITCH AND DGX-2: NVLINK-SWITCHING CHIP AND SCALE-UP COMPUTE SERVER.” https://www.hotchips.org/hc30/2conf/2.01_Nvidia_NVswitch_HotChips2018_DGX2NVS_Final.pdf, 2018. [Accessed Nov. 11, 2019].

[14] MPI Forum, “MPI Forum.” https://www.mpi-forum.org/, 2019. [Accessed Dec. 12, 2019].

[15] The Open MPI Project, "Open MPI: Open Source High Performance Computing." https://www.open-mpi.org/, 2019. [Accessed Dec. 12, 2019].

[16] NVIDIA Corporation, "IBM Spectrum MPI — NVIDIA Developer." https://developer.nvidia.com/ibm-spectrum-mpi, 2019. [Accessed Dec. 5, 2019].

[17] MPICH, "MPICH — High-Performance Portable MPI." https://www.mpich.org/, 2019. [Accessed Dec. 12, 2019].

[18] C. Ramseyer, "PCI Express 4.0 Brings 16 GT/s And At Least 300 Watts At The Slot." https://www.tomshardware.com/news/pcie-4.0-power-speed-express,32525.html, August 2016. [Accessed Dec. 10, 2019].

[19] HowStuffWorks, "How PCI Express Works — HowStuffWorks." https://computer.howstuffworks.com/pci-express.htm, 2019. [Accessed Dec. 16, 2019].

[20] N. Corporation, "NVLink High-Speed GPU Interconnect — NVIDIA Quadro." https://www.nvidia.com/en-us/design-visualization/nvlink-bridges, 2019. [Accessed Dec. 10, 2019].

[21] M. N. Farooqi, T. Nguyen, W. Zhang, A. S. Almgren, J. Shalf, and D. Unat, "Asynchronous AMR on Multi-GPUs." https://link.springer.com/chapter/10.1007/978-3-030-34356-9_11, 2019. [Accessed Dec. 15, 2019].

[22] NVIDIA Corporation, "NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch." https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, May 2018. [Accessed Nov. 7, 2019].

[23] G. Dearth and V. Venkataraman, "S8688: INSIDE DGX-2." http://on-demand.gputechconf.com/gtc/2018/presentation/s8688-extending-the-connectivity-and-reach-of-the-gpu.pdf, March 2018. [Accessed Nov. 12, 2019].

[24] Docker Inc., "Docker overview — Docker Documentation." https://docs.docker.com/get-started/overview/, 2020. [Accessed Mar. 15, 2020].

[25] Docker Inc., "DockerWhat is a Container? — App Containerization — Docker." https://www.docker.com/resources/what-container, 2020. [Accessed Mar. 15, 2020].

[26] R. Olson, J. Calmels, F. Abecassis, and P. Rogers, "NVIDIA Docker: GPU Server Application Deployment Made Easy — NVIDIA Developer Blog." https://developer.nvidia.com/blog/nvidia-docker-gpu-server-application-deployment-made-easy/, 2020. [Accessed Mar. 15, 2020].

[27] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite." https://www.researchgate.net/publication/220938804_The_Scalable_HeterOgeneous_Computing_SHOC_benchmark_suite, 2010. [Accessed Nov. 12, 2019].

[28] A. Danalis, G. M. C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "GitHub - The SHOC Benchmark Suite - Wiki." https://github.com/vetter/shoc/wiki, 2014. [Accessed Nov. 12, 2019].

[29] Programiz, "Programiz: Learn to Code for Free." `https://www.programiz.com/dsa/radix-sort`, 2020. [Accessed Apr. 15, 2020].

[30] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "GitHub - The SHOC Benchmark Suite." `https://github.com/vetter/shoc`, 2014. [Accessed Nov. 12, 2019].

[31] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 12 1995.

[32] G. Barlas, "Chapter 7 - the thrust template library," in *Multicore and GPU Programming* (G. Barlas, ed.), pp. 527 – 573, Boston: Morgan Kaufmann, 2015.

[33] J. Adams, "Bonding energy models," in *Encyclopedia of Materials: Science and Technology* (K. J. Buschow, R. W. Cahn, M. C. Flemings, B. Ilschner, E. J. Kramer, S. Mahajan, and P. Veyssière, eds.), pp. 763 – 767, Oxford: Elsevier, 2001.

[34] F. Petrovič, D. Střelák, J. Hozzová, J. Ol'ha, R. Trembecký, S. Benkner, and J. Filipovič, "A benchmark set of highly-efficient cuda and opencl kernels and its dynamic autotuning with kernel tuning toolkit," *Future Generation Computer Systems*, vol. 108, p. 161–177, Jul 2020.

[35] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019.

[36] B. van Werkhoven, "GitHub - Kernel Tuner." `https://github.com/benvanwerkhoven/kernel_tuner`, 2019. [Accessed May. 20, 2020].

[37] C. Nugteren and V. Codreanu, "CLTune: A Generic Auto-Tuner for OpenCL Kernels," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pp. 195–202, 2015.

[38] C. Nugteren, "CNugteren/CLCudaAPI: A portable high-level API with CUDA or OpenCL back-end." `https://github.com/CNugteren/CLCudaAPI`, 2015. [Accessed Mar. 27, 2020].

