BAT: A Benchmark suite for AutoTuners

Ingunn Sund*, Knut A. Kirkhorn*, Jacob O. Tørring, and Anne C. Elster

Norwegian University of Science and Technology (NTNU), Trondheim, Norway
Contact: {jacob.torring, elster}@ntnu.no

Abstract. An autotuner takes a parameterized code as input and tries to optimize the code by finding the best possible values for a given architecture. To our knowledge, there are currently no standardized benchmark suites for comparing and testing autotuners. Developers of autotuners thus make their own when presenting and comparing autotuners. We thus present BAT, a Benchmark suite for AutoTuners with HPC-based parameterized GPU programs. CUDA programs and kernels from "The Scalable Heterogeneous Computing (SHOC) Benchmark" are parameterized. BAT 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. BAT contains 9 different HPC benchmarks that provide a large search space of autotuning parameters, and are modified to suite many different autotuners. BAT also includes a CLI that facilitates autotuning with the benchmarks.
Our benchmark suite is tested with four different autotuners, OpenTuner, Kernel Tuner, CLTune and KTT. They differ in setup and how they tune. The impact of the different benchmark parameters on the running time across architectures is analyzed. Test systems used include a DGX-2, IBM Power System AC922 with Tesla V100-SXM2 32 GB GPUs, an RTX Titan, a GeForce GTX 980 and a server with 20 Tesla T4 GPUs.

Keywords: Autotuning · Benchmarking · CUDA · SHOC

1 Introduction

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 and include GPUs.

In *autotuning*, one takes a parameterized code as input and tries to find the best possible values for the tuning parameters defined. The challenge is that optimal parameters might be different for different devices and architectures, and that the optimal parameters might not be possible to find in a reasonable time frame. The best values can also depend on the input format or input size.

* These two authors contributed equally

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 [Pet+20].

Benchmarking is often performed to reveal strengths and weaknesses in computer systems by running the same program on multiple systems. It can even be used to identify faults with software or hardware.

To our knowledge, there are currently no standardized benchmark suites for autotuners. Developers of autotuners often make their own benchmarks when presenting and comparing autotuners. However, it is cumbersome for the developers to make tunable benchmarks every time an autotuner is made. This can also lead to issues with benchmarks being tailored for a given autotuner. We have therefore developed a benchmarking suite for autotuners that future standards can build upon.

2 Background

GPUs (Graphics Processing Units) are specialized units in computers optimized for data parallelism. They were originally created for improving graphics computations. In recent years, they have also shown to be incredibly useful for many HPC and AI applications. 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.

GPU Programming 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.

In CUDA threads are grouped into thread *blocks* that execute together on a Streaming Multiprocessor. It is common to set threads per block (block size) as a multiple of 32, for best performance. This is because the threads in a block are divided into groups of size 32, known as warps, that executes the same instruction.

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 multiple iterations and partly 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 compiler options available for compilers like G++, GCC and NVCC that can be useful when optimizing execution time [NVI20].

SHOC (Scalable HeterOgeneous Computing) is a benchmark suite [Dan+10] consisting of a collection of benchmarks for both single and multi GPU systems. The benchmarks consist of standard HPC algorithms with both CUDA and OpenCL versions. 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. The following level 1 algorithms from SHOC are parameterized in BAT.

BFS (Breadth-First Search) algorithms are search algorithms used on trees or graphs. They work by traversing one depth from the root node before it moves on to the next depth. 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. 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 1 000, 10 000, 100 000, or 1 000 000 when running the benchmark. The algorithm can be run on multiple GPUs, but only in the embarrassingly parallel mode.

SpMV (Sparse Matrix-Vector multiplication) is where a sparse matrix and a dense vector is multiplied with a dense vector as result. 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.

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.

MD5 Hash (Message-Digest algorithm 5) 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.

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

Scan The Parallel Prefix Sum algorithm, Scan, 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.

SHOC's implementation 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.

Radix Sort One of the benchmarks in SHOC is for measuring sorting performance on the device using an implementation of the radix sort algorithm [Pro20]. This is a sorting algorithm that groups the digits by its position and compares each digit one at a time in the selected positions.

SHOC's implementation sorts unsigned integer key-value pairs and supports the problem sizes 1, 8, 48 and 98 MB. The benchmark's performance of the sort kernel is measured in GB per seconds.

