CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **00**:1–13
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Autotuning CUDA Compiler Parameters for Heterogeneous Applications using the OpenTuner Framework

Pedro Bruel, Marcos Amarís and Alfredo Goldman

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

SUMMARY

A Graphics Processing Unit (GPU) is a parallel computing coprocessor specialized in accelerating vector operations. The enormous heterogeneity of parallel computing platforms justifies and motivates the development of automated optimization tools and techniques. The Algorithm Selection Problem consists in finding a combination of algorithms, or a configuration of an algorithm, that optimizes the solution of a set of problem instances. An autotuner solves the Algorithm Selection Problem using search and optimization techniques.

In this paper we implement an autotuner for the CUDA compiler's parameters using the OpenTuner framework. The autotuner searches for a set of compilation parameters that optimizes the time to solve a problem. We analyse the performance speedups, in comparison with high-level compiler optimizations, achieved in three different GPU devices, for 17 heterogeneous GPU applications, 12 of which are from the Rodinia Benchmark Suite. The autotuner often beat the compiler's high-level optimizations, but underperformed for some problems. We achieved over 2x speedup for *Gaussian Elimination* and almost 2x speedup for *Heart Wall*, both problems from the Rodinia Benchmark, and over 4x speedup for a matrix multiplication algorithm.

Received . . .

KEY WORDS: Autotuning, GPUs, Compilers, CUDA, OpenTuner

1. INTRODUCTION

The usage of and reliance on parallel computing models and architectures became increasingly predominant since their emergence. The enormous heterogeneity of these platforms complicates the task of hand-optimizing parallel computing programs, justifying and motivating the development of automated tools and techniques for program optimization.

A Graphics Processing Unit (GPU) is a parallel computing coprocessor specialized in accelerating vector operations such as graphics rendering. The General Purpose computing on Graphics Processing Unit methodology, or GPGPU, consists in providing accessible programming interfaces for languages such as C and Python that enable the use of GPUs in different parallel computing domains. The Compute Unified Device Architecture (CUDA) is a GPGPU platform introduced by the NVIDIA corporation.

The program autotuning problem fits in the framework of the Algorithm Selection Problem, introduced by Rice in 1976 [1]. 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 according to performance metrics such as the time to solve the problem instance, the accuracy of the

[*] Correspondence to: Instituto de Matemática e Estatística (IME), - CCSL - Universidade de São Paulo (USP), R. do Matão, 1010 – Cidade Universitária, São Paulo – SP, 05508-090

solution and the energy consumed. The set of all possible algorithms and configurations that solve a problem defines a *search space*. Various optimization techniques search this space, guided by the performance metrics, for the algorithm or configuration that best solve the problem.

There are specialized autotuners for domains such as matrix multiplication [2], dense [3] or sparse [4] matrix linear algebra, and parallel programming [5]. Other autotuning frameworks provide more general tools for the representation and search of program configurations, enabling the implementation of autotuners for different problem domains [6, 7].

The OpenTuner framework [6] provides tools for the implementation of autotuners for various problem domains. It implements different search techniques that explore the same search space for program optimizations. Running and measuring program execution time — that is, the empirical exploration of the search space — is done sequentially. The framework also provides support for parallel compilation.

In this paper we implemented an autotuner for the CUDA compiler using the use the OpenTuner framework [6] and used it to search for the compilation parameters that optimize the performance of 17 heterogeneous GPU applications, 12 of which are from the Rodinia Benchmark Suite [8]. We used 3 different NVIDIA GPUs in the experiments, the Tesla K40, the GTX 980 and the GTX 750.

Our main contribution is to show that it is possible to optimize code written for GPUs by automatically tuning just the parameters of the CUDA compiler. We propose a thorough methodology for analysing result correctness and checking for invalid flag combinations and compilation errors. The optimization achieved by autotuning often beat the compiler high-level optimization options, such as `-O1`, `-O2` and `-O3`. The autotuner found compilation options that achieved over 2x speedup for the *Gaussian Elimination* problem from the Rodinia Benchmark Suite, almost 2x speedup for *Heart Wall* problem, also from Rodinia, and over 4x speedup for one of the matrix multiplication optimizations in our benchmark, in comparison with the high-level compiler 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 work related to autotuning and the optimization of GPU programs, discusses NVIDIA GPU architectures and the OpenTuner framework. Section 3 describes the GPU testbed, the algorithm benchmark, the compiler parameters that define search space and the implementation of the autotuner. Section 4 presents the results of the autotuning experiments, and discusses the performance of the autotuner and the selection of compiler flags. Section 5 concludes the paper.