[39] F. Petrovič, "Fillo7/KTT: Kernel Tuning Toolkit." `https://github.com/Fillo7/KTT`, 2017. [Accessed Mar. 28, 2020].

[40] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), p. 303–316, Association for Computing Machinery, 2014.

[41] A. Sclocco, "isazi/TuneBench: Simple tunable OpenCL kernels for many-core accelerators.." `https://github.com/isazi/TuneBench`, 2016. [Accessed Mar. 29, 2020].

[42] A. Rasch and S. Gorlatch, "ATF: A generic directive-based auto-tuning framework," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 5, p. e4423, 2019. e4423 cpe.4423.

[43] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[44] S. C. Foundation, "How to mix C and C++, C++ FAQ." `https://isocpp.org/wiki/faq/mixing-c-and-cpp`, 2020. [Accessed Jun. 5, 2020].

[45] Inspur Systems, "AGX-5 - Inspur Systems." `https://www.inspursystems.com/product/agx-5/`, 2018. [Accessed Dec. 15, 2019].

[46] P. Alcorn, "Inside The World's Largest GPU: Nvidia Details NVSwitch." `https://www.tomshardware.com/news/nvidia-dgx-2-worlds-largest-gpu-nvswitch,37661.html`, August 2018. [Accessed Dec. 20, 2019].

A Parameter Research

This section contains tables with several parameters used in the examples in auto-tuners described in this thesis. The parameters from Kernel Tuner, CLTune and KTT were found in their GitHub repositories: `github.com/benvanwerkhoven/kernel_tuner`, `github.com/CNugteren/CLTune` and `github.com/Fillo7/KTT` respectivly. The parameters from OpenTuner were found in the paper describing the framework: *OpenTuner: An Extensible Framework for Program Autotuning* by J. Ansel et al. [40]. This appendix was done in collaboration with Ingunn Sund.

A.1 Kernel Tuner

Table 28: Parameters used in Convolution example in Kernel Tuner.

Parameter	**Search Space**	**Explanation if Provided**
`filter_height`	[i for i in range(3,35,2)]	
`filter_width`	[i for i in range(3,35,2)]	
`block_size_x`	[16*i for i in range(1,9)]	
`block_size_y`	[2**i for i in range(6)]	
`tile_size_x`	[i for i in range(1,9)]	
`tile_size_y`	[i for i in range(1,9)]	
`use_padding`	[0,1]	Padding in shared memory
`read_only`	[0,1]	Read-only cache

Table 29: Parameters used in Convolution Streams example in Kernel Tuner.

Parameter	**Search Space**	**Explanation if Provided**
`block_size_x`	[16*i for i in range(1,17)]	
`block_size_y`	[2**i for i in range(5)]	
`tile_size_x`	[2**i for i in range(4)]	
`tile_size_y`	[2**i for i in range(4)]	
`num_streams`	[2**i for i in range(6)]	

Table 30: Parameters used in Expdist example in Kernel Tuner.

Parameter	**Search Space**	**Explanation if Provided**
`block_size_x`	[2**i for i in range(5,10)]	
`block_size_y`	[2**i for i in range(6)]	
`tile_size_x`	[2**i for i in range(4)]	
`tile_size_y`	[2**i for i in range(4)]	
`use_shared_mem`	[0, 1]	

Table 31: Parameters used in Matrix Multiplication example in Kernel Tuner.

Parameter	Search Space	Explanation if Provided
`block_size_x`	[16*2**i for i in range(3)]	
`block_size_y`	[2**i for i in range(6)]	
`tile_size_x`	[2**i for i in range(4)]	
`tile_size_y`	[2**i for i in range(4)]	

Table 32: Parameters used in Point-in-Polygon example in Kernel Tuner.

Parameter	Search Space	Explanation if Provided
`block_size_x`	[32*i for i in range(1,32)]	Block size is a multiple of 32
`tile_size`	[1] + [2*i for i in range(1,11)]	
`between_method`	[0, 1, 2, 3]	
`use_precomputed_slopes`	[0, 1]	
`use_method`	[0, 1]	

Table 33: Parameters used in Reduction example in Kernel Tuner.

Parameter	Search Space	Explanation if Provided
`block_size_x`	[2**i for i in range(5,11)]	
`use_shuffle`	[0, 1]	Shuffle instructions (CUDA only)
`vector`	[2**i for i in range(3)]	Vector type
`num_blocks`	[2**i for i in range(5,16)]	The number of thread blocks the kernel is executed with

Table 34: Parameters used in SpMV example in Kernel Tuner.

Parameter	Search Space	Explanation if Provided
`block_size_x`	[32*i for i in range(1,33)]	
`threads_per_row`	[1, 32]	
`read_only`	[0, 1]	

Table 35: Parameters used in Stencil example in Kernel Tuner.

Parameter	Search Space	Explanation if Provided
`block_size_x`	[32*i for i in range(1,9)]	
`block_size_y`	[2**i for i in range(6)]	

Table 36: Parameters used in Texture example in Kernel Tuner.

