Autotuning GPU Compiler Parameters Using OpenTuner

Pedro Bruel[1], **Marcos Amarís**[1] **and Alfredo Goldman**[1]

[1]Instituto de Matemática e Estatística (IME)
Universidade de São Paulo (USP)
R. do Matão, 1010 – Vila Universitária, São Paulo – SP, 05508-090

Abstract. *Graphics Processing Units (GPUs) are specialized coprocessors that were initially conceived for the purpose of accelerating vector operations, such as graphics rendering. Writing and configuring efficient algorithms for GPU devices is still a hard problem. The Algorithm Selection Problem consists of finding a combination of algorithms, or a configuration of an algorithm, that optimizes the solution of a given problem instance or set of instances. An autotuner is a program solves the Algorithm Selection Problem automatically. In this paper we implement an autotuner for the compilation flags of GPU algorithms, using the OpenTuner framework. The autotuner produces a set of compilation flags that aims to optimize the time to solve a given problem for a specific GPU device. We analyse the performance gains of tuning the compilation flags for heterogeneous GPU algorithms across three different GPU devices. We show that it is possible to gain performance by automatically and empirically selecting a set of compilation flags for the same GPU algorithm in different devices. In one of the experimental settings we were able to achieve a 30% speedup in comparison with the compiler high-level optimization options.*

1. Introduction

The popularity of heterogeneous parallel computing platforms raised as parallel computing emerged in the last decade. Consequently, the need for optimized algorithms for these platforms has also raised. The increase in diversity and availability of parallel computing hardware makes the task of hand-optimizing programs increasingly hard, justifying automated strategies to tuning and configuring parallel programs.

The most widely used GPUs are produced by NVIDIA. They are built with a set of Streaming Multiprocessors (SMs) each containing several cores called Scalar Processors (SPs), a set of Special Function Units (SFUs) and a number of load/store units. The multiprocessors execute asynchronously, in parallel. The SM schedules threads in groups of 32 parallel threads called warps, which can use load/store units concurrently, allowing simultaneous reads from memory for these threads.

The Compute Unified Device Architecture (CUDA) is a high-level platform for developing GPU applications. It extends the C language and provides a compiler that translates it into a pseudo-assembly PTX (Parallel Thread Execution) code, which is executed by NVIDA GPUs. CUDA applications are organized in kernels, which are functions executed on GPUs. These kernels are composed of thread blocks, each containing hundreds of threads.

The performance of a kernel execution in a GPU depends largely on the optimization of the accesses to data in the memory hierarchy. Threads within a block can cooperate by sharing data through shared memory. This kind of memory is on-chip in each multiprocessor and has a very small latency. The global memory bandwidth of a GPU can be largely improved by combining the load/store requests from different threads of a single warp in a single memory request, in a process called coalescing [1]. Coalescing occurs when the threads access contiguous global memory addresses, which allows the usage of multiple load/store units available per SM.

The description of the Algorithm Selection Problem was first published by Rice in 1976 [2]. The problem consists in, given a set of *algorithms* and a set of *problems*, finding a *mapping* of algorithms to problems that minimizes the time to solve all problems in the set. The *algorithms* that compose a set can represent different abstractions, such as programs, heuristics, or configurations. The set of *problems* usually contains instances of a problem. The Algorithm Selection Problem is hard. Its NP-completeness has been proved when calculating static distributions of algorithms in parallel machines [3]. Its Undecidability in the general case was also shown [4].

Autotuning is a technique for solving the Algorithm Selection Problem in an automated and empirical way, accounting for architecture and problem features. The idea is to use the performance-impacting features of architectures, problems and algorithms in a domain to describe a *search space*. Optimization and Machine Learning techniques tailored for an application domain are then used to empirically search this space for optimal algorithm-to-problem mappings. Autotuning systems, or autotuners, typically target a specific domain such as matrix multiplication [5] and dense or sparse matrix linear algebra [6, 7]. In this paper we use the OpenTuner framework [8] to implement an autotuner for the space defined by the CUDA [1] compilation parameters.