Triad The Triad benchmark in SHOC is based on the STREAM (Sustainable Memory Bandwidth in High Performance Computers) TRIAD benchmark [McC95]. STREAM is a benchmark set consisting of the benchmarks `COPY`, `SCALE`, `SUM` and `TRIAD`.

SHOC's implementation of the triad benchmark uses single precision computation with a problem size ranging from 16 KB to 64 MB and measures the sustainable memory performance of a series of dot product operations [Dan+10]. In SHOC this benchmark does not have the possibility to select different problem sizes, and by default it tests all different sizes.

Reduction Another benchmark in the SHOC benchmark suite is the reduction benchmark. Reduction is an algorithm that takes an input array of numbers and returns a single number using an operation on all the numbers. A type of reduction is a sum reduction, where the sum operation are applied to each element in the sequence. Other operators can be, minimum, maximum and count [Bar15, p. 546].

In SHOC the benchmark measures the performance of a large sum reduction operation [Dan+10]. This reduction benchmark is implemented for both single and double floating-point precision and have problem sizes ranging from 1 MB to 64 MB. There is also implemented a true parallel version for this benchmark. In this TP version, the data is communicated between the nodes using MPI. [Dan+10]

Molecular Dynamics The molecular dynamics (MD) benchmark in SHOC measures the performance of an MD problem called Lennard-Jones potential [Ada01]. This is a pair potential for calculating the potential energy between two atoms.

In SHOC, the force potential is computed in the MD benchmark and each thread on the GPU computes acceleration for a single atom with impact from the other atoms in the given space [Dan+10]. This benchmark uses problem sizes based on number of atoms to compute, ranging from 12288 atoms to 73728 atoms. The input data for the kernel is both single and double floating-point precision.

2.1 Selected Autotuners

OpenTuner is a framework for building program autotuners described by J. Ansel et al. in "OpenTuner: An extensible framework for program autotuning" [Ans+17]. 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. 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. OpenTuner provides a range of 12 available algorithms to perform the autotuning and also has support for adding new search techniques [Ans+17].

Kernel Tuner Kernel Tuner is an autotuner [Wer18] that 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.

CLTune CLTune is an autotuner for tuning CUDA and OpenCL kernels [NC17]. 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.

KTT Kernel Tuning Toolkit (KTT) is an autotuner that focuses on autotuning CUDA and CLTune kernels [Pet+20]. 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.

2.2 Selected GPU Systems

The IBM Power System AC922 is a system designed for giving great performance to data analytics, HPC applications and especially AI training. The system has two IBM POWER9 processors with 32 cores and 128 threads each. The system also includes 4 Nvidia V100 32GB GPUs.

The NVIDIA DGX targets deep learning and complex AI applications. The DGX-2 used in our experiments consists of 16 Tesla V100 GPUs with 32 GB of memory each.The DGX-2 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. 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 [Ish+18].

Our T4 System contains 20 Nvidia Tesla T4 GPUs with 16 GB each and two Intel Xeon Gold 6230 CPUs with 20 cores and 40 threads each. Lastly, we also include a system with an RTX Titan and a system with a GTX 980.

2.3 Related Work

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 examples with very large search spaces. The search space of these benchmarks are large enough such that it is infeasible to brute-force the solutions.

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 their paper [Pet+20]. The kernels are highly efficient, but they generally do not have a large search space, and are possible to brute-force in less than a day on modern hardware. 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.

A. Rasch et al. [RHG18] describes the autotuning framework ATF. They also compare ATF with OpenTuner and CLTune with the help of a set of parameterized benchmarks. 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 [Scl17]. The project does not seem to be actively maintained and lacks documentation for setting up and testing the benchmark suite.

3 The Benchmark Suite

BAT[1] 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.

The benchmark suite is composed of 9 different benchmarks with several autotuning parameters. A summary of the search space for each available benchmark can also be found in Table 1, where MBS is the Maximum Block Size for the GPU for that benchmark.

The most common parameters in the algorithms are block size, single or double precision types, loop unrolling, work per thread, using texture memory or not, function inlining, various options for compiler optimization, loop unrolling and the number of GPUs to use when possibility for multi-GPU run.

Some algorithms also contains constraints where certain parameter values are dependent on each other. Additional details about the benchmark suite and its evaluation is also available in two previous theses on the topic [Kir20; Sun20].

Table 1: Search space for each parameterized benchmark.

