Master's thesis

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

Ingunn Sund

BAT: A Benchmark suite for AutoTuners

Development of BAT and Tuning on 20x Tesla T4 GPUs and More

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

NTNU
Norwegian University of
Science and Technology

Ingunn Sund

BAT: A Benchmark suite for AutoTuners

Development of BAT and Tuning on 20x Tesla T4 GPUs 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

NTNU
Norwegian University of
Science and Technology

Abstract

Autotuning solves the performance portability challenge when creating applications that will be run on different architectures. An autotuner is a program that takes a parameterized code as input and tries to find the best possible values for the tuning parameters defined. Currently, to our knowledge, there are no standardized benchmark suites for autotuners for comparing and testing. Developers of autotuners makes their own benchmarks when presenting and comparing autotuners.

As a possible solution to the challenge of benchmarking autotuners, we present **BAT: a Benchmark suite for AutoTuners**. This thesis describes the development of BAT and how to use BAT to evaluate known autotuners on different architectures. As part if this work, CUDA programs and kernels from "The Scalable Heterogeneous Computing (SHOC) Benchmark" were parameterized.

BAT is a benchmark suite with HPC based, parameterized algorithms in CUDA with GPU focus. It contains a varied selection of benchmarks of different complexity that can utilize multiple GPUs on one system, either by running the same program and computations on multiple nodes, or by splitting the work between nodes.

The benchmark suite is tested with four different autotuners that differs in setup and how they tune. These are OpenTuner, Kernel Tuner, CLTune and KTT. All the benchmarks are modified to suite a lot of different autotuners. A handy feature from BAT for testing is its CLI that makes it easier to run autotuning with the benchmarks.

BAT is a joint development with Knut Kirkhorn. The difference is that we parameterized our own separate algorithms and tested our algorithms on different multi-GPU systems. This thesis focuses on testing on an IBM Power System AC922 with four Tesla V100-SXM2 32 GB GPUs and a Server with 20 Tesla T4 GPUs.

Sammendrag

Autotuning løser ytelsesportabilitetsutfordringen når man lager applikasjoner som skal kjøres på forskjellige arkitekturer. En autotuner er et program som tar en parameterisert kode som input og prøver å finne de best mulige verdiene for et sett med parametrene. For øyeblikket, så vidt vi vet, er det ingen standardiserte "benchmark suites" for autotunere for sammenligning og testing. Utviklere av autotunere lager egne "benchmark suites" når de presenterer og sammenligner autotunere.

Som en mulig løsning på utfordringen med "benchmarking" av autotunere, presenterer vi **BAT: a Benchmark suite for AutoTuners**. Denne oppgaven beskriver utviklingen av BAT og hvordan man bruker BAT til å evaluere kjente autotunere på forskjellige arkitekturer. Som en del av dette, ble CUDA-programmer og "kernels" fra "The Scalable Heterogeneous Computing (SHOC) Benchmark" parametrisert.

BAT er en "benchmark suite" med HPC-baserte, parametrerte algoritmer i CUDA med GPU-fokus. Den inneholder et variert utvalg av "benchmarks" med forskjellig kompleksitet som kan bruke flere GPUer på ett system, enten ved å kjøre det samme programmet og beregninger på alle noder, eller ved å dele arbeidet mellom noder.

"Benchmark suiten" er testet med fire forskjellige autotunere som er forskjellige i oppsett og hvordan de "tuner". Disse er OpenTuner, Kernel Tuner, CLTune og KTT. Alle "benchmarkene" er modifisert slik at de passer til mange forskjellige autotunere. En praktisk funksjon fra BAT for testing er kommandolinjegrensesnittet som gjør det lettere å kjøre autotuning med "benchmarkene".

BAT er et felles utviklingsprosjekt med Knut Kirkhorn. Forskjellen er at vi parametrerte våre egne separate algoritmer og testet algoritmene våre på forskjellige multi-GPU-systemer. Denne oppgaven fokuserer på testing på en IBM Power System AC922 med fire Tesla V100-SXM2 32 GB GPUer og en server med 20 Tesla T4 GPUer.

Acknowledgments

I would like to give a huge thanks to my supervisor, Professor Anne C. Elster for her support and helpful insights.

I would also like to thank my collaborator, Knut Kirkhorn, for a great partnership and collaboration on this project.

PhD student and HPC-Lab Admin Jacob O. Tørring provided invaluable system support and helped suggest the topic of this thesis, and Rolf Harald Dahl, IT support at our Dept (IDI), was very responsive and helpful regarding updating the IBM Power System AC922 server used, which really facilitated this work.

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

List of Tables

List of Listings

List of Abbreviations

Table 1: Abbreviations and explanations.

Abbreviation	**Explanation**
AI	Artificial Intelligence
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
BAT	Benchmark suite for AutoTuners
BFS	Breadth First Search
FLOPS	Floating Point Operations Per Second
GPU	Graphics Processing Unit
HPC	High Performance Computing
MD5	Message-Digest algorithm 5
MPI	Message Passing Interface
OpenCL	Open Computing Language
PCIe	Peripheral Component Interconnect Express
RT	Ray Tracing
SM	Streaming Multiprocessor
SpMV	Sparse Matrix-Vector multiplication
TC	Tensor Cores

Chapter 1

Introduction

The ever-growing interest for faster computers and High Performance Computing in the recent years has led to a great deal of research and progress in this field. The GPU, which were originally created for computer graphics, is today designed to be more focused on computations in HPC and AI applications.

A challenge when creating applications that will be run on different architectures, is that the optimal program for one architecture might not be optimal for another architecture. This issue of performance portability is becoming more important as systems are becoming more heterogeneous. Autotuning can help programs with this.

Autotuning

An autotuner is a program that takes a parameterized code as input and tries to find the best possible values for the tuning parameters defined. The problem autotuners are made to solve is that the optimal parameters might be different for different devices and architectures, and the optimal parameters might not be possible to find in a reasonable time frame.

A set of optimal values for one architecture might not be optimal for others. If the same code should run efficient on a CPU and a GPU, the optimal parameter values might be different. The best values can also be dependent on the input format or the input size. It is not efficient to manually optimize the code for different systems.

All types of code can be autotuned, but some autotuners are specific to only tuning code for one language or style. Some autotuners can tune a full program and others will only tune a GPU kernel.

Autotuners will often use different search techniques to find the best parameter values. Examples of this is brute force, genetic algorithms and annealing search.

The autotuners will also often support parameter constraints, which is where the tuning space of one or more parameters is restricted based on the value of one or more parameters. [1]

A variant of the autotuning problem is hyperparameter optimization, which is optimization of hyperparameters in a machine learning algorithm. [2]

Benchmarking

In the computing field, benchmarking is the process of running tests to compare and analyze the performance of software and hardware. Benchmarking is often performed to reveal strengths and weaknesses in systems by comparing machines running the same program. It can even be used to identify faults with software or hardware. Benchmarks can also be a useful tool for finding out if a program will run differently on different architectures and in different scenarios.

If the goal is to measure and compare performance of software on different machines, it is preferable to have the same environment for the platforms that is used during the benchmarking. [3]

For almost every type of framework or program, it can be beneficial to compare them. This is also true for autotuners, and is the challenge addressed in this thesis. Currently, to our knowledge, there are no standardized benchmark suites for autotuners. Developers of autotuners makes their own benchmarks when presenting and comparing autotuners. However, it is not efficient for the developers to make tunable benchmarks every time an autotuner is made. This can also lead to issues with the benchmarks being tailored for the autotuners.

1.1 Thesis Goals

As a possible solution to the challenge of benchmarking autotuners, we present **BAT: a Benchmark suite for AutoTuners**.

This project was done in collaboration with Knut A. Kirkhorn, another master student from the NTNU HPC-lab, also supervised by Professor Elster.

This thesis describes how we developed BAT and how to use BAT to evaluate known autotuners on different architectures. As part if this work, we parameterized CUDA programs and kernels. We based our work on The Scalable Heterogeneous Computing (SHOC) Benchmark [4] - a known and much used benchmark suite. This led to an analysis of how successful it was to to base BAT on SHOC.

This thesis also describes testing of BAT. The benchmark suite was tested with different autotuners to ensure that BAT would work with different types of autotuners. During testing, the parameter values were compared on different systems. Kirkhorn and I collaborated on making the benchmark suite, but parameterized our own separate algorithms, tested our own finished algorithms, and did our own separate analysis for those algorithms.

With BAT we have laid the foundation for a benchmark suite for autotuners that is easy to extend to eventually possibly become a standardized benchmark suite.

Our hypothesis is that a benchmark suite for autotuners based on SHOC will have the potential to eventually become an ideal benchmark suite for autotuners. Some questions related to the hypothesis I would like to answer in this thesis are:

- Is SHOC is a good benchmark suite to base the benchmark suite on?
- Will there be a lot of rewriting of the code to ensure that the benchmark is enough parameterized?

- Will it be able to have a GPU and multi GPU focus?
- Will the benchmark suite work with different types of autotuners?
- Will the parameters added during the parameterization have different optimal values for different systems?

1.2 Contributions

The following list are the main contributions from this thesis.

- A set of requirements for how an ideal benchmark suite for autotuners should be.
- A benchmark suite (BAT) with different parameterized CUDA benchmarks with HPC focus. BAT has compatibility for many autotuners, is easy to use and well documented.
- Tested the benchmark suite with four different autotuners on multiple single and multi-GPU systems. This was done to make sure that everything works. Results from testing BAT are also discussed, and some parameters are evaluated.
- Evaluation of the known autotuners OpenTuner, Kernel Tuner, CLTune and KTT. This answers questions like what is missing and what is their best qualities. This analysis could be helpful for someone making their own autotuner.

1.3 Outline

The rest of this thesis consists of the following chapters:

- **Chapter 2. Background** gives an introduction to the different topics needed to understand the work from this thesis.
- **Chapter 3. SHOC, Selected Autotuners and Test Benches** describes the SHOC benchmark suite, the algorithms used for parameterization, and the systems used for testing the finished benchmark suite.
- **Chapter 4. Related Work** contains related work to this thesis.
- **Chapter 5. Plan for the Benchmark Suite** describes the motivation for choosing SHOC, how Kirkhorn and I split the work and what should be included in a benchmark suite.
- **Chapter 6. Making the Benchmark Suite** contains the process of making the benchmark suite. This includes parameterization of the benchmarks and how we made BAT user friendly.

- **Chapter 7. Testing the Benchmark Suite** includes a description of how the benchmarks were combined with the autotuners. The different systems used for testing are described. This chapter also includes a description of what to test and why, and how to run tests.
- **Chapter 8. Experiments and Discussion** describes and discusses the results from running the benchmarks. It also contains evaluation of the autotuners and BAT.
- **Chapter 9. Conclusion and Future Work** includes a summary of the results and important parts of the thesis. This section also addresses possible future work.

The appendices are:

- **Appendix A. Parameter Research** has lists of parameters used in different parameterized programs.
- **Appendix B. BAT User Guide** includes a copy of BAT's `README.md` documentation file that consists of a description of BAT and how to use the project.
- **Appendix C. System Information** contains command line output information from the different machines used in the experiments. This is system information about CPU, GPU and interconnects.
- **Appendix D. Setup** has a setup guide for setting up the autotuning benchmark tests.

The following items are also attached with the delivery of the thesis:

1. **BAT** is the source code for BAT, including the part that Knut Kirkhorn did.
2. **BAT-results** includes the test results from running the benchmarking, also including results Kirkhorn collected.
3. **Investigating New GPU Features for Performance** is the specialization project report by Knut Kirkhorn and I. This was a project done as preliminary work for this thesis.

Chapter 2

Background

This chapter gives an overview of GPUs and how to communicate between GPU and CPU, as well as a brief description of Docker, the virtualization framework used when benchmarking.

2.1 The Graphics Processing Unit

Graphics Processing Units (GPUs) are specialized units in computers optimized for performing operations on data in parallel. They were originally created for improving graphics for video games and other graphic heavy applications. In recent years they have also been incredibly useful for computations in High Performance Computing (HPC) and Artificial Intelligence (AI) applications. [5]

GPUs are great at performing the same type of small operation fast and multiple times in parallel. This stands in contrast to the Central Processing Unit (CPU), which specializes in serial computing and processes a sequence of operations very well. [6] In Figure 2.1, the architectural differences are shown. The GPU has more cores for data processing, and the CPU has more components for cache and control.

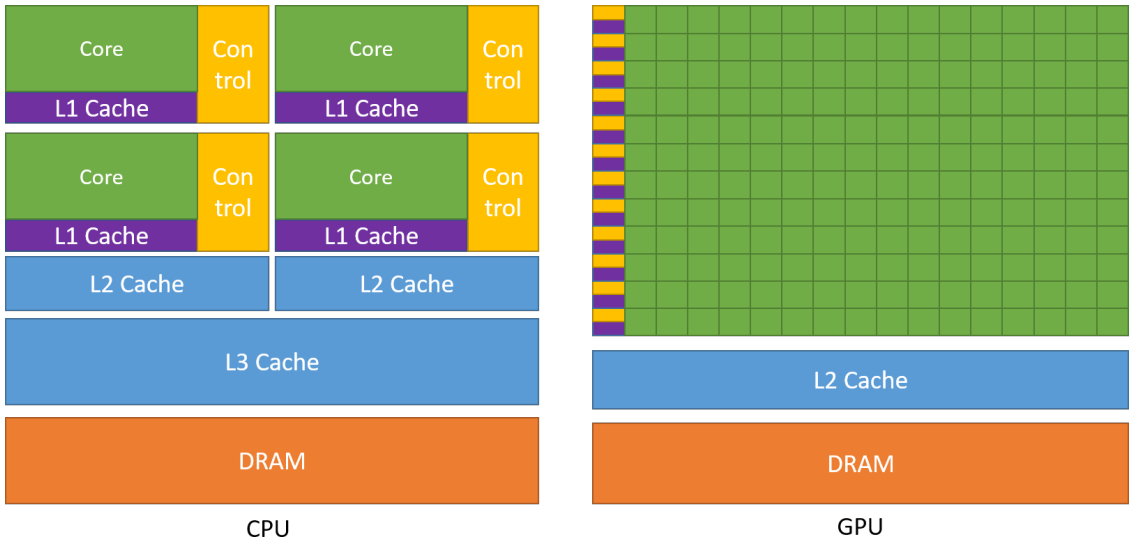

Figure 2.1: Example of architecture differences on CPUs and GPUs [6]. Figure used with permission from NVIDIA.

GPU Programming

There are multiple ways to develop programs that are able run on the GPU. Two ways are with CUDA and OpenCL.

CUDA

CUDA (Compute Unified Device Architecture) is a platform and programming model for parallel computing developed by NVIDIA. CUDA makes it possible to directly use NVIDIA GPUs in programs with programming languages like C, C++, Fortran and others. [6].

With C++ CUDA it is possible to define and launch CUDA kernels that are executed with threads in parallel on the GPU. A kernel needs a `__device__` or `__global__` declaration specifier to be executed on the GPU. To set how many threads to be used, a special syntax is used when the kernel is called that sets the number of blocks (grid size) and threads per block (block size): `<<<blocks, threads>>>`. The total thread amount will be the grid size multiplied with the block size. This syntax for setting blocks and threads per block can also be set for multiple dimensions (x, y, z). [6] Figure 2.2 shows an example of how the blocks can be divided for processing on different GPU architectures according to the number of streaming multiprocessors available.

It is generally common to set threads per block (block size) as a multiple of the warp size, 32, for best performance. The threads in a block are divided into warps that executes the same instruction. [7]

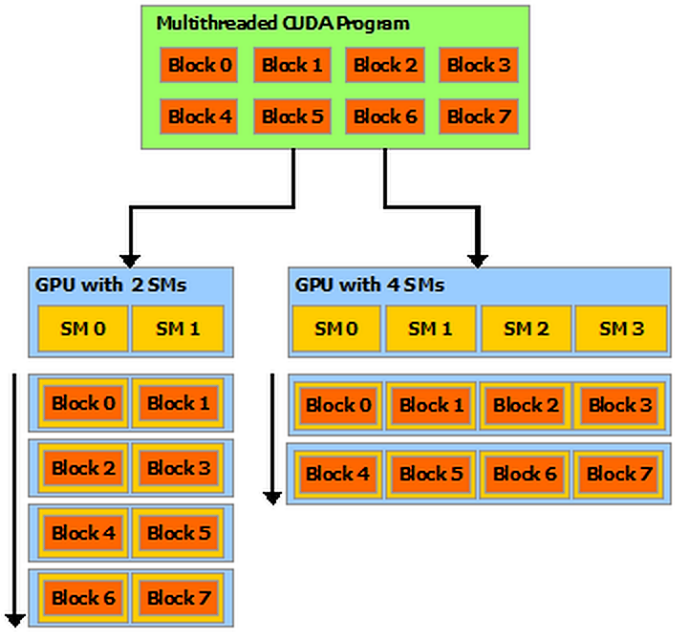

Figure 2.2: Example of how the blocks can be scheduled to run on the SMs when the number of SMs is different on different architectures [6]. Figure used with permission from NVIDIA.

Listing 2.1 shows an example of a kernel that is launched with N blocks with one thread per block. For this kernel, the thread ID will be equal to the block ID, which can be found with the `blockIdx.x` variable. If this example would have been launched with more threads per block, the thread ID would have to be calculated like this: `int threadId = blockIdx.x * blockDim.x + threadIdx.x`.

```
// Kernel
__global__ void add(int *a, int *b, int *c) {
    int threadId = blockIdx.x;
    c[threadId] = a[threadId] + b[threadId];
}

int main() {
    ...
    // Launches the kernel with N blocks with 1 thread per block
    add<<<N,1>>>(a, b, c);
    ...
}
```

Listing 2.1: CUDA kernel example.

OpenCL

The OpenCL framework works similarly to CUDA, which means it can be used for writing programs with kernels that can run on GPUs. OpenCL can also launch kernels with setting

grid and block size, but the terms are different, as can be seen in Table 2.1. [8] The CUDA terms will be used through this thesis as standard.

Table 2.1: Terms in CUDA and OpenCL. [8]

CUDA	**OpenCL**
Streaming multiprocessor	Compute unit
Thread	Work-item
Block	Work-group
Grid	N-D range
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

Optimization Techniques

Setting optimal values for grid and block size can help reduce execution times for a kernel. Other techniques for optimizing GPU code can be loop unrolling. Unrolling a loop means writing out what happens in every iteration and eliminating the loop. This optimization can sometimes reduce execution times.

Choosing a memory type like shared memory or texture memory instead of global memory can also potentially affect the time used for the program.

There are many different compiler options available for compilers like G++, GCC and NVCC that can be useful when optimizing execution time. [6]

2.2 GPU and CPU Communication

When combining several systems, either server, workstations or several GPU, one can use MPI. The following describes MPI i a bit more detail as well as PCI Express, NVLink and NVSwitch the three types of communication links available on the NVIDIA GPUs benchmarked in this thesis.

MPI

This section about MPI is taken from my specialization project which can be found as an attachment to the 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 for communication between the nodes. [9] There exist many different implementations of MPI, such as Open MPI [10], Spectrum MPI [11] and MPICH [12].

2.2.1 PCI Express

This section about PCI Express is taken from my specialization project which can be found as an attachment to the 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 connection between the GPU and CPU is normally 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. [13] [14]

2.2.2 NVLink 2.0 and NVSwitch

This section about NVLink and NVSwitch is taken from my spezialisation project which can be found as an attachment to the thesis.

NVIDIA NVLink is a GPU interconnect which offers much faster data transfer and is more scalable than using the PCIe. [15] NVLink can be used for both GPU to GPU and CPU to GPU connection. Each lane in the NVLink has a transfer rate of 25 GT/s. [16, 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. [17, p. 3]

NVIDIA DGX-2 is a system that uses NVSwitch between GPUs, this is illustrated in Figure 2.3 below.

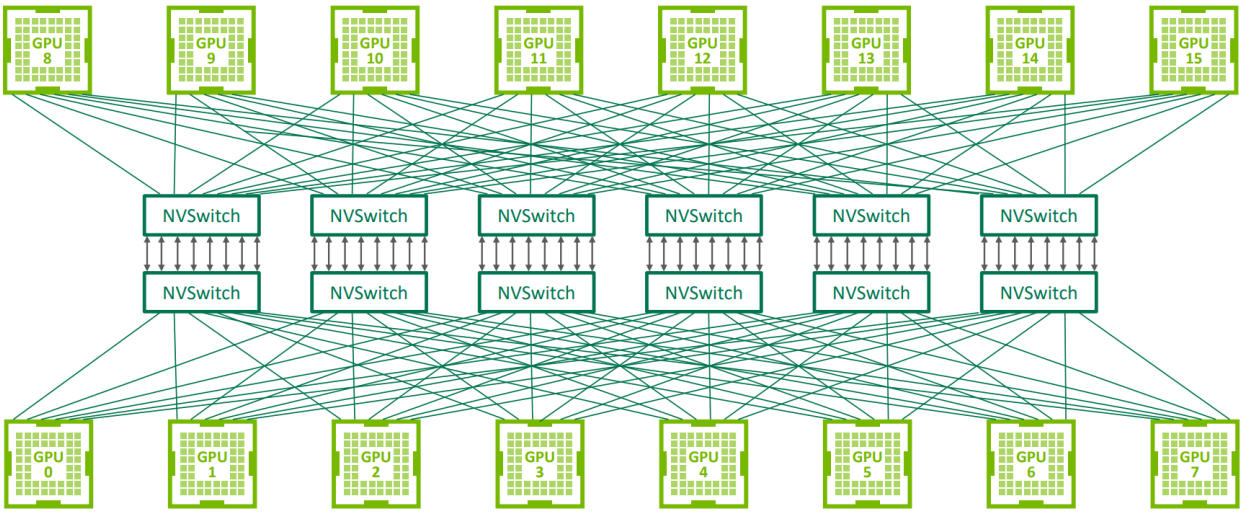

Figure 2.3: NVSwitch topology on DGX-2 [18, p. 8]. Figure used with permission from NVIDIA.

2.3 Docker

Docker is a platform that makes it possible to have a relatively isolated environment for programs to run in. This is helpful for making results easier to reproduce and it will ensure that the program has the same dependencies on every computer it runs on. Another positive trait from Docker is that it protects the host machine by not requiring to permanently download and switch between versions of dependencies. [19]

Docker containers are runnable instances of Docker images which are the results from building a Dockerfile. A Dockerfile is a special file that contains all the setup and dependencies for creating the environment. Figure 2.4 illustrates the relationship between the Docker components.

A part of Docker is the Docker Engine that hosts containers. There is also a client that sends requests to the Docker Engine to, among other things, build and run containers. Something that makes using Docker very practical in many cases is that it is possible to run multiple containers on the same machine at the same time. [20]

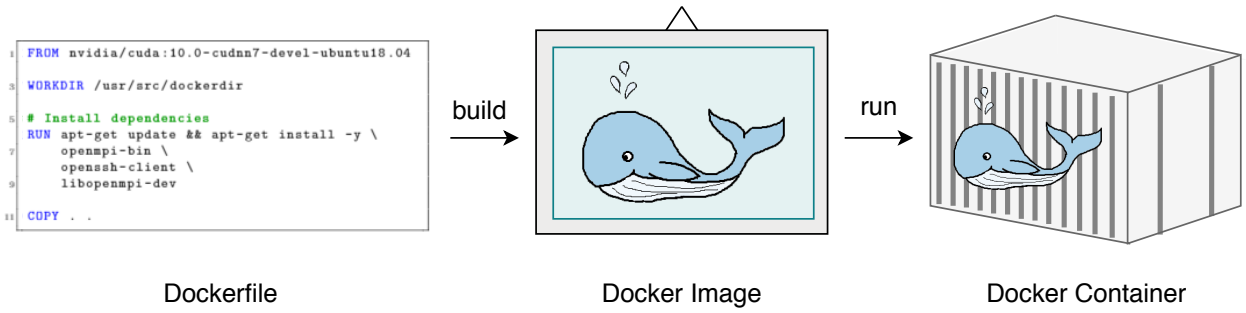

Figure 2.4: Relationship between Docker components. Created with inspiration from a figure from a Docker blog post [21].

For some applications it is necessary to use GPU acceleration. For this purpose, an extension of Docker called NVIDIA Docker can be used to utilize NVIDIA GPUs in containers. [22]

Chapter 3

SHOC, Selected Autotuners and Test Benches

This chapter gives an overview of the benchmarking suite SHOC, a description of the autotuners used in this thesis. The systems used for testing are also described.

3.1 SHOC Benchmark Suite and Algorithms

The Scalable HeterOgeneous Computing (SHOC) benchmark suite is a collection of benchmarks made to be used on single and multi GPU systems. The benchmarks consist of standard HPC algorithms with both CUDA and OpenCL versions. Most of the benchmarks can be run both serial and parallel. The parallel version runs on multiple nodes or devices with MPI. SHOC offers two different ways of running benchmarks parallel, Embarrassingly Parallel (EP) and True Parallel (TP). When the benchmark is EP, the same benchmark is run on all nodes without communication or collaboration. When TP is activated, the task is split between the different nodes and they collaborate to find the solution.

The SHOC benchmarks are divided into three levels. Level 0 benchmarks are focused on measuring low level performance like the performance of the bus between CPU and GPU. Level 1 benchmarks consist of common parallel algorithms often used in bigger applications. Level 2 has benchmarks for real application kernels. [4] [23]

The following level 1 algorithms from SHOC are some algorithms especially used in this thesis.

3.1.1 BFS

The breadth first search (BFS) algorithm is a search algorithm used on trees or graphs. It works by traversing one depth from the root node before it moves on to the next depth. [24] The BFS version used in SHOC performs search on an undirected k-way tree, which is a tree where each node has at most `k` children. [25]

An example of BFS on such graph can be seen in Figure 3.1. The dashed arrow symbolizes the traversal path: `0-1-2-3-4-5-6-7-8-9`. The algorithm will traverse all nodes on Level 0 before it explores Level 1, Level 2 and finally Level 3.