Parameter	Search Space	Explanation if Provided
`block_size_x`	[16, 32]	
`block_size_y`	[16, 32]	
`oldiw`	[1024]	
`oldih`	[1024]	
`newiw`	[1024]	
`newih`	[1024]	

Table 37: Parameters used in Vector Add example in Kernel Tuner.

Parameter	Search Space	Explanation if Provided
`block_size_x`	[128+64*i for i in range(15)]	

Table 38: Parameters used in Zero Mean Filter example in Kernel Tuner.

Parameter	Search Space	Explanation if Provided
`block_size_x`	[32*i for i in range(1,9)]	
`block_size_y`	[2**i for i in range(6)]	

A.2 CLTune

Table 39: Parameters used in Simple example in CLTune.

Parameter	Search Space	Explanation if Provided
`GROUP_SIZE`	[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]	

Table 40: Parameters used in Convolution Simple example in CLTune.

Parameter	Search Space	Explanation if Provided
`TBX`	[8, 16, 32]	Work group size dim x (threads in block)
`TBY`	[8, 16, 32]	
`WPTX`	[1, 2, 4]	Work Per Thread dim X
`WPTY`	[1, 2, 4]	
`VECTOR`	[1, 2, 4]	

Table 41: Parameters used in Convolution example in CLTune.

Parameter	**Search Space**	**Explanation if Provided**
`TBX`	[8, 16, 32, 64]	Work group size dim x (threads in block)
`TBY`	[8, 16, 32, 64]	
`LOCAL`	[0, 1, 2]	
`WPTX`	[1, 2, 4, 8]	Work Per Thread dim X
`WPTY`	[1, 2, 4, 8]	
`VECTOR`	[1, 2, 4]	
`UNROLL_FACTOR`	[1, FS]	FS = Filter size
`PADDING`	[0, 1]	
`TBX_XL`	[8,9,10,11,12,13,14,15,16,17,18,19, 20,21,22,23,24,25,26,32,33,34,35,36, 37,38,39,40,41,42,64,65,66,67,68,69, 70,71,72,73,74]	
`TBY_XL`	[8,9,10,11,12,13,14,15,16,17,18,19, 20,21,22,23,24,25,26,32,33,34,35,36, 37,38,39,40,41,42,64,65,66,67,68,69, 70,71,72,73,74]	

Table 42: Parameters used in GEMM example in CLTune.

Parameter	**Search Space**	**Explanation if Provided**
`MWG`	[16, 32, 64, 128]	Tile size dim M
`NWG`	[16, 32, 64, 128]	Tile size dim N
`KWG`	[16, 32]	Tile size dim K
`MDIMC`	[8, 16, 32]	Threads per work group in M dim
`NDIMC`	[8, 16, 32]	Threads per work group in N dim
`MDIMA`	[8, 16, 32]	Tile dimension
`NDIMB`	[8, 16, 32]	Tile dimension
`KWI`	[2, 8]	Unroll loop factor
`VWM`	[1, 2, 4, 8]	Vector width of matrix
`VWN`	[1, 2, 4, 8]	Vector width of matrix
`STRM`	[0, 1]	Strided access M dim
`STRN`	[0, 1]	Strided access N dim
`SA`	[0, 1]	Shared memory matrix A
`SB`	[0, 1]	Shared memory matrix B
`PRECISION`	[32, 64]	Precision for data types

A.3 KTT

Table 43: Parameters used in Conv 3D example in KTT.

Parameter	**Search Space**	**Explanation if Provided**
ALGORITHM	[0, 1, 2]	0: Reference kernel, 1: Blocked kernel, 2: Sliding plane kernel
TBX	[8, 16, 32, 64]	
TBY	[8, 16, 32, 64]	
TBZ	[1, 2, 4, 8, 16, 32]	
LOCAL	[0, 1, 2]	
WPTX	[1, 2, 4, 8]	
WPTY	[1, 2, 4, 8]	
WPTZ	[1, 2, 4, 8]	
VECTOR	[1, 2, 4]	
ATOMICS	[0, 1]	
UNROLL_FACTOR	[1, FS]	
CONSTANT_COEFF	[0, 1]	
CACHE_WORK_TO_REGS	[0, 1]	
REVERSE_LOOP_ORDER	[0, 1]	
REVERSE_LOOP_ORDER2	[0, 1]	
REVERSE_LOOP_ORDER3	[0, 1]	
PADDING	[0, 1]	
Z_ITERATIONS	[4, 8, 16, 32]	
TBX_XL	[1, 2, 3, 4, 8, 9, 10, 16, 17, 18, 32, 33, 34, 64, 65, 66]	Helper parameter for number of threads if LOCAL=2
TBY_XL	[1, 2, 3, 4, 8, 9, 10, 16, 17, 18, 32, 33, 34, 64, 65, 66]	Helper parameter for number of threads if LOCAL=2
TBZ_XL	[1, 2, 3, 4, 8, 9, 10, 16, 17, 18, 32, 33, 34, 64, 65, 66]	Helper parameter for number of threads if LOCAL=2

Table 44: Parameters used in Coulomb Sum 2D example in KTT.