2. RELATED WORK AND BACKGROUND

Rice's conceptual framework [1] formed the foundation of autotuners in various problem domains. In 1997, the PHiPAC system [2] 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.* [3] introduced the ATLAS project, that optimizes dense matrix multiplication routines. The OSKI [4] library provides automatically tuned kernels for sparse matrices. The FFTW [9] library provides tuned C subroutines for computing the Discrete Fourier Transform. Periscope [10] is a distributed online autotuner for parallel systems and single-node performance. In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [5] implements abstractions for OpenMP, MPI and OpenCL, and generates optimized parallel code for heterogeneous multi-core architectures.

Some systems provide generic tools that enable the implementation of autotuners in various domains. PetaBricks [11] 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 [7] applies stochastic local search methods for algorithm configuration and parameter tuning. The OpenTuner framework [6] 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 [12] and a framework for recursive parallel algorithm optimization [13].

DOI: 10.1002/cpe

The accuracy of a GPU performance model is subject to low level elements such as instruction pipeline usage and small cache hierarchies. A GPU's performance approaches its peak when the instruction pipeline is saturated, but becomes unpredictable when the pipeline is under-utilized [14, 15]. Considering the effects of small cache hierarchies [16, 17] and memory-access divergence [18, 19] is also critical to a GPU performance model.

Guo and Wang [20] introduce a framework for autotuning the number of threads and the sizes of blocks and warps used by the CUDA compiler for sparse matrix and vector multiplication GPU applications. Li *et al.* [21] discuss the performance of autotuning techniques in highly-tuned GPU General Matrix to Matrix Multiplication (GEMMs) routines, highlighting the difficulty in developing optimized code for new GPU architectures. Grauer-Gray *et al.* [22] autotune an optimization space of GPU kernels focusing on tiling, loop permutation, unrolling, and parallelization. Chaparala *et al.* [23] autotune GPU-accelerated Quadratic Assignment Problem solvers. Sedaghati *et al.* [24] build a decision model for the selection of sparse matrix representations in GPUs.

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 and heterogeneous parallel applications.

2.1. *The OpenTuner Framework*

OpenTuner search spaces are defined by *Configuration*s, that are composed of *Parameter*s of various types. Each type has restricted bounds and manipulation functions that enable the exploration of the search space. OpenTuner implements ensembles of optimization techniques that perform well in different problem domains. The framework uses *meta-techniques* to coordinate the distribution of resources among techniques. Results found during search are shared through a database. An OpenTuner application can implement its own search techniques and meta-techniques, making the ensemble more robust. OpenTuner's source code is available[†] under the MIT License.

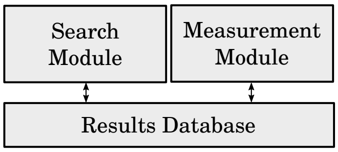

Figure 1. Simplified OpenTuner Architecture.

Figure 1 shows a high-level view of OpenTuner's architecture. Measurement and searching are done in separate modules, whose main classes are called *drivers*. The search driver requests measurements by registering configurations to the database. The measurement driver reads those configurations and writes back the desired results. Using the measurement driver available in the framework, the measurements are performed sequentially. Before running the user-defined measurement function, the OpenTuner measurement driver calls compilation hooks that can be run in parallel using Python threads.

OpenTuner implements optimization techniques such as the Nelder-Mead [25] simplex method and Simulated Annealing [26]. A resource sharing mechanism, called *meta-technique*, aims to take advantage of the strengths of each technique by balancing the exploitation of a technique that has produced good results in the past and the exploration of unused and possibly better ones.

2.2. *NVIDIA GPU Microarchitecture*

NVIDIA GPU architectures have multiple asynchronous and parallel Streaming Multiprocessors (SMs) which contain Scalar Processors (SPs), Special Function Units (SFUs) and load/store units.

[†] Hosted at GitHub: `github.com/jansel/opentuner` [Accessed on 10 February 2015]

DOI: 10.1002/cpe

Each group of 32 parallel threads scheduled by and SM, or *warp*, is able to read from memory concurrently [27]. The Tesla, Fermi, Kepler and Maxwell NVIDIA architectures vary in a large number of features, such as number of cores, registers, SFUs, load/store units, on-chip and cache memory sizes, processor clock frequency, memory bandwidth, unified memory spaces and dynamic kernel launches. Those differences are summarized in the Compute Capability (C.C.) of an NVIDIA GPU.

