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. *In this paper we implement an autotuner for the compilation flags of GPU algorithms using the OpenTuner framework. An autotuner is a program that finds a combination of algorithms, or a configuration of an algorithm, that optimizes the solution of a given problem instance or set of instances. We analyse the performance gained after autotuning compilation flags for parallel algorithms in three GPU devices, and show that it is possible to improve upon the high-level optimizations of the CUDA compiler. One of the experimental settings achieved a 30% speedup.*

1. Introduction

The popularity of heterogeneous parallel computing platforms has been raising since parallel computing emerged in the last decade. Consequently, the need for optimized algorithms for these platforms has also raised. The increasing diversity and availability of parallel computing hardware complicates the task of hand-optimizing programs, and justifies automated strategies to tune and configure parallel programs.

The most widely used GPUs are those produced by NVIDIA. They are built with a set of Streaming Multiprocessors (SMs), each containing several cores called Scalar Processors (SPs). They also contain a set of Special Function Units (SFUs) and a number of load/store units. The multiprocessors execute asynchronously and in parallel. The SM schedules parallel threads in groups of 32, called warps, which can use load/store units concurrently, enabling 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 pseudo-assembly Parallel Thread Execution code (PTX), which is executed by NVIDA GPUs. CUDA applications are organized in functions executed on the GPUs, called kernels. 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 enables the usage of multiple load/store units available per SM.

The program autotuning problem fits in the framework of the Algorithm Selection Problem, introduced by Rice in 1976 [2]. The objective of an autotuner is to select the best algorithm, or algorithm configuration, for each instance of a problem. Algorithms or configurations are selected by their performance, which is measured by the time to solve the problem instance, the accuracy of the solution or the energy consumed. The set of all possible algorithms and configurations that solve a problem define a *search space*. Guided by the performance metrics, various optimization techniques search this space for the algorithm or configuration that best solves the problem.

Autotuners can specialize in domains such as matrix multiplication [3], dense [4] or sparse [5] matrix linear algebra, and parallel programming [6]. Other autotuning frameworks provide more general tools for the representation and search of program configurations, enabling the implementation of autotuners for different problem domains [7, 8].

In this paper we use the OpenTuner framework [7] to implement an autotuner for the parameters of the CUDA [1] compiler. We use the 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 GPU code by autotuning the parameters of the CUDA compiler. The compilation parameters generated by the autotuner often beat the compiler high-level optimization options, such as `-O1`, `-O2` and `-O3`. For a certain program, a set of compilation parameters achieved a 30% speedup. The experiments' data also confirms 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 optimization of GPU programs. Section 3 describes the GPU testbed, the algorithm benchmark, 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

Rice's conceptual framework [2] formed the foundation of autotuners in various problem domains. In 1997, the PHiPAC system [3] used code generators and search scripts to automatically generate high performance code for matrix multiplication. Since then, systems tackled different domains with a diversity of strategies. Whaley *et al.* [4] introduced the ATLAS project, that optimizes dense matrix multiply routines. The OSKI [5] library provides automatically tuned kernels for sparse matrices. The FFTW [9] library provides tuned C subroutines for computing the Discrete Fourier Transform. In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [6] implements abstractions for OpenMP, MPI and OpenCL, and generates optimized parallel code for heterogeneous multi-core architectures.

Some autotuning systems provide generic tools that enable the implementation of autotuners in various domains. PetaBricks [10] is a language, compiler and autotuner that introduces abstractions, such as the "*either...or*" construct, that enable programmers to define multiple algorithms for the same problem. The ParamILS framework [8] applies

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

stochastic local search methods for algorithm configuration and parameter tuning. The OpenTuner framework [7] provides ensembles of techniques that search spaces of program configurations. Bosboom *et al.* and Eliahu use OpenTuner to implement a domain specific language for data-flow programming [11] and a framework for recursive parallel algorithm optimization [12].

GPU applications hide pipeline computation and communication latency by executing interleaved parallel threads. When the instruction pipeline is saturated, the performance of the application run close to the peak. When the pipeline is under-utilized the performance can be very unpredictable and harder to model [13, 14]. It is critical to model GPU caches [15] and consider the effects of memory access divergence [16, 17].