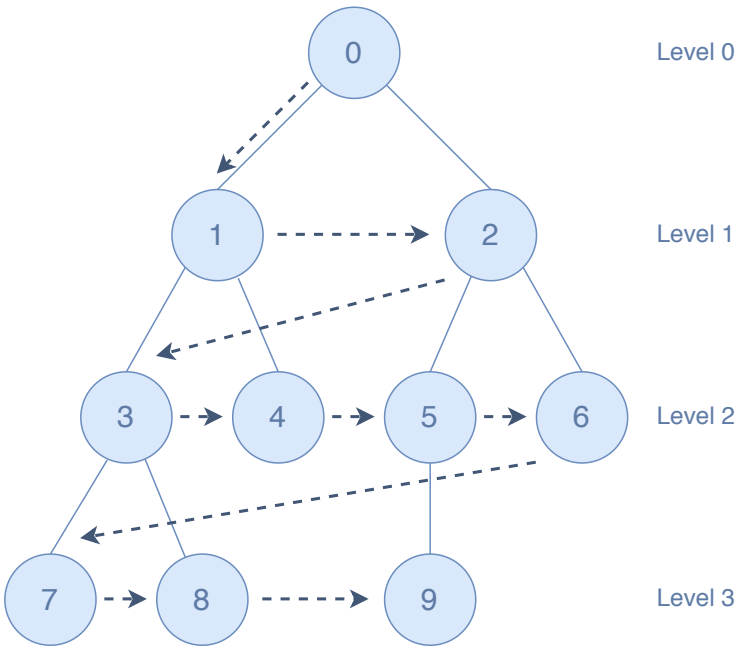

Figure 3.1: An undirected tree where the dashed arrow shows the traversal path when using the BFS algorithm.

SHOC's version of BFS measures performance when the algorithm is used on a random graph, and the number of graph vertices can be chosen as 1000, 10000, 100000, or 1000000 when running the benchmark. The algorithm can be run on multiple GPUs, but only in the embarrassingly parallel mode. [26]

3.1.2 SpMV

Sparse Matrix-Vector multiplication (SpMV) is where a sparse matrix and a dense vector is multiplied with a dense vector as result. A sparse matrix or vector signify a matrix or vector where most values are zero, the opposite of this is a dense matrix or vector. Figure 3.2 shows an example of SpMV on the $Ax = y$ format where the colored elements symbolizes non-zero values.

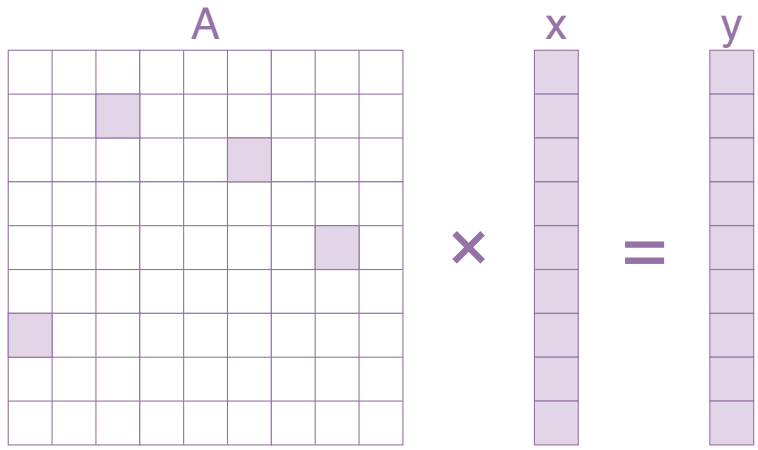

Figure 3.2: Example of a Sparse Matrix-Vector multiplication (SpMV).

SHOC's implementation of SpMV has benchmarks for both Compressed Sparse Row (CSR) and ELLPACK-R, two formats for storing sparse matrices when performing SpMV. There are also benchmarks for both normal and padded data for CSR.

The difference between CSR and ELLPACK-R can be seen in the following example. A is a sparse matrix where only non-zero values are visible. This example matrix has more non-zero values than it would in SHOC for a more intuitive example.

$$A = \begin{bmatrix} & 7 & 8 & \\ 1 & & & \\ & 1 & & 5 \\ 5 & & & \end{bmatrix}$$

When storing the matrix in CSR format, three arrays are stored. A data array stores the non-zero values in the order by going row by row in the matrix. An indices array stores the column numbers for the non-zero values. The last array is a pointer array that points to what the data index the different rows starts with. [27] The CSR format for matrix A can be seen below:

$$ptr = \begin{bmatrix} 0 & 2 & 3 & 5 & 6 \end{bmatrix} \quad indices = \begin{bmatrix} 1 & 2 & 0 & 1 & 3 & 0 \end{bmatrix} \quad data = \begin{bmatrix} 7 & 8 & 1 & 1 & 5 & 5 \end{bmatrix}$$

The ELLPACK-R format stores the matrix in three small matrices. A data matrix stores the non-zero values with padding at the end. The padding is noted with * in the example below. The next matrix stores the column number for all the values. The last matrix holds the number of non-zero values of each row. [28] An example of the ELLPACK-R format can be seen below for matrix A.

$$data = \begin{bmatrix} 7 & 8 \\ 1 & * \\ 1 & 5 \\ 5 & * \end{bmatrix} \quad col = \begin{bmatrix} 1 & 2 \\ 0 & * \\ 1 & 3 \\ 0 & * \end{bmatrix} \quad row\_length = \begin{bmatrix} 2 \\ 1 \\ 2 \\ 1 \end{bmatrix}$$

SHOC's implementation of the SpMV algorithm measures performance in both single and double precision. It is also possible to run SpMV embarrassingly parallel. For the CSR format, performance is measured for both a vector and a scalar version. This means that for the scalar version, one thread is used for calculations per row, but for the vector version, a warp is used per row. [29]

3.1.3 MD5 Hash

Message-Digest algorithm 5 (MD5) is a hashing algorithm that produces a 128-bit hash value. A hash algorithm is a one-way function that takes an input of arbitrary length and outputs a hash of a fixed length. To be considered a secure algorithm, the following requirements must be fulfilled:

Pre-Image Resistance: It should be computational infeasible to find the message from the hash.

Second Pre-Image Resistance (Weak Collision Resistance): Given a message x, It should be computational infeasible to find a message y that produces the same hash $(hash(x) = hash(y))$.

Strong Collision Resistance It should be computational infeasible to find any different input messages (x and y) that produces the same hash $(hash(x) = hash(y))$. [30]

Figure 3.3 shows an illustration of how a hash function works, with an example of a collision where two hashed messages produces the same input.

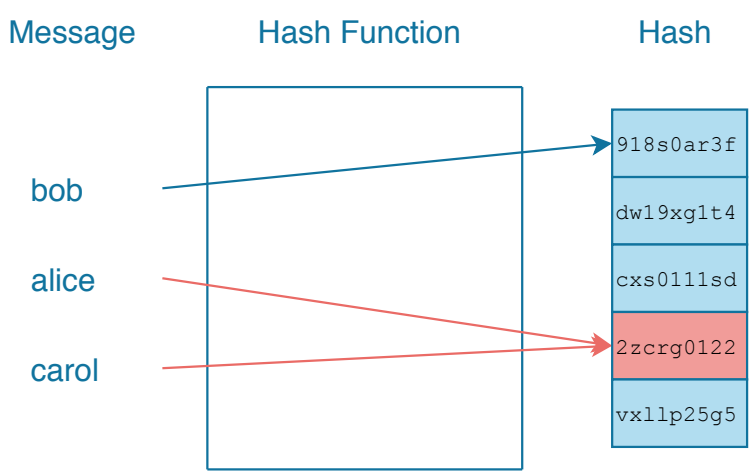

Figure 3.3: Illustration of how a hashing algorithm works with a collision.

MD5 was originally designed for hashing passwords. The algorithm has later been proven to be insecure for usage in cryptography because of collision vulnerability, but MD5 is still widely used, insecurely, for password hashing and for data integrity as an algorithm in checksums. [31]

The algorithm works by first padding the message to a bit length of 64 bits less than being divisible by 512. The length of the input message is then represented in 64 bits and appended to the padded message. The next step is to initialize the MD buffer which consists of four 32-bit words. After the initialization, the message can be processed in 16-bit word

blocks through stages called rounds with different operations. The processing result is a 128-bit hash value. [32]

In SHOC, the MD5 Hash algorithm have an option to choose between two types of round styles. The MD5 algorithm can be run on multiple GPUs as EP (embarrassingly parallel). [33]

3.1.4 Scan

The Scan algorithm, also called Parallel Prefix Sum algorithm, is an algorithm that computes the sum of the prefixes for each number in a sequence. This algorithm returns a sequence of sums that is the same length as the input sequence. Figure 3.4 shows an example of a naive parallel scan, where we can see that the output sequence for each index has the sum of the values up to, and including, the current value. [34]

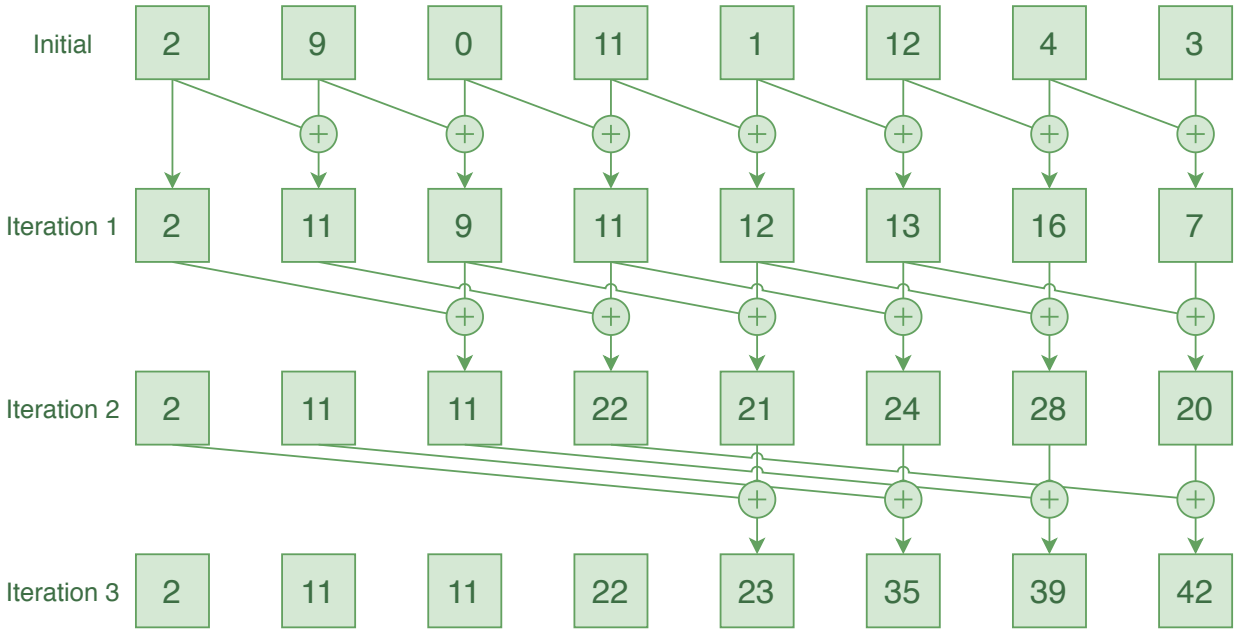

Figure 3.4: An example of a naive parallel scan.

SHOC's parallel implementation of the Scan algorithm works a bit different than the naive parallel version from Figure 3.4, but it is the same concept. SHOC's algorithm includes calculation for both single and double precision input sequences. It also has an implementation for true parallel (TP) computation that can run on multiple nodes. [35]

3.1.5 Stencil 2D

SHOC's Stencil 2D algorithm performs a 2D 9-point stencil computation. A stencil computation updates grid elements according to a pattern using neighboring elements. [36] SHOC have implemented a true parallel (TP) version of this algorithm that distributes work between multiple nodes. [37]

3.2 Selected Autotuners

The following sections contains descriptions of autotuner frameworks used in this thesis.

3.2.1 OpenTuner

OpenTuner is a framework for building program autotuners described by J. Ansel et al. in "OpenTuner: An extensible framework for program autotuning" [38]. The framework is written in Python and supports autotuning of programs written in different programming languages. Any compile or run commands needs to be set in the tuning file, which is also a Python file. In the tuning file, the parameters and their search space must be defined. The parameters are grouped in primitive parameters, with lower and upper bounds, and complex parameters for parameters that are not gradual.

Table 3.1 lists the search techniques, where the AUC Bandit Meta Technique is the default search technique. OpenTuner also has support for adding new search techniques. [38]

Table 3.1: Search techniques in OpenTuner.

Search Strategy	**Explanation**
Pure Random	Random search.
Nelder Mead	Nelder Mead search with variants "random", "regular", "right" and "multi".
Torczon	Torczon search with variants "random", "regular", "right" and "multi".
AUC Bandit Meta Technique	Search technique that combines differential evolution, greedy mutation and hill climbing by using AUC Bandit Meta technique.
AUC Bandit Mutation Technique	Mutation version of the AUC Bandit Meta technique.
Greedy Mutation	"Uniform" or "normal" greedy mutation search.
Differential Evolution	"Normal" or "composable" differential evolution.
Genetic algorithm	Genetic algorithm search.
Particle Swarm Optimization	Some different versions of particle swarm optimization.
Pattern Search	Pattern search.
Pseudo Annealing Search	Pseudo annealing search.
Grouping Genetic Algorithm	Grouping genetic algorithm search.

3.2.2 Kernel Tuner

Kernel Tuner is an autotuner presented by Ben van Werkhoven in a paper called "Kernel Tuner: A search-optimizing GPU code auto-tuner" [39]. This autotuner is written in Python and requires a python tuning file for preparing the autotuning. Kernel Tuner can tune both CUDA and OpenCL kernels with and without host code. To verify that a tuned kernel produces correct results, there is an option to add a list of correct results that Kernel Tuner can use for correction verification.

Kernel Tuner uses brute force as the default search technique, but it is possible to choose from nine additional techniques described in Table 3.2. [40]

Table 3.2: Search techniques in Kernel Tuner.

Search Strategy	**Explanation**
Brute Force	Runs tuning for every possible combination of values in the search space.
Random Sample	Runs tuning with values from a random fraction of the search space.
Minimize	Search technique that limits the search with minimizers.
Basinhopping	Search technique that limits the search with minimizers.
Differential Evolution	Differential Evolution search.
Genetic Algorithm	Genetic algorithm search with default population size of 20.
Particle Swarm Optimization	Particle Swarm Optimization with default swarm size of 20.
Firefly Algorithm	Firefly algorithm with default 20 fireflies.
Simulated Annealing	Simulated annealing search.
Bayesian Optimization	Bayesian optimization.

3.2.3 CLTune

CLTune is an autotuner for tuning CUDA and OpenCL kernels described in "CLTune: A Generic Auto-Tuner for OpenCL Kernels" by C. Nugteren et al. [41] The autotuner is written in C++, and needs a C++ file for setting up the autotuner and providing information to the kernel. To check for correctness, it is possible to provide a reference kernel with input that is guaranteed correct where the output will be compared with output from kernels during tuning.

Table 3.3 lists the search techniques in CLTune, where full search is the default. [42]

Table 3.3: Search techniques in CLTune.

Search Strategy	Explanation
Full search	Runs tuning for every possible combination of values in the search space.
Random search	Random search with a fraction of the search space as input.
Annealing	Simulated Annealing search where fraction of search space and max temperature need to be set.
Particle Swarm Optimization	Particle Swarm Optimization where fraction of search space and swarm size needs to be set.

3.2.4 KTT

Kernel Tuning Toolkit (KTT) is an autotuner that focuses on autotuning CUDA and CLTune kernels. It is described in "A Benchmark Set of Highly-efficient CUDA and OpenCL Kernels and its Dynamic Autotuning with Kernel Tuning Toolkit" by F. Petrovič et al. [1] KTT is based on CLTune, and the main part used from CLTune is the Annealing search, the generation of kernel configuration and tuning parameter restrictions. To check the kernel results for correctness, KTT needs a reference kernel, like CLTune also requires.

Table 3.4 shows the search techniques that can be used in KTT, where brute force is default. [43]

Table 3.4: Search techniques in KTT.

Search Strategy	Explanation
Brute Force (Full Search)	Runs tuning for every possible combination of values in the search space.
Random Search	Random search.
Simulated Annealing	Simulated annealing. This is the exact algorithm as in CLTune. Maximum temperature has to be specified.
MCMC	Markov Chain Monte Carlo method.

3.3 GPUs Selected for Benchmarks

In this section the GPUs used in this thesis are described.

3.3.1 NVIDIA GeForce GTX 980

This section about the NVIDIA GeForce GTX 980 is taken from my specialization project which can be found as an attachment to the 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. [44]

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. [45]

3.3.2 NVIDIA Tesla V100

This section about the NVIDIA Tesla V100 is taken from my specialization project which can be found as an attachment to the 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. [46, 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. [47]

3.3.3 NVIDIA Titan RTX

This section about the NVIDIA Titan RTX is taken from my specialization project which can be found as an attachment to the 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. [48]

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 the card that came with the Turing architecture is Ray Tracing (RT) cores. RT Cores provides more realistic 3D rendering. [49, p. 4]

3.3.4 NVIDIA Tesla T4

The NVIDIA Tesla T4 is, like the Titan RTX, also a graphics card based on the Turing architecture. It has 16 GB of GDDR6 memory that has 300 GB/s bandwidth. This graphic card has 320 Turing Tensor Cores and 2560 CUDA Cores. Similarly, to the Titan RTX, the Tesla T4 also has Ray Tracing cores. [50]

3.4 Selected Multi GPU Systems

This section describes multi-GPU systems used in this thesis.

3.4.1 IBM Power System AC922

This section about the IBM Power System AC922 is taken from my specialization project which can be found as an attachment to the 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 mostly 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. [51] [52]

The Power AC922 supports up to 4 or 6 NVIDIA TeslaV100 GPUs depending on the model, where the GPUs can have 16GB or 32GB memory. [46, 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 3.5. 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 GPU and CPU 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 3.5), 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. [46, p. 12-15]

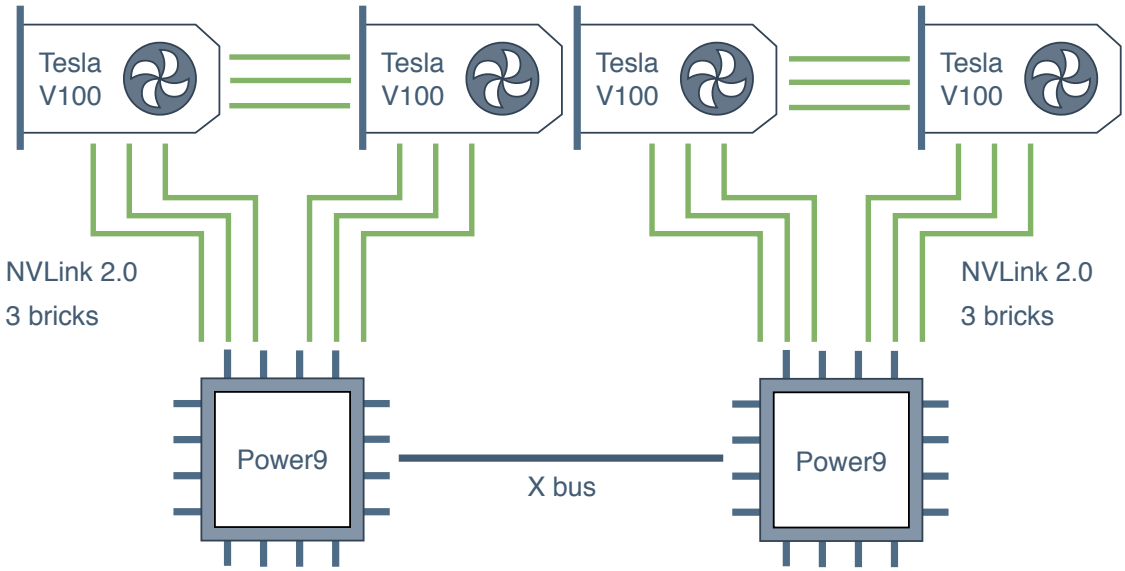

Figure 3.5: Interconnect diagram for IMB Power System AC922 with four GPUs. Figure made in collaboration with Knut Kirkhorn.

3.4.2 NVIDIA DGX-2

This section about NVIDIA DGX-2 is taken from my specialization project which can be found as an attachment to the thesis.

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. [53] 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 2.3. 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.

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 3.6. It can achieve the maximum performance for deep learning applications of 2 petaFLOPS which means that this system can be well suited for large workloads. [54]

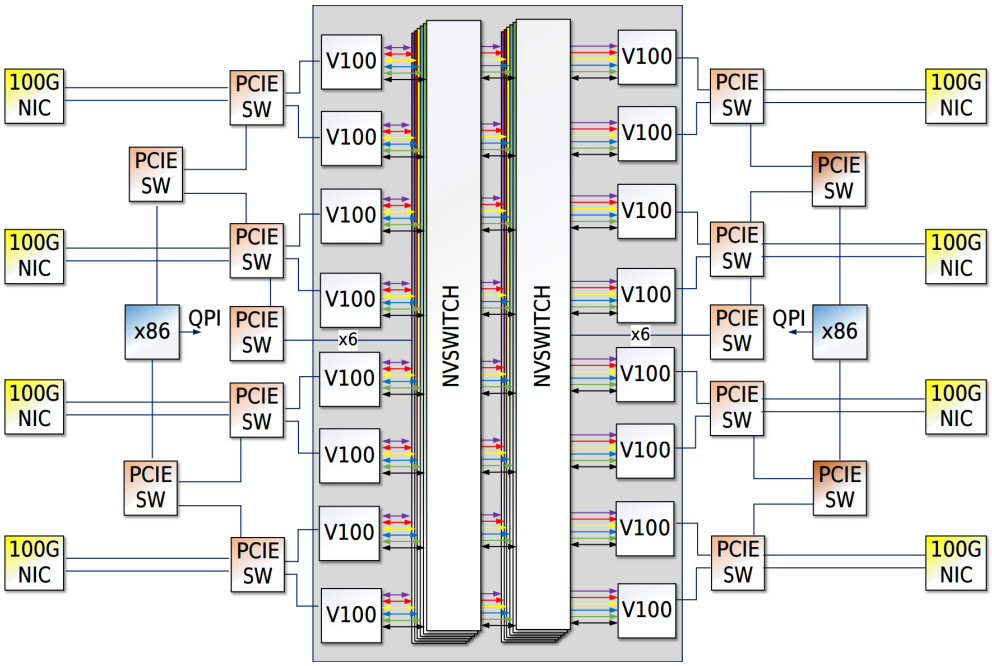

Figure 3.6: Interconnect diagram for DGX-2 [54, p. 19]. Figure used with permission from NVIDIA.

Chapter 4

Related Work

The part about my specialization project is taken and partly rewritten from the abstract from my specialization project which can be found as an attachment to the thesis.

"Investigating New GPU Features for Performance" is the specialization project report by Knut Kirkhorn and I done as preliminary work for the master thesis. This project consisted of a report that compared different GPUs and multi-GPU systems to evaluate performance for new hardware features. Some of these features are Tensor Cores, NVLink and NVSwitch. Multi-GPU systems with special interconnect configurations were benchmarked and compared. The purpose of this evaluation was to find out which systems or GPUs could be good for which tasks.

The systems and GPUs that were benchmarked was NVIDIA DGX-2 and two versions of the IBM Power System AC922, GeForce GTX 980 and Titan RTX. The benchmarking was done with the benchmark suites SHOC, DeepBench, Tartan and Scope.

The results from the benchmarking were among other things 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 would therefore depend on what kind of application 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 best possible performance, depending on the application. An interesting result for the DGX-2 was that there was no significant difference in the performance for the GPU-GPU communication over NVSwitches for any GPU combination.

Even though there is no standardized, easy to use benchmark suite for testing autotuners, most autotuners have various parameterized code examples for testing said autotuner.

OpenTuner includes several different examples with a very high search space. A lot of the examples are not possible to brute force in our lifetime with the technology we have today.

Kernel Tuner also includes a set of parameterized examples, but these have a much lower search space than the examples from OpenTuner. Most of these examples can be brute forced in some minutes to an hour. CLTune's example set is similar to the set in Kernel Tuner.

KTT advertises a benchmark set in the paper "A Benchmark Set of Highly-efficient CUDA and OpenCL Kernels and its Dynamic Autotuning with Kernel Tuning Toolkit"

by F. Petrovič et al. [1]. The kernels are highly-efficient, but they do not have a huge search space, and will probably mostly be possible to brute force in maximum a day. KTT evaluated their benchmark set by comparing the algorithms to their theoretical peak. They also used their set for demonstrating that autotuning for different systems is important for performance portability.

"ATF: A Generic Auto-Tuning Framework" by A. Rasch et al. [55] describes the autotuning framework ATF. This paper also compares ATF with OpenTuner and CLTune with the help of a set of parameterized algorithms. They measured the runtime of tuned kernels and found that ATF's tuned kernels has better speedups compared to the kernels of OpenTuner and CLTune. ATF compared the autotuners by using the same tuning techniques, with presumably the same options or tuning time limit.

TuneBench is a GitHub repository with a set of simple, tunable OpenCL kernels. [56] The documentation does not specify if this set of kernels are meant for benchmarking or what the intended use is.

Chapter 5

Plan for the Benchmark Suite

This chapter describes a bit more detail of our motivation, how this work is built on our specialization project, and a more detailed overall plan for the design of our benchmarking framework.

5.1 Motivation for Choosing SHOC

Since several autotuners focuses on tuning GPU kernels, Knut Kirkhorn and I knew we wanted to include HPC-based benchmarks that are oriented around GPU code. This led to us deciding to include both single kernels and kernels with attached host code in our benchmark suite for inclusivity.

We both obtained experience with the benchmark suite SHOC from our specialization project. SHOC is a known project that has been used by many others for benchmarking over the last years. In addition to this, SHOC focuses on multi GPU benchmarking, which is something that is very prominent in HPC. Considering all these points, we decided to use SHOC benchmarks in our benchmark suite for autotuners by parameterize algorithms from SHOC to make them tunable. Because of SHOC's good reputation, we thought that this would be better than to implement new algorithms from scratch. SHOC's reputation also made us decide to not use resources to perform an evaluation of the algorithms to find out how efficient they were compared to theoretical peak, as were done in KTT. SHOC has implementations for both CUDA and OpenCL, but we chose to only parameterize for one of them, and since we have experience with CUDA, we decided to parameterize the CUDA kernels.