Parameter	Search Space	Explanation if Provided
`INNER_UNROLL_FACTOR`	[0, 1, 2, 4, 8, 16, 32]	
`USE_CONSTANT_MEMORY`	[0, 1]	
`VECTOR_TYPE`	[1, 2, 4, 8]	
`USE_SOA`	[0, 1, 2]	
`OUTER_UNROLL_FACTOR`	[1, 2, 4, 8]	
`WORK_GROUP_SIZE_X`	[4, 8, 16, 32]	
`WORK_GROUP_SIZE_Y`	[1, 2, 4, 8, 16, 32]	

Table 45: Parameters used in BICG example in KTT.

Parameter	Search Space	Explanation if Provided
`FUSED`	[0, 1, 2]	
`BICG_BATCH`	[1, 2, 4, 8, 16, 32, 64]	
`USE_SHARED_MATRIX`	[0, 1]	
`USE_SHARED_VECTOR_1`	[0, 1]	
`USE_SHARED_VECTOR_2`	[0, 1]	
`USE_SHARED_REDUCTION_1`	[0, 1]	
`USE_SHARED_REDUCTION_2`	[0, 1]	
`ATOMICS`	[0, 1]	
`UNROLL_BICG_STEP`	[0, 1]	
`ROWS_PROCESSED`	[128, 256, 512, 1024]	
`TILE`	[16, 32, 64]	

Table 46: Parameters used in Transpose example in KTT.

Parameter	Search Space	Explanation if Provided
`CR`	[0, 1]	
`LOCAL_MEM`	[0, 1]	
`PADD_LOCAL`	[0, 1]	
`WORK_GROUP_SIZE_X`	[1, 2, 4, 8, 16, 32, 64]	
`WORK_GROUP_SIZE_Y`	[1, 2, 4, 8, 16, 32, 64]	
`TILE_SIZE_X`	[1, 2, 4, 8, 16, 32, 64]	
`TILE_SIZE_Y`	[1, 2, 4, 8, 16, 32, 64]	
`DIAGONAL_MAP`	[0, 1]	

A.4 OpenTuner

Table 47: The top 10 most important compiler flags used for FFT and MM benchmarks in OpenTuner.

FFT	**Matrix Multiply**
`-fno-tree-vectorize`	`-fno-exceptions`
`-funroll-loops`	`-fwrapv`
`-fno-jump-tables`	`-funsafe-math-optimizations`
`-fno-inline`	`-param=large-stack-frame=65`
`-fno-ipa-pure-const`	`-fschedule-insns2`
`-fno-tree-cselim`	`-funroll-loops`
`-fno-rerun-cse-after-loop`	`-fno-ivopts`
`-fno-tree-forwprop`	`-param=sccvn-max-scc-size=2995`
`-fno-tree-tail-merge`	`-param=max-sched-extendregions-iters=2`
`-fno-cprop-registers`	`-param=slp-max-insns-inbb=1786`

Table 48: The top 10 most important compiler flags used for RT and TSP GA benchmarks in OpenTuner.

Ray Tracer	**TSP GA**
`-funsafe-math-optimizations`	`-freorder-blocks-and-partition`
`-ffinite-math-only`	`-funroll-all-loops`
`-frename-registers`	`-param=omega-max-geqs=64`
`-fwhole-program`	`-param=predictable-branch-outcome=2`
`-param=selsched-insns-to-rename=2`	`-param=min-insn-to-prefetch-ratio=36`
`-fno-tree-dominator-opts`	`-fno-rename-registers`
`-param=min-crossjump-insns=17`	`-param=max-unswitch-insns=200`
`-param=max-crossjump-edges=31`	`-param=omega-max-keys=2000`
`-param=sched-state-edge-probcutoff=17`	`-param=max-delay-slot-live-search=83`
`-param=sms-loop-average-countthreshold=4`	`-param=prefetch-latency=50`

B Repository Readme

This appendix includes documentation for the project written in collaboration with Ingunn Sund.

README.md

BAT

> A standardized benchmark suite for auto-tuners

BAT is a standardized benchmark suite for auto-tuners that is based on benchmarks from SHOC and contains benchmarks for CUDA programs. The benchmarks are for both whole programs and kernel-code. BAT will save all your `JSON` and `CSV` results to an own results directory after auto-tuning is completed. Then it will parse specified files and print out the best parameters found by the auto-tuner. The parameters and other benchmarking information will be printed out prettified in the terminal.

This benchmark suite will be useful for you if you're making your own auto-tuner and want to use the benchmarks for testing or would like to compare your auto-tuner to other known auto-tuners. BAT can also be used to check how a parameter's value changes for different architectures.

Parameters

Parameters and search space for the algorithms can be seen in the `src` directory.

Prerequisites

- Python 3 (Or Docker, see section Within a Docker container)

Set up auto-tuner benchmarks

Without using Docker, the following steps are required to download and install the auto-tuners:

- OpenTuner
  - Can be downloaded along other needed dependencies by calling `pip3 install -r requirements.txt` from the tuning_examples/opentuner directory.
- Kernel Tuner
  - Can be downloaded along other needed dependencies by calling `pip3 install -r requirements.txt` from the tuning_examples/kernel_tuner directory.
- CLTune
  - Need to set the environment variable `KTT_PATH=/path/to/KTT` for using the benchmarks.
- KTT
  - Need to set the environment variable `CLTUNE_PATH=/path/to/CLTune` for using the benchmarks.