The availability of GPU hardware, tools and frameworks enable beginner and expert developers to take advantage of GPU architectures. However, knowledge about the architecture of specific GPUs is required to achieve optimal performance for particular algorithms [18, 19]. Some of the work on GPU autotuning [20, 21, 22] aim to tune parameters such as block sizes, tiling techniques, transfers between single and double precision, loop permutations and unrolling.

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

This section discusses the experiments with OpenTuner, the GPU algorithms implemented, and the GPU architectures used as testbed.

Testbed

We performed experiments in three Kepler architecture GPUs, with compute capability (*c.c.*) 3.x: the GeForce GTX-680; the Tesla-K20; and the Tesla-K40. Note that the GTX-680 has the fastest clock, the least amount of *L2* cache and global memory, and also the smallest bandwidth and bus. Table 1 summarizes their specifications.

Model	c.c.	Global Memory	Bus	Bandwidth	L2	SM/Cores	Clock
GTX-680	3.0	2 GB	256-bit	192.2 GB/s	512 KB	8/1536	1006 Mhz
Tesla-K20	3.5	4 GB	320-bit	208 GB/s	1280 KB	13/2496	706 Mhz
Tesla-K40	3.5	12 GB	384-bit	276.5 GB/s	1536 KB	15/2880	745 Mhz

Table 1. Hardware specifications of the GPUs of the testbed

GPU Algorithms

The benchmark used in the experiments comprises four optimization strategies for *matrix multiplication* and one solution for the *maximum sub-array problem*. The remaining of this section discusses the implementation of these algorithms.

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

Non-coalesced accesses happen when global memory contains irregular references to data. The optimizations affect only the performance of the communication between threads. In optimization (#2), the data access pattern is modified, enabling coalesced accesses to data in the global memory. Optimization (#3) uses shared memory to load data from global memory, processing them with a lower latency. Similarly to optimization (#2), optimization (#4) changes the coalesced memory accesses from the source code of optimization (#3). All performance measurements for these optimizations used square matrices, with $N = 1024$.

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 gene identification, analysis of sequences of protein and DNA, and identification of hydrophobic regions. The maximum sub-array problem can be solved sequentially in $O(N)$ comparisons [23], and in $O(N/t)$ with a parallel solution [24], where t is the number of threads.

The implementation [25] used in this paper creates a kernel with 4096 threads, divided in 32 blocks with 128 threads. The N elements are divided in intervals of N/t, and each block receives a portion of the array. The blocks use the shared memory for storing segments, 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 memory for later processing. All performance measurements for the Maximum Sub-array Problem used arrays with $N = 134217728$.

Compilation Parameters

Table 2 shows the subset of the CUDA configuration parameters used in the experiments. These options target different compilation steps: the *PTX* optimizing assembler; the *NVLINK* linker; and the *NVCC* compiler. We compared the performance of programs generated by tuned parameters with the standard compiler optimizations, namely `--opt-level=0,1,2,3`. The `--opt-level` parameter was also among the parameters that could be selected by the tuner. We did not use compiler options that target the host linker or the library manager, since they do not affect performance.

Autotuning GPU Applications

OpenTuner search spaces are defined by *Configurations*, composed of different *Parameter* types. Each type has restricted bounds, and implements its own manipulation functions, enabling the exploration of the search space. OpenTuner implements ensembles of optimization techniques that perform well in different problem domains.

Results found during the search process are shared between techniques through a common database. OpenTuner uses *meta-techniques* for coordinating the distribution of resources between techniques. An OpenTuner application can implement its own search techniques and meta-techniques.

Step	Options
NVCC	`prec-sqrt`, `relocatable-device-code`, `no-align-double`, `use-fast-math`, `gpu-architecture`, `ftz`, `prec-div`
PTX	`def-load-cache`, `opt-level`, `fmad`, `allow-expensive-optimizations`, `maxrregcount`
NVLINK	`preserve-relocs`

Options	`gpu-architecture`	`opt-level`	`def-load-cache`	`maxrregcount`
Values	`sm_20`, `sm_21`, `sm_30`, `sm_32`, `sm_35`	`0 - 1`	`ca`, `cg`, `cv`, `cs`	`16 - 64`

Table 2. The set of compiler parameters used by the autotuner and their allowed values, grouped by CUDA compilation step. Parameters whose values are not listed are boolean.