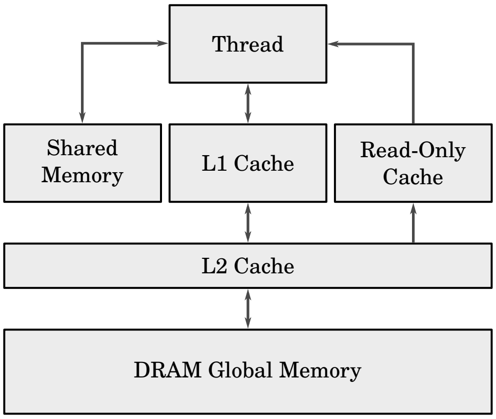

Figure 2. Memory hierarchy of threads in a kernel executed in Kepler architectures, adapted from [28]

The hierarchical memory of an NVIDIA GPU contains global and shared portions. Global memory is big, off-chip, has a high latency and can be accessed by all threads of the kernel. Shared memory is small, on-chip, has a low-latency and can be accessed only by threads in a same SM. Each SM has its own shared L1 cache, and new architectures have coherent global L2 caches. Optimizing thread accesses to different memory levels is essential to achieve good performance. Figure 2 shows the memory access hierarchy in a Kepler architecture.

2.3. *Compute Unified Device Architecture (CUDA)*

The CUDA programming model and platform enables the use of NVIDIA GPUs for scientific and general purpose computation. A single *master* thread runs in the CPU, launching and managing computations on the GPU. Data for the computations has to be transferred from the main memory to the GPU's memory. Multiple computations launched by the master thread, or *kernels*, can run asynchronously and concurrently. If the threads from a same warp must execute different instructions the CUDA compiler must generate code to branch the execution correctly, making the program lose performance due to this *warp divergence*.

The CUDA language extends C and provides a multi-step compiler, called *NVCC*, that translates CUDA code to Parallel Thread Execution code, or *PTX*. *NVCC* uses the host's C++ compiler in several compilation steps, and also to generate code to be executed in the host. The final binary generated by *NVCC* contains code for the GPU and the host[29]. When *PTX* code is loaded by an application at run-time, it is compiled to binary code by the host's device driver. This binary code can be executed in the device's processing cores, and is architecture-specific. The targeted architecture can be specified using *NVCC* parameters[30].

3. EXPERIMENTS

In this section we present the GPU testbed, the algorithm benchmark and the autotuner implementation and its search space.

DOI: 10.1002/cpe

3.1. GPU Testbed

To be able to show that different GPUs require different options to improve performance, and that it is possible to achieve speedups in different hardware, we wanted to tune our benchmark for different NVIDIA microarchitectures. We selected the Tesla K40, the GTX 750 and the GTX 980. Using the GK110B chip, the Tesla K40 is a Kepler microarchitecture GPU, the oldest GPU of the testbed. The GTX 750 is the first Maxwell microarchitecture GPU and uses the GM107 chip. The GTX 980 is the latest Maxwell GPU and uses the GM204 chip. Table I summarizes the hardware characteristics of the three GPUs.

Model	C.C.	Global Memory	Bus	Bandwidth	L2	Cores/SM	Clock
Tesla-K40	3.5	12 GB	384-bit	276.5 GB/s	1.5 MB	2880/15	745 Mhz
GTX-750	5.0	1 GB	128-bit	86.4 GB/s	2 MB	512/4	1110 Mhz
GTX-980	5.2	4 GB	256-bit	224.3 GB/s	2 MB	2048/16	1216 Mhz

Table I. Hardware specifications of the GPUs in the testbed

3.2. Algorithm Benchmark

We composed a benchmark with 17 heterogeneous GPU applications. The benchmark contains 4 optimization strategies for *matrix multiplication* (counted as a single application) [30], 1 vector addition problem [30], 1 solution for the *maximum sub-array problem* [31], 2 *sorting* algorithms [32] and 12 applications from the Rodinia Benchmark Suite [33]. The remainder of this section discusses some details of these algorithms, and introduces a three-letter code for each application, used in Section 4.

Matrix Multiplication We used four different memory access optimizations: global memory with non-coalesced accesses (MMU); global memory with coalesced accesses (MMG); shared memory with non-coalesced accesses to global memory (MSU); and shared memory with coalesced accesses to global memory (MMS). All performance measurements for these optimizations used square matrices, with $N = 8192$. The run-time complexity for a sequential matrix multiplication algorithm using two matrices of size $N \times N$ is $O(N^3)$. In a CUDA application with N^2 threads, the run-time complexity is $O(N)$

Vector Addition Algorithm For two vectors A and B, the Vector Addition (VAD) $C = A + B$ is obtained by adding the corresponding components. In a GPU algorithm each thread performs an addition of a position of the vectors A and B and stores the result in the vector C. The number of GPU threads used in this problem is equal to the size of the vectors. All performance measurements for this problem used arrays of size $N = 4194304$.