Running benchmarks

```
# Run all benchmark for all auto-tuners
python3 main.py

# Run the `sort` benchmark for all auto-tuners
python3 main.py -b sort

# Run all benchmarks for auto-tuner `OpenTuner`
python3 main.py -a opentuner

# Run benchmark `scan` for auto-tuner `CLTune`
python3 main.py -b scan -a cltune
```

Command-line arguments

`--benchmark [name]` , `-b [name]`

Default: `none`

Benchmark to run. Example: `sort` . If no benchmark is selected, all benchmarks are ran for selected auto-tuner(s).

`--auto-tuner [name]` , `-a [name]`

Default: `none`

Auto-tuner to run benchmarks for. Example: `ktt` . If no auto-tuner is selected, all auto-tuners are selected for benchmarking.

`--verbose` , `-v`

Default: `false`

If all `stdout` and `stderr` should be printed out during building of the benchmark(s). By default it does not print out the information during the building.

`--size [number]` , `-s [number]`

Default: `1`

Problem size for the data in the benchmarks. By default it uses a problem size of `1` . This is up to the specific auto-tuner to handle.

`--technique [name]` , `-t [name]`

Default: `brute_force`

Tuning technique to use for benchmarking. If no technique is specified, the brute force technique is selected. This is up to the specific auto-tuner to handle.

Add your own auto-tuner

It is easy to add new auto-tuner implementations for the benchmarks, just follow these steps:

1. Implement the benchmark(s) you want with your auto-tuner. If your auto-tuner tunes a whole program, the benchmarks can be found in src/programs. However if you have an auto-tuner that tunes kernels, the benchmarks can be found in src/kernels, and you have to generate the input data. Generating of input data can be done like in the KTT examples found tuning_examples/ktt.

2. Store your auto-tuner implementation of a benchmark inside a auto-tuner subdirectory in tuning_examples. The path to the benchmark implementation should look similar to `./tuning_examples/kernel_tuner/sort/`.
3. Create a `config.json` file in the same directory as the auto-tuner with content similar to this:

```
{
    "build": [
        "make clean",
        "make"
    ],
    "run": "./sort",
    "results": [
        "best-sort-results.json"
    ]
}
```

Content of `config.json`

- `build` : A list of commands that will be ran before the `run` command. Note, it does not work correctly with `&&` between commands. This is because of a limitation in the package subprocess to run the commands in Python. A solution is therefore to split them in a list.
- `run` : The command to run the auto-tuning benchmark.
- `results` : A list of result files that contains the best parameters found in the auto-tuner benchmark. These will be printed out by BAT after the auto-tuning is completed.

Within a Docker container

Building

Here are some examples of how to build the different auto-tuner Docker images:

```
# Build OpenTuner Dockerfile
$ docker build -t bat-opentuner -f docker/opentuner.Dockerfile .

# Build Kernel Tuner Dockerfile
$ docker build -t bat-kernel_tuner -f docker/kernel_tuner.Dockerfile .

# Build CLTune Dockerfile
$ docker build -t bat-cltune -f docker/cltune.Dockerfile .

# Build KTT Dockerfile
$ docker build -t bat-ktt -f docker/ktt.Dockerfile .
```

Running

Here are some examples of how to run the different auto-tuner Docker containers:

```
# Run the KTT container
$ docker run -ti --gpus all bat-ktt

# Example of running container detatched
$ docker run -d -ti --gpus all bat-ktt

# Open a shell into a detatched container
$ docker exec -it <container-id> sh

# After this the commands shown in the `Running benchmarks` section can be used
# Example:
$ main.py -b sort -a ktt -t mcmc -s 4
```

C System Information

The system information in this appendix is collected in collaboration with Ingunn Sund and some parts are from my specialization project.

C.1 NVIDIA GeForce GTX 980 Based System

C.1.1 GPU

```
ingunsu@hpclab04:~$ nvidia-smi topo -m
        GPU0    CPU Affinity    NUMA Affinity
GPU0     X      0-7             N/A

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA
    nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host
    Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the
    CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe
    Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

Listing 53: Topology for GTX 980 system

```
ingunsu@hpclab04:~$ nvidia-smi nvlink --status -i 0
ingunsu@hpclab04:~$
```

Listing 54: NVLink status for GTX 980 system. No results because the GPU does not have the possibility for NVLink connections

```
ingunsu@hpclab04:~$ nvidia-smi
Sat Oct 10 16:18:39 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.51.06    Driver Version: 450.51.06    CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  GeForce GTX 980     On   | 00000000:01:00.0 Off |                  N/A |
| 27%   30C    P8    13W / 180W |    154MiB /  4041MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Listing 55: Information about the GTX 980 GPU when running the nvidia-smi command

C.1.2 CPU

```
ingunsu@hpclab04:~$ lscpu
Architecture:         x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:           Little Endian
CPU(s):               8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):            1
NUMA node(s):         1
Vendor ID:            GenuineIntel
CPU family:           6
Model:                94
Model name:           Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
Stepping:             3
CPU MHz:              800.251
CPU max MHz:          4200.0000
CPU min MHz:          800.0000
BogoMIPS:             7999.96
Virtualization:       VT-x
L1d cache:            32K
L1i cache:            32K
L2 cache:             256K
L3 cache:             8192K
NUMA node0 CPU(s):   0-7
```