The autotuner we implemented uses OpenTuner's parameter types to represent CUDA compilation options. Compiler flags and multi-valued parameters are represented by *EnumParameter*s, and numerical parameters by *IntegerParameter*s. The same encoding is used to implement a GCC autotuner [7].

The unrestricted search in the space of all compilation options generates many invalid combinations due to incompatible flags or architecture restrictions. Since the autotuner has no knowledge about the search space or the programs being tuned, it could accept the very fast results produced when there are compilation or execution errors. Our implementation always checks for errors during the compilation phase and verifies program results, considering only the performance of correct programs.

The incorrect programs generated during autotuning can be used to better understand and prune the search space, enabling search techniques to test only valid combinations and achieve better results faster.

The code for the autotuner and all the results for the experiments are available[2] under the GNU General Public License. The core functions of the autotuner are `run` and `manipulator`. The `manipulator` function builds the representation of the compiler parameters using OpenTuner's types. The `run` function compiles and runs programs, checking for errors in both steps.

We did not allow the autotuner to produce invalid compilation options. Thus, if the tuner finds errors during the compilation phase it halts with an error value. Next, if it finds errors during execution of the program the autotuner saves the compilation options

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

to a file and assigns a penalty value to the configuration. Otherwise, it returns the runtime of the program as the fitness measure of the configuration. All tuning runs were arbitrarily 1 hour long.

4. Results

This section comprises two parts. The first part presents the performance improvement achieved by the autotuned compilation options in the five algorithms of the benchmark. The second part discusses the performance and accuracy of the autotuner.

4.1. Performance Gains

The boxplots[3] in Figures 1, 2, 3 and 4 present the performance distributions, over 20 measurements, of the high-level optimizations and 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 represent 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, circles represent the outliers.

The autotuned options achieved gains in performance of up to 30% in comparison with the high-level CUDA options for the matrix multiplication algorithm with non-coalesced accesses to global memory. The performance improvement was up to 9.22% for the algorithm with coalesced accesses to global memory. 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 for the algorithm with shared memory and coalesced accesses to global memory in the GTX-680 GPU.

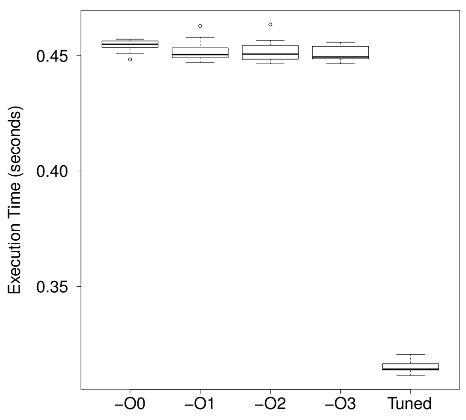

Figure 1. Boxplots for the GTX-680 GPU, comparing autotuned results and high-level compiler optimization options for the matrix multiplication algorithm with non-coalesced global memory accesses (optimization #1).

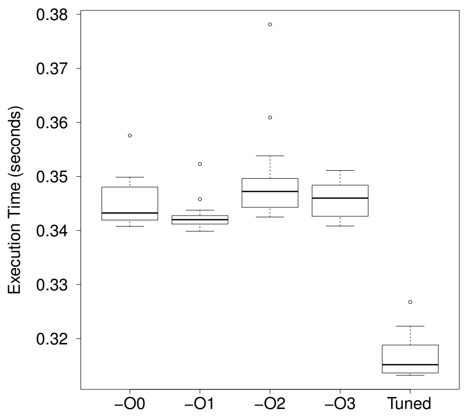

Figure 2. Boxplots for the GTX-680 GPU, comparing autotuned results and high-level compiler optimization options for the matrix multiplication algorithm with coalesced global memory accesses (optimization #2).

[3]The boxplots were made with standard implementations available in the R language.

Figure 3 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 4 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.