5.2 Planning the Test Setup

Knut Kirkhorn and I decided to split the multi-GPU machines we had access to between us when performing tests with the finished benchmark suite. I would test on an IBM Power System AC922 with four Tesla V100-SXM2 32 GB GPUs and a machine with 20 Tesla T4 GPUs. Kirkhorn would test on an IBM Power System AC922 with two Tesla V100-SXM2 16 GB GPUs and the DGX-2 with 16 Tesla V100-SXM3 32 GB GPUs. We would in addition to this test 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. I would test on one GPU of DGX-2 and Kirkhorn would test on one GPU of the Tesla T4 based system.

We decided to test the finished benchmark suite with four different autotuners: OpenTuner, Kernel Tuner, CLTune and KTT. These autotuners were chosen because they are quite different in setup and how and what they use as tuning input. We figured that they would be representative to a lot of types of autotuners.

5.3 Choosing SHOC Algorithms to Parameterize

To decide which algorithms to parameterize from SHOC, we researched autotuners and their example tunable algorithms. *Appendix A: Parameter Research* shows most of the algorithms used in Kernel Tuner, CLTune, KTT and OpenTuner. The code examples from the autotuners are often typical HPC algorithms that are similar to the algorithms used in the Level 1 benchmarks from SHOC. We decided to parameterize most of the Level 1 algorithms from SHOC and divided them between us to ensure similar work amount. I would parameterize BFS, SpMV, MD5 Hash and Scan, Kirkhorn would parameterize Triad, MD, Reduction and Sort. We would in collaboration parametrize the Stencil 2D algorithm because we wanted to test this algorithm on multiple multi-GPU systems. We ensured that both had one algorithm each that had a true parallel (TP) implementation, which is Scan, Reduction and Stencil 2D.

To find which typical parameters that should be defined in the benchmarks, we researched the examples in the autotuners Kernel Tuner, CLTune, KTT and OpenTuner. *Appendix A: Parameter Research* has tables with parameters collected from these autotuners. This research gave us an idea of what to look for when parameterizing to make sure that the parameters are what is wanted for a benchmark suite. The parameter research also showed us that the search space sizes were different for the autotuners, and not always as big as expected. OpenTuner were the autotuner that focused most on having a big search space in the examples.

5.4 The Lack of Documentation

Lack of documentation is often an issue for projects. As mentioned in *4 Related Work*, TuneBench, and other projects that contains parameterized programs, could be potential benchmark sets for autotuners, but has no documentation to back this up. To have bad or missing documentation can be a flaw because potential users does not know what the project is, how they can use it or if they will benefit from using it. This can potentially make them refrain from using it because of the risk of having to familiarize themselves with a project and abandon it later if it was not what they were looking for. A big aspect when wanting the new benchmark suite for autotuners to potentially be a standard, is making it user friendly, and for this it is necessary with good documentation.

Looking at other benchmark suites that focuses on providing benchmarks to run on different computers, like SHOC, DeepBench, Tartan and Scope, they all have in common

that they can be run from one place in the directory with an easy command line interface which makes it very user friendly. Therefore, we decided to implement a similar command line interface.

5.5 Requirements for an Ideal Benchmark Suite

After researching benchmark suites, autotuners and algorithms the autotuners used, Knut Kirkhorn and I defined a set of requirements for measuring success. The goal is to meet most of these requirements to some degree. But this thesis can not realistically fulfill all. The requirements for an ideal HPC benchmark suite for autotuners with a GPU focus are:

- 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 autotuners. If the autotuner does not support certain parameters or the autotuner only supports tuning of kernels, there should still not be a problem using the benchmarks.
  - Benchmarks that have been well tested with different autotuners and on different machines.
  - Examples of how to use the benchmarks with autotuners.
  - A way to compare autotuners with other autotuners.
- 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.

The requirement that says that "the benchmark suite should have a way to compare autotuners with other autotuners" is not a requirement that is focused on for this thesis. It requires more research for finding a good way to give the autotuners a score to compare them with each other.

Our hypotheses is that a benchmark suite for autotuners based on SHOC can fulfill many of the requirements listed above, and for it to have the potential to become an ideal benchmark suite. Some questions related to this hypothesis should be answered later in this thesis:

- Is SHOC is a good benchmark suite to base the benchmark suite for autotuners on?
- Will there be a lot of rewriting of the code to ensure that the benchmark is enough parameterized?
- Will this benchmark suite, that is based on SHOC, be able to fulfill most of the points from the ideal benchmark suite requirements list?
- Will it be able to have a GPU and multi GPU focus?
- Will the benchmark suite work with different types of autotuners?
- Will the parameters added during the parameterization have different optimal values for different systems?

Chapter 6

Making the Benchmark Suite

This section will describe the process of parameterizing the algorithms and which decisions were made to ensure that the benchmark suite would be user friendly. All code before parameterization can be found in SHOC's GitHub repository [23]. All code after parameterization can be found as an attachment to this thesis.

6.1 Parameterizing the Algorithms

The process I went through when parameterizing each of the algorithms was to extract the algorithm from SHOC, find the compiler commands, and run a version of the algorithm without the rest of SHOC. The next step was to analyze the algorithm and try to identify parts of the code that could use parameters. I kept the list of parameters from *Appendix A Parameter Research* in mind when looking for these potential parameters. When identifying a potential parameter, I tried to set different values for the parameter and figure out which possible values it could have. When deciding on a parameter, I implemented the algorithm with OpenTuner to try more values than is efficient to do manually. Some parameters were chosen not to necessarily make the code faster, but to produce give more alternatives and better test autotuners.

A very common parameter is the block size for kernel launching. It is common to set the block size as a multiple of the warp size (32), but it can be set to any size between 1 and maximum for the GPU architecture, which for all GPUs used in this thesis is 1024.

Another parameter I used several times is to loop unroll or not. It is also possible to set a parameter as the unroll factor, but some of the autotuners cannot handle setting a factor for the compiler directive that needs to be set to unroll. With the `#pragma` directive, the compiler can be instructed to not unroll a loop or to unroll a specific way. By adding `#pragma unroll` directly in front of a loop, it will instruct the compiler to unroll this loop completely if possible. The compiler cannot unroll the loop if the number of iterations for the loop is not known at compile time. The loops unrolled in the parameterization later in this chapter all have a size that can be determined before run time. To provide a factor for how many iterations, `N`, to unroll `#pragma unroll N` needs to be set. To hinder loop unrolling the factor has to be set to `1` and the directive will be `#pragma unroll 1`. [6]. To check that the loop is unrolled or not unrolled I ran the command:

```
cuobjdump -sass <objectfile>.o
```

This command will print the CUDA kernel assembly code. Two files from different runs with different parameter values can then be compared. There are no guarantees for a loop to be unrolled even if the assembly code is different, but if the code is identical for two runs where one of them should be unrolled and one not, it is safe to say that it did not work.

Function inlining is a parameter I added in one of the algorithms. To inline or not can be done with the identifiers `__forceinline__` and `__noinline__`. [6] To see if the option did anything, I checked the assembly code with the same command as for loop unrolling.

The rest of the parameters I implemented can be seen in the next section, where the parameterization of BFS, SpMV, MD5 Hash, Scan and Stencil2D is described.

6.1.1 BFS

Block Size

Block size is one parameter that almost every GPU code can have, which is the threads per block when launching a GPU kernel. Before parameterization, the block size were set as the maximum threads per block for the GPU (Listing 6.1). The GPUs used in the experiments of this thesis all have a maximum of 1024 threads per block.

```
numBlocks = (int)ceil((double)numVerts/(double)devProp.maxThreadsPerBlock)
    ;

...

// Kernel Launch
BFS_kernel_warp<<<numBlocks,devProp.maxThreadsPerBlock>>>(d_costArray,
    d_edgeArray,
     d_edgeArrayAux, W_SZ, CHUNK_SZ, numVerts, iters, d_flag);
```

Listing 6.1: Setting block size in `BFS.cu` before parameterization.

The `numVerts` variable is the number of execution times for the kernel. This variable is divided by the number of threads per block to get the number of blocks. I decided to replace the block size of maximum threads per block with a parameter with a range of 1 to maximum threads per block, this can be seen in Listing 6.2. I also restricted the search space to only go up to the problem size, `numVerts`, for problem sizes smaller than 1024 to avoid doing unnecessary tuning, since ceil(1000/1000) = 1 and ceil(1000/1024) = 1.

```
int numThreads = BLOCK_SIZE;
numBlocks = (int)ceil((double)numVerts/(double)numThreads);

...

// Kernel Launch
BFS_kernel_warp<<<numBlocks, numThreads>>>(d_costArray,d_edgeArray,
    d_edgeArrayAux,
     W_SZ, CHUNK_SZ, numVerts, iters, d_flag);
```

Listing 6.2: Setting block size in `BFS.cu` after parameterization.

Chunk Factor

A constant used in SHOC's implementation of BFS is `CHUNK_SZ` and can be seen in Listing 6.3. The constant is set to 32 and symbolizes the number of vertices each warp processes. The warp size is 32, which means one thread processes one vertex. I decided to multiply this constant with a parameter called `CHUNK_FACTOR`.

```
int CHUNK_SZ = 32;

...

BFS_kernel_warp<<<numBlocks,numThreads>>>(d_costArray,d_edgeArray,
    d_edgeArrayAux, W_SZ, CHUNK_SZ, numVerts, iters, d_flag);
```

Listing 6.3: Chunk size in `BFS.cu` before parameterization.

Instead of passing chunk size as an argument to the kernel, I decided to set the `CHUNK_FACTOR` parameter directly in the kernel and multiply it by 32 to get the chunk size. This can be seen in Listing 6.4. To avoid creating unnecessary threads that is not going to be used, I divided the total number of blocks by the chunk factor. Listing 6.5 shows how `numBlocks` is initialized after parameterization.

```
int CHUNK_SZ = CHUNK_FACTOR*32;
```

Listing 6.4: Chunk size and chunk factor in `bfs_kernel.cu` after parameterization.

```
int numThreads = BLOCK_SIZE;
int numBlocks = (int)ceil((double)numVerts/(double)numThreads/(double)
    CHUNK_FACTOR);
```

Listing 6.5: Chunk factor in `BFS.cu` after parameterization.

The possible parameter values I chose for this parameter were 1, 2, 4 and 8. This was done to find out if a bigger chunk size would lead to performance improvements.

Texture Memory

A way to change how the memory is accessed is using texture memory. This type of memory is often used for 2D textures but can also be used for one dimensional data. Texture memory is a read-only memory type.

I implemented two ways to use texture memory, texture references and texture objects (also called bindless textures), for two different read only arguments passed to the kernels. The texture objects are a newer way to implement texture memory than with texture references. Using texture objects are also said to improve the performance of texture memory. [57] I implemented both types for checking if this performance difference would be noticeable.

The texture memory parameters are called `TEXTURE_MEMORY_EA1` for the `edgeArray` array and `TEXTURE_MEMORY_EAA` for the `edgeArrayAux` array. The first parameter ends with `EA1` instead of `EA` because some issues were experienced in one of the autotuners when the name of one of the parameter were included in another parameter name. Both parameters have possible values 0, 1 and 2. 0 for not using texture memory, 1 for using the older texture references and 2 for using the newer texture objects.

Listing 6.6 shows how the texture memory were initialized for `TEXTURE_MEMORY_EA1` in the host code. This was done in the same way for `TEXTURE_MEMORY_EAA`. Listing 6.7 shows how the kernel looked like without texture memory, before parameterization, and Listing 6.8 show how the kernel looked like parameterized with texture memory options.

```
cudaTextureObject_t textureObjEA = 0;
#if TEXTURE_MEMORY_EA1 == 2
  cudaResourceDesc resDescEA1;
  memset(&resDescEA1, 0, sizeof(resDescEA1));
  resDescEA1.resType = cudaResourceTypeLinear;
  resDescEA1.res.linear.devPtr = d_edgeArray;
  resDescEA1.res.linear.desc.f = cudaChannelFormatKindUnsigned;
```

```
  resDescEA1.res.linear.desc.x = 32;
  resDescEA1.res.linear.sizeInBytes = sizeof(unsigned int) * (numVerts+1);
  cudaTextureDesc texDescEA1;
  memset(&texDescEA1, 0, sizeof(texDescEA1));
  texDescEA1.readMode = cudaReadModeElementType;
  CUDA_SAFE_CALL(cudaCreateTextureObject(&textureObjEA, &resDescEA1, &
   texDescEA1,
    NULL));
#elif TEXTURE_MEMORY_EA1 == 1
  //Bind a 1D texture to the edgeArray array
  CUDA_SAFE_CALL(cudaBindTexture(0, textureRefEA, d_edgeArray,
    sizeof(unsigned int) * (numVerts+1)));
#endif
```

Listing 6.6: Initializing texture memory in `BFS.cu` after parameterization.

```
__global__ void BFS_kernel_warp(
  unsigned int *levels,
  unsigned int *edgeArray,
  unsigned int *edgeArrayAux,
  int W_SZ,
  int CHUNK_SZ,
  unsigned int numVertices,
  int curr,
  int *flag) {

    ...

  for(int v=v1; v< chk_sz-1+v1; v++) {
    if(levels[v] == curr) {
      unsigned int num_nbr = edgeArray[v+1]-edgeArray[v];
      unsigned int nbr_off = edgeArray[v];
      for(int i=W_OFF; i<num_nbr; i+=W_SZ) {
        int v = edgeArrayAux[i + nbr_off];

        ...
```

Listing 6.7: BFS kernel in `bfs_kernel.cu` before parameterization.

```
texture<unsigned int, 1, cudaReadModeElementType> textureRefEA;
texture<unsigned int, 1, cudaReadModeElementType> textureRefEAA;

extern "C" __global__ void BFS_kernel_warp(
  unsigned int *levels,
  unsigned int *edgeArray,
  cudaTextureObject_t textureObjEA,
  unsigned int *edgeArrayAux,
  cudaTextureObject_t textureObjEAA,
  int W_SZ,
  unsigned int numVertices,
  int curr,
  int *flag) {

  ...
```

```
16
  for (int v=v1; v< chk_sz-1+v1; v++) {
18    if (levels[v] == curr) {
        #if TEXTURE_MEMORY_EA1 == 2
20        unsigned int num_nbr = tex1Dfetch<unsigned int>(textureObjEA, v+1)
    -
              tex1Dfetch<unsigned int>(textureObjEA, v);
22        unsigned int nbr_off = tex1Dfetch<unsigned int>(textureObjEA, v);

24      #elif TEXTURE_MEMORY_EA1 == 1
          unsigned int num_nbr = tex1Dfetch(textureRefEA, v+1) -
26            tex1Dfetch(textureRefEA, v);
          unsigned int nbr_off = tex1Dfetch(textureRefEA, v);
28
        #else
30        unsigned int num_nbr = edgeArray[v+1]-edgeArray[v];
          unsigned int nbr_off = edgeArray[v];
32
        #endif
34
        for (int i=W_OFF; i<num_nbr; i+=W_SZ) {
36        #if TEXTURE_MEMORY_EAA == 2
            int v = tex1Dfetch<unsigned int>(textureObjEAA, (i + nbr_off));
38        #elif TEXTURE_MEMORY_EAA == 1
            int v = tex1Dfetch(textureRefEAA, (i + nbr_off));
40        #else
            int v = edgeArrayAux[i + nbr_off];
42        #endif

44        ...
```

Listing 6.8: Setting texture memory in `bfs_kernel.cu` after parameterization.

6.1.2 SpMV

Sparse matrix algorithms and storage formats lend themselves well to optimalizations. Following are some important ones we chose to focus on.

Matrix Storing Format

Since the SpMV algorithm for SHOC runs with two different alternatives for storing matrices, I decided to set the format type as a parameter. To avoid adding another restriction for if the CSR matrix is padded, I included this specification in the list of value alternatives. There are also two different variation of the CSR format, one where a single thread is used for processing a matrix row, and one where a warp (32 threads) is used for processing a row, called scalar and vector respectively. Listing 6.9 shows how the function for each format is chosen after parameterizing the algorithm. The compiler directives `#if`, `#elif`, `#else` and `#endif` were used for setting format type because these will be processed at compile-time instead of run-time.

```
// 0: ellpackr, 1: csr-normal-scalar, 2: csr-padded-scalar, 3: csr-normal-vector, 4: csr-padded-vector
#if (FORMAT == 1 || FORMAT == 3)
  // Test CSR kernels on normal data
  cout << "CSR Test\n";
  csrTest<floatType, texReader>(op, h_val, h_cols, h_rowDelimiters, h_vec, h_out, numRows, nItems, refOut, false);
#elif (FORMAT == 2 || FORMAT == 4)
  // Test CSR kernels on padded data
  cout << "CSR Test -- Padded Data\n";
  csrTest<floatType, texReader>(op, h_valPad, h_colsPad, h_rowDelimitersPad, h_vec, h_out, numRows, nItemsPadded, refOut, true);
#else
  // FORMAT == 0
  // Test ELLPACKR kernel
  cout << "ELLPACKR Test\n";
  ellPackTest<floatType, texReader>(op, h_val, h_cols, h_rowDelimiters, h_vec, h_out, numRows, nItems, refOut, false, paddedSize);
#endif
```

Listing 6.9: Setting format in SpMV in `spmv.cu` after parameterization.

Block Size

The block size for the kernels set as 128 in SHOC, and the grid size as the total number of rows divided by the block size for ELLPACK-R and the scalar version of CSR. For the vector version, the grid size was set as the same as the other version but also divided by the warp size, 32.

I decided to remove the initialization of the `BLOCK_SIZE` constant and use it as a parameter with possible values ranging from 1 to the maximum block size. Listing 6.10 shows the code before parameterization. I also added a constraint for the `FORMAT` and `BLOCK_SIZE` parameters:

```
FORMAT < 3 or (BLOCK_SIZE % 32 == 0)
```

The `BLOCK_SIZE` parameter needs to be divisible by 32 (warp size) if the format type is CSR vector.

```
// spmv.h
static const int BLOCK_SIZE = 128;

// spmv.cu
// In csrTest(...)
int nBlocksScalar = (int) ceil((floatType) numRows / BLOCK_SIZE);
int nBlocksVector = (int) ceil(numRows / (floatType)(BLOCK_SIZE / WARP_SIZE));

...
// CSR scalar kernel
spmv_csr_scalar_kernel<floatType, texReader><<<nBlocksScalar, BLOCK_SIZE>>>(d_val, d_cols, d_rowDelimiters, numRows, d_out);
```

```
...
// CSR vector kernel
spmv_csr_vector_kernel<floatType, texReader><<<nBlocksVector, BLOCK_SIZE
    >>>(d_val, d_cols, d_rowDelimiters, numRows, d_out);

...
// In ellpackrTest(...), cmSize = numRows
int nBlocks = (int) ceil((floatType) cmSize / BLOCK_SIZE);

...
// ELLPACK-R kernel
spmv_ellpackr_kernel<floatType, texReader><<<nBlocks, BLOCK_SIZE>>>(d_val,
     d_cols, d_rowLengths, cmSize, d_out);
```

Listing 6.10: Setting block size for SpMV in `spmv.cu` and `spmv.h` before parameterization.

Precision

SHOC's version of the SpMV algorithm does the calculation with two different precision modes, single and double. This led to choosing precision as a parameter, with possible values 32 for single precision and 64 for double precision. The alteration can be seen in Listing 6.11, where the precision mode is inserted in the templated function `RunTest`.

```
#if PRECISION == 32
  RunTest<float, texReaderSP>(op, probSizes[sizeClass]);
#else // PRECISION == 64
  RunTest<double, texReaderDP>(op, probSizes[sizeClass]);
#endif
```

Listing 6.11: Setting precision for SpMV in `spmv.cu` after parameterization.

Loop Unrolling

In the unparameterized code, there was an unrolled loop in the CSR vector kernel, which can be seen in Listing 6.12. I rolled the loop and added `#pragma` directives for unroll or not unrolling, which can be seen in Listing 6.13. With this being the second or inner loop, I called the parameter `UNROLL_LOOP_2` and gave it the possible values 0 (not unroll) and 1 (unroll whole loop).

```
// In spmv_csr_vector_kernel
// Reduce partial sums
if (id < 16) partialSums[t] += partialSums[t+16];
if (id <  8) partialSums[t] += partialSums[t+ 8];
if (id <  4) partialSums[t] += partialSums[t+ 4];
if (id <  2) partialSums[t] += partialSums[t+ 2];
if (id <  1) partialSums[t] += partialSums[t+ 1];
```

Listing 6.12: Unrolled loop in SpMV CSR vector kernel in `spmv.cu` before parameterization.

```
// In spmv_csr_vector_kernel
// Reduce partial sums
if (id < 16) {
  #if UNROLL_LOOP_2
  #pragma unroll
  #else
  #pragma unroll(1)
  #endif
  for (int i = 4; i >= 0; i--) {
    int l = 1 << i;
    if (id < l) partialSums[t] += partialSums[t+l];
  }
}
```

Listing 6.13: Loop in SpMV CSR vector kernel in `spmv_kernel.cu` after parameterization.

Since this loop is only used in the CSR vector kernel, the parameter should not have multiple value possibilities for other formats than for CSR vector. This is why a constraint had to be introduced setting `UNROLL_LOOP_2` to 0 if `FORMAT` is less than 3.

`FORMAT > 2 or UNROLL_LOOP_2 < 1`

Texture Memory

The implementation from SHOC were already using the older version of texture memory, so I decided to make the usage of texture memory into a parameter by adding an alternative of not using texture memory. Listing 6.14 shows the main change that had to be done to have an option to disable the use of texture memory. This is the same for the kernel launching of all types of matrix storage formats. There were also added some changes in accessing values from the array similarly to the earlier texture memory example for BFS.

```
spmv_csr_scalar_kernel<floatType, texReader><<<nBlocksScalar, BLOCK_SIZE
    >>>(d_val,
```

```
  d_cols, d_rowDelimiters,
  #if TEXTURE_MEMORY == 0
  d_vec,
  #endif
  numRows, d_out);
```

Listing 6.14: Texture memory for SpMV in `spmv.cu` after parameterization.

6.1.3 MD5 Hash

Block Size

The block size for the MD5 Hash algorithm can also be turned into a parameter. In SHOC, the block size, `nthreads`, is set to 384, but it can be any integer below the maximum threads per block. The code before parameterization can be seen in Listing 6.15. The changes to the algorithm when setting a block size parameter is shown in Listing 6.16. This parameter, like the earlier block size parameters, gets the value range of 1 to maximum block size.

```
int nthreads = 384;
size_t nblocks = ceil((double(keyspace)/double(valsPerByte))/double(
    nthreads));
```

Listing 6.15: Block size for MD5 Hash in `MD5hash.cu` before parameterization.

```
int nthreads = BLOCK_SIZE;
size_t nblocks = ceil((double(keyspace)/double(valsPerByte))/double(
    nthreads));
```

Listing 6.16: Block size for MD5 Hash in `md5hash_kernel.cu` after parameterization.

MD5 Round Style

The implementation of MD5 Hash in SHOC provided two different round styles. Which round style to use were set as a parameter, `ROUND_STYLE`, with 0 as a style that is using temporary variables, and 1 as a style that in-places via shift. Listing 6.17 shows the parameter in action in the final code.

```
// Here, we pick which style of ROUND we use.
#if ROUND_STYLE == 0
    #define ROUND ROUND_USING_TEMP_VARS
#else
    #define ROUND ROUND_INPLACE_VIA_SHIFT
#endif
```

Listing 6.17: Round style for MD5 Hash in `md5hash_kernel.cu` after parameterization.

Loop Unrolling

The MD5 Hash algorithm had multiple unrolled loops in the kernels from the SHOC implementation. I rolled up the loops and added `#pragma` directives for loop unrolling. Listing

6.18 shows the first unrolled loop before parameterization, and Listing 6.19 shows the rolled loop after. Listing 6.20 and Listing 6.21 shows the second and third loop before and after parameterization, respectively.

```
__host__ __device__ void IndexToKey(unsigned int index, int byteLength,
    int valsPerByte, unsigned char vals[8]) {
  vals[0] = index % valsPerByte;
  index /= valsPerByte;

  vals[1] = index % valsPerByte;
  index /= valsPerByte;

  vals[2] = index % valsPerByte;
  index /= valsPerByte;

  vals[3] = index % valsPerByte;
  index /= valsPerByte;

  vals[4] = index % valsPerByte;
  index /= valsPerByte;

  vals[5] = index % valsPerByte;
  index /= valsPerByte;

  vals[6] = index % valsPerByte;
  index /= valsPerByte;

  vals[7] = index % valsPerByte;
  index /= valsPerByte;
}
```

Listing 6.18: Unrolled loop in `IndexToKey` kernel in `MD5hash.cu` before parameterization.

```
void IndexToKey(unsigned int index, int byteLength, int valsPerByte,
    unsigned char vals[8]){
  #if UNROLL_LOOP_1
  #pragma unroll
  #else
  #pragma unroll(1)
  #endif
  for (int i = 0; i < 8; i++) {
    vals[i] = index % valsPerByte;
    index /= valsPerByte;
  }
}
```

Listing 6.19: Rolled loop in `IndexToKey` in `md5hash_kernel.cu` after parameterization.

```
...
foundKey[0] = key[0];
foundKey[1] = key[1];
foundKey[2] = key[2];
foundKey[3] = key[3];
foundKey[4] = key[4];
```

```
foundKey[5] = key[5];
foundKey[6] = key[6];
foundKey[7] = key[7];
foundDigest[0] = digest[0];
foundDigest[1] = digest[1];
foundDigest[2] = digest[2];
foundDigest[3] = digest[3];
...
```

Listing 6.20: `FindKeyWithDiges_Kernel` in `MD5hash.cu` before parameterization.

```
...
#if UNROLL_LOOP_2
#pragma unroll
#else
#pragma unroll(1)
#endif
for (int i = 0; i < 8; i++) {
  foundKey[i] = key[i];
}

#if UNROLL_LOOP_3
#pragma unroll
#else
#pragma unroll(1)
#endif
for (int i = 0; i < 4; i++) {
  foundDigest[i] = digest[i];
}
...
```

Listing 6.21: `FindKeyWithDiges_Kernel` in `md5hash_kernel.cu` after parameterization.