Listing 56: Information about the CPU in the GTX 980 based system when running the lscpu command

C.2 NVIDIA Titan RTX Based System

C.2.1 GPU

```
ingunsu@hpclab15:~$ nvidia-smi topo -m
        GPU0    CPU Affinity    NUMA Affinity
GPU0     X      0-15            N/A

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA
    nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host
    Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the
    CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe
    Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

Listing 57: Topology for Titan RTX system

```
ingunsu@hpclab15:~$ nvidia-smi nvlink --status -i 0
GPU 0: TITAN RTX (UUID: GPU-8583ed85-a5e2-eeb3-178a-5921ab72dcf3)
         Link 0: <inactive>
         Link 1: <inactive>
```

Listing 58: NVLink status for Titan RTX system. The system has a possibility for two NVLink connections, but they are not in use on this specific computer.

```
ingunsu@hpclab15:~$ nvidia-smi
Sun Oct 11 19:44:13 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 455.23.05    Driver Version: 455.23.05    CUDA Version: 11.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  TITAN RTX           On   | 00000000:01:00.0 Off |                  N/A |
| 41%   27C    P8    15W / 280W |     20MiB / 24220MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Listing 59: Information about the Titan RTX GPU when running the nvidia-smi command

C.2.2 CPU

```
ingunsu@hpclab15:~$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              16
On-line CPU(s) list: 0-15
Thread(s) per core:  2
Core(s) per socket:  8
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               158
Model name:          Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
Stepping:            13
CPU MHz:             799.828
CPU max MHz:         5000.0000
CPU min MHz:         800.0000
BogoMIPS:            7200.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            16384K
NUMA node0 CPU(s):   0-15
```

Listing 60: Information about the CPU in the RTX Titan based system when running the lscpu command

C.3 NVIDIA Tesla T4 Based System

```
selbu:~$ nvidia-smi nvlink --status -i 0
selbu:~$
```

Listing 61: NVLink status for part of NVIDIA Tesla T4 system. No results because the GPU does not have the possibility for NVLink connections

```
selbu:~$ nvidia-smi
Sun Oct 11 23:33:42 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.100      Driver Version: 440.100      CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:1A:00.0 Off |                    0 |
| N/A   36C    P8    10W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
...
```

Listing 62: Information about the first GPU when running the nvidia-smi command on the NVIDIA Tesla T4 system

C.4 DGX-2

```
ingunsu@DGX-2:~$ nvidia-smi topo -m
      GPU0 G1  G2  G3  G4  G5  G6  G7  G8  G9  G10 G11 G12 G13 G14 G15   CPU Affinity
GPU0   X   NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   0-23,48-71
GPU1  NV6   X  NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   0-23,48-71
GPU2  NV6 NV6   X  NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   0-23,48-71
GPU3  NV6 NV6 NV6   X  NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   0-23,48-71
GPU4  NV6 NV6 NV6 NV6   X  NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   0-23,48-71
GPU5  NV6 NV6 NV6 NV6 NV6   X  NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   0-23,48-71
GPU6  NV6 NV6 NV6 NV6 NV6 NV6   X  NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   0-23,48-71
GPU7  NV6 NV6 NV6 NV6 NV6 NV6 NV6   X  NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   0-23,48-71
GPU8  NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   X  NV6 NV6 NV6 NV6 NV6 NV6 NV6   24-47,72-95
GPU9  NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   X  NV6 NV6 NV6 NV6 NV6 NV6   24-47,72-95
GPU10 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   X  NV6 NV6 NV6 NV6 NV6   24-47,72-95
GPU11 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   X  NV6 NV6 NV6 NV6   24-47,72-95
GPU12 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   X  NV6 NV6 NV6   24-47,72-95
GPU13 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   X  NV6 NV6   24-47,72-95
GPU14 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   X  NV6   24-47,72-95
GPU15 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6 NV6   X    24-47,72-95

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA
    nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host
    Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU
    )
  PXB  = Connection traversing multiple PCIe switches (without traversing the PCIe
    Host Bridge)
  PIX  = Connection traversing a single PCIe switch
  NV#  = Connection traversing a bonded set of # NVLinks

ingunsu@DGX-2:~$ nvidia-smi topo -mp
     G0   G1   G2   G3   G4   G5   G6   G7   G8   G9   G10  G11  G12  G13  G14  G15
G0   X    PIX  PXB  PXB  NODE NODE NODE NODE SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS
G1   PIX  X    PXB  PXB  NODE NODE NODE NODE SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS
G2   PXB  PXB  X    PIX  NODE NODE NODE NODE SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS
G3   PXB  PXB  PIX  X    NODE NODE NODE NODE SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS
G4   NODE NODE NODE NODE X    PIX  PXB  PXB  SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS
G5   NODE NODE NODE NODE PIX  X    PXB  PXB  SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS
G6   NODE NODE NODE NODE PXB  PXB  X    PIX  SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS
G7   NODE NODE NODE NODE PXB  PXB  PIX  X    SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS
G8   SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS  X    PIX  PXB  PXB  NODE NODE NODE NODE
G9   SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS  PIX  X    PXB  PXB  NODE NODE NODE NODE
G10  SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS  PXB  PXB  X    PIX  NODE NODE NODE NODE
G11  SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS  PXB  PXB  PIX  X    NODE NODE NODE NODE
G12  SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS  NODE NODE NODE NODE X    PIX  PXB  PXB
G13  SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS  NODE NODE NODE NODE PIX  X    PXB  PXB
G14  SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS  NODE NODE NODE NODE PXB  PXB  X    PIX
G15  SYS  SYS  SYS  SYS  SYS  SYS  SYS  SYS  NODE NODE NODE NODE PXB  PXB  PIX  X
```

