An Empirical Evaluation of GPGPU Performance Models

Souley Madougou[1], Ana Lucia Varbanescu[1], Cees de Laat[1], and Rob van Nieuwpoort[2]

[1] University of Amsterdam, Amsterdam, The Netherlands
[2] Netherlands eScience Center, Amsterdam, The Netherlands

Abstract. Computing systems today rely on massively parallel and heterogeneous architectures to promise very high peak performance. Yet most applications only achieve small fractions of this performance. While both programmers and architects have clear opinions about the causes of this performance gap, finding and quantifying the real problems remains a topic for performance modeling tools. In this paper, we sketch the landscape of modern GPUs' performance limiters and optimization opportunities, and dive into details on modeling attempts for GPU-based systems. We highlight the specific features of the relevant contributions in this field, along with the optimization and design spaces they explore. We further use a typical kernel example (tiled dense matrix multiplication) to assess the efficacy and usability of a set of promising approaches. We conclude that the available GPU performance modeling solutions are very sensitive to applications and platform changes, and require significant efforts for tuning and calibration when new analyses are required.

Keywords: performance modeling, performance analysis and prediction, GPU architectures.

1 Introduction

Today's computing landscape is dominated by parallel, heterogeneous systems. This evolution has been triggered by the significant performance gains that many-core architectures can offer (e.g., over 1 TFLOP for a Xeon Phi CPU [1] or a regular, consumer GPU card [2,3]).

Attaining the best possible performance on such platforms for real-world applications is very challenging because it requires a lot of manual tuning: the algorithms require tailored implementations for specific hardware, and a large design and optimization space must be (manually) explored. In this trial-and-error process, both design and program tuning are based on a mix of designers' and programmers' expertise. As a result, the optimization space is only partially explored and most applications run far below their own peak performance of the given hardware system [4,5].

We believe this practice of searching for performance needs to change. Instead of using word-of-mouth and primitive, in-house performance models and empirical program tuning, we advocate a systematic approach through and beyond

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 165–176, 2014.
© Springer International Publishing Switzerland 2014

existing performance modeling and engineering tools, which have been found very useful for traditional parallel systems. However, heterogeneous computing still focuses much more on application case-studies [6] and programming systems [7,8] than on developing and using performance modeling techniques. Yet, the "affordability" of high performance promised by the ubiquity of heterogeneous computing can only be enjoyed with the support of the appropriate performance modeling tools and approaches.

The first step in this direction is to provide a clear picture of the tools we have available now. Thus, the contribution of this work is twofold. First, we sketch the performance modeling landscape for heterogeneous systems by focusing on representative and promising methods. Second, for our empirical evaluation, we select seven different approaches to performance modeling specific to GPUs. For each one of them, we highlight the methodology and the specific features, the design and/or optimization space it explores, its possible shortcomings; we further apply it to a tiled dense matrix multiplication kernel from the CUDA SDK on four platforms from two generations of NVIDIA GPUs (GTX480, Tesla C2050, GTX-Titan, and/or K20) and report our results.

2 GPU Execution Model and Performance Factors

In this section, we briefly describe the execution model and some architectural features of modern GPUs that either contribute to or limit the performance of GPGPU applications. We believe that any performance modeling approach for the latter should capture some or all of those features; moreover, when analyzing a set of selected GPGPU performance models in Section 3, we rely on those features to assess the adequacy of the model to current GPUs. Without loosing generality, we will use NVIDIA's Compute Unified Device Architecture (CUDA) terminology for concepts pertaining to GPUs and their programming.

2.1 Execution Model

The processing power of GPUs resides in their array of Streaming Multiprocessors (SMs). When a GPGPU program invokes a kernel grid, the multiple thread blocks (TBs) of the grid are distributed to the SMs that have the necessary computing resources. Threads in a TB execute concurrently, and multiple TBs will execute concurrently. As TBs terminate, new blocks are launched on vacated SMs. Instructions are pipelined to harness instruction-level parallelism (ILP) inside a thread, and thread-level parallelism (TLP) is achieved in hardware.

