Application-independent Autotuning for GPUs

Martin TILLMANN, Thomas KARCHER, Carsten DACHSBACHER,
Walter F. TICHY

Karlsruhe Institute of Technology
Am Fasanengarten 5, Karlsruhe, Germany

Abstract. Autotuning is an established technique for adjusting performance-critical parameters of applications to their specific run-time environment. In this paper, we investigate the potential of online autotuning for general purpose computation on GPUs. Our application-independent autotuner *AtuneRT* optimizes GPU-specific parameters such as block size and loop-unrolling degree. We also discuss the peculiarities of autotuning on GPUs. We demonstrate tuning potential using CUDA and by instrumenting the parallel algorithms library Thrust. We evaluate our online autotuning approach with various GPUs and sample applications.

1. Introduction

Autotuning has emerged as an effective and automated technique for optimizing parallel applications. Most work in the autotuning area has concentrated on application specific tuning parameters such as the replication factor for tasks or the number of threads in data parallel operations. On graphic processing units (GPUs), there are various tunable parameters that are application independent. These parameters include the number of warps combined into blocks and the degree of loop-unrolling. These parameters must be specified in every GPU call. This paper shows how an automatic tuner can optimize block size and loop unrolling and presents the performance gains of four examples on three current GPUs.

2. Online Autotuning

Autotuning is an automatic optimization technique that adjusts tuning parameters of an application or its execution environment, with the aim of optimizing some criterion such as execution time, memory usage or energy consumption. An online autotuner operates while the application is running and can therefore adjust to varying input or environment characteristics. An autotuner drives a feedback loop that (1) measures one or more sections of a program, and (2) adjusts the tuning parameter configuration. A search-based algorithm such as *Nelder-*

Mead [1] guides the process. An autotuner may walk into local minima and settle on a suboptimal configuration.

Many existing autotuners use application-specific knowledge about tuning parameters to speed up the search. ATLAS [2] exploits specific knowledge about matrix sizes and shapes, MATE [3] tunes MPI-specific parameters, and PetaBricks [4] chooses among alternative algorithms. Application-independent autotuners such as Active Harmony [5] and our own autotuner [6] refrain from using such knowledge in exchange for generality.

In this work, we adapt our general-purpose run-time autotuner *AtuneRT* to optimize GPU kernels. *AtuneRT* is the successor of *Atune* [7], an off-line autotuner. The following API calls of *AtuneRT* make applications autotunable:

- $addParameter(\&param, range, default)$ informs the autotuner about the tunable variable *param*. The autotuner is allowed to vary *param* within the range given, starting with *default*. The variable could be, for instance, the number of threads to spawn.
- $startMeasurement()$ defines the beginning of a measurement section. Whenever control flow passes this call, the autotuner initializes measurement.
- $stopMeasurement()$ defines the end of a measurement section. The autotuner stops the measurement and calculates a new tuning parameter configuration.

3. Autotuning on the GPU

Modern graphics processors provide massive parallelism for a wide range of algorithms. To fully utilize GPUs it is important to optimize memory access patterns, balance workloads, and minimize control-flow costs. Optimizing on the GPU is a non-trivial task, given the peculiarities of these multi-threading architectures [8].

3.1. Kernel launch configuration

In this work, we employ NVIDIA's CUDA architecture. CUDA organizes threads in a hierarchy. A group of 32 threads form a warp, and several warps form a block. And finally, blocks are organized in a grid. Figure 1 illustrates the thread hierarchy in CUDA.

The overall thread number for a given GPU kernel is usually dictated by the number of data elements the kernel operates on. However the number of threads per block, called the block size, is variable and can be chosen at run-time.

GPU resources such as temporary registers and shared memory are partitioned among the warps and blocks. Due to limited resources available on the GPU not all warps can be active at once.

Occupancy is the ratio of active warps to the total number of warps that can run in parallel. Obviously, it is a good idea to keep as many blocks operating at all times as possible. An important performance aspect is the trade-off between occupancy and the workload per thread. As stated in the "CUDA C Best Practices Guide" [9] higher occupancy might not always result in higher performance. It
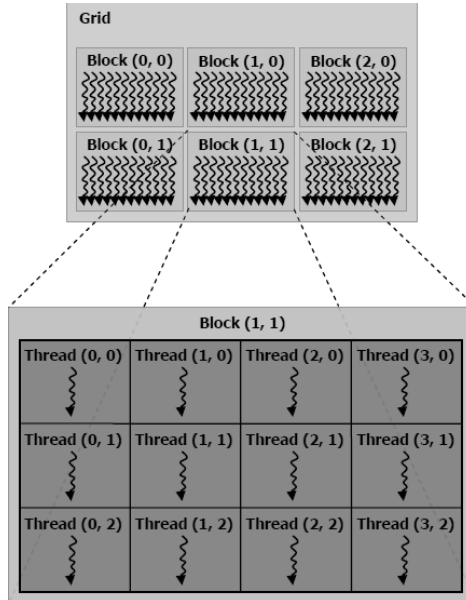
Figure 1. CUDA Thread Model. Image courtesy of NVIDIA.