The search spaces are defined by instantiating the different *Parameter* types provided by the framework. Each parameter type implements its own manipulation functions, which allow the search techniques to navigate the values allowed to each parameter, exploring the search space defined by the user. OpenTuner implements ensembles of optimization and Machine Learning techniques that perform well in different problem domains, and are used to search user-defined search spaces. The results found during the search process are shared between techniques through a common results database. OpenTuner uses *meta-techniques* for coordinating the distribution of resources between techniques in an ensemble. An OpenTuner application can implement its own search techniques and meta-techniques, adding them to existing ensembles or creating completely new ones.

In this paper we use the OpenTuner framework to implement an autotuner for the parameters of the CUDA compiler. We use this autotuner to search for the compiler parameter sets that optimize the performance of five different GPU applications. Our main contribution is to show that it is possible to optimize legacy code written for GPUs by automatically tuning the parameters of the CUDA compiler. The optimization achieved by autotuning often beats the compiler high-level optimization options, such as `-O1`, `-O2` and `-O3`. In one of the experimental settings the autotuner produced compilation

[1] http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc

options that achieved a 30% speedup in comparison with the high-level optimizations. We also show that the compilation parameters that optimize an algorithm for a given GPU architecture will not always achieve the same performance in different hardware.

The rest of this paper is organized as follows. Section 2 presents related work in autotuning and the optimization of GPU programs. Section 3 describes the GPU testbed and algorithms, the compilation parameters used for tuning and the implementation of the autotuner. Section 4 presents the results of the autotuning experiments, and discusses these results. Section 5 concludes.

2. Related Work

The autotuning technique has been used since as early as 1997, when the PHiPAC system [5] used code generators and search scripts to automatically generate high performance code for matrix multiplication. Since then the autotuning problem has been tackled in multiple domains, with a great variety of strategies. Whaley *et al.* [6] introduce the ATLAS project, that produced optimized dense matrix multiply routines. The OSKI [7] library provides automatically tuned kernels for sparse matrices. The FFTW [9] library provides tuned C subroutines for computing the Discrete Fourier Transform. More recently, there have been efforts in the implementation of generic tools for autotuning. PetaBricks [10] is a language, compiler and autotuner for generic programs, that introduce new abstractions such as the *either...or* keywords, which let programmers define multiple algorithms for the same problem. The OpenTuner framework [8] implements ensembles of search techniques that are used to search a user-defined space of program configurations. The OpenTuner framework has already been used to implement a domain specific language for data-flow programming [11] and a framework for the optimization of recursive parallel algorithms [12]. The ParamILS framework [13] implements state-of-the-art search methods for algorithm configuration and parameter tuning.

GPU applications can hide pipeline computation and communication latency by executing many parallel threads in an interleaved model. When the instruction pipeline is fully saturated, the performance of the application run close to the peak performance of the GPU. On the other hand, when the pipeline is under-utilized the performance of the application can be very unpredictable. The difficulty of modeling the instruction pipeline performance lies in the non-ideal situation [14, 15]. Based on recent work [16] in the impact of small cache hierarchies on performance, it is has become understood that it is critical to model GPU caches. These models must also carefully consider the effects of memory access divergence, which can significantly decrease the fidelity of the model [17, 18]. The models seek to improve the performance of GPU applications by using techniques of high and low-level parallel programming languages for GPUs, such as CUDA, OpenCL and PTX, to find bottlenecks or divergences in parallel branches.