Maximum Sub-Array Problem Let X be a sequence of N integer numbers $(x_1, ..., x_N)$. The Maximum Sub-Array Problem (MSA) 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 [34], and in $O(N/t)$ with a parallel solution [35], where t is the number of threads.

The implementation [31] 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 $N = 1073741824$.

Sorting Algorithms The benchmark contains two sorting algorithms, quicksort (QKS, with $N = 65536$) and bitonicsort (BTN, with $N = 4194304$) [‡]. Both algorithms sort random numbers sampled from a uniform distribution.

Rodinia Benchmark Suite Rodinia's [8] applications have implementations for multi-core CPUs and GPUs using three parallel programming models (OpenMP, CUDA and OpenCL). The benchmark was devised for heterogenous parallel computing research and its applications represent different high-level domains or behaviours, called the Berkeley dwarfs [36, 37]. Table II shows the Rodinia applications contained in our benchmark.

Application	Berkeley Dwarf[37]	Domain
B+Tree (BPT)	Graph Traversal	Search
Back Propagation (BCK)	Unstructured Grid	Pattern Recognition
Breadth-First Search (BFS)	Graph Traversal	Graph Algorithms
Gaussian Elimination (GAU)	Dense Linear Algebra	Linear Algebra
Heart Wall (HWL)	Structured Grid	Medical Imaging
Hot Spot (HOT)	Structured Grid	Physics Simulation
K-Means (KMN)	Dense Linear Algebra	Data Mining
LavaMD (LMD)	N-Body	Molecular Dynamics
LU Decomposition (LUD)	Dense Linear Algebra	Linear Algebra
Myocyte (MYO)	Structured Grid	Biological Simulation
Needleman-Wunsch (NDL)	Dynamic Programming	Bioinformatics
Path Finder (PTF)	Dynamic Programming	Grid Traversal

Table II. Rodinia applications used in the experiments

3.3. *The Autotuner*

Our implementation uses the OpenTuner framework's native parameter types to represent CUDA compilation options. 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 Jansel *et al.* [6] in the implementation of an autotuner for GCC compilation parameters.

The autotuner we implemented used the *multi-armed bandit with sliding window, area under the curve credit assignment* meta-technique, simply named *AUC Bandit*. AUC Bandit is OpenTuner's core meta-technique [6], and its ensemble of search techniques is composed by implementations of the Nelder-Mead algorithm and three variations of genetic algorithms. There is no guarantee that any tuning run will find the optimal solution, but the variance of the results found by different tuning runs decreases as tuning time progresses, as is shown in the results from the original OpenTuner paper [6].

The search in the space of CUDA options potentially generates invalid parameter combinations due to incompatible flags or architecture restrictions. To prevent these invalid configurations from misguiding the search techniques we first modified all programs in the benchmark so that they verify the correctness of their output using previously computed correct outputs. Then we made the autotuner check for errors during compilation and execution whenever testing a new configuration and assign a penalty value when finding errors. Errors could be used to acquire knowledge about the interactions and incompatibilities between parameters, which would enable the autotuner to

[‡] Obtained from: `http://digitalcommons.providence.edu/student_scholarship/7/` [Accessed on 10 February 2015]

prune the search space and find better solutions faster. The final verification step used to validate an autotuned result was to manually check the output of the CUDA toolkit's `nvprof` profiler, certifying that the CUDA kernels were launched and executed properly.

The code for our autotuner and all the experiments and results is available[§] under the GNU General Public License.

3.4. The Search Space

Flag	**Description**
`no-align-double`	Specifies that `malign-double` should not be passed as a compiler argument on 32-bit platforms. **Step**: NVCC
`use_fast_math`	Uses the fast math library, implies `ftz=true`, `prec-div=false`, `prec-sqrt=false` and `fmad=true`. **Step**: NVCC
`gpu-architecture`	Specifies the NVIDIA virtual GPU architecture for which the CUDA input files must be compiled. **Step**: NVCC **Values**: `sm_20`, `sm_21`, `sm_30`, `sm_32`, `sm_35`, `sm_50`, `sm_52`
`relocatable-device-code`	Enables the generation of relocatable device code. If disabled, executable device code is generated. Relocatable device code must be linked before it can be executed. **Step**: NVCC
`ftz`	Controls single-precision denormals support. `ftz=true` flushes denormal values to zero and `ftz=false` preserves denormal values. **Step**: NVCC
`prec-div`	Controls single-precision floating-point division and reciprocals. `prec-div=true` enables the IEEE round-to-nearest mode and `prec-div=false` enables the fast approximation mode. **Step**: NVCC
`prec-sqrt`	Controls single-precision floating-point squre root. `prec-sqrt=true` enables the IEEE round-to-nearest mode and `prec-sqrt=false` enables the fast approximation mode. **Step**: NVCC
`def-load-cache`	Default cache modifier on global/generic load. **Step**: PTX **Values**: `ca`, `cg`, `cv`, `cs`
`opt-level`	Specifies high-level optimizations. **Step**: PTX **Values**: `0 - 3`
`fmad`	Enables the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations (FMAD, FFMA, or DFMA). **Step**: PTX
`allow-expensive-optimizations`	Enables the compiler to perform expensive optimizations using maximum available resources (memory and compile-time). If unspecified, default behavior is to enable this feature for optimization level $\geqslant$O2. **Step**: PTX
`maxrregcount`	Specifies the maximum number of registers that GPU functions can use. **Step**: PTX **Values**: `16 - 64`
`preserve-relocs`	Makes the `PTX` assembler generate relocatable references for variables and preserve relocations generated for them in the linked executable. **Step**: NVLINK