can be beneficial to run applications at lower occupancy but increased workload per thread [10].

CUDA also allows explicit loop unrolling. The unrolling factor can be specified and it is the programmer's responsibility to ensure best performance.

Block size, per-thread workload, and degree of loop unrolling are tuning parameters that apply to any parallel application on CUDA. These parameters are collectively known as the kernel launch configuration.

The optimal values of these parameters are hard to determine as they not only depend on the algorithm but also the underlying architecture of the respective GPU and the characteristics of the input data. The performance impact of these parameters, in particular block size, is well-documented [11]. When tuning optimization parameters by hand, a multitude of restrictions and hardware idiosyncrasies have to be considered. Even though simple heuristics that maximize occupancy exists, the standard practice of finding the optimal kernel launch configuration is trial and error.

Autotuning removes the need for hand-tuning, because it empirically searches for the optimal configuration on any GPU. Furthermore, online autotuning allows the application to react to changes that occur at run-time, for example changes in the data size. As hardware-induced parameters affect the performance of almost all GPU programs, autotuning is applicable to a wide range of software.

In order to autotune parameters that need to be known at compile time, such as CUDA's preprocessor command for loop unrolling, we used code instantiation via templates. Code instantiation also benefits from compiler optimizations.

block size	average time	standard deviation
128	0.256	0.011
256	0.210	0.009
512	0.232	0.028
1024	0.251	0.022

Table 1. Average run-time and standard deviation of an empty kernel. All times are in milliseconds.

3.2. Measuring execution time on the GPU

One important difference to autotuning for CPUs is that measuring execution times is less accurate on GPUs. While CPUs provide exact cycle counters, GPU computation is initiated via driver calls (which might use internal caching/queuing) and the scheduling on the GPU is a black box for the programmer. On the GPU, one measures the time between the driver call and the return of the result. We insert events in the execution pipeline right before and after the kernel launch. Measuring the time difference of the events occurrences gives the best approximation of the kernel execution time. The associated, unpredictable overhead makes measurements on the GPU prone to inaccuracies for short running kernels. Figure 2 illustrates the launch of a GPU kernel and the timing events.

We used an empty kernel to measure the overhead of a kernel launch. The kernel launch time varies with changing block sizes. Table 1 lists the average time and standard deviation of 20 kernel launches on the GTX 680 graphics card. Many algorithms call multiple kernel which accumulates the overhead and potential measuring errors. It is therefore important to measure time frames that are long enough to dampen differences in the kernel launch time.
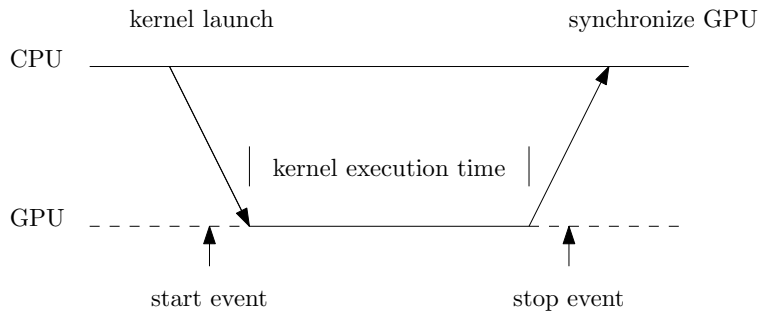

Figure 2. Measuring kernel execution time with events.

In the next sections we discuss sample applications and show their behaviour during autotuning. We evaluated them on the following three GPUs:

- Geforce GTX680 (GK104, 4096 MBytes memory, GPU clock rate 1058 MHz, memory clock rate 1502 MHz, 1536 CUDA cores)
- Geforce GTX470 (GF100, 1280 MBytes memory, GPU clock rate 607 MHz, memory clock rate 838 MHz, 448 CUDA cores)
- Quadro 6000 (GF100GL, 6144 MBytes memory, GPU clock rate, 574 MHz, memory clock rate 750 MHz, 448 CUDA cores)

4. Results

We evaluated autotuning on the GPU with four applications that come directly from NVIDIA's CUDA tool kit and are in most cases already set up for benchmarking. The sample applications documentation discusses performance implications of the kernel launch configurations but in all cases the parameters were selected by trial and error for one particular GPU.