The availability of GPU hardware, tools and frameworks allow beginner as well as expert developers to design and implement algorithms that take advantage of GPU architectures. However, knowledge about the architecture of specific GPUs is required to achieve optimal performance for particular algorithms [19, 20]. For this reason, the development of highly optimized code for a new architecture is not an easy task. Some of the work on GPU autotuning [21, 22, 23] aim to tune parameters such as block sizes, tiling techniques, transfers between single and double precision, loop permutations and

unrolling. They worked under the assumption that these parameters were enough to capture the algorithmic design space and attain a large fraction of peak performance. In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [24] implements abstractions for OpenMP, MPI and OpenCL. The INSIEME compiler is able to generate optimized parallel code for heterogeneous multi-core architectures.

To the best of our knowledge, this is the first work that tackles the autotuning problem in GPUs by tuning the parameters of the CUDA compiler, using the OpenTuner framework. This approach has the potential to improve the performance of legacy code in a variety of parallel computing platforms, across heterogeneous applications.

3. Experiments

In this section we discuss the experiments with OpenTuner, the GPU algorithms implemented, and the GPU architectures used as testbed.

Testbed

We performed autotuning experiments over three different GPUs, the GeForce GTX-680, the Tesla-K20 and the Tesla-K40. All three GPUs are Kepler architectures (Compute Capability 3.X). Table 1 summarizes their specifications.

Model	C.C.	GM	Bus Width	BW	L2	Cores/SM	GPU Clock
GTX-680	3.0	2 GB	256-bit	192.2 GB/s	512 KB	1536/8	1006 Mhz
Tesla-k20	3.5	4 GB	320-bit	208 GB/s	1280 KB	2496/13	706 Mhz
Tesla-k40	3.5	12 GB	384-bit	276.5 GB/s	1536 KB	2880/15	745 Mhz

Table 1. Hardware characteristics of GPUs were used for the testbed

GPU Algorithms

We autotuned the compiler parameters for a benchmark of five different GPU applications. The benchmark is composed of four optimization strategies for *matrix multiplication* [25] and one solution for the *maximum sub-array problem* [26]. The remaining of this section discusses the implementation of these algorithms.

Matrix Multiplication We used four different optimization techniques regarding the use of the available memories for the matrix multiplication application: (#1) global memory with non-coalesced accesses; (#2) global memory with coalesced accesses; (#3) shared memory with non-coalesced accesses to global memory; and (#4) shared memory with coalesced accesses to global memory. The running time of a matrix multiplication for two matrices of size $N \times N$ is proportional to $O(N^3)$ in the sequential algorithm and to $O(N)$ in a CUDA application that uses N^2 threads.

Non-coalesced accesses occur because of irregular references to data in global memory. The matrix multiplication optimizations affect only the performance of the communication between threads. For optimization (#2), we changed the data access pattern to allow coalesced access to data in global memory. Optimization (#3) uses shared memory to load data from global memory and to process them with a lower latency of communication. Similarly to optimization (#2), optimization (#4) is obtained by changing the

coalesced memory access from the source code of optimization (#3). All performance measurements for these optimizations used square matrices where $N = 1024$.

Maximum Sub-array Problem Let X be a sequence of N integer numbers $(x_1, ..., x_N)$. The maximum sub-array problem consists of finding the contiguous sub-array within X which has the largest sum of elements. The solution for this problem is frequently used in computational biology for, for example, gene identification, analysis of sequences of protein and DNA, and identification of hydrophobic regions. The maximum sub-array problem can be solved with $O(N)$ comparison operations [27] and a parallel solution for this problem was developed using Coarse Grained Model [28], resulting in $O(N/t)$ comparisons, where t is the number of threads.

In the implementation [26] used in this paper, we created a kernel with 4096 threads divided in 32 thread blocks with 128 threads on each. The N elements are divided in intervals of N/t elements, one per block and each block receive a portion of the array. The blocks use the shared memory for storing segments of its interval, which are read from the global memory using coalesced accesses. Each interval is reduced to a set of 5 integer variables, which are stored in vector of size $5 \times t$ in global memory. This vector is then transferred to the CPU main memory RAM for later processing. All performance measurements for the Maximum Sub-array Problem used arrays where $N = 134217728$.