Function Inlining

The `md5_2words` kernel already used the `inline` function identifier. I wanted to include a parameter for inlining, and decided to add the `__forceinline__` and `__noinline__` options to the `md5_2words` kernel and to the `IndexToKey` kernel. Listing 6.22 shows how the inlining is used with the kernels after parameterization.

```
__host__ __device__
#if INLINE_1
__forceinline__
#else
__noinline__
#endif
void md5_2words(unsigned int *words, unsigned int len, unsigned int *
    digest){

...

__host__ __device__
#if INLINE_2
```

```
__forceinline__
#else
__noinline__
#endif
void IndexToKey(unsigned int index, int byteLength, int valsPerByte,
    unsigned char vals[8]) {
...
```

Listing 6.22: Inline functions for MD5 Hash in `md5hash_kernel.cu` after parameterization.

Work per Thread

I added a new parameter, `WORK_PER_THREAD_FACTOR`, to set different work per thread factor. This is a factor that says how much work a thread processes. Listing 6.23 shows the kernel before adding this parameter, and Listing 6.24 shows it after. Listing 6.25 shows that the total number of blocks needs to be divided by the new parameter to avoid generating excessive threads.

```
__global__ void FindKeyWithDigest_Kernel(...) {
  int threadid = blockIdx.x*blockDim.x+threadIdx.x;
  int startindex = threadid * valsPerByte;
```

Listing 6.23: Thread work for MD5 Hash in `MD5hash.cu` before parameterization.

```
__global__ void FindKeyWithDigest_Kernel(...) {
  for (int k = 0; k < WORK_PER_THREAD_FACTOR; k++) {
    int threadid = (blockIdx.x*blockDim.x+threadIdx.x) *
   WORK_PER_THREAD_FACTOR+k;
    int startindex = threadid * valsPerByte;
```

Listing 6.24: Work per thread for MD5 Hash in `md5hash_kernel.cu` after parameterization.

```
int nthreads = BLOCK_SIZE;
size_t nblocks  = ceil((double(keyspace) / double(valsPerByte)) / double(
    nthreads) / double(WORK_PER_THREAD_FACTOR));
```

Listing 6.25: Work per thread for MD5 Hash in `md5hash.cu` after parameterization.

6.1.4 Scan

For the Scan algorithm, files for both multi- and single-GPU implementations were parameterized.

Grid and Block Size

For SHOC's scan algorithm, both grid size and block size are set as constant integers, grid size as 64 and block size as 256. Listing 6.26 shows the before in SHOC and Listing 6.27 shows the after, when the constants have been replaced with parameters.

```
int num_blocks  = 64;
int num_threads = 256;
```

Listing 6.26: Setting grid size and block size in `Scan.cu` in SHOC before parameterization.

```
int num_blocks  = GRID_SIZE;
int num_threads = BLOCK_SIZE;
```

Listing 6.27: Setting grid size and block size in `scan.cu` after parameterization.

The possible values for `GRID_SIZE` and `BLOCK_SIZE` are power of two values with some exceptions. The `GRID_SIZE` values can be 1, 2, 4, 8, 16, 32, 64, 128, 256 or 512, and `BLOCK_SIZE` can be 16, 64, 128, 256, 512. `GRID_SIZE` also has to be less than or equal to `BLOCK_SIZE` which creates this constraint:

`GRID_SIZE <= BLOCK_SIZE`

Precision

The Scan algorithm had benchmarks for both single and double precision, which makes precision a good candidate for a parameter. Listing 6.28 shows the code after adding precision as a parameter.

```
#if PRECISION == 32
  RunTest<float, float4>("Scan-SP", op);
#else
  RunTest<double, double4>("Scan-DP", op);
#endif
```

Listing 6.28: Setting precision in `scan.cu` after parameterization.

Loop Unrolling

The Scan algorithm, like earlier algorithms, has possibilities for adding loop unrolling parameters. In Listing 6.29 two unrolled loops are shown and in Listing 6.30 the loops are rolled and has `#pragma` directives for unrolling or rolling the loops.

```
template <class T, int blockSize>
__device__ T scanLocalMem(const T val, volatile T* s_data) {
  ...
  T t;
  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();

  if (blockSize > 32) {
    t = s_data[idx -  32];  __syncthreads();
    s_data[idx] += t;       __syncthreads();
  }
  if (blockSize > 64) {
    t = s_data[idx -  64];  __syncthreads();
    s_data[idx] += t;       __syncthreads();
  }
  if (blockSize > 128) {
    t = s_data[idx -  128]; __syncthreads();
    s_data[idx] += t;       __syncthreads();
  }
  if (blockSize > 256) {
    t = s_data[idx -  256]; __syncthreads();
    s_data[idx] += t;       __syncthreads();
  }
  if (blockSize > 512) {
    t = s_data[idx -  512]; __syncthreads();
    s_data[idx] += t;       __syncthreads();
  }
  ...
```

Listing 6.29: Loop unrolling in `scan_kernel.h` in SHOC before parameterization.

```
template <class T, int blockSize>
__device__ T scanLocalMem(const T val, volatile T* s_data) {
  ...
  T t;
  #if UNROLL_LOOP_1
  #pragma unroll
  #else
  #pragma unroll(1)
  #endif
  for(int i = 0; i < 5; i++) {
    t = s_data[idx - (1<<i)];  __syncthreads();
    s_data[idx] += t;          __syncthreads();
  }

  if (blockSize > 32) {
    #if UNROLL_LOOP_2
    #pragma unroll
    #else
    #pragma unroll(1)
    #endif
    for(int i = 5; i < 10; i++) {
      int num = 1<<i;
      if (blockSize > num) {
        t = s_data[idx - num];  __syncthreads();
        s_data[idx] += t;       __syncthreads();
```

```
      }
    }
  }
  ...
```

Listing 6.30: Loop unrolling in `scan_kernel.cu` after parameterization.

Number of GPUs

Since the Scan algorithm has the possibility to run on multiple GPUs as true parallel, I wanted to add the number of GPUs as a parameter. Listing 6.31 shows the run command for the algorithm when running the true parallel version. `GPUS` is the number of GPUs and `DEVICE_LIST` is a list of device IDs. This parameter takes the first `n` GPU IDs when `GPUS = n`.

```
mpirun -np {GPUS} --allow-run-as-root ./scan -d {DEVICE_LIST}
```

Listing 6.31: Example of setting number of GPUs when running Scan.

Compiler Options

Other possible parameters for the Scan algorithm are compiler options. There are numerous different possible compiler options to add when using compilers like G++, GCC and NVCC. One parameter I chose to add was to use fast math or not. This is an option to use the fast math library for NVCC or not. [58] Listing 6.32 shows how this is added to the compile command.

```
nvcc -use_fast_math
```

Listing 6.32: Example of setting fast math in compiler string for the Scan algorithm.

Another compiler option is compiler optimization level. This can be set for the host and for the GPU code, as can be seen in Listing 6.33. The possible values for these parameters are 0, 1, 3 and 4.

```
nvcc -O{OPTIMIZATION_LEVEL_HOST} -Xptxas -O{OPTIMIZATION_LEVEL_DEVICE}
```

Listing 6.33: Example of setting optimization level in compiler string for scan algorithm.

Another NVCC compiler option is the `-maxrregcount` option which specifies the maximum amount of registers that GPU kernels are allowed to use. [58] The possible values for this parameter are -1, 20, 40, 60, 80, 100 and 120, where -1 is supposed to symbolize not setting the maximum register count. This is up to the autotuner to handle. The possible values for the parameter were found in collaboration with Knut Kirkhorn, who also used this parameter in the Reduction algorithm.

```
nvcc -maxrregcount={MAX_REGISTERS}
```

Listing 6.34: Example of setting maximum registers in a compiler string for Scan algorithm.

6.1.5 Stencil 2D

The Stencil 2D algorithm was parameterized in collaboration with Knut Kirkhorn.

Number of GPUs

We chose to only have one parameter, the number of GPUs, for this algorithm. This was to find out if the autotuners would choose the best amount of GPUs when using the multi-GPU systems. Listing 6.35 shows how this is set when running the algorithm. In our specialization project, we saw a difference in performance when using multiple GPUs for Stencil 2D. It showed that generally using more GPUs lead to better GFLOPS.

```
mpirun -np {GPUS} --allow-run-as-root ./stencil2d -d {DEVICE_LIST}
```

Listing 6.35: Example of setting number of GPUs when running Stencil 2D.

6.1.6 The Final Parameters

This section contains tables for every algorithm with available parameters, explanations and search space.

BFS

Table 6.1 shows the final parameters for the BFS algorithm.

Table 6.1: Parameters in the BFS algorithm.

Parameter	Explanation	Search Space
`BLOCK_SIZE`	The block size used for launching the kernel.	Integers from 1 to maximum threads per block for the architecture (1024 for this thesis) or to 1000 if the problem size is the lowest.
`CHUNK_FACTOR`	Factor multiplied with a chunk size (which is the work per thread).	[1, 2, 4, 8]
`TEXTURE_MEMORY_EA1`	Not use texture memory, use texture reference (older version) or use texture object (newer version).	[0, 1, 2]
`TEXTURE_MEMORY_EAA`	Not use texture memory, use texture reference (older version) or use texture object (newer version).	[0, 1, 2]

SpMV

Table 6.2 shows the final parameters for the SpMV algorithm and Table 6.3 shows the parameter restrictions.

Table 6.2: Parameters in the SpMV algorithm.

Parameter	Explanation	Search Space
`BLOCK_SIZE`	The block size used for the kernel.	Integers from 1 to maximum threads per block for the architecture (1024 for this thesis).
`PRECISION`	Use single or double precision on variables.	[32, 64]
`FORMAT`	Which format to use for SpMV. 0: ELLPACK-R 1: CSR Normal Scalar 2: CSR Padded Scalar 3: CSR Normal Vector 4: CSR Padded Vector	[0, 1, 2, 3, 4]
`UNROLL_LOOP_2`	Not unroll or unroll loop.	[0, 1]
`TEXTURE_MEMORY`	Not use texture memory or use the newer version of texture memory (texture objects).	[0, 1]

Table 6.3: Parameter restrictions for the SpMV algorithm.

Restriction	Explanation
`FORMAT < 3 or` `BLOCK_SIZE % 32 == 0`	Format 3 or 4, which is the CSR Vector formats, needs to have block size that is divisible by 32.
`FORMAT > 2 or` `UNROLL_LOOP_2 < 1`	The loop exists only in the CSR vector kernel, which means that the loop unroll parameter should be 0 when the format is something else than CSR vector.

MD5 Hash

Table 6.4 shows the final parameters for the MD5 Hash algorithm.

Table 6.4: Parameters in the MD5 Hash algorithm.

Parameter	Explanation	Search Space
BLOCK_SIZE	The block size used for the kernel.	Integers from 1 to maximum threads per block for the architecture (usually 1024).
ROUND_STYLE	Which MD5 round style to use.	[0, 1]
UNROLL_LOOP_1	Not unroll or unroll loop 1.	[0, 1]
UNROLL_LOOP_2	Not unroll or unroll loop 2.	[0, 1]
UNROLL_LOOP_3	Not unroll or unroll loop 3.	[0, 1]
INLINE_1	Not inline or inline kernel 1.	[0, 1]
INLINE_2	Not inline or inline kernel 2.	[0, 1]
WORK_PER_THREAD_FACTOR	Factor for setting how much work a thread should do.	[1, 2, 3, 4, 5]

Scan

Table 6.5 shows the final parameters for the Scan algorithm and Table 6.6 shows the parameter restrictions.

Table 6.5: Parameters in the Scan algorithm.

Parameter	Explanation	Search Space
BLOCK_SIZE	The block size used for the kernels.	[16, 64, 128, 256, 512]
GRID_SIZE	The grid size used for the kernels.	[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
PRECISION	Use single or double precision.	[32, 64]
UNROLL_LOOP_1	Not unroll or unroll loop 1.	[0, 1]
UNROLL_LOOP_2	Not unroll or unroll loop 2.	[0, 1]
USE_FAST_MATH	If fast math will be used when compiling.	[0, 1]
OPTIMIZATION_LEVEL_HOST	Compiler optimization level for host code.	[0, 1, 2, 3]
OPTIMIZATION_LEVEL_DEVICE	Compiler optimization level for GPU code.	[0, 1, 2, 3]
MAX_REGISTERS	The amount of max registers to be used for the GPU kernels. -1 means to not set this compiler option.	[-1, 20, 40, 60, 80, 100, 120]
GPUS	The number of GPUs used in the computations.	All integers between 1 and the number of GPUs available.

Table 6.6: Parameter restrictions for the Scan algorithm.

Restriction	Explanation
`GRID_SIZE <= BLOCK_SIZE`	Grid size needs to be less than or equal to block size.

Stencil 2D

Table 6.7 shows the final parameters for the Stencil 2D algorithm.

Table 6.7: Parameters in the Stencil 2D algorithm.

Parameter	Explanation	Search Space
`GPUS`	The number of GPUs used in the computations.	All integers between 1 and the number of GPUs available.

6.1.7 Total Parameter Search Space

Table 6.8 shows the total amount of value combinations for the algorithms without considering the parameter constraints. The table also includes the total amount if the block size is 1024.

Table 6.8: The total amount of value combinations for the algorithms.

Algorithm	Combination Calculation	Total Combinations
`BFS`	`MAX_BLOCK_SIZE` $\cdot 4 \cdot 3 \cdot 3$	`MAX_BLOCK_SIZE` $\cdot$ 36 *when 1024 block size => 36 864*
`SpMV`	`MAX_BLOCK_SIZE` $\cdot 2 \cdot 5 \cdot 2 \cdot 2$	`MAX_BLOCK_SIZE` $\cdot$ 40 *when 1024 block size => 40 960*
`MD5 Hash`	`MAX_BLOCK_SIZE` $\cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 5$	`MAX_BLOCK_SIZE` $\cdot$ 320 *when 1024 block size => 327 680*
`Scan`	$5 \cdot 10 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 4 \cdot 4 \cdot 7 \cdot$ `NUMBER_OF_GPUS`	89 600 $\cdot$ `NUMBER_OF_GPUS`
`Stencil 2D`	`NUMBER_OF_GPUS`	`NUMBER_OF_GPUS`

6.2 Making a User Friendly Benchmark Suite

We decided to name the benchmark suite BAT: A Benchmark suite for AutoTuners.

To create a benchmark suite that is easy to use and not just a collection of parameterized algorithms, Knut Kirkhorn and I implemented a Python script, `main.py`, that runs the benchmarks with autotuners added to the project from the root directory. For an autotuner to be added to the project, a folder with the benchmark-autotuner implementations needs to be included in `tuning_examples`. The next step is to add a `config.json` file inside all the folders for every benchmark implemented with the autotuner. This file specifies configurations for running the autotuner implementation with the given algorithm.

The command line interface (CLI) provides multiple ways to run the benchmarks: a single benchmark for a single autotuner, all available benchmarks for one autotuner, one benchmark algorithm with all of the autotuners added, or every benchmark with every autotuner. The `main.py` script will copy the results to a `results` folder in the top directory. Having all the results saved in one place makes it more practical to run multiple benchmarks at the same time. The CLI will also print useful information when benchmarking, which includes error messages if something goes wrong.

The CLI provides a way to set arguments for which benchmark and autotuner to use. Other arguments like verbosity, problem size and search technique are also possible to set values for. Listing 6.36 shows all the arguments with descriptions from the `main.py` file.

```
parser.add_argument("--benchmark", "-b", type=str, default=None, help="
    name of the benchmark (e.g.: sort)")
parser.add_argument("--auto-tuner", "-a", type=str, default=None, help="
    auto-tuner to benchmark (e.g.: opentuner)")
parser.add_argument("--verbose", "-v", action="store_true", help="print
    stdout and stderr from building of benchmarks")
parser.add_argument("--size", "-s", type=int, default=1, help="problem
    size to the benchmark(s) (e.g.: 2)")
parser.add_argument("--technique", "-t", type=str, default="brute_force",
    help="tuning technique to use for the benchmark(s) (e.g.: annealing)")
```

Listing 6.36: Possible CLI arguments for running benchmarks.

The `main.py` script can be found in the attached code, and a guide to constructing `config.json` files can be found in the `README.md` file in *Appendix B BAT User Guide*.

For a benchmark suite to actually be used it should be user friendly, easy to understand and intuitive to use. One of the steps Knut Kirkhorn and I took to meet this requirement was to make the source code structured in different folders. Figure 6.1 shows how BAT is structured. The `src` folder has two separate folders with `kernels` and `programs`. The `programs` folder contains both host code and GPU code, instead of retrieving the kernel from the `kernels` folder. This was done mainly for it to be easier for someone to extract only the code they want to use if they do not want to use the whole benchmark suite.

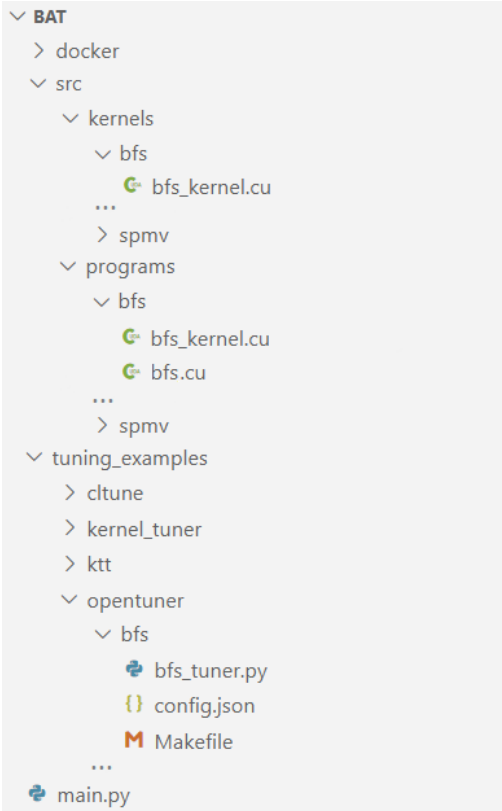

Figure 6.1: Selected part of the project structure in BAT.

The `tuning_examples` folder contains autotuner implementations for the programs or kernels. The content in this folder is useful to get an example of how to set up autotuning with the kernels or programs. The implementations for KTT and CLTune uses the kernels, and OpenTuner and Kernel Tuner uses the programs.

The benchmark suite includes Dockerfiles for easier testing, with one Dockerfile for each autotuner. The dependencies are set up in the files which makes the autotuning run without any additional setup or installation, as long as Docker is installed.

An important way to make a project user friendly is to have documentation that explains what the project is, who will benefit from using it and how to use it. Knut Kirkhorn and I made a `readme.md` file for documentation that incorporates this information. The file can be seen in *Appendix B BAT User Guide*. In addition to this information, the documentation links to another file that has a list of all parameters with their search space, similar to the content in section *6.1.6 The Final Parameters*. This file can be seen in the attached code.

BAT Users

Kirkhorn and I defined a list of possible users of this benchmark suite. These are the types of users that can potentially benefit from using BAT. Some of these types of users are part of the reasoning for the structure and the CLI.

- A user that wants to have a set of parameterized kernels or programs to test an autotuner with. This user only wants the `kernels` or `programs` folders.
- A user that wants to use the whole benchmark suite and add a new autotuner to the `tuning_examples` folder. The motivation for this can be to compare the new autotuner with the existing ones, or to just use the CLI to easier run benchmarks with the added autotuner.
- A user that want to use the whole benchmark suite and only use the autotuners that already are added for running benchmark tests on different machines or architectures. This can be done to see if specific parameters change values on different architectures.

Chapter 7

Testing the Benchmark Suite

Testing the benchmark suite is an important part for making sure that the benchmark suite will work for multiple types of autotuners. The testing was done by implementing the benchmarks with some known autotuners. The autotuners that were chosen for this were OpenTuner, Kernel Tuner, CLTune and KTT. KTT is based on CLTune, but outside of that, all the autotuners are a bit different with different setup.

The autotuners are often built different and takes different code as input. An example of this is that not every autotuner has possibility to set texture memory if they only take a kernel as input.

All of the benchmarks from BAT is implemented with all of the mentioned autotuner to make sure that all benchmarks would work for different types of autotuners. All the implementation can be found in the attached code. Knut Kirkhorn and I collaborated on figuring out how to set up the autotuners with the parameterized benchmarks, and Kirkhorn had main responsibility for CLTune and KTT, while I had main responsibility for OpenTuner and Kernel Tuner.

For some of the autotuners it is possible to set which type of measurement to use when ranking the tuning combinations. But since some of them only can compare based on time used, we decided to use time as measurement for all autotuners.

7.1 Autotuner Implementations

The following section will contain descriptions of how I implemented the autotuners with the different benchmarks, with most focus on OpenTuner and Kernel Tuner.

7.1.1 OpenTuner

When implementing a parameterized algorithm with OpenTuner, a Python file have to be made with tuning information. One of the Python packages I needed was the `numba` package to get the maximum number of threads for the GPUs. To utilize the `numba` package, it was easier to use Python 3 than Python 2 that OpenTuner uses. When changing to Python 3, I had to make a change to OpenTuner in file `opentuner/search/manipulator.py`. "`with`

`open(filename, 'wb') as fd:"` on line 100 were changed to "`with open(filename, 'w') as fd:`". A GitHub fork with this change can be found here: `github.com/ingunnsund/opentuner`

Going through the process of setting up an OpenTuner implementation. With one implementation at a time, but most focus on one of them.

Since OpenTuner tunes a whole program, it supports all of the parameters defined in *6.1 Parameterizing the Algorithms*. The autotuner even supports compiler options.

BFS

Listing 7.1 show the manipulator part of the tuning file. This is the part where the parameters and their search space are set. `IntegerParameter` are used for parameters where the search spaces are continuous, and `EnumParameter` for the search spaces that are a set of values that are not necessary continuous.

The maximum thread per block is found for the current device, and used for the maximum block size when setting the possible parameter values.

```
class BFSTuner(MeasurementInterface):

  def manipulator(self):
    gpu = cuda.get_current_device()
    sizes = [1000, 10000, 100000, 1000000]
    numVerts = sizes[argparser.parse_args().size - 1]

    min_size = 1
    max_size = min(numVerts, gpu.MAX_THREADS_PER_BLOCK)

    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(IntegerParameter('BLOCK_SIZE', min_size,
   max_size))
    manipulator.add_parameter(EnumParameter('CHUNK_FACTOR', [1, 2, 4, 8]))
    ...

    return manipulator
```

Listing 7.1: OpenTuner: BFS manipulator

In Listing 7.2 it is shown how the compiler make commands are generated, where the parameters and values are set in the compile string. There are error checks for compiling and running the code. A check for errors in the code is also added on line 49.

```
start_path = '../../../src/programs'

...

  def run(self, desired_result, input, limit):
    """
    Compile and run a given configuration then
    return performance
    """
    args = argparser.parse_args()

    cfg = desired_result.configuration.data
```

```
    compute_capability = cuda.get_current_device().compute_capability
    cc = str(compute_capability[0]) + str(compute_capability[1])

    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}/bfs/BFS.cu'
    ...
    make_program += '-D{0}={1}'.format('CHUNK_FACTOR',cfg['CHUNK_FACTOR'])
    make_program += ' -D{0}={1} \n'.format('BLOCK_SIZE',cfg['BLOCK_SIZE'])

    if args.parallel:

      ...

    else:
      make_serial_start = f'nvcc -I {start_path}/common/ -I {start_path}/cuda-common/ -g -O2 -c -o Graph.o {start_path}/common/Graph.cpp \n'
      ...
      compile_cmd = make_serial_start + make_program + make_serial_end

    compile_result = self.call_program(compile_cmd)
    assert compile_result['returncode'] == 0

    program_command = './BFS -s ' + str(args.size)
    if args.parallel:
      # Select number below max connected GPUs
      chosen_gpu_number = min(args.gpu_num, 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}'
    else:
      run_cmd = program_command

      run_result = self.call_program(run_cmd)

      # Check that error code and error output is ok
      assert run_result['stderr'] == b''
      assert run_result['returncode'] == 0

      result = {'parameters': cfg, 'time': run_result['time']}
      self.all_results.append(result)
      return Result(time=run_result['time'])
```

Listing 7.2: OpenTuner: BFS run function

Listing 7.3 contains a function for saving the results. The tuning script can also take size, number of GPUs and a parallel boolean as arguments.

```
  def save_final_config(self, configuration):
    """called at the end of tuning"""
    print("Optimal parameter values written to results.json:", configuration.data)
    with open('all-results.json', 'w') as f:
        json.dump(self.all_results, f, indent=4)
```

```
    # Update configuration with problem size and tuning technique
    configuration.data["PROBLEM_SIZE"] = argparser.parse_args().size
    configuration.data["TUNING_TECHNIQUE"] = argparser.parse_args().
   technique

    self.manipulator().save_to_file(configuration.data, 'results.json')

if __name__ == '__main__':
  argparser = opentuner.default_argparser()
  argparser.add_argument('--size', type=int, default=1, help='problem size
    of the program (1-4)')
  argparser.add_argument('--gpu-num', type=int, default=1, help='number of
    GPUs')
  argparser.add_argument('--parallel', action="store_true", help='run on
   multiple GPUs')
  BFSTuner.main(argparser.parse_args())
```

Listing 7.3: OpenTuner BFS safe config

SpMV

The SpMV implementation is quite similar to the BFS implementation. The only notable difference in the structure of the tuning file is the parameter constraints in SpMV, as can be seen in Listing 7.4. OpenTuner does not have a built-in way to set parameter constraints, but the time can be set as infinity to make sure that OpenTuner knows that this is not a good combination of values.

The rest of this tuning file can be found in the code attached to this thesis.

```
def run(self, desired_result, input, limit):

    ...

    # Check constraints for the parameters
    if not (cfg['FORMAT'] < 3 or cfg['BLOCK_SIZE'] % 32 == 0):
        return Result(time=float("inf"), state="ERROR", accuracy=float("-
   inf"))

    if not (cfg['FORMAT'] > 2 or cfg['UNROLL_LOOP_2'] < 1):
        return Result(time=float("inf"), state="ERROR", accuracy=float("-
   inf"))
```

Listing 7.4: OpenTuner: SpMV run function

MD5 Hash

There are no notable changes from the MD5 Hash tuning file compared to the earlier examples. The tuning file can be found in the code attached to this thesis.