4.1. Merge Sort

The merge sort sample implementation uses three different kernels to sort an array of 2^{22} elements. Each kernel's block size can be configured independently. Autotuning these block sizes leads to a 3-dimensional parameter space. A complete measurement of the parameter space can be seen in figure 3.
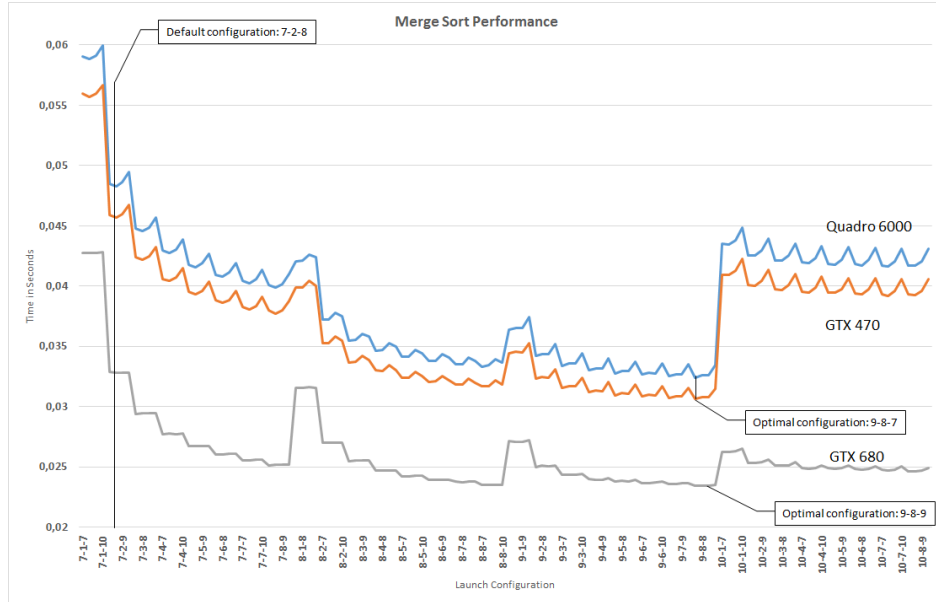

Figure 3. Merge Sort Parameter Space, block sizes of three kernels.

The three tested GPUs react similarly to changes of the kernel launch configuration, however the respective optimal values are different. Table 2 lists the achieved speed-up and how many iterations it took for the autotuner to find the optimal parameters.

4.2. N-Body Simulation

An N-body simulation is a simulated dynamic system of mass points that influence each others acceleration. An all-pairs approach with a computational complexity of $O(N^2)$ is used to compute the position of each mass point in the next time step. The implemented kernel computes the gravitation for each point by looping

	default time	optimal time	speed-up	autotuning iterations
Quadro 6000	0.0483	0.0324	32.81%	17
GTX 470	0.0457	0.0307	32.85%	20
GTX 680	0.0328	0.0234	28.64%	20

Table 2. Execution times in seconds. Speed-up of the optimal configuration relative to the default parameters.

over the other mass points and calculating their gravitational pull. The simulation implementation yields two parameters of interest:

- the block size of the kernel, and
- the degree of loop unrolling.

The documentation provided by NVIDIA discusses the effects of loop unrolling in relation to the number of simulated points. However, the results presented in the documentation use trial and error to find good values for a certain graphics card and a fixed input size. With autotuning it is possible to find the optimal tuning values for different hardware and varying input sizes. We measure the performance while simulating 131072 mass points.

Figures 4 and 5 show the performance in the complete parameter space for the Quadro 6000 and GTX 680. The GTX 470 has the same architecture as the Quadro 6000, with the main difference being the clock speed. The performance graphs of those two GPUs therefore have the same shape, with a small offset in the y-axis.

The degree of loop unrolling has a significant impact on the performance of the kernel. On the Quadro 6000 the unrolled kernel runs about 23% faster than the a kernel without loop unrolling. Important to note is that the loop unrolling parameter and the block size are not independent of each other, as can be seen in figure 4. Tuning the parameters independently will result in a non-optimal configuration.