Compilation Parameters

We used the CUDA configuration parameters listed in the CUDA Toolkit Documentation [1] to pass specific options to the PTX optimizing assembler, the linker, and the CUDA compiler. We compared the performance of the compilation parameters found by the autotuner with the standard options provided by the compiler to optimize code, namely `--opt-level=0,1,2,3`. We did not use compiler options that targeted the host linker or the library manager, since they do not affect performance.

Autotuning GPU Applications

The autotuner implemented for the GPU algorithms compilation uses the native parameter types of the OpenTuner framework to represent the available CUDA compilation options. Compiler flags and multi-valued parameters are represented as single *EnumParameter*s, and parameters that accept numerical values are represented as *IntegerParameter*s. The same encoding was used in another work for implementing a GCC compilation parameter tuner [8].

The search performed in the space of all CUDA compilation options has the potential to generate many invalid combinations of options due to incompatible flags or architecture restrictions. Since the autotuner has no knowledge about the search space or the programs that are being tuned, it could misinterpret as good the very fast results produced when there are errors during execution or compilation. To prevent this, our implementation always checks for errors during the compilation phase, and the tuned programs always verify their results. This allows the autotuner to only consider the performance of correct programs. Incorrect programs are the result of incompatibilities of flags, and these results can be used to better understand the search space.

If the behavior of the search space defined by the compiler options was already known, an autotuner could be implemented that considered the interactions and incompatibilities between options. This would generate a pruned search space and allow the search techniques to test only valid combinations, which would achieve better results faster.

The code for the implemented autotuner, as well as all the results for the experiments, is available[2] under the GNU General Public License. The `manipulator` function builds the representation of the compiler parameters using the native types of the OpenTuner framework. The utility functions `parse_flags` and `parse_config` help to construct a valid compilation string from a configuration received from a search technique. Finally, the `run` function compiles the target program and runs the binary, checking for errors in both phases. If errors were found during execution, the autotuner saves the compilation options to a file, and assigns a penalty value to this configuration. If errors were found during the compilation of the binaries, the autotuner halts its execution. Otherwise, the autotuner returns the runtime of the program as the fitness measure.

4. Results

This section is divided in two parts. In the first part we present the performance gains achieved by the autotuned compilation options in each of the five algorithms that compose the application benchmark. In the second part we discuss the performance and accuracy of the autotuner.

4.1. Performance Gains

The boxplots[3] in Figures 1, 2, 5 and 6 depict the distributions of 20 measurements of the performances of binaries compiled with high-level optimization options and with the autotuned options. The results for the high-level options `--opt-level=0,1,2,3` were denoted by *-O0*, *-O1*, *-O2* and *-O3*. The results for the autotuned configurations were denoted by the keyword *Tuned*.

The black band inside the boxes represents the median of the measurements. The lower and upper bounds of the box represent, respectively, the first and third quartiles of the data. The whiskers represent the third and first quartile plus and minus the *inner quartile range* times 1.5. Finally, the circles represent the outliers.

For the matrix multiplication algorithm with non-coalesced accesses to global memory, denoted by optimization #1, the autotuned compilation options achieved gains in performance of up to 30% in comparison with the high-level CUDA compilation options. For the matrix multiplication algorithm with coalesced accesses to global memory, denoted by optimization #2, the performance gains achieved were up to 9.22%. Both results were achieved in the GTX-680 GPU, and are summarized in Figures 1 and 2.

The autotuner also achieved a 2.7% speedup, in comparison with high-level optimizations, for the matrix multiplication algorithm with shared memory and coalesced accesses to global memory for the GTX-680 GPU.

A summarization of our results is presented in Figure 4. The autotuned solutions did not improve upon the high-level optimizations for the matrix multiplication algorithm with optimization #4, in any of the GPUs of the testbed.

[2] https://github.com/phrb/gpu-autotuning

[3] All boxplots were made using the standard implementations available for the R language.