Benchmark	**Search Space**	**Cardinality**	**Dimensionality**
BFS	$MBS \times 4 \times 3 \times 3$	36 864	4
SpMV	$MBS \times 2 \times 5 \times 2 \times 2$	40 960	5
MD5 Hash	$MBS \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 5$	327 680	8
Scan	$5 \times 10 \times 2 \times 2 \times 2 \times 2$	89 600 $\times GPUS$	10
	$\times 4 \times 4 \times 7 \times GPUS$		
Sort	$2 \times 2 \times 3 \times 3 \times 7 \times 7 \times 2 \times 2 \times 2$	14 112	9
Triad	$MBS \times 10 \times 2 \times 2$	40 960	4
Reduction	$\lfloor log_2(MBS) \rfloor \times GPUS$	394 240	11
	$\times 11 \times 2 \times 4 \times 4 \times 2 \times 7 \times 2 \times 2 \times 2$	$\times GPUS$	
MD	$MBS \times 2 \times 2 \times 5$	20 480	4

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 different with different setups.

[1] https://github.com/NTNU-HPC-Lab/BAT

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

Most of the benchmarks from BAT are implemented with the mentioned autotuners to make sure that all benchmarks would work for different types of autotuners. One example of a benchmark that did not work with an autotuner is Scan with CLTune. This is because CLTune can only autotune single kernels and the Scan algorithm has sequential kernels where the result is dependent on data from other kernels.

4 Evaluation of the Benchmark Suite

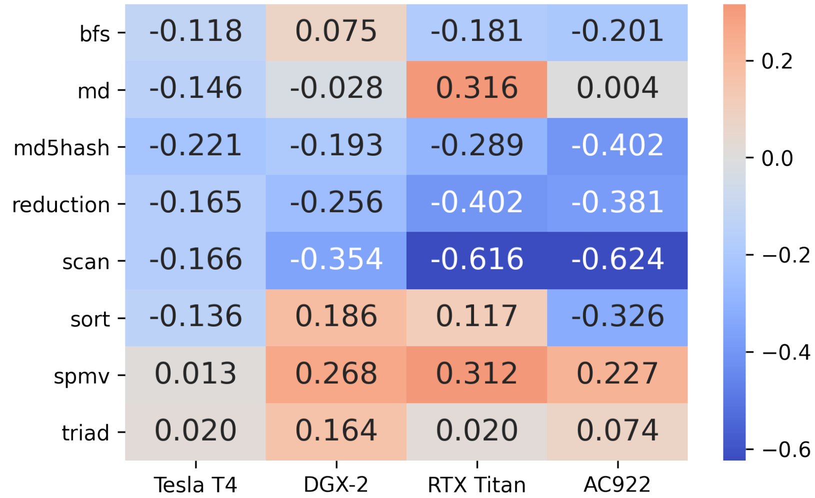

Fig. 1: Correlation between block size and running time for each system and benchmark.

Fig. 1 indicates how the block size is correlated with running time for each benchmark for four of our architectures. A negative value for the correlation indicates that a higher value for the block sizes provides a lower running time for the kernel, and vice versa.

From this we see that for most benchmarks a higher block size is usually better, however for the SpMV and Triad benchmark lower block sizes can be better. This behavior also varies between different systems, where the RTX Titan on the MD benchmark generally performs worse with higher block sizes. This behavior is also shown in Fig. 3 later in this Section.

BFS We wanted to see how the time varied for the values of block size in BFS. The circles on the graph in Fig. 2 are block sizes that are divisible by 32. The single block size 16 is also marked. Values below around 15 are outside of the graph due to their high running time. In the figure, we can see that generally the values divisible by 32 seem to be the sizes that leads to the fastest run time. For the GeForce GTX980, block sizes divisible by 32 are the fastest up to around block size 150. When the block size increases for this GPU, we can see that the values divisible by 32 actually leads to higher run time.

This figure 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.

In Fig. 2 we can see that the system with the GeForce GTX980 is the one that uses the most time when running autotuning with KTT. Since this is a more common consumer GPU, this is expected. One thing that might not be obvious before seeing the results, is that the autotuning uses less time on one single GPU on the DGX-2 system than on one single GPU on the IBM Power System AC922.

The two systems have practically the same GPU model, however the difference in performance might come from a difference in the interconnects. For the Power AC922, NVLink is used between CPU and GPU, while the DGX-2 has PCIe switches between CPU and GPU.