Scan

The Scan algorithm also has a constraint, that is set in a similar way to the constraints in SpMV. The compiler flags parameters for this algorithm can be set in the compile string, as can be seen in Listing 7.5 in line 5-13. The number of GPUs parameter is set in the run string on line 23.

For this algorithm, there exists two parallel versions (EP and TP), where the difference is in the compile commands.

The rest of this tuning file can be found in the code attached to this thesis.

```
def run(self, desired_result, input, limit):

    ...

    use_fast_math = ''
    if cfg['USE_FAST_MATH']:
        use_fast_math = '-use_fast_math '

    max_registers = f'-maxrregcount={cfg["MAX_REGISTERS"]} '
    if cfg['MAX_REGISTERS'] == -1:
        max_registers = ''

    make_program = f'nvcc -O{cfg["OPTIMIZATION_LEVEL_HOST"]} {use_fast_math}{max_registers}-Xptxas -O{cfg["OPTIMIZATION_LEVEL_DEVICE"]},-v -gencode=arch=compute_{cc},code=sm_{cc} -I {start_path}/cuda-common -I {start_path}/common -c {start_path}/scan/scan.cu'
    make_program += ' -D{0}={1}'.format('PRECISION',cfg['PRECISION'])
    ...
    make_program += ' -D{0}={1} \n'.format('BLOCK_SIZE',cfg['BLOCK_SIZE'])

    ...

    program_command = './scan -s ' + str(args.size)
    if args.parallel == 1 or args.parallel == 2:
        chosen_gpu_number = cfg['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}'
    else:
        run_cmd = program_command
```

Listing 7.5: OpenTuner: Scan run function

Stencil 2D

There are no notable changes from the Stencil 2D tuning file compared to the earlier examples. The tuning file can be found in the code attached to this thesis.

7.1.2 Kernel Tuner

Kernel Tuner is also an autotuner where the tuning setup file is a Python file. This autotuner can tune both singular kernels and whole programs. The program consists of host code and a kernel that returns a unit for performance measurement, for example the time used.

BFS

I originally tried to tune just the BFS kernel, but even though texture memory could be set, it could not be used as a parameter. I then decided to implement the host code as well as the kernel. This host code is to the host code in the programs folder.

Listing 7.6 shows how the algorithm is implemented with Kernel Tuner in the tuning file. First, command line arguments are processed, then tuning parameters with search space are set. After this the tuning happens and then the results are saved.

```
# Setup CLI parser
parser = argparse.ArgumentParser(description="BFS tuner")
parser.add_argument("--size", "-s", type=int, default=1, help="problem
    size to the benchmark (e.g.: 2)")
parser.add_argument("--technique", "-t", type=str, default="brute_force",
    help="tuning technique to use for the benchmark (e.g.: annealing)")
arguments = parser.parse_args()

size = arguments.size
gpu = cuda.get_current_device()
sizes = [1000, 10000, 100000, 1000000]
vertices = sizes[size - 1]

# Use host code in combination with CUDA kernel
kernel_files = ['bfs_host.cu', '../../../src/programs/bfs/bfs_kernel.cu']

min_block_size = 1
max_block_size = min(vertices, gpu.MAX_THREADS_PER_BLOCK)

tune_params = dict()
tune_params["BLOCK_SIZE"] = [i for i in range(min_block_size,
    max_block_size+1)]
tune_params["CHUNK_FACTOR"] = [1, 2, 4, 8]
tune_params["TEXTURE_MEMORY_EA1"] = [0, 1]
tune_params["TEXTURE_MEMORY_EAA"] = [0, 1]

strategy_options = {}
if arguments.technique == "genetic_algorithm":
    strategy_options = {"maxiter": 50, "popsize": 10}

tuning_results = tune_kernel("RunBenchmark", kernel_files, vertices, [],
    tune_params, strategy=arguments.technique, lang="C", block_size_names=[
    "BLOCK_SIZE"],
    compiler_options=["-I ../../../src/programs/bfs/", "-I ../../../src/
    programs/common/", "-I ../../../src/programs/cuda-common/", f"-
    DPROBLEM_SIZE={size}"],
    iterations=2, strategy_options=strategy_options)

```

```
# Save the results as a JSON file
with open("bfs-results.json", 'w') as f:
    json.dump(tuning_results, f, indent=4, cls=NumpyEncoder)

# Get the best configuration
best_parameter_config = min(tuning_results[0], key=lambda x: x['time'])
best_parameters = dict()

# Filter out parameters from results
for k, v in best_parameter_config.items():
    if k not in tune_params:
        continue

    best_parameters[k] = v

# Add problem size and tuning technique to results
best_parameters["PROBLEM_SIZE"] = size
best_parameters["TUNING_TECHNIQUE"] = arguments.technique

# Save the best results as a JSON file
with open("best-bfs-results.json", 'w') as f:
    json.dump(best_parameters, f, indent=4, cls=NumpyEncoder)
```

Listing 7.6: Kernel Tuner: BFS Tuner

SpMV

The SpMV implementation is mostly similar, with the exception of parameter restrictions. Setting restrictions can be seen in Listing 7.7. The rest of the tuning file can be found in the attached code.

```
restrict = ["FORMAT < 3 or BLOCK_SIZE % 32 == 0", "FORMAT > 2 or (
    UNROLL_LOOP_2 < 1)"]

tuning_results = tune_kernel("RunBenchmark", kernel_files, size, [],
    tune_params, strategy=arguments.technique, restrictions=restrict, lang=
    "C", block_size_names=["BLOCK_SIZE"], compiler_options=["-I ../../../
    src/kernels/spmv/", "-I ../../../src/programs/common/", "-I ../../../
    src/programs/cuda-common/", f"-DPROBLEM_SIZE={sizeIndex}"], iterations
    =2, strategy_options=strategy_options)
```

Listing 7.7: Kernel Tuner: BFS Tuner

MD5 Hash

The MD5 implementation is similar to the earlier implementations. The tuning file can be found in the attached code.

Scan

The Scan implementation is similar to the earlier implementations. The tuning file can be found in the attached code.

There is an option to set compiler options, but it is not possible to use a tuning parameter for the options. Multiple GPUs are not possible to use for Kernel Tuner.

Stencil 2D

Stencil 2D is not implemented with Kernel Tune because it is only possible to use one GPU in code for this autotuner.

7.1.3 CLTune

To implement a parameterized algorithm with CLTune, a C++ file needs to be included to setup the tuning. CLTune only tunes kernels and they need to have a `extern C` term in front of the kernel to be able to be tuned. CLTune also does not support templated kernels, which means i had to add helper kernels to then call the templated kernels.

To check for correctness, a reference kernel needs to be added with parameter values that are guaranteed to give correct answers. Since CLTune only tunes kernels, the input has to be generated. This was not that big of a problem since the tuning file is in C++, same as the host code that generates input in the programs.

BFS

Listing 7.8 includes the most important part of the CLTune kernel. CLTune did not include support for texture memory, which lead to setting the texture memory value to 0. This shows that it is important to be able to disable the use of texture memory for benchmarks.

```
int main(int argc, char* argv[]) {
    ...
    // Generate data
    ...

    // Get the maximum threads per block
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, 0);
    unsigned int maxThreads = min(deviceProp.maxThreadsPerBlock, size);
    // Define a vector of block sizes from 1 to maximum threads per block
    vector<long unsigned int> block_sizes = {};
    for(int i = 1; i < (maxThreads+1); i++) {
        block_sizes.push_back(i);
    }

    // Add kernel
    size_t kernel_id = auto_tuner.AddKernel({kernelFile}, kernelName, {
   numVerts}, {1});

    // Add parameters to tune
    auto_tuner.AddParameter(kernel_id, "BLOCK_SIZE", block_sizes);
```

```
    auto_tuner.AddParameter(kernel_id, "CHUNK_FACTOR", {1, 2, 4, 8});

    // Set the different block sizes (local size) multiplied by the base
   (1)
    auto_tuner.MulLocalSize(kernel_id, {"BLOCK_SIZE"});
    // Divide the total number of threads by the chunk factor
    auto_tuner.DivGlobalSize(kernel_id, {"CHUNK_FACTOR"});

    // Set reference kernel for correctness verification and compare to
   the computed result
    auto_tuner.SetReference({referenceKernelFile}, kernelName, {numVerts},
    {512});

    // Add arguments for kernel
    auto_tuner.AddArgumentOutput(costArray);
    auto_tuner.AddArgumentInput(edgeArrayVector);
    auto_tuner.AddArgumentInput(edgeArrayAuxVector);
    auto_tuner.AddArgumentScalar(32);
    auto_tuner.AddArgumentScalar(numVertsInt);
    auto_tuner.AddArgumentScalar(0);
    auto_tuner.AddArgumentOutput(flag);

    ...

    auto_tuner.Tune();

...
```

Listing 7.8: CLTune: BFS Tuner

SpMV

A helper kernel had to be added for the SpMV kernels because they use templates for setting precision. One issue that arose is that there was no way to set the data type as a parameter for the input values to the kernels. Listing 7.9 shows the helper kernel with the additional input kernel parameters that had to be added to make sure that all the different data could be provided to the different kernels.

With this solution, the reference kernel does not work, since the output results that is compared will sometimes be of different data type. This lead to correctness verification having to be split up.

```
extern "C" __global__ void
spmv_kernel(float * valSP_csr,
        double * valDP_csr,
        float * valSP_csr_pad,
        double * valDP_csr_pad,
        float * valSP_ellpackr,
        double * valDP_ellpackr,
        const int    * __restrict__ cols_csr,
        const int    * __restrict__ cols_csr_pad,
        const int    * __restrict__ cols_ellpackr,
        const int    * __restrict__ rowDelimiters,
```

```
        const int    * __restrict__ rowDelimiters_pad,
        const int    * __restrict__ rowLengths,
        float * vecSP,
        double * vecDP,
        const int dim,
        const int dim_pad,
        float * outSP,
        double * outDP
    ) {

    #if PRECISION == 32
    // Single precision
        #if (FORMAT == 1 || FORMAT == 2)
        // CSR scalar
            #if (FORMAT == 1)
            // Normal
                spmv_csr_scalar_kernel<float>(valSP_csr, cols_csr, rowDelimiters, vecSP, dim, outSP);
            #else
            // Padded
                spmv_csr_scalar_kernel<float>(valSP_csr_pad, cols_csr_pad, rowDelimiters_pad, vecSP, dim_pad, outSP);
            #endif
        #elif (FORMAT == 3 || FORMAT == 4)
    ...
```

Listing 7.9: CLTune: SpMV reference kernel

Listing 7.10 shows setting the parameter constraints for the SpMV algorithm. The rest of the tuning file can be found in the attached code.

```
...

// Add constraint for only using block sizes that are a multiple of 32 for CSR vector format (format 3 or 4)
auto blockSizeLimit = [] (std::vector<size_t> v) {
    return (v[1] < 3 || v[0] % 32 == 0);
};
auto_tuner.AddConstraint(kernel_id, blockSizeLimit, {"BLOCK_SIZE", "FORMAT"});

// Add constraint for only unrolling loop 2 when the format is CSR Vector
auto unrollLoop2 = [] (std::vector<size_t> v) {
    return (v[0] > 2 || v[1] < 1);
};
auto_tuner.AddConstraint(kernel_id, unrollLoop2, {"FORMAT", "UNROLL_LOOP_2"});

// Add a constraint for what to multiply the total thread size by. If the format is the CSR vector format (3 or 4)
// the total number of threads should be multiplied by 32.
auto threadMultiply = [] (std::vector<size_t> v) {
    if (v[1] > 2) {
        return v[0] == 32;
    } else {
```

```
        return v[0] == 1;
    }
};
auto_tuner.AddConstraint(kernel_id, threadMultiply, {"THREADS_PER_ROW", "
    FORMAT"});

// Multiply the base number (1) of threads per block with the parameter
    value
auto_tuner.MulLocalSize(kernel_id, {"BLOCK_SIZE"});
// Multiply the total thread number by 32 if the format is 3 or 4, if not
    multiply by 1.
auto_tuner.MulGlobalSize(kernel_id, {"THREADS_PER_ROW"});

...
```

Listing 7.10: CLTune: SpMV tuner

MD5 Hash

The MD5 Hash implementation with CLTune is similar to the others and the tuning file can be found in the attached code.

One problem with the MD5 Hash implementation was that CLTune did not support the data type `char` for input data. This was an relatively easy thing to fix in CLTune, and resulted in a GitHub fork: `github.com/ingunnsund/CLTune`.

Scan

The Scan algorithm did not work with CLTune because Scan has three different kernels that is run after each other. The second kernel, `scan_single_block`, did not work with generated input because the input varies in size and values based on grid size and problem size.

Stencil 2D

CLTune did not have support for running multi-GPU code, which means that there was not point in implementing a version for Stencil 2D.

7.1.4 KTT

Implementing the algorithms in KTT were similar to CLTune. A C++ file is used for the tuning code and a reference kernel is mainly used for reference checking.

KTT does not support templated kernels or texture memory, which means that the same issues for the CLTune implementation is present for KTT as well.

One problem with KTT was with the kernel launch parameters. KTT says in its documentation that the `global size` parameter in the `addKernelFromFile()` function is the same as grid size in CUDA [59]. The `global size` parameter is actually the total thread number. This lead to some confusion, and what made it worse was that the launch dimensions printed during tuning were not correct. They were printed before any thread modifiers were added.

BFS

The most important part of the tuning file for the BFS algorithm can be seen in Listing 7.11. The `setThreadModifier()` functions in the listing modifies the local and global sizes. When modifying the global size, the total number of threads needs to be divided by `BLOCK_SIZE` and `CHUNK_FACTOR`. KTT divides this global size by `BLOCK_SIZE` after the thread modifying, but does not use the `ceil()` function. This means that the division might lead to not enough threads being generated. A way to fix this is to calculate the global size as the final grid size, and then multiply with the `BLOCK_SIZE` again. When KTT later divides by `BLOCK_SIZE`, it will always return an integer value.

```
const ktt::DimensionVector gridSize(numVerts);
const ktt::DimensionVector blockSize;
const ktt::DimensionVector blockSizeReference(512);

...

// Add parameters to tune
auto_tuner.addParameter(kernelId, "BLOCK_SIZE", block_sizes);
auto_tuner.addParameter(kernelId, "CHUNK_FACTOR", {1, 2, 4, 8});

// Multiply block size base (1) by BLOCK_SIZE parameter value
auto_tuner.setThreadModifier(kernelId, ktt::ModifierType::Local, ktt::
    ModifierDimension::X, "BLOCK_SIZE", ktt::ModifierAction::Multiply);

// Divide total size by block size and chunk factor (and multiply with
    block size again (because KTT will divide) to make sure number of
    threads is rounded up)
auto globalModifier = [](const size_t size, const std::vector<size_t>&
    vector) {
     return int(ceil(double(size) / double(vector.at(0)) / double(vector.at
    (1)))) * vector.at(0);
};
auto_tuner.setThreadModifier(kernelId, ktt::ModifierType::Global, ktt::
    ModifierDimension::X,
     std::vector<std::string>{"BLOCK_SIZE", "CHUNK_FACTOR"}, globalModifier
    );
```

Listing 7.11: KTT: BFS tuner

SpMV

The SpMV implementation with KTT is similar to the earlier implementations and can be found in the code attached.

MD5 Hash

The MD5 Hash implementation with KTT is similar to the earlier implementations and can be found in the code attached.

Scan

The Scan algorithm was possible to implement in KTT, but a tuning manipulator needed to be added for running kernels after each other and share data. Listing 7.12 shows how the kernels were run after each other.

The reference check for this implementation is different than for the others. A function to check correctness against a CPU calculation were used as correctness verification.

```
void launchComputation(const ktt::KernelId kernelId) override {
    vector<ktt::ParameterPair> parameterValues = getCurrentConfiguration()
    ;
    size_t blockSize = getParameterValue("BLOCK_SIZE", parameterValues);
    size_t gridSize = getParameterValue("GRID_SIZE", parameterValues);
    size_t totalSize = blockSize * gridSize;

    resizeArgumentVector(blockSumsIdf, blockSize, true);
    resizeArgumentVector(blockSumsIdd, blockSize, true);

    // Run "reduce" kernel
    runKernel(kernelIds[0], ktt::DimensionVector(totalSize, 1, 1), ktt::
    DimensionVector(blockSize, 1, 1));

    // Run "scan_single_block_helper" kernel
    // Grid size for this kernel is 1. Block size is chosen as the global
    size because it will be divided by block size in runKernel()
    runKernel(kernelIds[1], ktt::DimensionVector(blockSize, 1, 1), ktt::
    DimensionVector(blockSize, 1, 1));

    // Run "bottom_scan_helper" kernel
    runKernel(kernelIds[2], ktt::DimensionVector(totalSize, 1, 1), ktt::
    DimensionVector(blockSize, 1, 1));
}
```

Listing 7.12: KTT: SpMV tuning manipulator

Stencil 2D

The Stencil 2D algorithm was not implemented for KTT because KTT does not support multi-GPU algorithms.

7.1.5 Viable Parameters for the Autotuners

Since not every parameter for every algorithm is possible to implement in all of the autotuners, tables with a summary of viable parameters for the autotuners can be found below for BFS in Table 7.1, SpMV in Table 7.2, MD5 Hash in Table 7.3, Scan in Table 7.4 and Stencil 2D in Table 7.5.

Table 7.1: Parameters from BFS that is used with the autotuners

	OpenTuner	**Kernel Tuner**	**CLTune**	**KTT**
BLOCK_SIZE	✓	✓	✓	✓
CHUNK_FACTOR	✓	✓	✓	✓
TEXTURE_MEMORY _EA1	✓	✓	✗	✗
TEXTURE_MEMORY _EAA	✓	✓	✗	✗

Table 7.2: Parameters from SpMV that is used with the autotuners

	OpenTuner	**Kernel Tuner**	**CLTune**	**KTT**
BLOCK_SIZE	✓	✓	✓	✓
PRECISION	✓	✓	✓	✓
FORMAT	✓	✓	✓	✓
UNROLL_LOOP_2	✓	✓	✓	✓
TEXTURE_MEMORY	✓	✓	✗	✗

Table 7.3: Parameters from MD5 Hash that is used with the autotuners

	OpenTuner	**Kernel Tuner**	**CLTune**	**KTT**
BLOCK_SIZE	✓	✓	✓	✓
ROUND_STYLE	✓	✓	✓	✓
UNROLL_LOOP_1	✓	✓	✓	✓
UNROLL_LOOP_2	✓	✓	✓	✓
UNROLL_LOOP_3	✓	✓	✓	✓
INLINE_1	✓	✓	✓	✓
INLINE_2	✓	✓	✓	✓
WORK_PER_THREAD _FACTOR	✓	✓	✓	✓

Table 7.4: Parameters from Scan that is used with the autotuners

	OpenTuner	**Kernel Tuner**	**CLTune**	**KTT**
BLOCK_SIZE	✓	✓	✗	✓
GRID_SIZE	✓	✓	✗	✓
PRECISION	✓	✓	✗	✓
UNROLL_LOOP_1	✓	✓	✗	✓
UNROLL_LOOP_2	✓	✓	✗	✓
USE_FAST_MATH	✓	✗	✗	✗
OPTIMIZATION _LEVEL_HOST	✓	✗	✗	✗
OPTIMIZATION _LEVEL_DEVICE	✓	✗	✗	✗
MAX_REGISTERS	✓	✗	✗	✗
GPUS	✓	✗	✗	✗

Table 7.5: Parameters from Stencil2D that is used with the autotuners

	OpenTuner	**Kernel Tuner**	**CLTuner**	**KTT**
GPUS	✓	✗	✗	✗

7.2 Systems Used for Testing

The systems used for testing were one computer with a NVIDIA GeForce GTX 980 GPU, one computer with a NVIDIA Titan RTX GPU, one GPU from the NVIDIA DGX-2 system, a IBM Power System AC922 with four GPUs and a system with 20 NVIDIA Tesla T4 GPUs.

7.2.1 NVIDIA GeForce GTX 980 Based System

This section about GeForce GTX 980 is taken from my specialization project which can be found as an attachment to the 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 7.6. *Appendix C System Information* has additional information about the system in some listings. Listing C.1 shows the topology and Listing C.2 shows that the system can not use NVLinks. Extra GPU and CPU information can be found in Listing C.3 and C.4 respectively.

Table 7.6: Hardware specification for NVIDIA GeForce GTX 980 based computer.

	GeForce GTX 980 system
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

7.2.2 NVIDIA Titan RTX Based System

This section about Titan RTX is taken from my specialization project which can be found as an attachment to the 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 7.7. More information about the system can be found in listings in *Appendix C System Information*. Listing C.5 shows the topology, Listing C.6 shows that the graphics card can have two NVLinks, Listing C.7 displays extra GPU information and Listing C.8 shows extra CPU information.

Table 7.7: Hardware specification for NVIDIA Titan RTX based computer.

	Titan RTX system
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

7.2.3 IBM Power System AC922

This section about IBM Power System AC922 is taken from my specialization project which can be found as an attachment to the thesis.

The Power AC922 system is of the model 8335-GTH, which among other things means that it has air cooling instead of water cooling [46]. The system has two POWER9 processors and four NVIDIA Tesla V100-SXM2 GPUs with 32 GB memory each. The system is named Yme and the hardware specifications are shown in Table 7.8. See Listing C.11 and C.12 in *Appendix C System Information* for GPU and CPU information respectively. The listings show that the SXM2 model of the Tesla V100 has a power limit of 300W.

The Power AC922 has two GPUs connected to each processor with three NVLink 2.0 bricks between. There are also three NVLink 2.0 bricks between the two GPUs connected to each CPU. Figure 3.5 from *2 Background* shows an illustration of the architecture. For the printed topology see Listing C.9 in *Appendix C System Information*. All four GPUs in this system have all six NVLinks 2.0 bricks active. This can be seen in Listing C.10. The system is divided into two NUMA nodes where each node has one CPU and two GPUs.

Table 7.8: IBM Power System AC922 hardware specification

	Power AC922 (name: Yme)
CPU	2x 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 128 GB GPU memory
GPU	4x NVIDIA Tesla V100-SXM2, 32 GB. 640 Tensor Cores.
GPU-GPU Interconnect	NVLink 2.0
CPU-GPU Interconnect	NVLink 2.0
OS	Red Hat Enterprise Linux 7.6

7.2.4 NVIDIA DGX-2

This section about DGX-2 is taken from my specialization project which can be found as an attachment to the 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 [60]. In Listing C.14 from *Appendix C System Information* 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 [61] 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 C.13. The system is divided into two NUMA nodes with 8 GPUs per node. The graphic cards has 640 Tensor Cores and more specifications can be seen in Table 7.9.

Table 7.9: NVIDIA DGX-2 hardware specification.

	DGX-2 (name: Heid)
CPU	2x 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	16x NVIDIA Tesla V100-SXM3, 32GB. 640 Tensor Cores.
GPU-GPU Interconnect	NVSwitch
CPU-GPU Interconnect	PCIe
OS	Ubuntu 18.04.3

7.2.5 NVIDIA Tesla T4 Based Multi GPU System

The system with 20 Tesla T4 graphic cards has two Intel Xeon Gold 6230 CPUs and 378 GB of main memory. The GPUs have 16 GB memory each, bringing the total GPU memory for the system to 320 GB. More system information can be found in Table 7.10 and in *Appendix C System Information* with CPU information in Listing C.19, GPU information in Listing C.17 and NVLink information in Listing C.16. Topology information can be found in Figure C.1 and Listing C.18.

Table 7.10: Tesla T4 based multi GPU system hardware specification.

	Tesla T4 Based Multi GPU System (name: Selbu)
CPU	2x 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	20x NVIDIA Tesla T4, 16GB. 320 Tensor Cores.
GPU-GPU Interconnect	PCIe
CPU-GPU Interconnect	PCIe
OS	Ubuntu 18.04.5

Figure 7.1 illustrates the system topology that were derived from the command results.

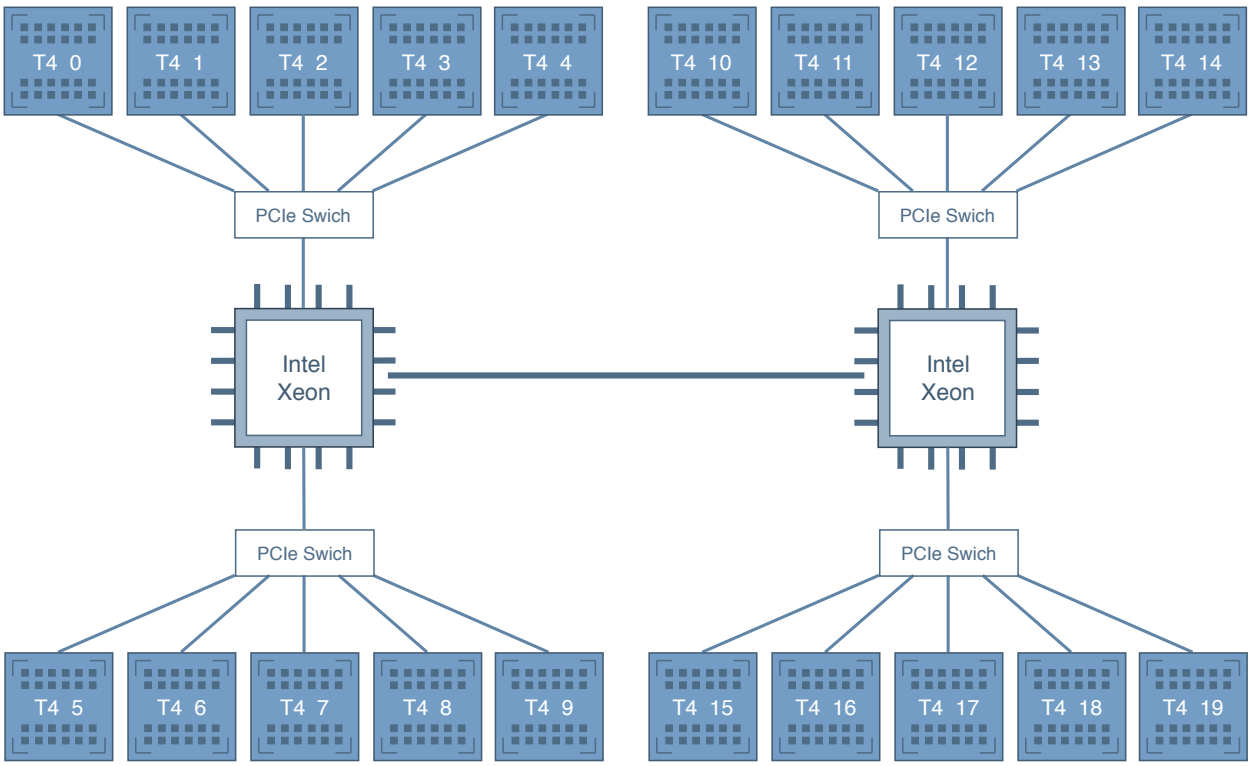

Figure 7.1: Topology of the Tesla T4 based machine.