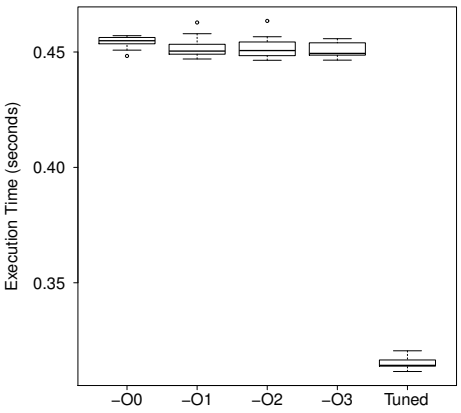

Figure 1. Boxplots for the GTX-680 GPU, comparing autotuned results and high-level compiler optimization options for the matrix multiplication algorithm with non-coalesced global memory accesses (optimization #1).

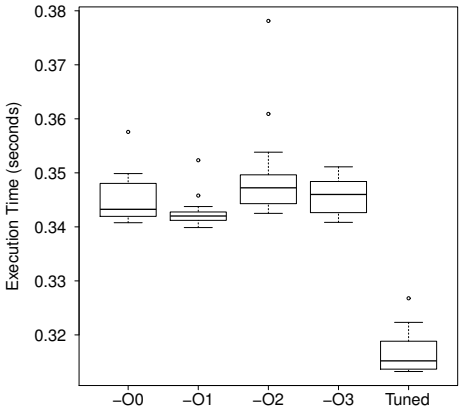

Figure 2. Boxplots for the GTX-680 GPU, comparing autotuned results and high-level compiler optimization options for the matrix multiplication algorithm with coalesced global memory accesses (optimization #2).

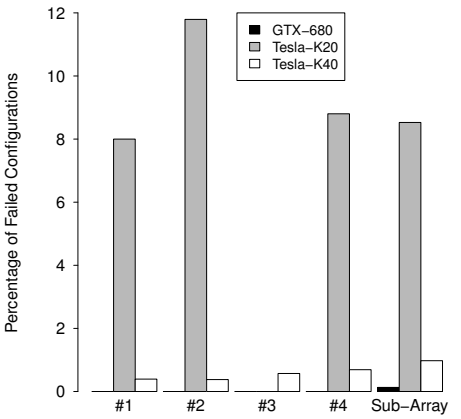

Figure 3. Percentage of all tested configurations that compiled the GPU programs without errors, but generated binaries that produced incorrect results.

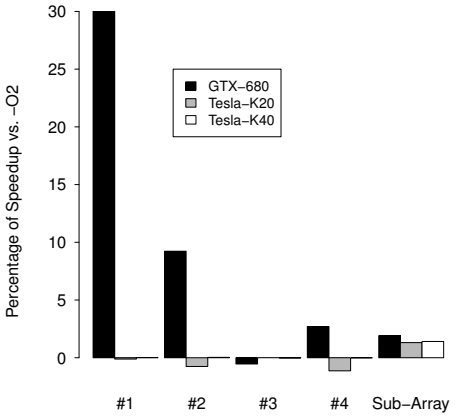

Figure 4. Summary of the speedups achieved versus *-O2* for the algorithms in the benchmark, in each architecture of the testbed. The different matrix multiplication optimizations were denoted by the numbers assigned to them in Section 3, and the maximum sub-array problem was denoted by *Sub-Array*.

Overall, the GTX-680 GPU had the best results. This GPU has the fastest clock but the least amount of L2 cache in the testbed. It was also the only GPU in which the applications could be compiled with a Compute Capability 3.2, specified by the `sm_32` parameter.

Figure 5 presents a speedup of the autotuned solution of up to 1.40% in the Tesla K-40 GPU for the maximum sub-array problem. Finally, Figure 6 shows a speedup of 1.31% of the autotuned solution in the Tesla-K20 GPU for the same problem. Still on the maximum sub-array problem, the autotuned solution achieved a 1.91% speedup in comparison with the compiler high-level optimizations, in the GTX-680 GPU.