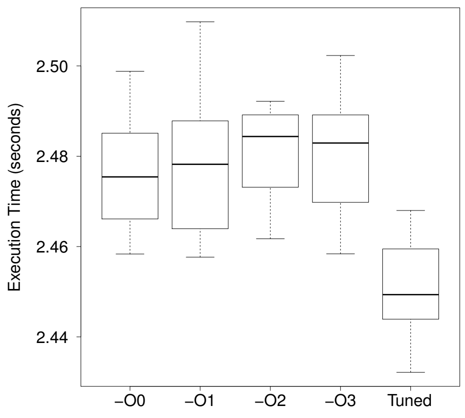

Figure 3. Boxplots for the Tesla-K40 GPU, comparing autotuned results and high-level compiler optimization options for the maximum sub-array problem.

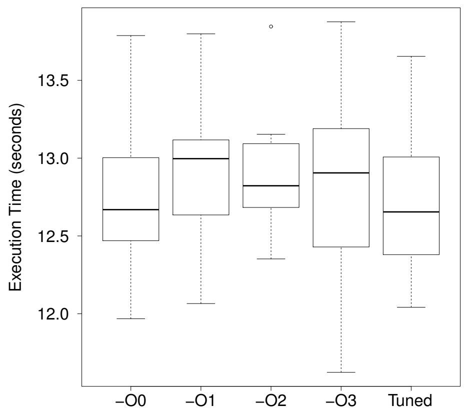

Figure 4. Boxplots for the Tesla-K20 GPU, comparing autotuned results and high-level compiler optimization options for the maximum sub-array problem.

A summarization of the tuning results is presented in Figure 6. The autotuned solutions did not achieve improvements for optimization #3 in any of the GPUs of the testbed. The autotuner achieved speedups for the GTX-680 in the algorithms #1, #2, #4 and *Sub-Array*. Experiments in the Tesla-K20 had the worst improvement results, the autotuner produced configurations worse than the high-level optimizations in algorithms #1, #2 and #4. The autotuner improved upon high-level options in all GPUs for the *Sub-Array* algorithm.

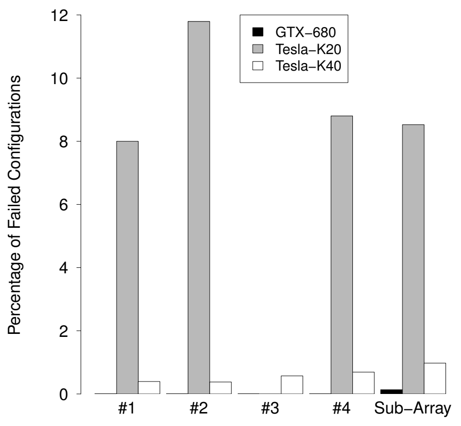

Figure 5. Percentage of all tested configurations that compiled the GPU programs without errors, but generated binaries that produced incorrect results.

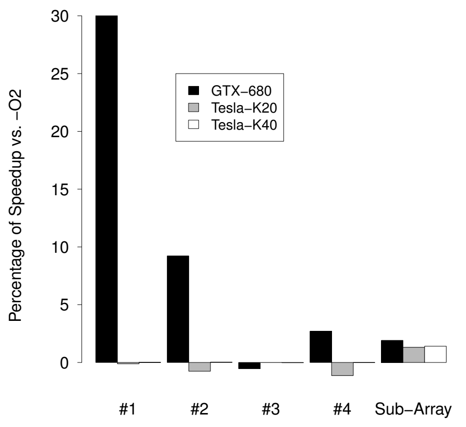

Figure 6. Summary of the speedups achieved versus *-O2* for the algorithms in the benchmark.

Autotuning in the GTX-680 presented the best results. Table 1 shows that this GPU has the fastest clock, the least amount of *L2* cache and global memory, and also the smallest bandwidth and bus in the testbed. We believe that the GTX-680 hardware limitations relative to the other GPUs in the testbed left more cache and memory optimization opportunities for the compiler. The GTX-680 was also the only GPU in which applications compiled with a compute capability 3.2, specified by the `sm_32` parameter, did produce correct results.

The autotuner logged the configurations that *passed the compilation phase* and generated binaries that *did not crash* but still produced incorrect results. Assertions in the programs exposed these cases, considered to be *failed configurations*. Figure 5 summarizes the failed configurations. The Tesla-K20 had the highest percentage of such configurations, for all algorithms.