Table III. Description of flags in the search space

Table III details the subset of the CUDA configuration parameters used in the experiments [¶]. The parameters 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`. Different `--opt-levels` could also be selected during tuning. We did not use compiler options that target the host linker or the library manager since they do not affect performance. The size of

[§] Hosted at GitHub: `https://github.com/phrb/gpu-autotuning` [Accessed on 10 February 2015]
[¶] Adapted from: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc [Accessed on 10 February 2015]

the search space defined by all possible combinations of the flags in Table III is in the order of 10^6 making hand-optimization or exhaustive searches very time consuming.

4. RESULTS

This section presents the speedups achieved for all algorithms in the application benchmark, highlights the most significant speedups, and discusses the performance and accuracy of the autotuner.

4.1. Performance Improvements

All boxplots presented in this section were made using the standard implementations available for the R language. 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.

Figures 3, 4, 5 and 6 compare the distributions of 10 perfomance measurements of binaries compiled with high-level compiler optimizations and with autotuned compiler options. The results for the high-level optimizations `--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*.

Figure 3 shows that the autotuned solution for the Heart Wall problem (HWL) in the Tesla K40 achieved over 2x speedup in comparison with the high-level CUDA optimizations. Figure 4 shows, in the GTX 980, almost 2.5x speedup for the Gaussian Elimination problem (GAU). Figure 5 show 10% speedup of the autotuned solution for the Path Finder problem (PTF) in the Tesla K40. Figure 6 shows over 5% speedup of the autotuned solution, also in the Tesla K40, for the Myocyte problem (MYO).

Figures 7, 8, 9 and 10 present summaries of the results. The autotuner did not find solutions that improved upon the high-level optimizations for the problems BTN and MSU in any of the GPUs of the testbed, but it found solutions that achieved speedups for at least one GPU for the other problems.

We do not know precisely which hardware characteristics impacted performance the most. Although more experiments are needed to confirm the following hypothesis, we believe that the Maxwell GPUs, the GTX 980 and GTX 750, had differing results from the Tesla K40 because they are consumer grade GPUs, producing not so precise results and with different default optimizations. The similarities between the compute capabilities of the GTX GPUs could also explain the observed differences from the K40. Finally, the overall greater computing power of the GTX 980 could explain its differing results from the GTX 750, since the GTX 980 has a greater number of Cores, SMs/Cores, Bandwidth, Bus, Clock and Global Memory.

4.2. Autotuner Performance

This section presents an assessment of the autotuner's performance. Figures 11, 12, 13 and 14 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. The Figures show the evolution of the autotuned solution for the best results in our experiments, the Heart Wall problem (HWL) and the Gaussian Elimination problem (GAU) in the GTX 980, and the Path Finder problem (PTF) and the Myocyte problem (MYO) in the Tesla K40.

The points in each graph represent the performance, in the y-axis, of the best configuration found at the corresponding tuning time, shown in the x-axis. The leftmost point in each graph represents the performance of a configuration chosen at random by the autotuner. Each subsequent point represents the performance of the best configuration found so far.