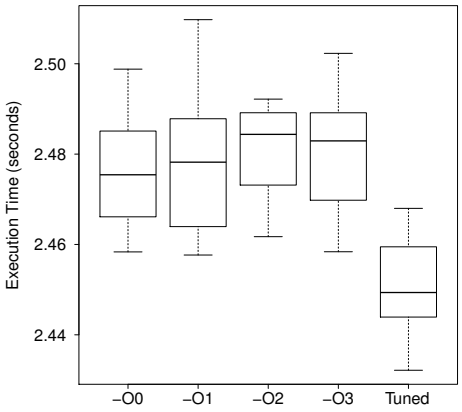

Figure 5. Boxplots for the Tesla-K40 GPU, comparing autotuned results and high-level compiler optimization options for the maximum sub-array problem.

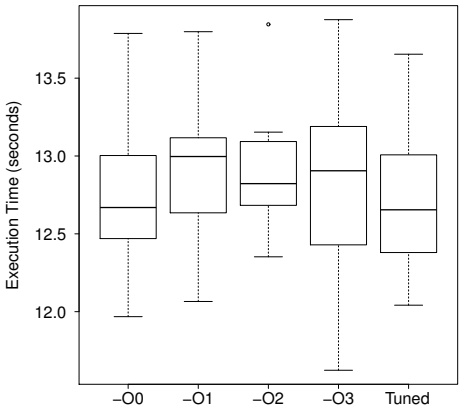

Figure 6. Boxplots for the Tesla-K20 GPU, comparing autotuned results and high-level compiler optimization options for the maximum sub-array problem.

The autotuner saved to a log file the compiler configurations that passed the compilation phase with no errors but generated binaries that did not produce the correct results for the applications. Figure 3 summarizes the results for these failed configurations.

The Tesla-K20 GPU had the highest percentage of configurations that compiled but produced binaries that failed the assertions, for all algorithms. All failed configurations for the Telsa-K20 specified the `sm_32` Compute Capability, and many of them enabled the `--allow-expensive-optimizations` flag.

Configurations that used the `sm_32` Compute Capability failed in all GPUs of the testbed except the GTX-680, for the maximum sub-array problem. In the GTX-680 GPU, a configuration using the `sm_32` flag was the best, but the failed configurations for the sub-array algorithm exposed conflicts between the flags `--fmad=true` and `--ftz=false`.

4.2. Autotuner Performance

This section presents an assessment of the performance of the autotuner implemented with the OpenTuner framework. Figures 7 to 9 present the performance of the best solution found by the autotuner versus the time these solutions were found, in seconds since the beginning of the tuning process.

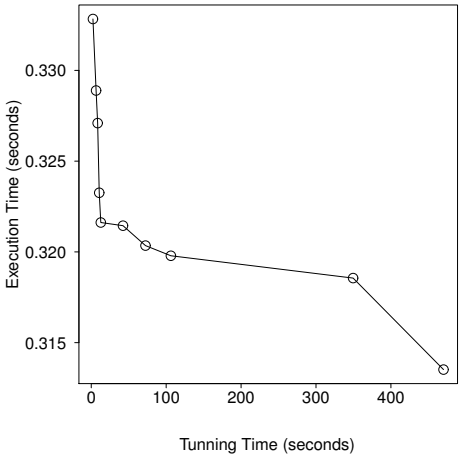

Figure 7. Best solutions found by the autotuner over time for the matrix multiplication algorithm with coalesced accesses to global memory (optimization #2) in the GTX-680 GPU.