The achieved speed-up and the number of autotuning iterations it took to find the optimum are listed in table 3. Even though the kernel was already hand optimized, *AtuneRT* was able to find a better configuration on every GPU.

	default time	optimal time	speed-up	autotuning iterations
Quadro 6000	6.361	6.304	1.0%	26
GTX 470	6.044	5.953	1.5%	20
GTX 680	2.937	2.824	3.8%	14

Table 3. Execution times in seconds. Speed-up of the optimal configuration relative to the default parameters.

4.3. Thrust Scan

Thrust[1] is a C++ template library for CUDA, similar to the Standard Template Library (STL), providing various algorithmic building blocks, such as scan, and
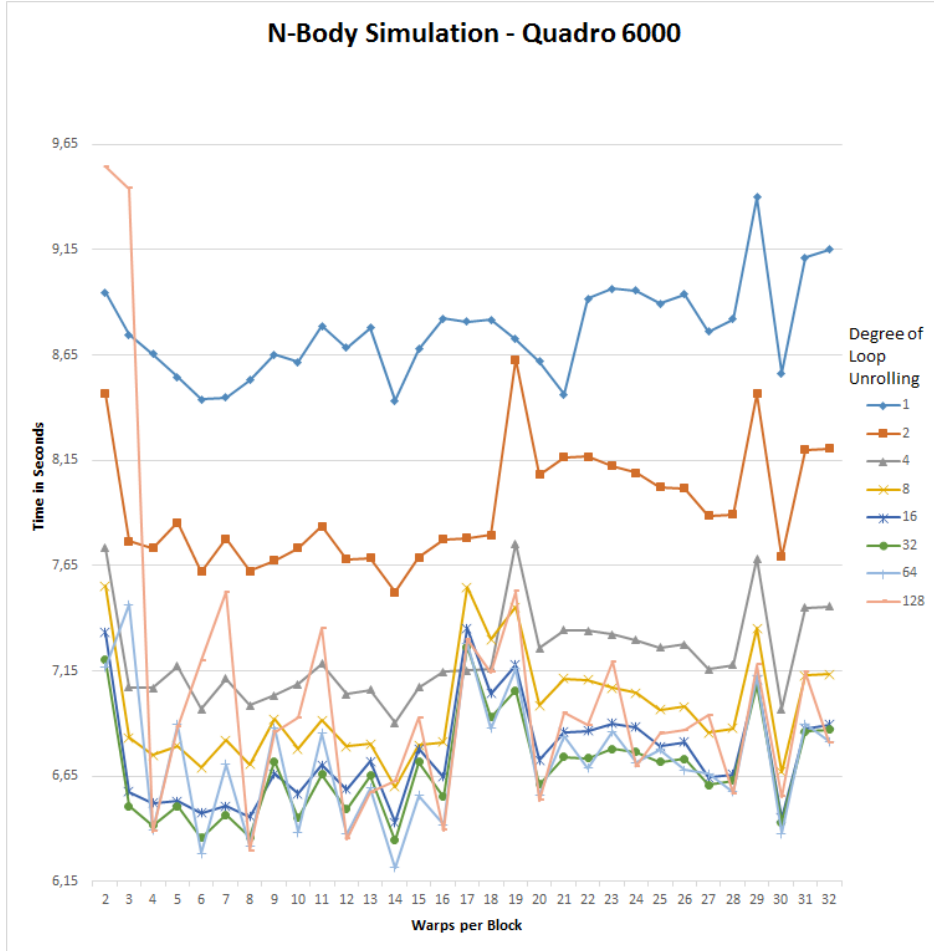
---

[1]http://docs.nvidia.com/cuda/thrust/

Figure 4. N-Body simulation – Performance of the Quadro 6000.

reduction. Thrust's algorithms either use hard-coded values or simple heuristics to determine the block size of the CUDA kernels. We applied autotuning to these parameters to optimize the kernel launch configuration.

We benchmarked 1000 calls to the `thrust::inclusive_scan`-function computing a prefix sum over 2^{24} random integers. The results are listed in table 4. The performance for all possible block sizes can be found in figure 6.

On the GTX 680 we can achieve a 13% performance gain over the non-tuned Thrust function. Our autotuner *AtuneRT* confirms that 224 is the optimal block size for the `thrust::inclusive_scan`-function on the Quadro 6000.