7.3 What to Test and Why

The following points are the main parts I wanted to cover when performing tests with the autotuners.

- **Ensure that the parameterized algorithms runs with different autotuners, and evaluate the autotuner results.**

  The benchmark suite needs to be able to be used with different autotuners. This will also help with evaluation popular autotuners. This point can be fulfilled by running different search techniques with the benchmarks.

- **Evaluating the parameters and the search space.**

  Find out if the parameter values varies on different machines. This can be done by running brute force for algorithm on every system

7.3.1 What to Run From Each Autotuner

Knut Kirkhorn and I decided to run the same set of tests for our benchmarks.

Opentuner has an implementation for every parameter, but does not have an option for brute force. It is not an autotuner that is supposed to use for problems where the best combination can be found in a appropriate time frame. For OpenTuner we wanted to run all benchmarks for all systems, including multi-GPU systems, with the default search technique: `AUC Bandit Meta Technique`.

We wanted to run brute force for Kernel Tuner, since this is the autotuner that supports the most parameters of the autotuners with brute force technique. We also wanted to test another search technique for Kernel Tuner, and decided to try the Genetic Algorithm technique with maximum iterations of 50 and population size of 10.

For KTT we wanted to run brute force search and an additional search technique. We chose MCMC for this technique.

Since CLTune has the same possible parameters for the benchmarks as KTT, but without possibility for Scan, we did not feel the need to run brute force search for this autotuner. We chose to run multiple iterations of PSO and Annealing search with swarm sizes 1, 5 and 20 for PSO, and maximum temperature values 0.1, 2 and 10 for Annealing search. The CLTune tests were performed with MD5 Hash and SpMV.

7.4 Running the Tests

The tests were run using Docker, with the Dockerfiles listed in *Appendix D Dockerfiles*, on all the systems except the IBM Power System AC922, which does not have Docker.

There were some problems with setting up packages and dependencies when trying to run the benchmarks on the IBM Power System AC922. There were some outdated packages and other small issues, which eventually lead to Kernel Tuner not being able to run on this machine.

A guide to how to run the tests can be found in *Appendix D Set Up* and *Appendix B BAT User Guide*. If Docker is not available on the machine, the steps in the Dockerfiles have to be followed manually.

Chapter 8

Experiments and Discussion

This chapter describes and discusses the results from running the benchmarks. It also contains an evaluation of the autotuners used and an evaluation of our BAT framework.

8.1 Evaluation of Autotuning Results

All results can be found in folder `BAT-results` in the attached code.

Several of the results are based on the average of multiple runs. The brute force results however are based on one run, since they used longer time.

8.1.1 KTT

For KTT, all the benchmarks were run with search strategy MCMC. This strategy did not restrict the search space and ended up with running just the same amount of combinations as the brute force strategy did. After checking the results, I found out that every combination is run once, like brute force, but in a different order.

KTT was also used for benchmarking with brute force for problem size 4. All the benchmarks except for MD5 Hash finished in an acceptable time frame. MD5 Hash even used too long time when setting the problem size to 1. It could probably still have finished in under a day with problem size 1 if I had not stopped it.

Evaluating block size parameter for BFS

I wanted to see how the time varied for the values of block size in BFS. Figure 8.1 shows the time used when the value of block size varies. The circles on the graph are block sizes that are divisible by 32. Block size 16 is also marked. Values below around 15 are outside of the graph because they use a lot of time.

In the figure, we can see that the values divisible by 32 seems to be the ones that leads to the shortest run time for block sizes up to around 400. After a while we can see that the values divisible by 32 actually leads to higher run time.

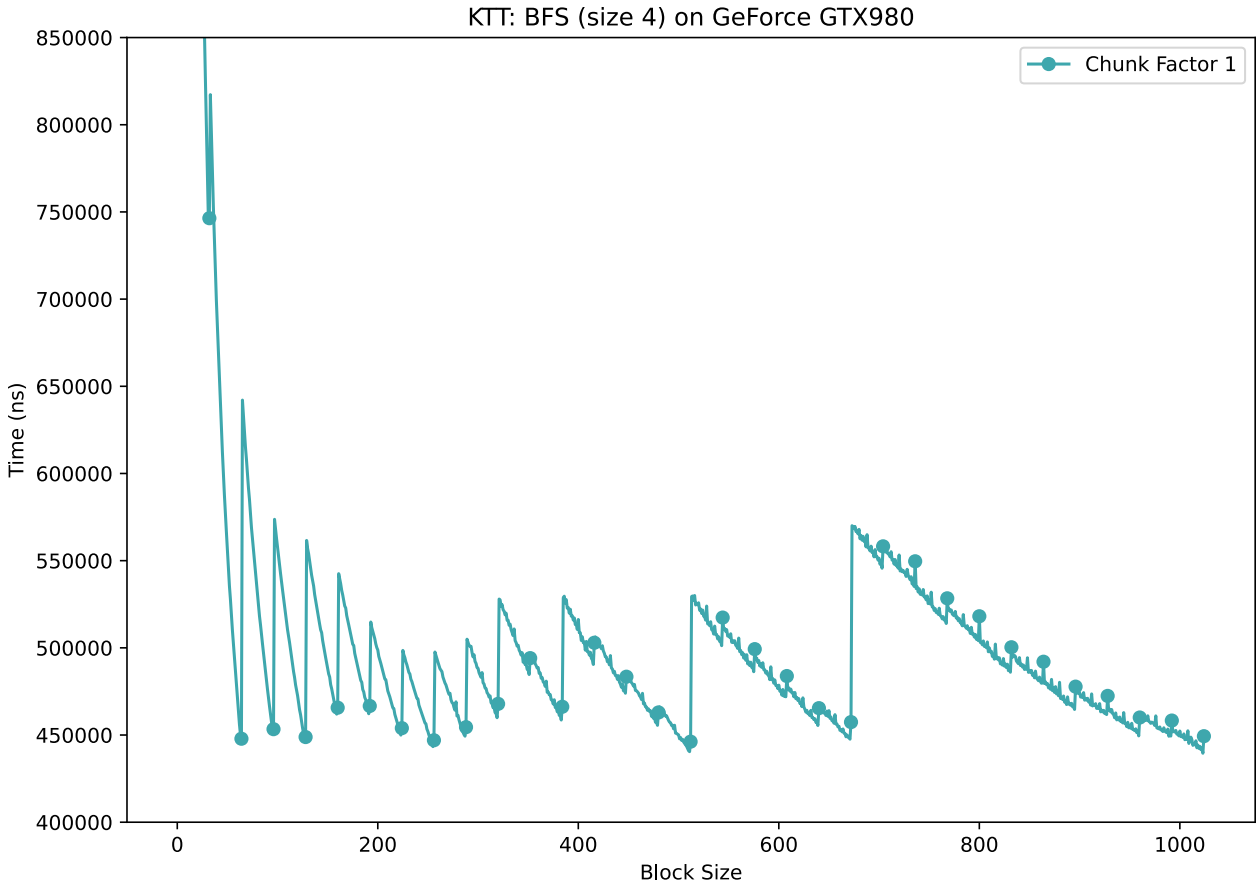

Figure 8.1: KTT: BFS (size 4) on GeForce GTX980. Chunk factor 1.

When zooming in on the graph in Figure 8.2 and Figure 8.3, we can see that the value divisible by 32 leads to higher time than the block size just before.

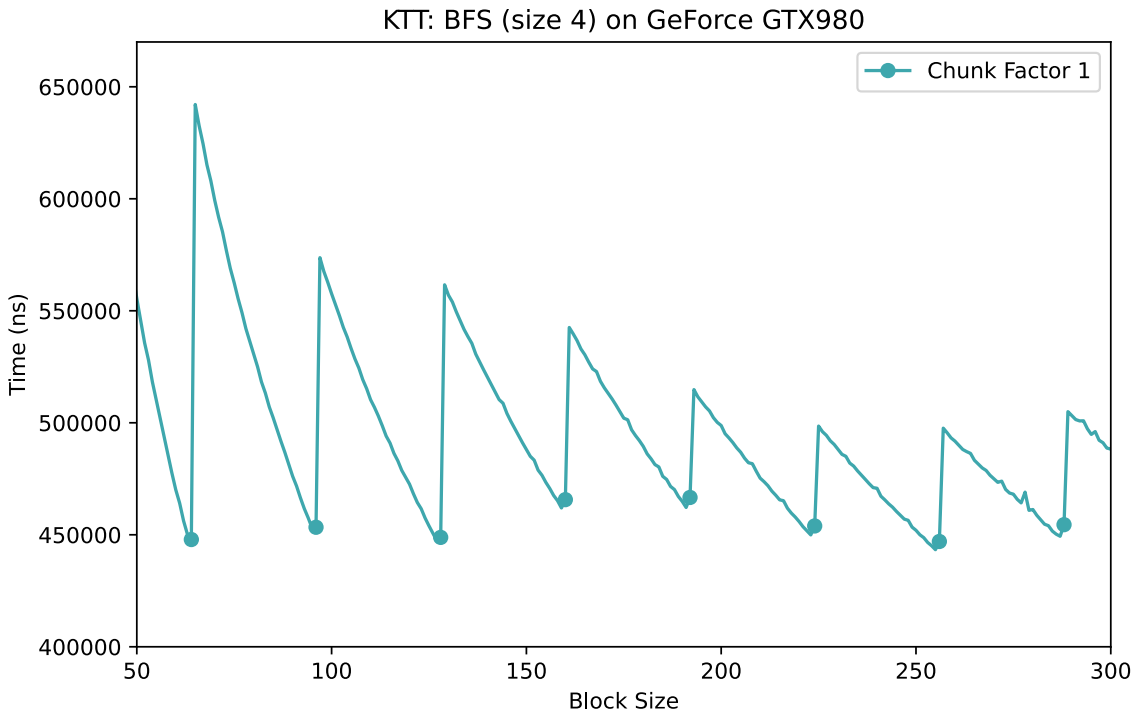

Figure 8.2: KTT: BFS (size 4) on GeForce GTX980. Chunk factor 1 - zoomed in version 1.

In Figure 8.3, the block size 672 is marked with a circle, and is divisible by 32. The block size just before, 671, shows lower time than for 672.

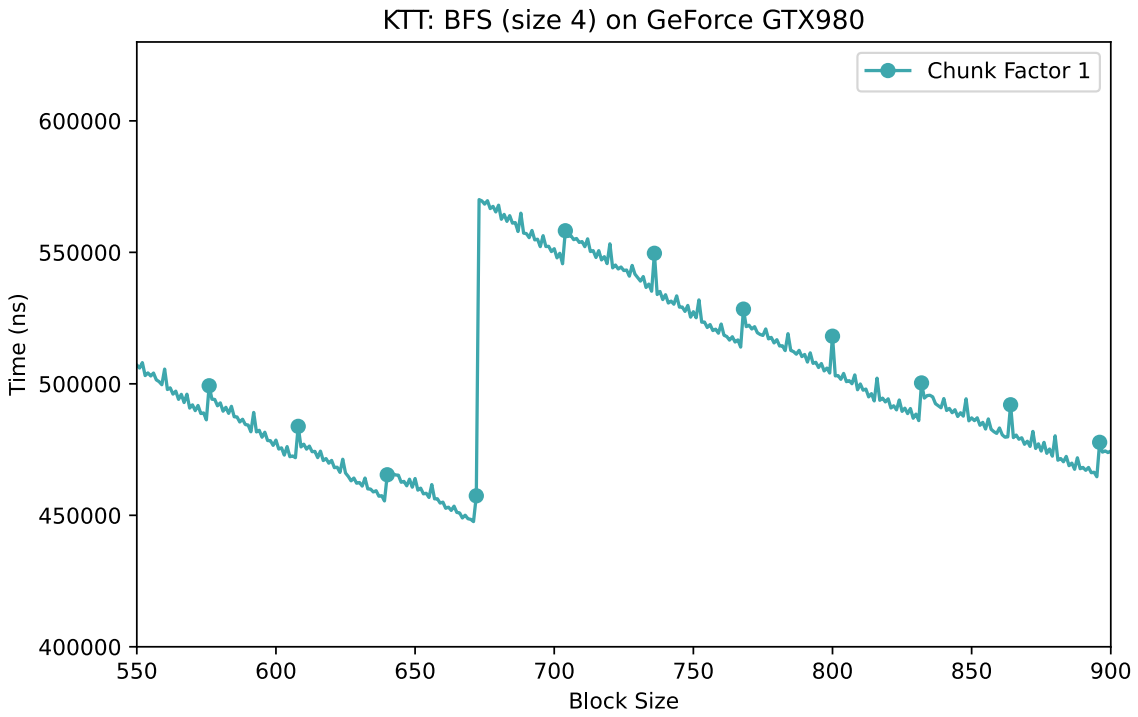

Figure 8.3: KTT: BFS (size 4) on GeForce GTX980. Chunk factor 1 - zoomed in version 2.

When comparing the time for the different chunk factors, we can see in Figure 8.4, that generally a chunk factor of 2 gives better time. This can be seen easier in the zoomed in versions in Figure 8.5 and Figure 8.6.

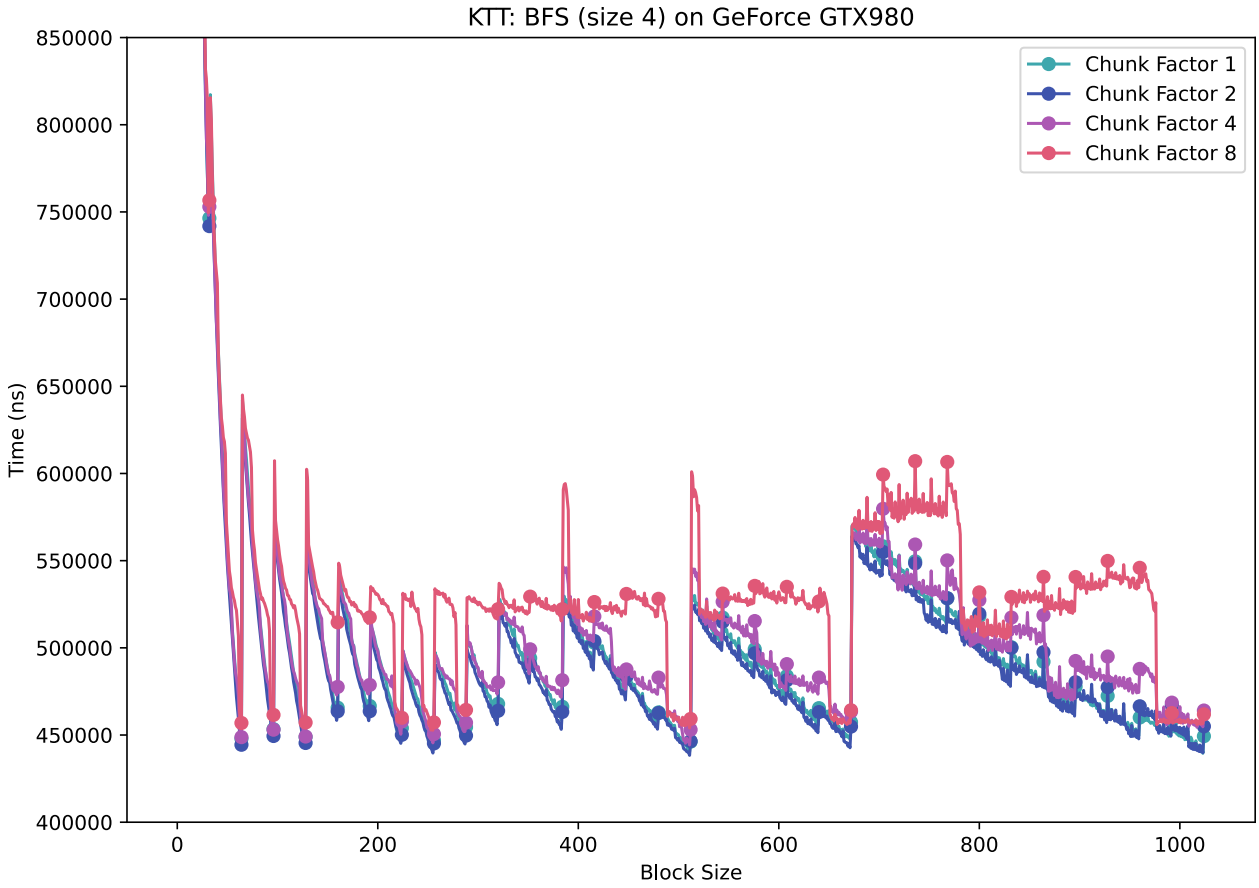

Figure 8.4: KTT: BFS (size 4) on GeForce GTX980. All chunk factors.

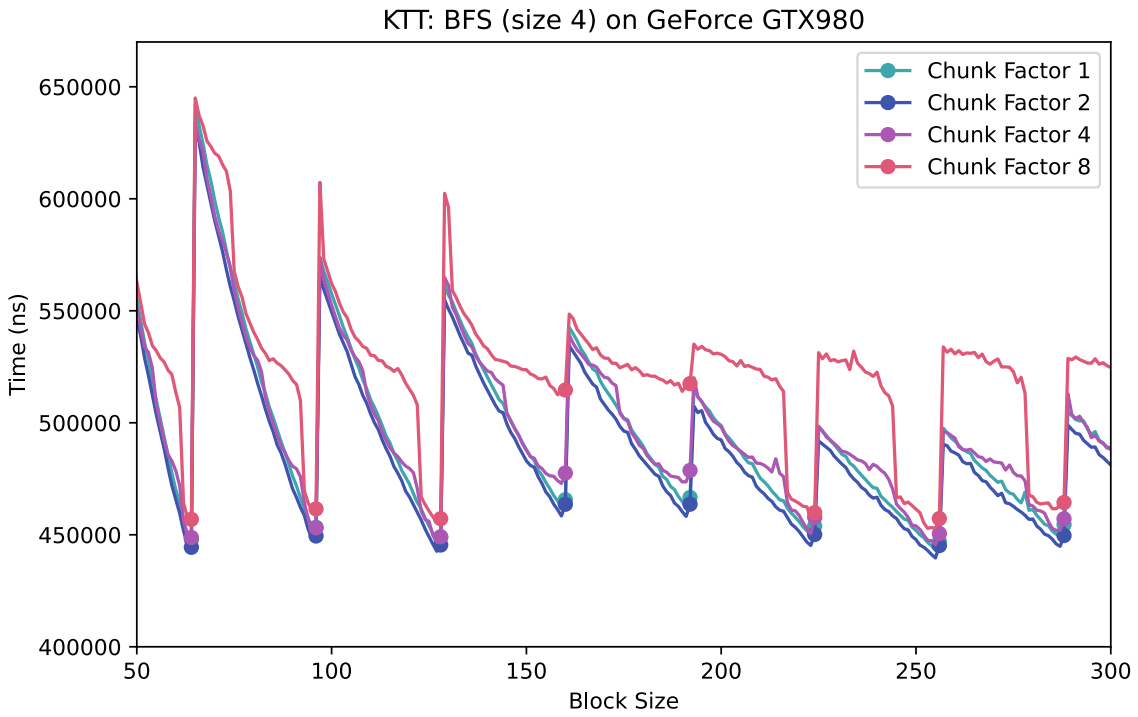

Figure 8.5: KTT: BFS (size 4) on GeForce GTX980. All chunk factors - zoomed in version 1.

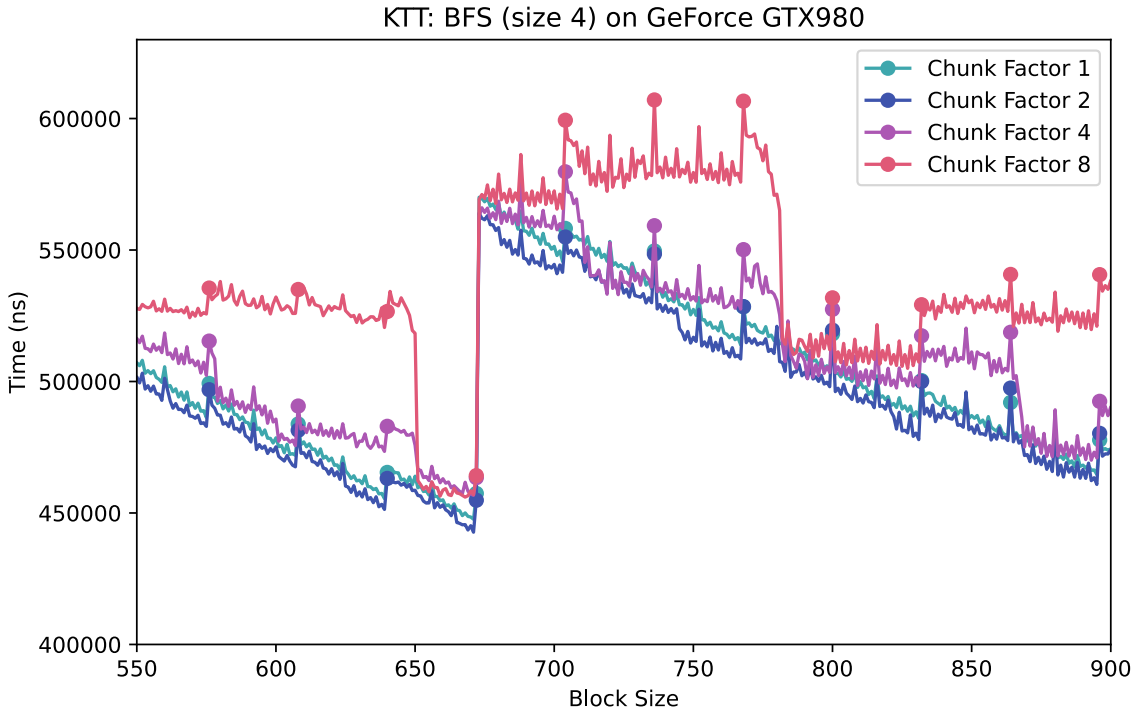

Figure 8.6: KTT: BFS (size 4) on GeForce GTX980. All chunk factors - zoomed in version 2.

To find out why the block size just before the one divisible by 32 is the one that leads to the best time, I decided to see how the time varied with different block sizes on different machines. This is done with chunk factor 2 on all of the systems in Figure 8.7. Chunk factor 2 were chosen because the results showed that this was the best chunk factor for all systems. Figure 8.8 shows a zoomed in version. These figures shows that the time varies with block size values differently on different systems. All graphs except for the GeForce GTX980 shows a trend of the block sizes divisible by 32 being the block sizes that leads to lower time used. It is unclear why it is different for the GeForce GTX980, and needs to be researched more to reach a conclusion.

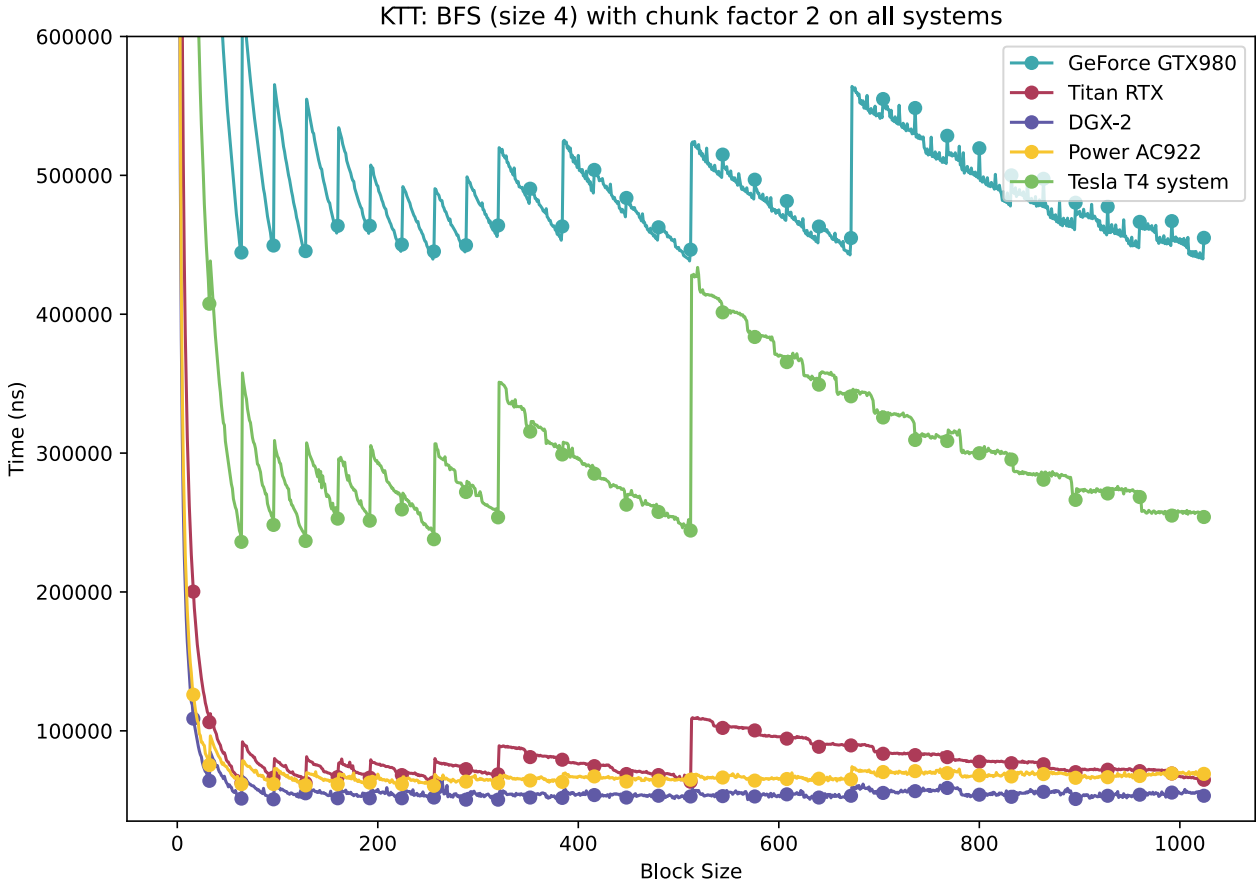

Figure 8.7: KTT: BFS (size 4) on all systems. Chunk factor 2.