SMs execute threads in groups of 32 (NVIDIA GPUs) parallel threads called *warps*. Each warp is scheduled for execution by a warp scheduler which issues the same instruction for all the warp's threads (Single Instruction Multiple Threads, SIMT, execution model). When threads diverge due to control flow, all divergent paths are executed serially with inactive threads in a path disabled.

2.2 Performance Factors and Optimization Space

At a very coarse level, performance on current GPUs is achieved by following a few basic optimization principles: *1*) maximize parallel execution, *2*) optimize memory usage in order to achieve maximum memory bandwidth and *3*) optimize instruction usage in order to achieve maximum instruction throughput. We detail each of these principles in the remainder of the section.

GPU kernels must expose *sufficient parallelism* to "fill" the platform. Specifically, for arithmetic operations, there should be sufficient independent instructions to accommodate multi-issue and hide latency. For memory operations, enough requests must be in flight in order to saturate the bandwidth. This is achieved by either having more independent work within a thread (ILP or independent memory accesses) or more concurrent threads or, equivalently, more concurrent warps (TLP). While ILP may be inherent to the code or deduced by the compiler, TLP and load-balance are decided by the execution configuration and the actual available resources. The ratio between the active warps on an SM and the maximum number of warps per SM is called *occupancy* in CUDA jargon, and it is a measure of utilization. Occupancy is limited by the number of registers and the amount of shared memory used by each thread block, and the thread count per block. Low occupancy always corresponds to performance degradation. The required occupancy depends on the kernel: a kernel exhibiting good ILP may need less occupancy, while a memory-bound kernel needs more to hide the high latency affecting memory accesses.

Global memory, the largest memory space with the greatest latency, is accessed at warp granularity via memory transactions of certain sizes (32, 64 or 128 bytes on current GPUs). When a warp executes a memory instruction, it attempts to *coalesce* the memory accesses of its composing threads into as few as possible such transactions, depending on the memory access patterns. The more transactions are necessary, the more unused data are transferred, limiting the achieved memory throughput. The number of necessary transactions varies among device generations (i.e., the compute capability). Access patterns along with the number of concurrent memory accesses in flight heavily impact the memory subsystem performance, leading to significant degradation when coalescing conditions are not met. Following Little's law, there should be $Latency \times Bandwidth$ bytes in transit in order to totally hide the memory latency. It is therefore crucial that a model is capable of evaluating the memory-level parallelism (MLP) available in a kernel.

Shared memory has higher ($\sim 10\times$) bandwidth and lower latency ($20 - 30\times$) than global memory. It is divided into equally-sized memory modules, called *banks*, which can be accessed simultaneously to increase performance. For example, N memory requests from N distinct banks can be serviced simultaneously; however, memory requests for two or more different words in the same bank lead to *bank conflicts* serialization, decreasing the achieved throughput significantly.

Additionally, most recent GPUs ship with L1 and L2 caches, which, if not game changers, at least disrupt the way shared memory and global memory are accessed. As L1 and the shared memory use the same region of on-chip memory

and their respective sizes are program configurable, their respective sizing has an impact on performance with regard to, for instance, register spilling and occupancy. Global memory operations now only reach the device DRAM on cache misses.

To summarize, we expect GPU performance models and their adjacent tools to capture and/or analyze parallelism and concurrency, occupancy, memory access patterns and their impact on global and local memory, as well as caching.

3 An Empirical Evaluation of GPU Performance Models

Our empirical evaluation starts from the assumption that performance modeling is based on a smart combination of application and architecture models. A performance modeling tool must be (1) *accurate*, (2) *fast* or *tunable* for accuracy versus speed and (3) *easy to use*, to favor adoption from the user community. We further favor performance modeling tools based on *reusable application and hardware models*. Application models should consider performance factors such as available parallelism and memory accesses and be hardware-agnostic. Likewise, hardware models must capture salient performance features such as the number of SMs, the SIMD width, amount of shared memory or of cache levels. Finally, an ideal modeling tool should not be limited to only predicting runtime. Instead, it should also highlight *performance bottlenecks* both from the application and the hardware, and eventually provide hints to fix them.