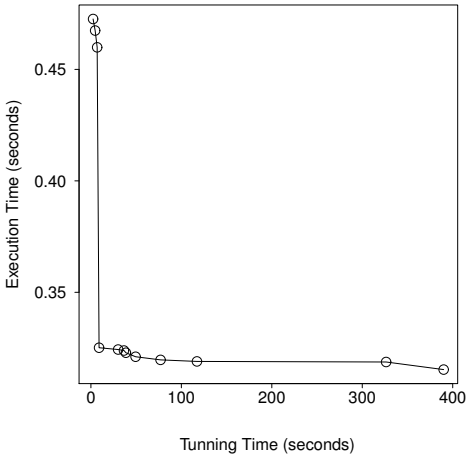

Figure 8. Best solutions found by the autotuner over time for the matrix multiplication algorithm with non-coalesced accesses to global memory (optimization #1) in the GTX-680 GPU.

The final version of the autotuner implemented in this paper returned the mean of 20 executions of a given configuration as the fitness measure for that configuration. This allowed the tuner to quickly find a solution with good performance in a single tuning run, as can be seen in Figures 7 and 8. Most tuning runs did not benefit of the entire length of the run, since the best solutions were found closer to the beginning of the tuning.

The quality of the solutions is confirmed by the results showed in the previous section. For instance, Figure 1 shows the measurement of the distribution of multiple executions of the tuned solution, whose median value is very close to the final best value reported by the autotuner, presented in Figure 8.

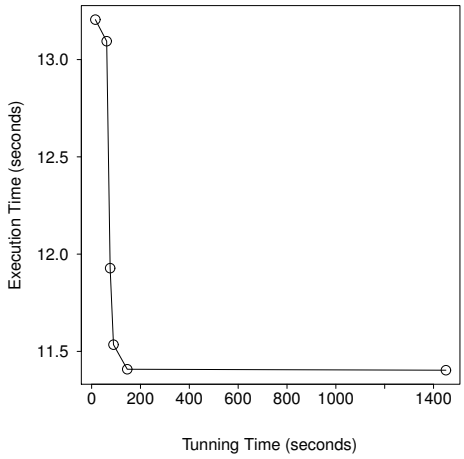

Figure 9. Best solutions found by the autotuner over time for the maximum sub-array problem in the Tesla-K20 GPU.

Figure 9 however, shows a fitness value for the final best solution that does not

correspond to the median of the values of its 20 executions, which is shown in Figure 6. This happened because the tuning run in Figure 9 was made with a previous version of the autotuner, that returned a single value as the fitness measure.

The explanation for this anomaly could be that the tuner found a single execution of a configuration that had a lower value than the current best. This value did not reflect the configuration's quality because it was privileged by fluctuations in the loads of the processor, GPU and operating system.

Therefore, the tuner was mislead and kept finding "good" solutions that benefited from fluctuations. This could also explain why the best solution in Figure 9 was found later in the tuning process. We suggest that the implementation of autotuners should always consider the mean of multiple executions, or some other representative metric, as the fitness value for a configuration or algorithm.

5. Conclusion

This paper showed that it is possible to improve the performance of legacy GPU code by applying empirical and automatic tuning techniques. We used the OpenTuner framework to implement an autotuner for the search space defined by the parameters of the CUDA compiler. We composed a benchmark of five different GPU applications, and compared their performance in a testbed of three different GPU architectures. The autotuner was able to achieve up to 30% of speedup in comparison with high-level compiler optimizations in one of the experimental settings. Future work on GPU program autotuning will include parameters for the GCC compiler, that composes the CUDA compilation chain, as well as application parameters. We would also like to experiment with the Programming by Optimization [29] design paradigm for GPU programming. Finally, we would like to extend the analysis of the interactions between compilation parameters and the improvements introduced by their usage.

To the best of our knowledge, this is the first work that applies autotuning techniques to CUDA compiler parameters for GPU applications using the OpenTuner framework, comparing the speedup achieved in different GPU architectures for heterogeneous applications.