Listing 63: Topology for DGX-2 (G=GPU). First matrix is the direct communication matrix, the second is PCI only.

```
ingunsu@DGX-2:~$ nvidia-smi nvlink --status -i 0
GPU 0: Tesla V100-SXM3-32GB (UUID: GPU-ad78d3a5-0a4f-ac16-0ea4-e02b88404047)
    Link 0: 25.781 GB/s
    Link 1: 25.781 GB/s
    Link 2: 25.781 GB/s
    Link 3: 25.781 GB/s
    Link 4: 25.781 GB/s
    Link 5: 25.781 GB/s

...

ingunsu@DGX-2:~$ nvidia-smi nvlink --status -i 15
GPU 15: Tesla V100-SXM3-32GB (UUID: GPU-4e5a4b58-56fc-114b-5de2-ee41540cc549)
    Link 0: 25.781 GB/s
    Link 1: 25.781 GB/s
    Link 2: 25.781 GB/s
    Link 3: 25.781 GB/s
    Link 4: 25.781 GB/s
    Link 5: 25.781 GB/s
```

Listing 64: NVLink status on DGX-2. The listing shows only GPU 0 and 15 because the command will print the same for every GPU in the system.

```
ingunsu@DGX-2:~$ nvidia-smi
Sun Dec 15 16:51:35 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.87.01    Driver Version: 418.87.01    CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM3...  On   | 00000000:34:00.0 Off |                    0 |
| N/A   33C    P0    49W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Listing 65: Information about the first GPU when running the nvidia-smi command on the DGX-2

C.4.1 CPUs

```
ingunsu@heid:~/results-bat/opentuner$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                96
On-line CPU(s) list: 0-95
Thread(s) per core:  2
Core(s) per socket:  24
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz
Stepping:              4
CPU MHz:               2687.537
CPU max MHz:           3700,0000
CPU min MHz:           1200,0000
BogoMIPS:              5400.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              33792K
NUMA node0 CPU(s):     0-23,48-71
NUMA node1 CPU(s):     24-47,72-95
```

Listing 66: Information provided about the CPUs in the DGX-2 when running the lscpu command.

C.5 IBM Power System AC922

```
-bash-4.2$ nvidia-smi topo -m
        GPU0  GPU1  CPU Affinity
GPU0     X    SYS    0-63
GPU1    SYS    X     64-127

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA
    nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host
    Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the
    CPU)
  PXB  = Connection traversing multiple PCIe switches (without traversing the PCIe
     Host Bridge)
  PIX  = Connection traversing a single PCIe switch
  NV#  = Connection traversing a bonded set of # NVLinks
```

Listing 67: Topology for IBM Power System AC922 (2 GPUs)

```
-bash-4.2$ nvidia-smi nvlink --status -i 0
GPU 0: Tesla V100-SXM2-16GB (UUID: GPU-d5cfaea6-0aca-7f9b-5ed1-957950b4f8f8)
	 Link 0: <inactive>
	 Link 1: 25.781 GB/s
	 Link 2: <inactive>
	 Link 3: 25.781 GB/s
	 Link 4: <inactive>
	 Link 5: 25.781 GB/s

-bash-4.2$ nvidia-smi nvlink --status -i 1
GPU 1: Tesla V100-SXM2-16GB (UUID: GPU-4822d295-9751-d6b8-bd93-7739510f189e)
	 Link 0: <inactive>
	 Link 1: 25.781 GB/s
	 Link 2: <inactive>
	 Link 3: 25.781 GB/s
	 Link 4: <inactive>
	 Link 5: 25.781 GB/s
```

Listing 68: NVLink status on IBM Power System AC922 (2 GPUs)

```
-bash-4.2$ nvidia-smi
Sun Dec 15 16:49:09 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000004:04:00.0 Off |                    0 |
| N/A   37C    P0    35W / 300W |      0MiB / 16160MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Listing 69: Information about the first GPU when running the nvidia-smi command on IBM Power System AC922 (2 GPUs).