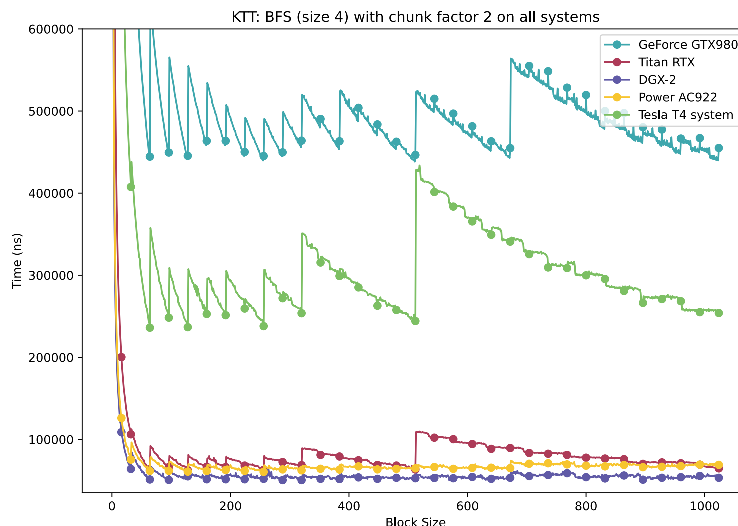

Fig. 2: KTT: BFS (size 4) on all systems. Chunk factor 2.

For the BFS benchmark on the AC922, RTX and T4 system, running time is most strongly inversely correlated with block size(-0.201, -0.118, -0.181), indicating that a larger block size is often better. For the same benchmark on the

DGX-2 however, surprisingly all of the parameters are correlated with running time(from 0.05-0.11), indicating that a low block size and few manual optimizations provides the best performance for this system. None of the other autotuning parameters had consistently a positive or negative impact on the performance across all systems.

SpMV We investigated how the best parameter values for SpMV differed for the different machines. By looking at the best results, we found out that to loop unroll or not varied much over the machines.

We also found 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.

This can also be seen in our correlation results, where the Format had a significant impact on the runtime of the program, with the parameter being strongly inversely correlated with runtime for the DGX-2(-0.461), RTX Titan(-0.359) and AC922(-0.233) and strongly correlated with runtime for the T4 system(0.712). The block size was correlated with runtime for all of the systems in this benchmark (0.013-0.312).

Scan For the brute force version of scan, 32 is shown to be the best value for precision for all the systems. The effect of applying loop unrolling varies greatly between different systems. When analyzing the Scan results, we found that block size and grid size of 128 or 256 gave generally the best run time.

For our correlation results for Scan the RTX Titan, DGX-2 and AC922 had a running time that was strongly inversely correlated with block size(-0.616, -0.354, -0.624) and grid size (-0.718, -0.377, -0.631). This indicates that a large block size and grid size has a very significant impact on the running time of the kernels for these architectures. For the T4 system, this correlation was much weaker (-0.166, -0.051). The optimization levels for the host-side code and the device-code also had a significant impact, with a large variance between architectures. For the RTX Titan, DGX-2, AC922 and T4 system respectively, Device-side optimizations were inversely correlated (-0.104, -0.247, 0.099, -0.412) apart from AC922, however host-side optimization had a negative correlation for AC922 and RTX Titan, yet positive for the others (-0.204, 0.437, -0.357, 0.426).

MD5 Hash Analyzing the MD5 Hash benchmark result showed that work-per-thread factor varied significantly for the different systems. This was also the situation for inline and loop unrolling.

The correlation results for the MD5 Hash benchmark show that block size and round style were inversely correlated with runtime for the DGX-2, RTX Titan, AC922 and T4 System (-0.193, -0.221, -0.289, -0.402 and -0.049, -0.018, -0.105, -0.051). The rest of the parameters showed no consistent trend.

Sort Using the sort benchmark implementation with Kernel Tuner, we could see that generally 512 for scan block size and 128 for sort block size gave the best run time.

Scan block size, scan data size and LSB loop unroll are inversely correlated with running time for all systems. Inline LSB has a strong correlation on DGX-2 (0.697), yet a negative correlation on the RTX, T4 and AC922 systems (-0.379, -0.078, -0.484). Indicating that inlining is greatly beneficial for all systems except for the DGX-2. The other parameters showed no consistent trends.

Triad When analyzing the triad results for autotuning with KTT, we could see that a precision of 32 gave the fastest result and that it generally had a low value for the work per thread parameter.