Acknowledgments

This work was partially supported by the São Paulo Research Foundation (FAPESP) (procs. #2012/23300-7), by CAPES and by CNPq. The authors would also like to thank the Nvidia Corporation for the donation of a GPU Tesla K-40.

References

[1] P. H. Ha, P. Tsigas, and O. J. Anshus, "The Synchronization Power of Coalesced Memory Accesses." in *DISC*, ser. Lecture Notes in Computer Science, G. Taubenfeld, Ed., vol. 5218. Springer, 2008, pp. 320–334.

[2] J. R. Rice, "The algorithm selection problem," 1976.

[3] M. Bougeret, P.-F. Dutot, A. Goldman, Y. Ngoko, and D. Trystram, "Combining multiple heuristics on discrete resources," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–8.

[4] H. Guo, "Algorithm selection for sorting and probabilistic inference: a machine learning-based approach," Ph.D. dissertation, Citeseer, 2003.

[5] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. New York, NY, USA: ACM, 2014, pp. 253–260.

[6] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–27.

[7] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[8] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 303–316.

[9] M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.

[10] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," *SIGPLAN Not.*, vol. 44, no. 6, pp. 38–49, Jun. 2009.

[11] J. Bosboom, S. Rajadurai, W.-F. Wong, and S. Amarasinghe, "Streamjit: a commensal compiler for high-performance stream programming," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2014, pp. 177–195.

[12] D. Eliahu, O. Spillinger, A. Fox, and J. Demmel, "Frpa: A framework for recursive parallel algorithms," Master's thesis, EECS Department, University of California, Berkeley, May 2015.

[13] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[14] Y. Zhang and J. Owens, "A quantitative performance analysis model for GPU architectures," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, Feb 2011, pp. 382–393.

[15] S. Liu, C. Eisenbeis, and J.-L. Gaudiot, "Speculative execution on gpu: An exploratory study," in *Parallel Processing (ICPP), 2010 39th International Conference on*, Sept 2010, pp. 453–461.

[16] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[17] D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. a. Pereira, "Divergence analysis," *ACM Transactions on Programming Languages and Systems*, vol. 35, no. 4, pp. 13:1–13:36, Jan. 2014.

[18] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," *SIGPLAN Not.*, vol. 45, no. 5, pp. 105–114, Jan. 2010.

[19] J. S. Kirtzic, "A Parallel Algorithm Design Model for the GPU Architecture," Ph.D. dissertation, Richardson, TX, USA, 2012, aAI3547670.

[20] T. Chen and Y.-K. Chen, "Challenges and opportunities of obtaining performance from multi-core cpus and many-core gpus," in *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, April 2009, pp. 613–616.

[21] P. Guo and L. Wang, "Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus," in *Computational and Information Sciences (ICCIS), 2010 International Conference on*, Dec 2010, pp. 1154–1157.

[22] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning gemm for gpus," in *Computational Science–ICCS 2009*. Springer, 2009, pp. 884–892.

[23] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *Innovative Parallel Computing (InPar), 2012*, May 2012, pp. 1–10.

[24] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–12.

[25] NVIDIA, *CUDA C: Programming Guide, Version 7.*, March 2015.

[26] C. Silva, S. Song, and R. Camargo, "A parallel maximum subarray algorithm on gpus," in *5th Workshop on Applications for Multi-Core Architectures (WAMCA 2014). IEEE Int. Symp. on Computer Architecture and High Performance Computing Workshops*, Paris, 2014, pp. 12–17.

[27] J. L. Bates and R. L. Constable, "Proofs As Programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 113–136, Jan. 1985.

[28] C. E. R. Alves, E. Cáceres, and S. W. Song, "BSP/CGM Algorithms for Maximum Subsequence and Maximum Subarray." in *PVM/MPI*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241. Springer, 2004, pp. 139–146.

[29] H. H. Hoos, "Programming by optimization," *Communications of the ACM*, vol. 55, no. 2, pp. 70–80, 2012.