4.4. Marching Cubes

Marching cubes is an algorithm often used in visualization for extracting a triangular mesh of an isosurface from three-dimensional scalar fields, e.g. a computer-tomography scan, stored as a 3D regular grid. It processes cells of the 3D grid
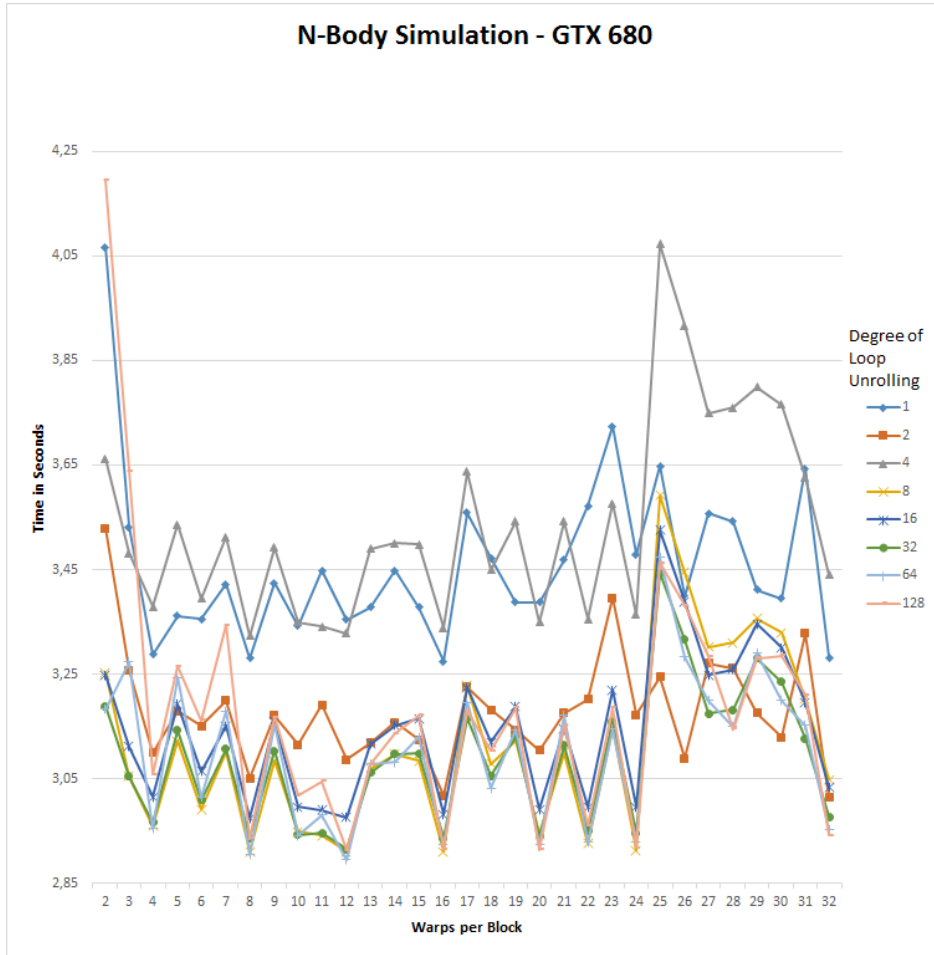
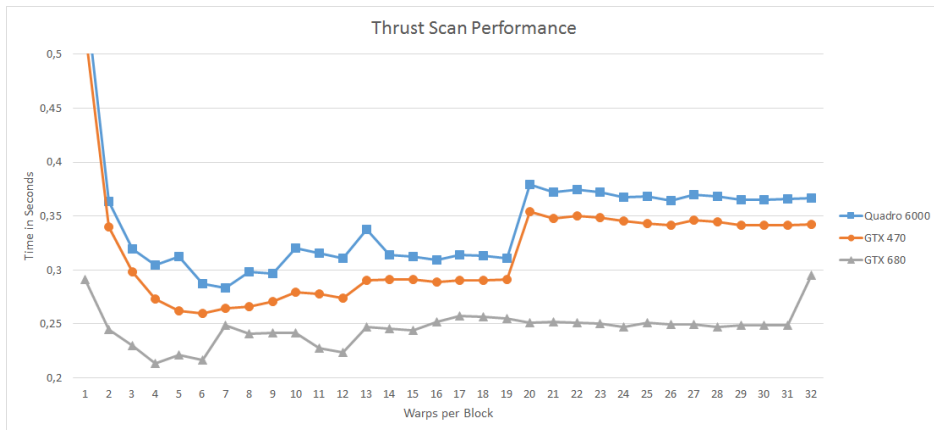**Figure 5.** N-Body simulation – Performance of the GTX 680.
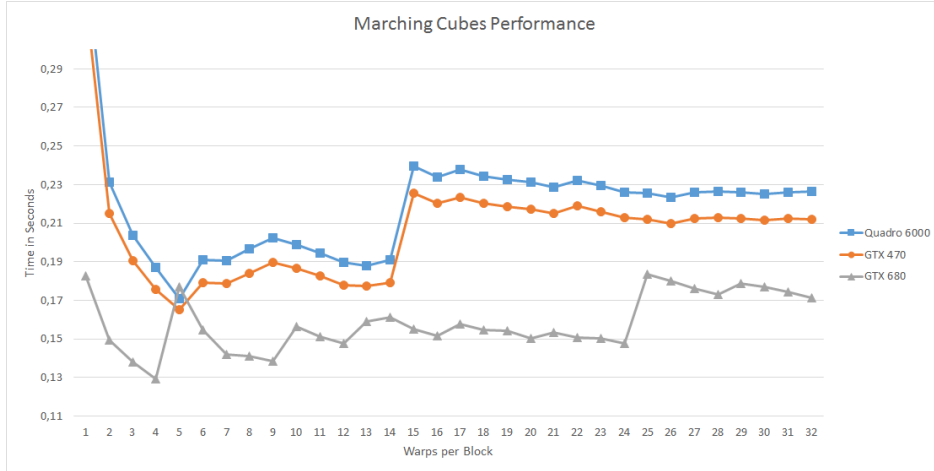