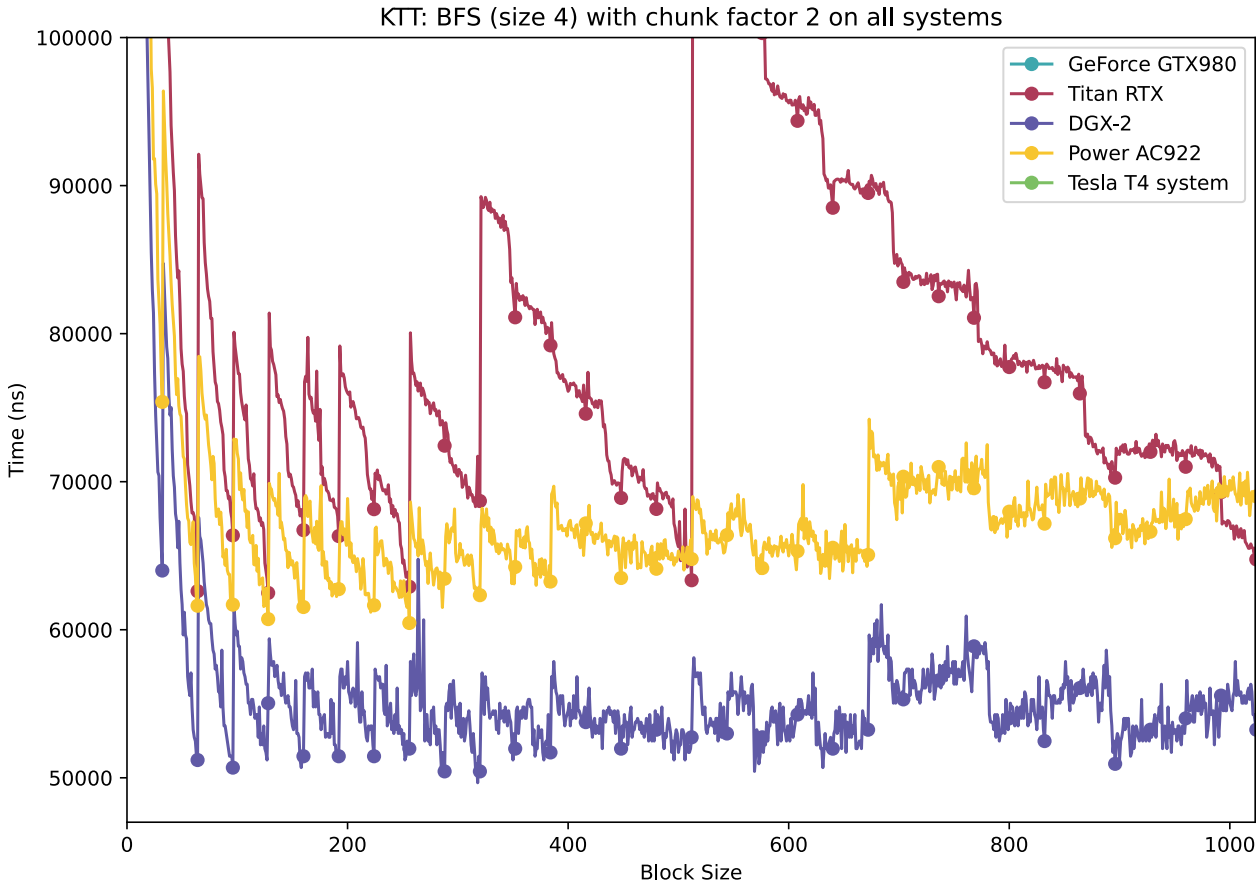

Figure 8.8: KTT: BFS (size 4) on all systems. Chunk factor 2 - zoomed in.

In Figure 8.7 we can see that the system with the GeForce GTX980 is the one that uses the most time for KTT. Since this is a more common, not that expensive, GPU, this is expected. One thing that might not be obvious before seeing the results, is that KTT uses less time on one GPU on the DGX-2 system than on one GPU on the IBM Power System AC922. The two systems has almost the same GPU, and in my specialization project I found that there was almost no difference in performance between them. The reason why one could think that the Power AC922 could use less time is because it has NVLink between CPU and GPU, while the DGX-2 has PCIe between CPU and GPU. This can make us assume that the CPUs in the DGX-2 might have better performance than the CPUs in Power AC922.

Evaluating Parameters for SpMV

I wanted to see how the best parameter values for SpMV differed for the different machines. By looking at the best results, I found out that to loop unroll or not varied much over the machines, and that more extensive analysis for this parameter is needed.

I found out that precision equal to 32 was the best on every machine for the tests. The best SpMV matrix format also varied some for the machines, but it seems that a format of 0, 3 or 4 is the best. These are the formats for ELLPACK-R, CSR Normal Vector and CSR Padded Vector, none for the CSR Scalar version. More analysis need to be done, but with

this evidence there might be better to use ELLPACK-R or a type of SCR Vector format for getting the lowest runtime.

Evaluating parameters for Scan

For the brute force version of scan, 32 is shown to be the best value for precision for all the systems. Loop unrolling shows a lot of varying in value. A more extensive analysis should be performed on this parameter to conclude which value is the best.

There were some differences in best block and grid size, but all of the block and grid sizes were over 128.

8.1.2 CLTune

All of the results ran through for CLTune. The results for the CLTune running should be more analyzed, but a quick analysis shows that for SpMV a precision of 32 is generally best.

An analysis of MD5 Hash on DGX-2 for Annealing search showed that generally a block size over 700 gave a better time. The loop unroll and inline values varied a lot, so a deeper analysis should be done to figure out which values are the best.

8.1.3 Kernel Tuner

For the Kernel Tuner autotuning, brute force did not finish in an acceptable time frame for BFS, SpMV and MD5 Hash. All the tuning for BFS, SpMV, Scan and MD5 Hash finished for the genetic algorithm tests.

BFS

Since KTT did not have texture memory support, BFS's texture memory values can be analyzed with the genetic algorithm results from Kernel Tuner. After analyzing the results, I found out that the best texture memory value vary a lot, and more analysis needs to be done to conclude anything.

Scan

When analyzing the Scan results I found that block size and grid size of 128 or 256 gave generally the best run time, and a precision of 32 was always the best.

8.1.4 OpenTuner

All the OpenTuner tests ran through.

MD5 Hash

Analyzing the MD5 Hash benchmark result showed that work-per-thread factor varies a lot over the different machines. This was also the situation for inline and loop unrolling. More research needs to be done for concluding anything for these parameters.

Scan

All the results showed that a GPU number of 1 is the best number of GPUs. The compiler option values changed a lot for the machines, so these parameters should be analyzed more.

Stencil 2D

The multi-GPU Stencil 2D benchmark were run on the IBM Power System AC922 and the Tesla T4 system.

Figure 8.9 shows the results when using problem size 1, and Figure 8.10 shows the results for size 4. We can see that the results are quite similar of both sizes, and the best number of GPUs is 1. We can see that for the IBM Power System AC922 machine, using 3 GPUs is almost as good as using 1 GPU.

If the goal is to run the algorithm as fast as possible, it is not efficient to split the work of these algorithms on these particular systems.

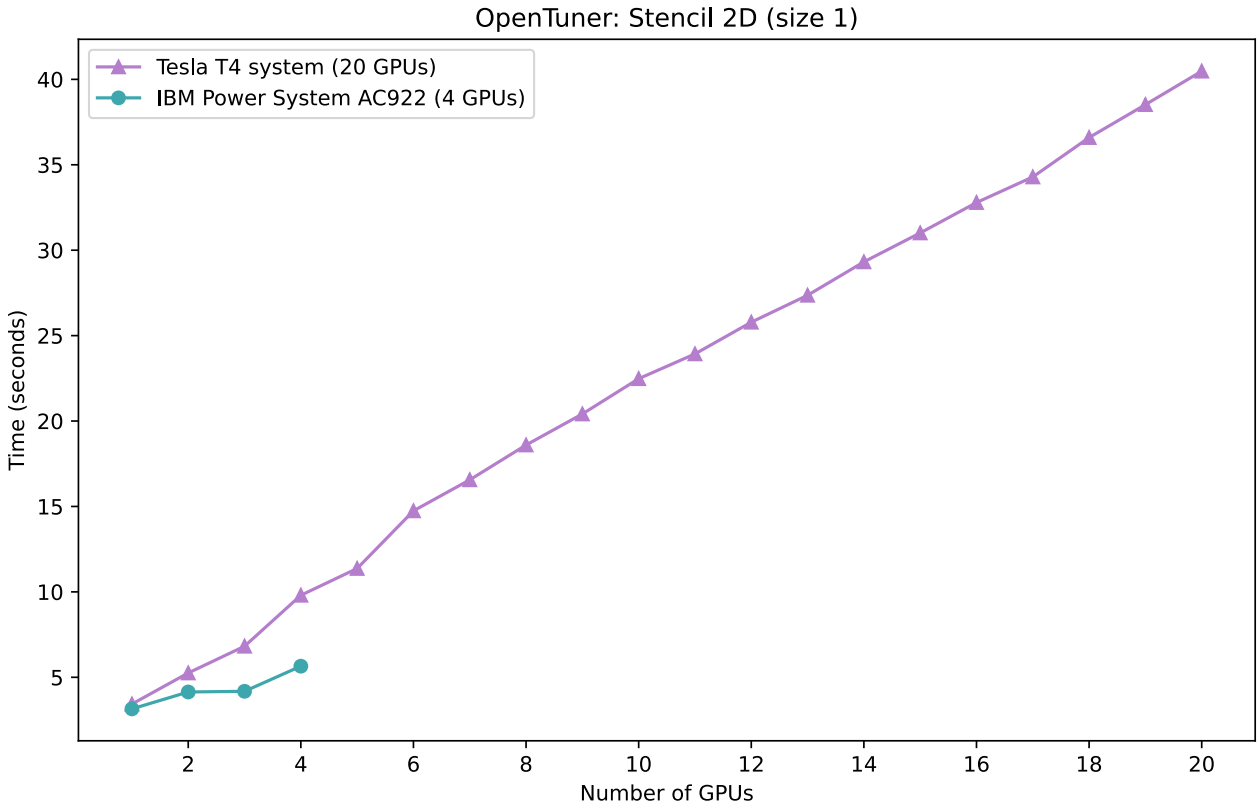

Figure 8.9: OpenTuner: Stencil 2D (size 1).

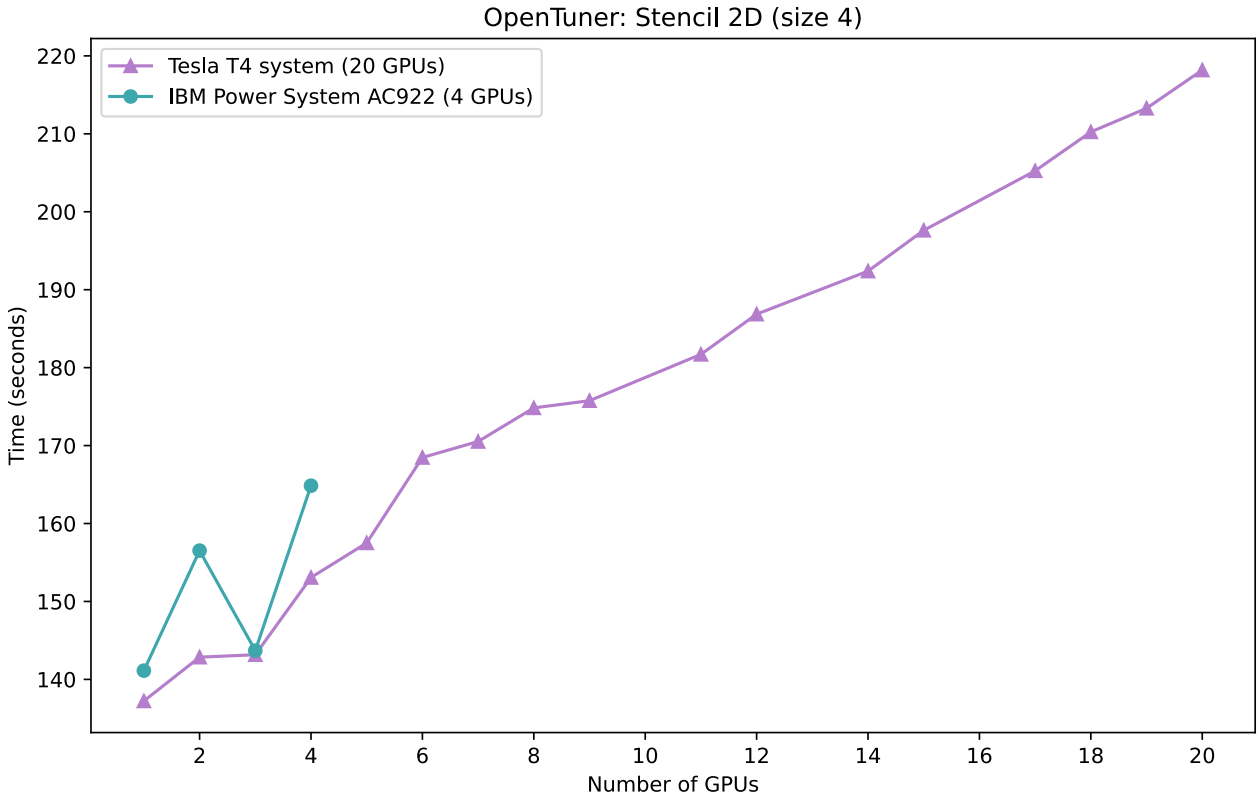

Figure 8.10: OpenTuner: Stencil 2D (size 4).

In my specialization project, I found out that Stencil 2D got better GFLOPS performance, for the more GPUs that were used for the IBM Power System AC922. This means that it could have been interesting to see how the graph would have been if the measuring unit had been GFLOPS instead of seconds.

8.2 Evaluation of the Autotuners

The original plan was to focus more on multi-GPU testing than the results shows, but most autotuners we used for testing did not support multi-GPU algorithms.

Since this thesis did not contain a way to directly compare the autotuners with a score, or something similar, the evaluation of the autotuners contains a description of what they are lacking and what they are great at.

OpenTuner

OpenTuner is a very general autotuner with several options for tuning. It is great that parameters can be included in the compile and run commands.

What OpenTuner is lacking is having enough documentation. There is no list of tuning techniques with explanations, and it is almost necessary to look at the examples to know how to set up the tuning. But after finding out how, it was quite simple to set up.

A positive aspect of OpenTuner is that it supports multi-GPU running, and all the parameters in the parameterized benchmarks works with OpenTuner.

Something I would like OpenTuner to add, is an option for brute force search. I would also like if OpenTuner had a built-in way to sett parameter constraints.

Kernel Tuner

The fact that Kernel Tuner supports tuning both singular kernels and host code including kernel code, is very practical. One negative part is that when tuning host code, it took way longer time than for tuning a singular kernel.

Kernel Tuner has good documentation and prints helpful information during the tuning. Most of the parameters from BAT worked with Kernel Tuner, but it was not possible to run the multi-GPU tuning benchmarks.

Kernel Tuner supports texture memory and compiler options for a kernel, but unfortunately not as parameters.

CLTune

CLTune has a bit more complimented setup than the other autotuners that is implemented in Python.

One part that CLTune could improve on is to give better error messages. Sometimes there is only a message that something is wrong, without any additional information.

A negative part of CLTune, is that it can only tune single-GPU non-templated kernels without support for texture memory. A positive to make up for this is that the tuning process is relatively fast. The search techniques works well, but I would like it if there was more techniques than four.

A part that CLTune is lacking, is correctness verification when there are multiple variations that can be correct. An example is when setting precision to single or double. The values cannot be compared because they are of different datatype.

KTT

The main improvement from CLTune to KTT is the possibility to run multiple kernels after each other.

One issue I had with KTT was that it was hard to see when there was an error or correctness verification failure in the printed info while tuning. It would have been easier to see if something was wrong if the text were red.

Some other issues i had with KTT were that the kernel launch parameters that is printed in the command line are wrong, and that I did not find a way to restrict the search space when testing for MCMC.

KTT has multiple of the same weaknesses as CLTune, with not supporting templated kernels, only be able to run on one GPU and few search techniques. KTT, like CLTune, is also lacking correctness verification when multiple variations can be correct.

8.3 Evaluation of BAT

I would like to revisit the requirements from *5. Plan for the Benchmark Suite* that were formulated in collaboration with Knut Kirkhorn. The requirements describes an ideal benchmark suite for autotuners.

The evaluation is only based on the algorithms I parameterized and tested, and the benchmark suite as a whole.

- **The benchmark suite should have:**
  - **HPC based benchmarks.**
    All of the algorithms used for benchmarks in BAT are common HPC algorithms from SHOC.
  - **Parameterized algorithms as benchmarks.**
    BFS, SpMV, MD5 Hash, Scan and Stencil 2D are parameterized with different amount of parameters, ranging from one to 10 parameters per algorithm.
  - **Varied selection of benchmarks with different degree of complexity and scope.** None of the algorithms are very complex, but they vary in degree of complexity. The BFS program consists of a small, simple kernel with some additional host code. The SpMV program has three different kernels to choose between (CSR scalar, CSR vector and ELLPACK-R). MD5 Hash has multiple kernels, but is still not that complex. Scan and Stencil 2D both has possibility for splitting work between multiple GPUs, which makes the benchmarks a bit more complex.
  - **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.**
    BAT consists only of CUDA kernels, with and without host code. Since SHOC also has OpenCL kernels, it could be possible to parameterize those too in the future.
  - **Benchmarks that can run on multi-GPU systems and distribute work on multiple nodes.**
    Scan and Stencil 2D can be run on multiple GPUs where the work is distributed among the nodes. They both have a parameter each for the number of GPUs to use when running the algorithm. Scan also has additional parameters.
  - **Support for different types of autotuners. If the autotuner does not support certain parameters or the autotuner only supports tuning of kernels, there should still not be a problem using the benchmarks.**
    Some changes had to be done to the benchmark suite to make alll autotuners work for it. The `extern C` option had to be added in front of some kernels. The templated kernels needed helper functions or removal of the templates to work with the autotuners that tunes kernels. Some parameters needed the option to not be included in the kernel, like texture memory that cannot be a parameter at kernel launch for CLTune and KTT.

- **Benchmarks that have been well tested with different autotuners and on different machines.**
  The benchmarks have been tested with OpenTuner, Kernel Tuner, CLTune and KTT on five different GPUs. Scan and Stencil 2D have also been tested on two multi GPU machines, IBM Power System AC922 and a machine with 20 Tesla T4 GPUs. Scan in CLTune were the only implementation that was not possible because of variable input for the second kernel. The autotuners that were used for testing are different in what they tune, which makes them a good representative for autotuners generally.
- **Examples of how to use the benchmarks with autotuners.**
  The benchmark suite includes a folder, `tuning_examples`, with OpenTuner, Kernel Tuner, CLTune and KTT implementations for every benchmark in the benchmark suite (except for Scan in CLTune). This provides different versions of how to use autotuners with the benchmarks.
- **A way to compare autotuners with other autotuners.**
  Since this is a time consuming task, we decided to not focus on this task for this thesis. A scoring system and guidelines of how to compare them has to be defined. A potential way to do this can be with measuring relative speedup of tuned code.

- **The parameterized algorithms should contain:**
  - **Both full programs and single GPU kernels.**
    BAT includes both full programs with host and GPU code in folder `/src/programs`, and singular kernels in folder `/src/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.**
    The algorithms have variation in search space size, where the smallest is `NUMBER_OF_GPUS` for Stencil 2D and the largest is 327 680 (if maximum block size is 1024) for MD5 Hash. As described earlier in this chapter, some algorithms were not efficient to brute force. But most can be done in under a week. Something to add to the list of future work is creating a bigger search space for some algorithms. It is still good to have a variation of search spaces for testing with different autotuners.
  - **Parameters that potentially could have different values on different machines or architectures.**
    Some of the parameters showed that they could have different best value for different systems. This was especially highlighted in the graphs for BFS block size. Another parameter that seems to change when the system changes is the SpMV storing format. The benchmarking results should be analyzed more to conclude further.
  - **Some benchmarks with possibility for restrictions or constraints on the parameters possible values.**
    Two of the algorithms have restrictions. In the Scan algorithm, `GRID_SIZE` has to

be smaller or equal to `BLOCK_SIZE`. For SpMV, when `FORMAT` is 3 or 4 `BLOCK_SIZE` has to be divisible by 32. SpMV also has a restriction for when `FORMAT` is less then 3, `UNROLL_LOOP_2` can only be 0, because this loop is in a kernel that is only used when format is 3 or 4.

- **The benchmark suite should be user friendly by being:**
  - **A well structured project.**
    The project is structured with a source folder that contains the kernels and programs. This makes it easier to only extract the parameterized code if this is preferable. The project also contains a folder with tuning examples that has the autotuner implementations. A `main.py` file can be used to run the benchmarks with the autotuners in a simple way. New autotuners can easily be added.
  - **Easy to use.**
    With the `main.py` file and the `config.json` files, the benchmarks are easy to run with a simple command. The results are saved in a result folder that makes it effortless to gather the results after running multiple benchmarks. New autotuners can be added as described in *Appendix B BAT User Guide*. Dockerfiles are also provided with the project, to make the benchmark suite easier to setup and use. These measures taken for making BAT easy to use, were quite helpful when running the benchmark tests.
  - **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.**
    We added a `readme.md` file to the project, which can be seen in *Appendix B BAT User Guide*. This file describes what BAT is, how to use the benchmark suite and who benefits from using it.

I also want to evaluate if the user needs are met after making the benchmark suite:

- **A user that wants to have a set of parameterized kernels or programs to test an autotuner with. This user only wants the `kernels` or `programs` folders.**

  This can easily be achieved by only using the `kernels` or `programs` folder, that can be extracted from the project.

- **A user that wants to use the whole benchmark suite and add a new autotuner to the `tuning_examples` folder.**

  How to do this is described in *Appendix B BAT User Guide*. The `main.py` script were very useful when running the benchmarks. There was no need for going up and down in the file structure to run benchmarks and save results.

- **A user that want to use the whole benchmark suite and only use the autotuners that already are added for running benchmark tests on different machines or architectures.**

  This can be done by following the guide in *Appendix D Setup*. At least some of the parameters changes best values when the system changes.

At last, the research questions can be answered and the hypothesis can be evaluated. The hypothesis was that a benchmark suite for autotuners based on SHOC can fulfill the requirements listed earlier to eventually become an ideal benchmark suite.

At this point, the benchmark suite does not fulfill all requirements for an ideal benchmark suite. But many of the points are fulfilled or close to be. The main shortage is that the search space is not very big. It is possible to brute force to get the best values, without using too long time. Another thing that would have been practical to include in BAT is kernels for OpenCL. This was a part of the requirements for an ideal benchmark suite, but not realistically something to fulfill in this thesis.

Another point that is missing is to have a way to compare autotuners with other autotuners. Since we have implemented a CLI, it would be very easy to add a scoring system to BAT. Defining a scoring system and guidelines for comparing, are two time consuming tasks that would not be realistically to include in this thesis.

- **Is SHOC is a good benchmark suite to base the benchmark suite for autotuners on?** SHOC has suitable, well written HPC algorithms with possibilities for running the algorithms parallel. These are all traits that are very beneficial to BAT. One thing that could have been better is if the algorithms had more parts to replace with potential parameters to increase search space.

- **Will it be a lot of rewriting of the code to ensure that the benchmark is enough parameterized?** I did not rewrite much code when parameterizing. But in the future, it can become necessary to rewrite some code for adding more parameters.

- **Will this benchmark suite, that is based on SHOC, be able to fulfill most of the points from the ideal benchmark suite requirements list?** BAT did fulfill most of the points from the requirements list.

- **Will it be able to have a GPU and multi-GPU focus?** The Benchmark suite are very GPU focused. Two of the algorithms I parameterized also has a number of GPUs parameter, which makes it multi-GPU focused. Unfortunately, none of these parameters showed using multiple GPUs were better than using one GPU. In the future, I would like to implement additional algorithms with multi-GPU focus.

- **Will the benchmark suite work with different types of autotuners?** The benchmark suite works very well with different types of autotuners. BAT was tested with four different types, which should be a good representative of other available autotuners.

- **Will the parameters added during the parameterization have different optimal values for different systems?** Some of the parameters had different optimal values for different systems. This could especially be seen for block size in the BFS algorithm.

Chapter 9

Conclusion and Future Work

This thesis presented the benchmark suite **BAT: A Benchmark suite for AutoTuners**. BAT is a possible solution to the fact that there is no standardized benchmark suite for autotuners. Autotuners should have a such benchmark suite to easier compare and test them.

BAT is a benchmark suite with HPC based, parameterized algorithms in CUDA with GPU focus. It contains a varied selection of benchmarks of different complexity as both singular GPU kernels and programs with both host code and GPU code. The benchmarks can utilize multiple GPUs on one system, either by running the same program and computations on multiple nodes, or by splitting the work between nodes.

The system was developed together with Knut Kirkhorn. The multi-GPU machines we had access to was split between us when performing tests with the finished benchmark suite. I tested on an IBM Power System AC922 with four Tesla V100-SXM2 32 GB GPUs and a machine with 20 Tesla T4 GPUs. Kirkhorn tested on an IBM Power System AC922 with two Tesla V100-SXM2 16 GB GPUs and the DGX-2 with 16 Tesla V100-SXM3 32 GB GPUs. We also did tests 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. I tested on one GPU of DGX-2 while Kirkhorn tested on one GPU of the Tesla T4 based system.