Note that the autotuner is able to quickly improve upon the initial random configuration, but the rate of improvement decays over tuning time. The duration of all tuning runs was two hours, or 7200 seconds. The rightmost point in each graph represents the performance of the last improving

DOI: 10.1002/cpe

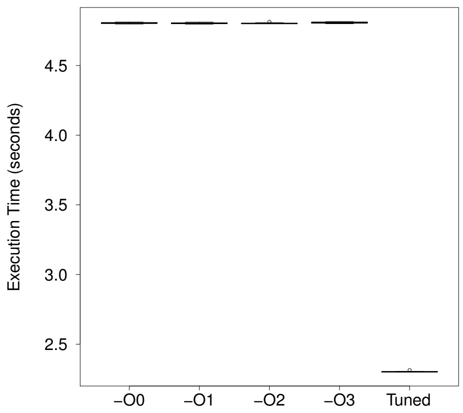

Figure 3. Boxplots for the Tesla K40, comparing autotuned results and high-level compiler optimizations for the Heart Wall problem (HWL)

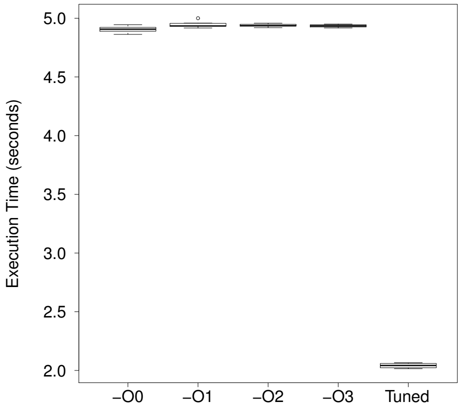

Figure 4. Boxplots for the GTX 980 GPU, comparing autotuned results and high-level compiler optimizations for the Gaussian Elimination problem (GAU)

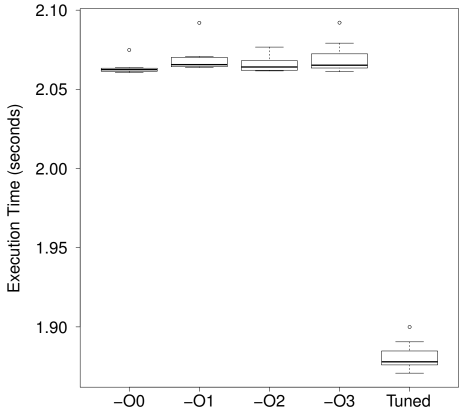

Figure 5. Boxplots for the Tesla K40, comparing autotuned results and high-level compiler optimizations for the Path Finder problem (PTF)

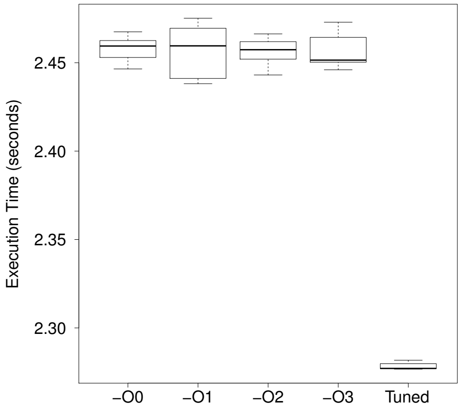

Figure 6. Boxplots for the Tesla K40, comparing autotuned results and high-level compiler optimizations for the Myocyte problem (MYO)

configuration found by the autotuner. After the tuning time of this configuration, the autotuner did not find any improving configuration until the end of its tuning run.

The autotuner used the mean of 10 performance measurements of a given configuration as the fitness value, reducing fluctuations and improving the measurement's accuracy. This is reflected in the proximity of the measurements reported by the autotuner during tuning, shown in figures 11, 12, 13 and 14 and the mean values of 10 measurements of the final solution, shown in the corresponding figures 3, 4, 5 and 6.

4.3. Parameter Selection

We attempted to associate compilation parameters to applications and GPUs using WEKA's [38] clustering algorithms. Although we could not find significant relations for most applications we detected that the `ftz=true` in MMS and the Compute Capabilities 3.0, 5.0 and 5.2 in GAU caused the speedups observed in the GTX 980 for these applications. Table IV shows clusters obtained with the K-means WEKA algorithm for autotuned parameter sets for the Rodinia Benchmark in the GTX 750. Unlike most clusters found for all GPUs and problems, these clusters did not contain an equal number of instances. The difficulty of finding associations between compiler optimizations, applications and GPUs justifies the autotuning of compiler parameters in the general case.