For the Triad benchmark, block size was consistently positively correlated with running time for the Titan RTX(0.019), DGX-2(0.164), AC922(0.074) and Tesla T4(0.020) systems. This indicates that higher block sizes might not necessarily improve the running time for this kernel. Precision was also strongly correlated with running time for all systems, indicating the natural conclusion that the single-precision performance of the kernel is higher than double-precision performance for all systems. The rest of the parameters showed no consistent trend.

Reduction The systems IBM Power System AC922 and DGX-2 found the best values for the precision to be 64, but on the other machines a precision of 32 was the best. For the grid size parameter, we could see that all the best values found, were larger values.

For the Reduction benchmark, the RTX Titan, DGX-2, AC922 and Tesla T4 system had an inverse correlation for block size (-0.402, -0.256, -0.381, -0.165) and grid size(-0.402, -0.359, -0.170, -0.435) with running time. Loop unrolling of two different functions in the reduction kernel also have potential to give a substantial speedup, based on their correlation factors. For the first loop unrolling the RTX Titan, DGX-2, AC922 and T4 systems had both a positive and negative correlation, depending on the system (0.172, -0.218, -0.044, -0.218). For the second function the loop unrolling was mostly inversely correlated (-0.158, -0.168, 0.036, -0.681). This indicates that loop unrolling generally leads to higher performance for this benchmark.

MD In Fig. 3 we can see that the parameter for block size is an impactful parameter because the best value changes for different architectures. Note, that behind the Power AC922 system plots, there are plots for a DGX-2 system. The very similar runtime is most likely due to the GPUs being practically the same model for these systems. We can see that for all the GPUs a very low block size results in a very poor runtime, and just by increasing this slightly we can see that the runtime improves significantly. However, the runtime of block size does not change any drastically when further increasing these sizes.

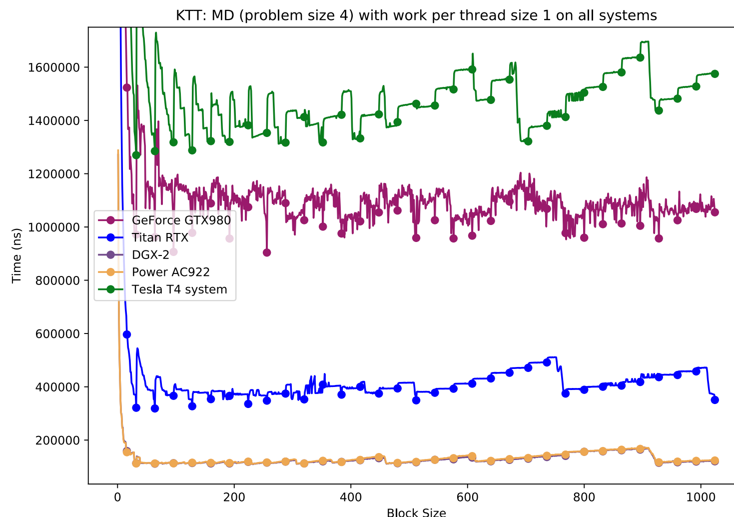

Fig. 3: Work per thread parameter and block size parameter for all systems for the MD benchmark in KTT.

In this figure, as well as in Fig. 2, the block sizes are divisible by 32 and the block size equal to 16 is marked with a circle. From this, we can notice better runtime for each block size value matching these values. This is perhaps due to the warp size being 32, where a multiple of 32 is often considered to be the optimal when launching kernels.

One observation in Fig. 3, is that despite NVIDIA Tesla T4 GPUs being a much newer generation graphics card and having generally better performance than the NVIDIA GeForce GTX 980, it performed worse. This is an interesting result and might be due to the difference in the interconnects and how the GPUs are connected to the CPUs in the two systems.

For the MD benchmark the RTX Titan, DGX-2, AC922 and T4 system all had a positive correlation with increased running time for use of texture memory (0.224, 0.657, 0.089, 0.138) and work per thread (0.317, 0.229, 0.160, 0.170). These results indicate that thread coarsening and use of texture memory is generally not beneficial for this benchmark.

5 Conclusion and Future Work

In this paper we presented BAT, an HPC-based benchmark suite with parameterized algorithms in CUDA. 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 tested hardware included the DGX-2, two of the IBM Power System AC922 with two and four Tesla V100-SXM2 32 GB GPUs and a server with 20 Tesla T4 GPUs. We also did tests on a system with consumer GPUs, including GTX 980 and Titan RTX.