The benchmark suite is tested with four different autotuners that differs in setup and how they tune. These are OpenTuner, Kernel Tuner, CLTune and KTT. All the benchmarks have been modified to suite a lot of different autotuners that have support for different parameter implementations. One example of this is that texture memory needs a possibility to be disabled because multiple autotuners that tunes kernels does not support using it, at least not as a tuning parameter.

BAT includes a `readme.md` file that contains documentation for the benchmark suite. This consists of a description of BAT, who can benefit from using it, and how a user can use it. The project contains examples of how to use the benchmarks with autotuners, in a `tuning_examples` folder. BAT also has a CLI that makes it easier to run autotuning with the benchmarks.

Evaluation of the parameterized benchmarks from BAT showed that especially the optimal block size for the BFS algorithm varies for different systems. This means that the parameter is good to tune for ensuring performance portability.

One of the research questions purposed was if SHOC is a good benchmark suite to base a benchmark suite for autotuners on. SHOC provided well written HPC algorithms with possibilities for parallel running. It worked well to base BAT on SHOC, but some challenges did appear throughout the thesis. The SHOC algorithms used templates on multiple kernels, which led to having to make helper kernels. Another challenge was that the search space did not get very big with the algorithms from SHOC. This can be solved in the future by doing more rewriting of the algorithms.

What BAT is missing is more analysis on how the parameters vary on different systems. Some of the results could not be analyzed within the time frame of this thesis.

Another feature it would have been nice to have for BAT is a way to compare the autotuners with a numeric value. This would need clear guidelines for comparison and a type of scoring. Results that could be compared for different autotuners could be time used for tuning or relative speedup for runtume of a tuned benchmark. A such scoring system could be easily added to BAT because of the CLI. This would be a great task for future work.

Other future work could be to perform parameter value analysis on other systems, and finding more autotuners to test the benchmark suite with, preferably more with support for multi-GPU running. It would also be interesting to compare the original values in SHOC to the results of autotuned programs to see the performance difference.

User testing of BAT would be helpful to actually find out how user friendly the benchmark suite is.

Other future work can be to parameterize the OpenCL versions of the algorithms from SHOC. There can also be done parameterization on the host code, even though BAT is focused on GPUs.

One practical feature autotuner developers could add to their tuning frameworks is the ability to split the tuning on different GPUs if the search technique allows it. This would have made the tuning process for brute force use significantly less time on the large multi-GPU machines.

Bibliography

[1] F. Petrovič, D. Střelák, J. Hozzová, J. Oľha, 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*, https://arxiv.org/abs/1910.08498, [Accessed April 27, 2020], Mar. 2020.

[2] Z. Wang, R. Egawa, R. Suda, and H. Takizawa, *Auto-tuning of Hyperparameters of Machine Learning Models*, https://www.researchgate.net/publication/329269354_Auto-tuning_of_Hyperparameters_of_Machine_Learning_Models, [Accessed September 2, 2020], Jan. 2018.

[3] S. Bouckaert, J. V.-V. Gerwen, I. Moerman, S. C. Phillips, J. Wilander, S. U. Rehman, W. Dabbous, and T. Turletti, *Benchmarking computers and computer networks*, http://www-sop.inria.fr/members/Thierry.Turletti/WP11.pdf, [Accessed June 12, 2020], May 2011.

[4] 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://dl.acm.org/doi/10.1145/1735688.1735702, [Accessed April 27, 2020], Mar. 2010.

[5] NVIDIA, *NVIDIA History*, https://www.nvidia.com/en-us/about-nvidia/corporate-timeline/, [Accessed July 23, 2020], 2020.

[6] NVIDIA Corporation, *CUDA C++ Programming Guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, [Accessed June 10, 2020], Jun. 2020.

[7] R. Crovella, *Why launch a multiple of 32 number of threads in CUDA?* https://stackoverflow.com/questions/26611241/why-launch-a-multiple-of-32-number-of-threads-in-cuda/26611959#26611959, [Accessed August 22, 2020], Oct. 2014.

[8] G.-J. V. den Braak, *Improving GPU performance - reducing memory conflicts and latency*, https://www.researchgate.net/publication/286174393_Improving_GPU_performance_-_reducing_memory_conflicts_and_latency, [Accessed August 23, 2020], Nov. 2015.

[9] MPI Forum, *MPI Forum*, https://www.mpi-forum.org/, [Accessed December 12, 2019], MPI Forum, 2019.

[10] The Open MPI Project, *Open MPI: Open Source High Performance Computing*, https://www.open-mpi.org/, [Accessed December 12, 2019], The Open MPI Project, 2019.

[11] NVIDIA Corporation, *IBM Spectrum MPI | NVIDIA Developer*, https://developer.nvidia.com/ibm-spectrum-mpi, [Accessed December 5, 2019], NVIDIA Corporation, 2019.

[12] MPICH, *MPICH | High-Performance Portable MPI*, https://www.mpich.org/, [Accessed December 12, 2019], MPICH, 2019.

[13] 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, [Accessed December 10, 2019], Aug. 2016.

[14] HowStuffWorks, *How PCI Express Works | HowStuffWorks*, https://computer.howstuffworks.com/pci-express.htm, [Accessed December 16, 2019], HowStuffWorks, 2019.

[15] N. Corporation, *NVLink High-Speed GPU Interconnect | NVIDIA Quadro*, https://www.nvidia.com/en-us/design-visualization/nvlink-bridges, [Accessed December 10, 2019], 2019.

[16] 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, [Accessed December 15, 2019], 2019.

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

[18] G. Dearth, V. Venkataraman (NVIDIA), *S8688: INSIDE DGX-2*, http://on-demand.gputechconf.com/gtc/2018/presentation/s8688-extending-the-connectivity-and-reach-of-the-gpu.pdf, [Accessed November 12, 2019], NVIDIA Corporation, Mar. 2018.

[19] Docker, *What is a Container? | App Containerization | Docker*, https://www.docker.com/resources/what-container, [Accessed April 28, 2020].

[20] Docker, *Docker overview | Docker Documentation*, https://docs.docker.com/get-started/overview/, [Accessed May 10, 2020], Apr. 2020.

[21] A. Iordache, *Containerized Python Development – Part 1*, https://www.docker.com/blog/containerized-python-development-part-1/, [Accessed September 2, 2020], Jul. 2020.

[22] NVIDIA, *NVIDIA Container Toolkit*, https://github.com/NVIDIA/nvidia-docker, [Accessed September 20, 2020], Sep. 2020.

[23] *GitHub: The SHOC Benchmark Suite*, https://github.com/vetter/shoc, [Accessed April 27, 2020], Apr. 2020.

[24] T. Cormen and D. Balkcom, *The breadth-first search algorithm*, https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/the-breadth-first-search-algorithm, [Accessed June 11, 2020], Khan Academy.

[25] V. Sangam, *N-ary tree or K-way tree data structure*, http://theoryofprogramming.com/2018/01/14/n-ary-tree-k-way-tree-data-structure/, [Accessed June 11, 2020], Jan. 2018.

[26] K. Spafford, *BFS - vetter/shoc Wiki*, https://github.com/vetter/shoc/wiki/BFS, [Accessed June 19, 2020], Feb. 2012.

[27] N. Bell and M. Garland, *Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors*, https://www.nvidia.com/docs/IO/77944/sc09-spmv-throughput.pdf, [Accessed June 19, 2020], 2009.

[28] S. Guo, Y. Dou, Y. Lei, Q. Wang, F. Xia, and J. Chen, *Designing Parallel Sparse Matrix Transposition Algorithm Using ELLPACK-R for GPUs*, https://link.springer.com/chapter/10.1007/978-3-662-49283-3_7, [Accessed June 19, 2020], Feb. 2016.

[29] K. Spafford, *Spmv - vetter/shoc Wiki*, https://github.com/vetter/shoc/wiki/Spmv, [Accessed June 19, 2020], Feb. 2012.

[30] Tutorialspoint, *Cryptography Hash functions*, https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm, [Accessed August 23, 2020].

[31] OWASP, *Password Storage Cheat Sheet*, https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html, [Accessed August 23, 2020], Aug. 2020.

[32] R. Rivest, *The MD5 Message-Digest Algorithm*, https://tools.ietf.org/html/rfc1321, [Accessed August 23, 2020], Apr. 1992.

[33] J. Meredith, *shoc/MD5Hash.cu - Github*, https://github.com/vetter/shoc/blob/master/src/cuda/level1/md5hash/MD5Hash.cu, [Accessed August 23, 2020], Aug. 2014.

[34] M. Harris, S. Sengupta, and J. D. Owens, *GPU Gems 3: Chapter 39. Parallel Prefix Sum (Scan) with CUDA*, https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda, [Accessed September 22, 2020], Aug. 2017.

[35] K. Spafford, *shoc/Scan.cu - Github*, https://github.com/vetter/shoc/blob/master/src/cuda/level1/scan/Scan.cu, [Accessed September 22, 2020], Sep. 2014.

[36] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, *High-Performance Code Generation for Stencil Computations on GPU Architectures*, http://web.cs.ucla.edu/~pouchet/doc/ics-article.12.pdf, [Accessed September 22, 2020], Jun. 2012.

[37] K. Spafford, *Stencil2D - vetter/shoc Wiki*, https://github.com/vetter/shoc/wiki/Stencil2d, [Accessed September 22, 2020], Feb. 2012.

[38] 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*, https://ieeexplore.ieee.org/document/7855909, [Accessed May 2, 2020], Feb. 2017.

[39] B. van Werkhoven, *Kernel Tuner: A search-optimizing GPU code auto-tuner*, https://www.sciencedirect.com/science/article/pii/S0167739X18313359, [Accessed April 27, 2020], Aug. 2018.

[40] B. van Werkhoven, *API Documentation - Kernel Tuner*, https://benvanwerkhoven.github.io/kernel_tuner/user-api.html, [Accessed April 27, 2020], 2016.

[41] C. Nugteren and V. Codreanu, *CLTune: A Generic Auto-Tuner for OpenCL Kernels*, https://arxiv.org/pdf/1703.06503.pdf, [Accessed April 29, 2020], Mar. 2017.

[42] C. Nugteren, *CLTune: Automatic OpenCL kernel tuning*, https://github.com/CNugteren/CLTune, [Accessed August 29, 2020], Jun. 2020.

[43] F. Petrovič, *KTT - Kernel Tuning Toolkit*, https://github.com/Fillo7/KTT, [Accessed August 29, 2020], Aug. 2020.

[44] TechPowerUp, *NVIDIA GeForce GTX 980*, https://www.techpowerup.com/gpu-specs/geforce-gtx-980.c2621, [Accessed December 11, 2019].

[45] NVIDIA Corporation, *Maxwell Architecture | NVIDIA Developer*, https://developer.nvidia.com/maxwell-compute-architecture, [Accessed December 16, 2019].

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

[47] 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, [Accessed November 5, 2019], Mar. 2018.

[48] NVIDIA Corporation, *TITAN RTX Ultimate PC Graphics Card with Turing | NVIDIA*, https://www.nvidia.com/en-us/deep-learning-ai/products/titan-rtx/, [Accessed December 10, 2019], NVIDIA Corporation, 2019.

[49] 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, [Accessed December 20, 2019], NVIDIA Corporation, 2018.

[50] NVIDIA Corporation, *NVIDIA T4 TENSOR CORE GPU*, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf, [Accessed October 20, 2020], NVIDIA Corporation, Mar. 2019.

[51] IBM Corporation, *IBM Power System AC922*, https://www.ibm.com/downloads/cas/6PRDKRJ0, [Accessed November 5, 2019], IBM Corporation, 2019.

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

[53] 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, [Accessed November 5, 2019], NVIDIA Corporation, Jul. 2019.

[54] 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, [Accessed November 11, 2019], NVIDIA Corporation, 2018.

[55] A. Rasch, M. Haidl, and S. Gorlatch, *ATF: A Generic Auto-Tuning Framework*, https://ieeexplore.ieee.org/document/8291912, [Accessed August 4, 2020], Feb. 2018.

[56] A. Sclocco, *TuneBench*, https://github.com/isazi/TuneBench, [Accessed August 4, 2020], Dec. 2017.

[57] M. Clark, *CUDA Pro Tip: Kepler Texture Objects Improve Performance and Flexibility*, https://developer.nvidia.com/blog/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility/, [Accessed May 5, 2020], Feb. 2013.

[58] NVIDIA, *NVCC :: CUDA Toolkit Documentation*, https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html, [Accessed October 29, 2020], Oct. 2020.

[59] F. Petrovič, *Kernel Tuning Toolkit - Documentation*, https://fillo7.github.io/KTT/classktt_1_1_tuner.html#a98f12acd6cabf05acc4085979573b86f, [Accessed October 20, 2020], Oct. 2020.

[60] Inspur Systems, *AGX-5 - Inspur Systems*, https://www.inspursystems.com/product/agx-5/, [Accessed December 15, 2019], 2018.

[61] 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, [Accessed December 20, 2019], Aug. 2018.

Appendix A

Parameter Research

This section contains tables with several parameters used in the examples in autotuners 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` respectively. The parameters from OpenTuner were found in the paper describing the framework: "OpenTuner: An Extensible Framework for Program Autotuning" by J. Ansel et al. [38]. This section was done in collaboration with Knut Kirkhorn.

Kernel Tuner

Table A.1: 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 A.2: 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 A.3: 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 A.4: 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 A.5: 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_precomputated_slopes`	[0, 1]	
`use_method`	[0, 1]	

Table A.6: 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 A.7: 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 A.8: 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 A.9: 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 A.10: 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 A.11: 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)]	

CLTune

Table A.12: 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 A.13: 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 A.14: 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 A.15: 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

KTT

Table A.16: 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 A.17: 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 A.18: 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 A.19: 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]	

OpenTuner

Table A.20: 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 A.21: 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

Appendix B

BAT User Guide

This appendix includes documentation for the project written in collaboration with Knut Kirkhorn.

README.md

> 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
```

Appendix C

System Information

The system information in this appendix is collected in collaboration with Knut Kirkhorn.

NVIDIA GeForce GTX 980 based system

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 C.1: Topology for GTX 980 system.

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

Listing C.2: 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 C.3: Information about the GTX 980 GPU when running the nvidia-smi command

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 C.4: Information about the CPU in the GTX 980 based system when running the lscpu command.

NVIDIA Titan RTX based system

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 C.5: 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 C.6: 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 C.7: Information about the Titan RTX GPU when running the nvidia-smi command.

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 C.8: Information about the CPU in the RTX Titan based system when running the lscpu command.

IBM Power System AC922

GPUs

```
-bash-4.2$ nvidia-smi topo -m
        GPU0    GPU1    GPU2    GPU3    CPU Affinity
GPU0     X      NV3     SYS     SYS     0-63
GPU1    NV3      X      SYS     SYS     0-63
GPU2    SYS     SYS      X      NV3     64-127
GPU3    SYS     SYS     NV3      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 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 C.9: GPU topology for Power AC922.

```
-bash-4.2$ nvidia-smi nvlink --status -i 0
GPU 0: Tesla V100-SXM2-32GB (UUID: GPU-8be05466-6c9b-4d91-a8c8-4ed680df64a5)
         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 C.10: NVLink status for Power AC922. The same results for all GPUs, where it says that all of the GPUs have six active NVLink connections.

```
-bash-4.2$ nvidia-smi
Wed Oct 14 01:02:39 2020
+-----------------------------------------------------------------------------+
| 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   36C    P0    39W / 300W |      0MiB / 32510MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  On   | 00000004:05:00.0 Off |                    0 |
| N/A   38C    P0    40W / 300W |      0MiB / 32510MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  On   | 00000035:03:00.0 Off |                    0 |
| N/A   35C    P0    39W / 300W |      0MiB / 32510MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  On   | 00000035:04:00.0 Off |                    0 |
| N/A   40C    P0    40W / 300W |      0MiB / 32510MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Listing C.11: Information about the GPUs in Power AC922 when running the nvidia-smi command.

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
```

Listing C.12: Information about the CPUs in Power AC922 when running the lscpu command.

NVIDIA DGX-2

There is no GPU topology for the DGX-2 presented in this thesis because the experiments uses only one GPU.

GPUs

```
ingunsu@heid:~$ 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
```

Listing C.13: NVLink status for one of the GPUs on the DGX-2. All 16 GPUs will show the same result, which is that the GPUs has six NVLink connections each.

```
ingunsu@heid:~$ nvidia-smi
Mon Oct 12 22:09:07 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.152.00    Driver Version: 418.152.00    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   43C    P0    90W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM3...  On   | 00000000:36:00.0 Off |                    0 |
| N/A   44C    P0    84W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM3...  On   | 00000000:39:00.0 Off |                    0 |
```

```
| N/A   46C    P0    53W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM3...   On  | 00000000:3B:00.0 Off |                    0 |
| N/A   47C    P0    54W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   4  Tesla V100-SXM3...   On  | 00000000:57:00.0 Off |                    0 |
| N/A   47C    P0    58W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   5  Tesla V100-SXM3...   On  | 00000000:59:00.0 Off |                    0 |
| N/A   46C    P0    52W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   6  Tesla V100-SXM3...   On  | 00000000:5C:00.0 Off |                    0 |
| N/A   43C    P0    85W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   7  Tesla V100-SXM3...   On  | 00000000:5E:00.0 Off |                    0 |
| N/A   58C    P0    96W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   8  Tesla V100-SXM3...   On  | 00000000:B7:00.0 Off |                    0 |
| N/A   31C    P0    48W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   9  Tesla V100-SXM3...   On  | 00000000:B9:00.0 Off |                    0 |
| N/A   47C    P0    53W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|  10  Tesla V100-SXM3...   On  | 00000000:BC:00.0 Off |                    0 |
| N/A   35C    P0    49W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|  11  Tesla V100-SXM3...   On  | 00000000:BE:00.0 Off |                    0 |
| N/A   49C    P0    52W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|  12  Tesla V100-SXM3...   On  | 00000000:E0:00.0 Off |                    0 |
| N/A   32C    P0    52W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|  13  Tesla V100-SXM3...   On  | 00000000:E2:00.0 Off |                    0 |
| N/A   31C    P0    48W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|  14  Tesla V100-SXM3...   On  | 00000000:E5:00.0 Off |                    0 |
| N/A   35C    P0    49W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|  15  Tesla V100-SXM3...   On  | 00000000:E7:00.0 Off |                    0 |
| N/A   34C    P0    52W / 350W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Listing C.14: Information provided about the GPUs in the DGX-2 when running the nvidia-smi command.

CPUs

```
ingunsu@heid:~$ 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 C.15: Information provided about the CPUs in the DGX-2 when running the lscpu command.

NVIDIA Tesla T4 based system

GPUs

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

Listing C.16: NVLink status for the first GPU on the Tesla T4 based system. No results means no 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 |
+-------------------------------+----------------------+----------------------+
|   1  Tesla T4            Off  | 00000000:1B:00.0 Off |                    0 |
| N/A   37C    P8    17W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla T4            Off  | 00000000:1C:00.0 Off |                    0 |
| N/A   38C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla T4            Off  | 00000000:1D:00.0 Off |                    0 |
| N/A   39C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   4  Tesla T4            Off  | 00000000:1E:00.0 Off |                    0 |
| N/A   39C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   5  Tesla T4            Off  | 00000000:3D:00.0 Off |                    0 |
| N/A   38C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   6  Tesla T4            Off  | 00000000:3E:00.0 Off |                    0 |
| N/A   38C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   7  Tesla T4            Off  | 00000000:3F:00.0 Off |                    0 |
| N/A   38C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   8  Tesla T4            Off  | 00000000:40:00.0 Off |                    0 |
| N/A   39C    P0    27W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   9  Tesla T4            Off  | 00000000:41:00.0 Off |                    0 |
| N/A   37C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

```
# nvidia-smi topo -m
        GPU0 GPU1  GPU2  GPU3  GPU4  GPU5  GPU6  GPU7  GPU8  GPU9  GPU10 GPU11 GPU12 GPU13 GPU14 GPU15 GPU16 GPU17 GPU18 GPU19 CPU Affinity
GPU0     X   PIX   PIX   PIX   PIX   NODE  NODE  NODE  NODE  NODE  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU1    PIX   X    PIX   PIX   PIX   NODE  NODE  NODE  NODE  NODE  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU2    PIX  PIX    X    PIX   PIX   NODE  NODE  NODE  NODE  NODE  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU3    PIX  PIX   PIX    X    PIX   NODE  NODE  NODE  NODE  NODE  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU4    PIX  PIX   PIX   PIX    X    NODE  NODE  NODE  NODE  NODE  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU5    NODE NODE  NODE  NODE  NODE   X    PIX   PIX   PIX   PIX   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU6    NODE NODE  NODE  NODE  NODE  PIX    X    PIX   PIX   PIX   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU7    NODE NODE  NODE  NODE  NODE  PIX   PIX    X    PIX   PIX   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU8    NODE NODE  NODE  NODE  NODE  PIX   PIX   PIX    X    PIX   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU9    NODE NODE  NODE  NODE  NODE  PIX   PIX   PIX   PIX    X    SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   0-19,40-59
GPU10   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS    X    PIX   PIX   PIX   PIX   NODE  NODE  NODE  NODE  NODE  20-39,60-79
GPU11   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   PIX    X    PIX   PIX   PIX   NODE  NODE  NODE  NODE  NODE  20-39,60-79
GPU12   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   PIX   PIX    X    PIX   PIX   NODE  NODE  NODE  NODE  NODE  20-39,60-79
GPU13   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   PIX   PIX   PIX    X    PIX   NODE  NODE  NODE  NODE  NODE  20-39,60-79
GPU14   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   PIX   PIX   PIX   PIX    X    NODE  NODE  NODE  NODE  NODE  20-39,60-79
GPU15   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   NODE  NODE  NODE  NODE  NODE   X    PIX   PIX   PIX   PIX   20-39,60-79
GPU16   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   NODE  NODE  NODE  NODE  NODE  PIX    X    PIX   PIX   PIX   20-39,60-79
GPU17   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   NODE  NODE  NODE  NODE  NODE  PIX   PIX    X    PIX   PIX   20-39,60-79
GPU18   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   NODE  NODE  NODE  NODE  NODE  PIX   PIX   PIX    X    PIX   20-39,60-79
GPU19   SYS  SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   SYS   NODE  NODE  NODE  NODE  NODE  PIX   PIX   PIX   PIX    X    20-39,60-79

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
```

Figure C.1: GPU topology for the Tesla T4 based system.

```
|   10  Tesla T4            Off  | 00000000:88:00.0 Off |                    0 |
| N/A   37C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   11  Tesla T4            Off  | 00000000:89:00.0 Off |                    0 |
| N/A   38C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   12  Tesla T4            Off  | 00000000:8A:00.0 Off |                    0 |
| N/A   38C    P0    25W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   13  Tesla T4            Off  | 00000000:8B:00.0 Off |                    0 |
| N/A   38C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   14  Tesla T4            Off  | 00000000:8C:00.0 Off |                    0 |
| N/A   38C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   15  Tesla T4            Off  | 00000000:B1:00.0 Off |                    0 |
| N/A   36C    P0    25W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   16  Tesla T4            Off  | 00000000:B2:00.0 Off |                    0 |
| N/A   36C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   17  Tesla T4            Off  | 00000000:B3:00.0 Off |                    0 |
| N/A   37C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   18  Tesla T4            Off  | 00000000:B4:00.0 Off |                    0 |
| N/A   37C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   19  Tesla T4            Off  | 00000000:B5:00.0 Off |                    0 |
| N/A   37C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Listing C.17: Information about the Tesla T4 GPUs when running the `nvidia-smi` command.

```
selbu:~$ nvidia-smi topo -p -i 0,1
Device 0 is connected to device 1 by way of a single PCIe switch.

selbu:~$ nvidia-smi topo -p -i 0,5
Device 0 is connected to device 5 by way of an on-CPU interconnect between PCIe host bridges
    .

selbu:~$ nvidia-smi topo -p -i 0,10
Device 0 is connected to device 10 by way of an SMP interconnect link between NUMA nodes.

selbu:~$ nvidia-smi topo -p -i 0,19
Device 0 is connected to device 19 by way of an SMP interconnect link between NUMA nodes.
```

Listing C.18: Information about the most direct path traversal between GPUs in the Tesla T4 based system when running the `nvidia-smi topo -p` command.

CPUs

```
selbu:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                80
On-line CPU(s) list:   0-79
Thread(s) per core:    2
Core(s) per socket:    20
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
```

```
CPU family:          6
Model:               85
Model name:          Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Stepping:            7
CPU MHz:             800.828
CPU max MHz:         3900.0000
CPU min MHz:         800.0000
BogoMIPS:            4200.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            28160K
NUMA node0 CPU(s):   0-19,40-59
NUMA node1 CPU(s):   20-39,60-79
```

Listing C.19: Information about the CPUs in the Tesla T4 based system when running the `lscpu` command.

Appendix D

Setup

The Docker images are built like in Listing D.1 and the containers are run like in Listing D.2. When running the Docker container, the benchmarks can be run like described in *Appendix B: BAT User Guide*.

```
docker build -f <path_to_dockerfile> -t <image_name> .
```

Listing D.1: Build Docker image.

```
docker run --rm -ti <image_name>
```

Listing D.2: Run Docker container.

When running code on the Tesla T4 based system, the Slurm queue system needs to be used. Listing D.3 shows the command for listing the Slurm queue, and Listing D.4 shows the command for reserving the whole machine and running the job. When running the job, a Docker container can be run inside the job.

```
squeue
```

Listing D.3: List Slurm queue.

```
srun -N1 -n1 -c80 -gres=gpu:T4:20 --partition=TDT4200 --time=02:00:00 -w
    selbu --pty slurm_init
```

Listing D.4: Run Slurm job for the whole Tesla T4 based machine.

Dockerfiles

```
# 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 D.5: Dockerfile for 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 \
    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 D.6: Dockerfile for OpenTuner

```
# 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 D.7: Dockerfile for 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 \
    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 D.8: Dockerfile for CLTune

Ingunn Sund

BAT: A Benchmark suite for AutoTuners

NTNU

Norwegian University of
Science and Technology