C.5.1 CPUs

```
-bash-4.2$ lscpu
Architecture:          ppc64le
Byte Order:            Little Endian
CPU(s):                128
On-line CPU(s) list:   0-127
Thread(s) per core:    4
Core(s) per socket:    16
Socket(s):             2
NUMA node(s):          6
Model:                 2.2 (pvr 004e 1202)
```

```
Model name:             POWER9, altivec supported
CPU max MHz:            3800.0000
CPU min MHz:            2300.0000
L1d cache:              32K
L1i cache:              32K
L2 cache:               512K
L3 cache:               10240K
NUMA node0 CPU(s):      0-63
NUMA node8 CPU(s):      64-127
NUMA node252 CPU(s):
NUMA node253 CPU(s):
NUMA node254 CPU(s):
NUMA node255 CPU(s):
```

Listing 70: Information provided about the CPUs in the IBM Power System AC922 (2 GPUs) when running the lscpu command.

D Setup

D.1 Dockerfiles

D.1.1 CLTune

```
# CUDA version 10.2
FROM nvidia/cuda:10.2-devel-ubuntu18.04

WORKDIR /usr/src/bat

RUN apt-get update && apt-get install -y \
    git \
    cmake \
    python3

# Download and build CLTune
RUN cd /usr/local \
    && git clone https://github.com/ingunnsund/CLTune \
    && cd CLTune \
    && mkdir build \
    && cd build \
    && cmake -DUSE_OPENCL=OFF .. \
    && make install

# Copy content
COPY . .

# Set the environment variable so other sources can use CLTune
ENV CLTUNE_PATH=/usr/local/CLTune

# Set the correct encoding for Python
ENV PYTHONIOENCODING=utf-8
```

Listing 71: Dockerfile for CLTune.

D.1.2 KTT

```
# CUDA version 10.2
FROM nvidia/cuda:10.2-devel-ubuntu18.04

WORKDIR /usr/src/bat

RUN apt-get update && apt-get install -y \
    git \
    wget \
    python3

# Download premake5 (dependency of KTT)
RUN wget https://github.com/premake/premake-core/releases/download/v5.0.0-alpha15/
   premake-5.0.0-alpha15-linux.tar.gz \
    && tar -xzf premake-5.0.0-alpha15-linux.tar.gz \
    && rm premake-5.0.0-alpha15-linux.tar.gz \
    && mv premake5 /usr/bin
```

```
# Set CUDA path required by KTT
ENV CUDA_PATH=/usr/local/cuda-10.2/

# Add temporary linking path for building KTT
ARG TEMP_LD_LIBRARY_PATH=${LD_LIBRARY_PATH}
ENV LD_LIBRARY_PATH=${CUDA_PATH}lib64/stubs:${LD_LIBRARY_PATH}

# Create symbolic link for CUDA libraries to be used during building
# This is due to libraries being different in build and run phase of Docker (See
    issue: https://github.com/NVIDIA/nvidia-docker/issues/775)
# This is for linking to work on Docker build for CUDA libs required by KTT
RUN ln -s /usr/local/cuda/lib64/stubs/libcuda.so /usr/local/cuda/lib64/stubs/
    libcuda.so.1

# Download and build KTT
RUN cd /usr/local \
    && git clone https://github.com/Fillo7/KTT \
    && cd KTT \
    && premake5 gmake \
    && cd build \
    && make config=release_x86_64

# Copy content
COPY . .

# Remove the symbolic link for the CUDA libraries as it is not needed anymore
RUN rm /usr/local/cuda/lib64/stubs/libcuda.so.1

# Reset the LD_LIBRARY_PATH
ENV LD_LIBRARY_PATH=${TEMP_LD_LIBRARY_PATH}

# Set the environment variable so other sources can use KTT
ENV KTT_PATH=/usr/local/KTT

# Set the correct encoding for Python
ENV PYTHONIOENCODING=utf-8
```

Listing 72: Dockerfile for KTT.

D.1.3 Kernel Tuner

```
# CUDA version 10.2
FROM nvidia/cuda:10.2-devel-ubuntu18.04

WORKDIR /usr/src/bat

# Install dependencies
RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip

# Copy content
COPY . .

# Install dependencies for Kernel Tuner
RUN cd tuning_examples/kernel_tuner && \
    pip3 install -r requirements.txt
```

```
# Set the correct encoding for Python
ENV PYTHONIOENCODING=utf-8
```

Listing 73: Dockerfile for Kernel Tuner.

D.1.4 OpenTuner

```
# CUDA version 10.2
FROM nvidia/cuda:10.2-devel-ubuntu18.04

WORKDIR /usr/src/bat

# Install dependencies
RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip \
    openmpi-bin \
    openssh-client \
    libopenmpi-dev

# Copy content
COPY . .

# Install dependencies for OpenTuner
RUN cd tuning_examples/opentuner && \
    pip3 install -r requirements.txt

# Due to a bug in OpenMPI (https://github.com/open-mpi/ompi/issues/4948)
ENV OMPI_MCA_btl_vader_single_copy_mechanism=none

# Set the correct encoding for Python
ENV PYTHONIOENCODING=utf-8
```

Listing 74: Dockerfile for OpenTuner.

D.2 Slurm

```
srun -N1 -n1 -c40 -gres=gpu:T4:1 --partition=TDT4200 --time=01:00:00 \
    -w selbu --pty slurm_init
```

Listing 75: Slurm reserving of a single NVIDIA T4 and 40 CPU cores.

Knut Aasgaard Kirkhorn

BAT: A Benchmark Suite for Auto-Tuners

NTNU
Norwegian University of
Science and Technology