DOI: 10.1002/cpe

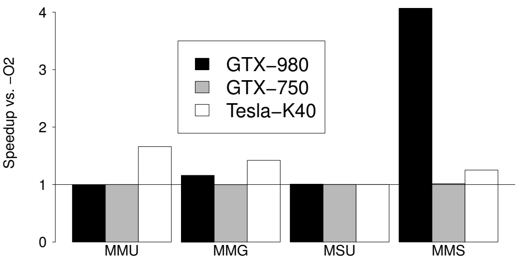

Figure 7. Summary of the speedups achieved versus *-O2* in the matrix multiplication optimizations

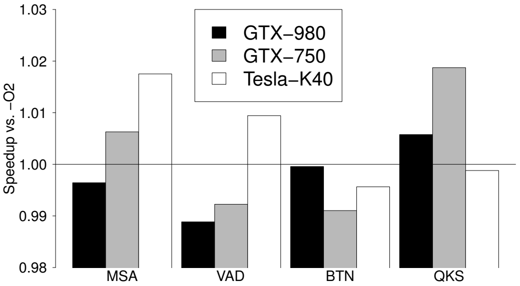

Figure 8. Summary of the speedups achieved versus *-O2* in the other independent applications

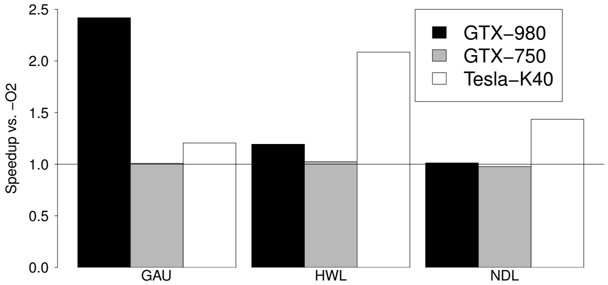

Figure 9. Summary of the biggest speedups achieved versus *-O2* in the Rodinia applications

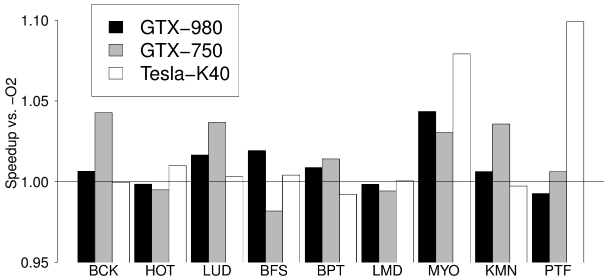

Figure 10. Summary of the smaller speedups achieved versus *-O2* in the Rodinia applications

Flag	Cluster 0 (17%)	Cluster 1 (83%)
no-align-double	on	on
use_fast_math	on	on
preserve-relocs	off	off
relocatable-device-code	true	false
ftz	true	true
prec-div	true	false
prec-sqrt	true	true
fmad	false	false
allow-expensive-optimizations	false	true
gpu-architecture	sm_20	sm_50
def-load-cache	cv	ca
opt-level	1	3
maxrregcount	42	44.6

Table IV. Parameter clusters for all Rodinia problems in the GTX 750

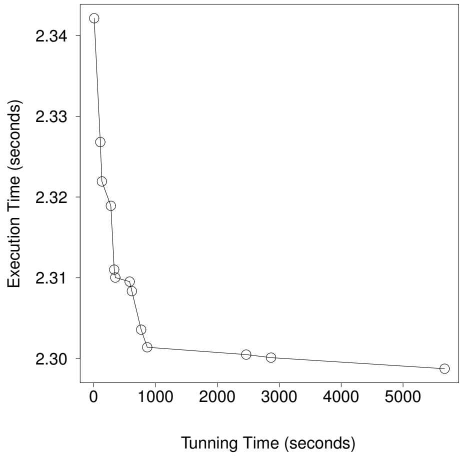

Figure 11. Best solutions found by the autotuner over time for the Heart Wall problem (HWL) in the Tesla K40

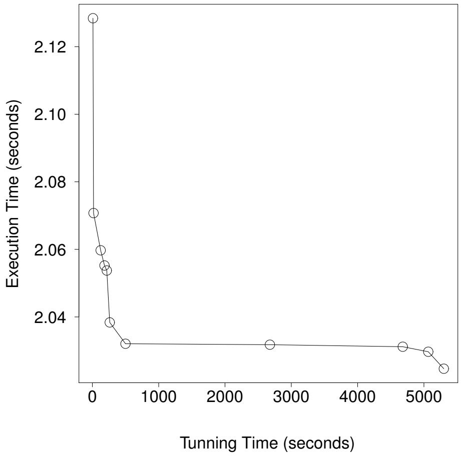

Figure 12. Best solutions found by the autotuner over time for the Gaussian Elimination problem (GAU) in the GTX 980

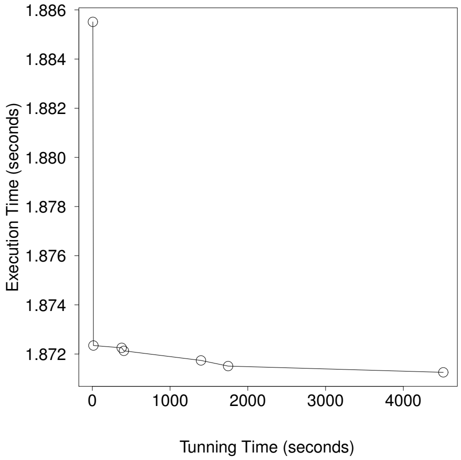

Figure 13. Best solutions found by the autotuner over time for the Path Finder problem (PTF) in the Tesla K40