Figure 6. Performance of Thrust's scan function.

Figure 7. Marching Cube – Performance.

	Thrust Scan		Marching Cubes	
	time in seconds	block size	time in seconds	block size
GTX680, no tuner	2.566	224	0.170	160
GTX680, with tuner	2.223	128	0.134	128
Quadro 6000, no tuner	10.025	224	0.179	160
Quadro 6000, with tuner	10.026	224	0.179	224

Table 4. Performance measurements for Thrust scan and Marching Cubes.

independently and determines the triangles required to represent the part of the isosurface passing through each cell.

Figure 7 shows that the best configuration of 5 warps per block for the GTX 470 and the Quadro 6000 does not transfer to the GTX 680. The optimal configuration for the GTX 470 and the Quadro 6000 perform poorly on the GTX 680.

The NVIDIA CUDA samples include a marching cube implementation that uses Thrust. Again we apply *AtuneRT* to the block size. The results listed in table 4 show that the standard block size gives poor performance on the GTX 680. The autotuner achieves a 20% speed-up. A block size of 224 is as good as the standard block size on the Quadro 6000.

5. Conclusion

We showed that autotuning is a feasible tool for optimizing GPU applications on multiple platforms. Preparing the applications for *AtuneRT* was easy: Three API calls sufficed for every example. These three calls could easily be placed into the kernel API call. In this way, autotuning requires no extra effort for the programmer. We showed the performance impacts of the tuning parameters and the importance to tune them independently for each GPU. Autotuning also provides a way to configure GPUs without having to deal with, or even know,

the hardware specifications. Our evaluation showed that the tuning parameter space is unintuitive. For every hand-tuned sample application *AtuneRT* was able to find the optimal kernel launch configuration on every tested GPU.

Due to the heterogeneity and wide variety of GPUs we expect autotuning to become an essential part in determining tuning parameters at run-time.

References

[1] John A. Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[2] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001.

[3] Anna Morajko, Tomàs Margalef, and Emilio Luque. Design and implementation of a dynamic tuning environment. *Journal of Parallel and Distributed Computing*, 67(4):474–490, 2007.

[4] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 85–96. IEEE Computer Society, April 2011.

[5] Cristian Ţăpuş, I-Hsin Chung, Jeffrey K Hollingsworth, et al. Active harmony: towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.

[6] Thomas Karcher and Victor Pankratius. Run-time automatic performance tuning for multicore applications. In *Euro-Par 2011 Parallel Processing*, pages 3–14. 2011.

[7] Frank Otto, Christoph A. Schaefer, Matthias Dempe, and Walter F. Tichy. A language-based tuning mechanism for task and pipeline parallelism. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 328–340, Berlin, Heidelberg, 2010.

[8] Henry Wong, Misel myrto Papadopoulou, Maryam Sadooghi-alv, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246, 2010.

[9] *CUDA C Best Practices Guide*, page 39. October 2012.

[10] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

[11] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R Llanos. Understanding the impact of cuda tuning techniques for fermi. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 631–639. IEEE, July 2011.