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 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 tune kernels do not support using it as a tuning parameter.

Our results show that the optimal parameters, including block size, change drastically between different architectures and benchmarks. This provides a strong argument for autotuning these algorithms to most efficiently utilize the underlying hardware.

BAT is publically available[2] and includes 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. BAT also has a CLI that makes it easier to run autotuning with the benchmarks.

Current and future work includes expanding the benchmark suite with more benchmarks and tunable parameters. This includes the option of adding OpenCL or HIP kernels to support benchmarking autotuners for AMD GPUs. We would also like to develop better tools for easily comparing different autotuners automatically on the benchmark suite. A scoring system for the performance of autotuners on benchmark suite would also be useful.

Acknowledgments

The authors would like to acknowledge the Faculty of Information Technology, Mathematics and Electrical Engineering and the Department of Computer Science at NTNU for their PhD stipend and support of our HPC-Lab that facilitated the development of this project.

References

[McC95] John McCalpin. "Memory bandwidth and machine balance in high performance computers". In: *IEEE Technical Committee on Computer Architecture Newsletter* (Dec. 1995), pp. 19–25.

[Ada01] J.B. Adams. "Bonding Energy Models". In: *Encyclopedia of Materials: Science and Technology*. Ed. by K.H. Jürgen Buschow et al. Oxford: Elsevier, 2001, pp. 763–767. ISBN: 978-0-08-043152-9. DOI: `https://doi.org/10.1016/B0-08-043152-6/00146-7`. URL: `http://www.sciencedirect.com/science/article/pii/B0080431526001467`.

[2] https://github.com/NTNU-HPC-Lab/BAT

[Dan+10] Anthony Danalis et al. *The Scalable Heterogeneous Computing (SHOC) benchmark suite*. https://dl.acm.org/doi/10.1145/1735688.1735702. [Accessed April 27, 2020]. Mar. 2010.

[Bar15] Gerassimos Barlas. "Chapter 7 - The Thrust template library". In: *Multicore and GPU Programming*. Ed. by Gerassimos Barlas. Boston: Morgan Kaufmann, 2015, pp. 527–573. ISBN: 978-0-12-417137-4. DOI: https://doi.org/10.1016/B978-0-12-417137-4.00007-1. URL: http://www.sciencedirect.com/science/article/pii/B9780124171374000071.

[Ans+17] Jason Ansel et al. *OpenTuner: An Extensible Framework for Program Autotuning*. https://ieeexplore.ieee.org/document/7855909. [Accessed May 2, 2020]. Feb. 2017.

[NC17] Cedric Nugteren and Valeriu Codreanu. *CLTune: A Generic Auto-Tuner for OpenCL Kernels*. https://arxiv.org/pdf/1703.06503.pdf. [Accessed April 29, 2020]. Mar. 2017.

[Scl17] Alessio Sclocco. *TuneBench*. https://github.com/isazi/TuneBench. [Accessed August 4, 2020]. Dec. 2017.

[Ish+18] Alex Ishii et al. *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.

[RHG18] Ari Rasch, Michael Haidl, and Sergei Gorlatch. *ATF: A Generic Auto-Tuning Framework*. https://ieeexplore.ieee.org/document/8291912. [Accessed August 4, 2020]. Feb. 2018.

[Wer18] Ben 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.

[Kir20] Knut Aasgaard Kirkhorn. "BAT: A Benchmark Suite for Auto-Tuners. Development of BAT and Tuning on DGX-2 and More". MA thesis. Trondheim, Norway: Dept. of Computer, Information Science, Norwegian University of Science, and Technology (NTNU), 2020.

[NVI20] NVIDIA Corporation. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed June 10, 2020]. June 2020.

[Pet+20] Filip Petrovič et al. *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.

[Pro20] Programiz. *Programiz: Learn to Code for Free*. https://www.programiz.com/dsa/radix-sort. [Accessed Apr. 15, 2020]. 2020.

[Sun20] Ingunn Sund. "BAT: A Benchmark suite for AutoTuners. Development of BAT and Tuning on 20x Tesla T4 GPUs and More". MA thesis. Trondheim, Norway: Dept. of Computer, Information Science, Norwegian University of Science, and Technology (NTNU), 2020.