All failed configurations in the Tesla-K20 and K40 for the #1 algorithm specify the `sm_32` compute capability. This exposes a bug in NVCC where a program fails silently, that is, it runs but produces incorrect results when compiled successfully with an unsupported compute capability. The failed configurations for the *Sub-Array* algorithm in the GTX-680 exposed conflicts between the simultaneous specification of `--fmad=true` and `--ftz=false`.

4.2. Autotuner Performance

This section presents an assessment of the performance of the autotuner. Figures 7 and 8 present the performance of the best solutions versus the time when they were found, measured in seconds of tuning.

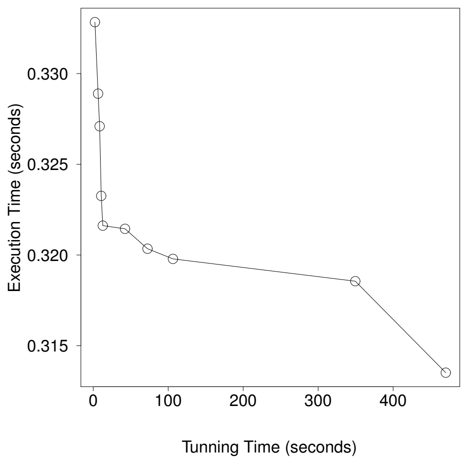

Figure 7. Best solutions found by the autotuner over time for the matrix multiplication algorithm with coalesced accesses to global memory (optimization #2) in the GTX-680 GPU.

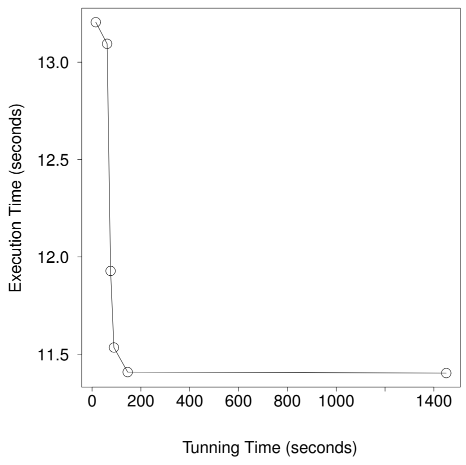

Figure 8. Best solutions found by the autotuner over time for the maximum sub-array problem in the Tesla-K20 GPU.

The fitness value in the final version of the autotuner was calculated as the mean of the performance over 20 executions of a configuration. This enabled the tuner to filter out the system fluctuations that usually interfere with the performance of a program, such as task scheduling and GPU load. Figure 7 is representative of the majority of tuning runs, which did not benefit from the entire hour of the run since the best solutions were found at the beginning.

To ensure the configurations produced were measured correctly we obtained the median of 20 executions of the solutions. Figure 2 shows that this median is very close to the best value reported by the autotuner, presented in Figure 7. Figure 8 shows a fitness value for the final best solution that is very distant from the median of 20 executions, which is shown in Figure 4. This happened because the tuning run presented in Figures 8 and 4 was made with a previous version of the autotuner, that returned a single value as the fitness measure.

In that case 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 its performance was privileged by system fluctuations. Therefore, the tuner was mislead and kept finding "good" solutions that benefited from fluctuations. This could also explain why the best solution in Figure 8 took longer to be found. The implementation of autotuners should always consider the mean of multiple executions, or some other representative metric, as the fitness value for configurations or algorithms.

5. Conclusion

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 applications, and compared their performance in three Kepler architecture GPUs. Although most experimental settings were not able to improve upon high-level optimizations, the autotuner was able to achieve a speedup of 30% in one of them. We consider these to be good results, since no domain-specific search techniques were implemented.

This paper showed that it is possible to improve the performance of legacy GPU code by applying empirical and automatic tuning techniques. We present results that confirm that different compilation options can be selected to achieve performance improvements in different GPUs. We expose a compiler option that causes programs to fail silently, that is, the program is compiled successfully and does not present runtime errors, but produces incorrect results.

Future work will include application parameters, and options for the GCC compiler, that also composes the CUDA compilation chain. We would also like to apply the Programming by Optimization [26] design paradigm for GPU and parallel programming. We will extend the analysis of the interactions between compilation parameters, aiming to explain 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 supported by the São Paulo Research Foundation (FAPESP) (procs. #2012 / 23300-7), CAPES, CNPq and Hewlett-Packard. We 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[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] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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