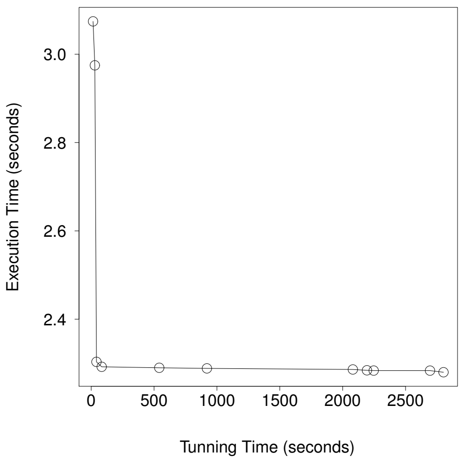

Figure 14. Best solutions found by the autotuner over time for the Myocyte problem (MYO) in the Tesla K40

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 17 heterogeneous applications and compared their performance in one Kepler and two Maxwell microarchitecture GPUs. Although the autotuner often beat the compiler's high-level optimizations it still underperformed for some problems. We achieved over 2x speedup for Gaussian Elimination (GAU) and almost 2x speedup for Heart Wall (HWL), and over 4x speedup for a matrix multiplication algorithm (MMS).

This paper showed that it is possible to improve the performance of GPU applications applying empirical and automatic tuning techniques. Our results and clustering attempts emphasize the importance of automatic optimization techniques by showing that different compilation options have to be selected in order to achieve performance improvements in different GPUs and that it is difficult to associate compiler optimization parameters, applications and GPUs.

Future work will include application parameters and options for the GCC compiler, which also composes the CUDA compilation chain. We would also like to apply the Programming by Optimization [39] design paradigm for GPU and parallel programming. We will continue to apply clustering algorithms to future experiments and we will perform more comprehensive experiments to explore the search space of CUDA compiler parameters. We hope these approaches will enable us to provide valuable guidelines for CUDA parameter selection in the future.

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

ACKNOWLEDGEMENTS

This work was supported by Hewlett-Packard, FAPESP (São Paulo Research Foundation, grant number #2012/23300-7), CAPES and CNPq. We also thank NVIDIA for the donation of a Tesla K40 GPU.

REFERENCES

1. J. R. Rice, "The algorithm selection problem," in *Advances in Computers 15*, 1976, pp. 65–118.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, Dec 2010, pp. 1–11.
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. M. Gerndt and M. Ott, "Automatic performance analysis with periscope," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 736–748, Apr. 2010.
11. 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.
12. 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.
13. 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.
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. M. Amarís, D. Cordeiro, A. Goldman, and R. Y. Camargo, "A simple bsp-based model to predict execution time in gpu applications," in *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*, Dec 2015, pp. 285–294.
16. T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, "A performance model for gpus with caches," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 1800–1813, July 2015.
17. J. Picchi and W. Zhang, "Impact of l2 cache locking on gpu performance," in *SoutheastCon 2015*, April 2015, pp. 1–4.
18. 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.
19. 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.
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. A. Chaparala, C. Novoa, and A. Qasem, "Autotuning gpu-accelerated qap solvers for power and performance," in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on*

Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on. IEEE, 2015, pp. 78–83.

24. N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 99–108.
25. J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.
26. S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi *et al.*, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.
27. NVIDIA Corporation, *CUDA C Best Practices Guide*, August 2014.
28. N. Corporation, "Web site: [Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110] Visited on Nov, 2015."
29. NVIDIA, *CUDA COMPILER DRIVER NVCC*, September 2015.
30. ——, *CUDA C: Programming Guide, Version 7.*, March 2015.
31. 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.
32. A. Sinha and K. Agrawa, "[Web site: Digital Commons at Providence College. Sorting in CUDA: by Ayushi Sinha] Visited on Jan, 2016."
33. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54.
34. 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.
35. 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.
36. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," TECHNICAL REPORT, UC BERKELEY, Tech. Rep., 2006.
37. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009.
38. G. Holmes, A. Donkin, and I. H. Witten, "Weka: A machine learning workbench," in *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*. IEEE, 1994, pp. 357–361.
39. H. H. Hoos, "Programming by optimization," *Communications of the ACM*, vol. 55, no. 2, pp. 70–80, 2012.