Guided by these requirements, the remainder of this section describes several GPU performance models, discussing their approaches, and results in analyzing a benchmark kernel. Given its popularity among such performance tools, we selected a tiled dense matrix multiplication (MM) kernel from the CUDA SDK as benchmark. To compute the product C of two square matrices A, and B, the kernel uses $t \times t$ block matrices (or tiles), with t divisible by both A and B size. A grid of $width_C/t \times height_C/t$ TBs is launched where each TB computes the elements of a different tile of C from a tile of A and a tile of B. The kernel is optimized by loading both tiles of A and B into shared memory to avoid redundant transfers from global memory.

3.1 PMAC Framework [9]

A relatively old framework for distributed systems analysis, PMAC is extended with diverse tools [10,11,12] for handling heterogeneity and evaluating the performance benefits of porting a kernel to a given accelerator. In a typical scenario, the PMAC Idiom Recognizer (PIR) can be used to automatically collect a set of well-known compute and memory access patterns (so-called idioms) present in a given source code [12]. After PIR has discovered idioms, their performance on a given accelerator is evaluated using micro-benchmarks. To this aim, the data footprint for each idiom needs to be captured using the PEBIL binary instrumentation tool [11]. Gather/Scatter and stream idioms have been tested on 2 types of accelerators, GPUs and an FPGA, showing 82% - 99% accuracy.

Evaluation. MM is only composed of the stream idiom, so we do not have to use PIR for this simple case. Predicting the performance of stream on an accelerator boils down to evaluating the time taken by memory operations as the framework assumes these are dominant in the execution time. Equation (1) estimates this time over all basic blocks pertaining to the idiom, with $MemRef_{i,j}$ being the number of references of basic block i of type j, $RefSize$ the reference size in bytes and $MemBW_{\text{stream}}$ the bandwidth of the stream idiom on the target accelerator.

$$MemTime = \sum_{i}^{allBB} \frac{\#MemRef_{i,j} \times RefSize}{MemBW_{\text{stream}}} \quad . \tag{1}$$

We run PEBIL on the MM binary to get all the basic blocks along with the memory references and their sizes. Using the CUDA version of MM, several runs with different matrix sizes must be performed to collect run-times. A model for $MemBW_{\text{stream}}$ is built from these data via regression, as shown in Fig. 01. The model itself is given by the following equation, with s being the data size:

$$MemBW_{\text{stream}}(s) = -0.0020\times\max(0, 3072-s)+0.0003\times\max(0, s-3072)+7.0709 \tag{2}$$

According to the framework, the value from (1) and the actual GPU time from running MM on the GPU should be similar. However, we found the outputs can differ significantly, as shown in Fig. 02. Additionally, some tools are not readily available (PIR), others are difficult to use and with no proper documentation (PEBIL). Finally, the framework provides only limited support for heterogeneity (idioms) and, in general, says nothing about bottlenecks and how to overcome them.

Summary. PMAC seems too complex and tedious to use, and only characterizes an application by idioms. The observed accuracy is low. Finally, (1) only works for memory-bound kernels.

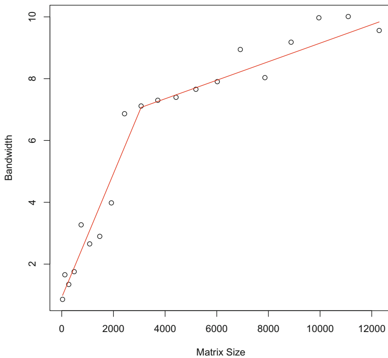

Fig. 1. Piecewise linear regression of the memory throughput versus matrix size data of MM kernel on NVIDIA Tesla C2050

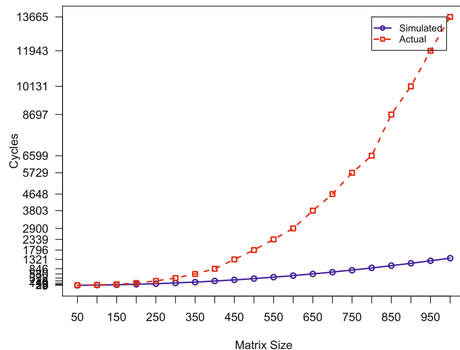

Fig. 2. Elapsed clock cycles for different matrix sizes for MM on NVIDIA Tesla C2050, as predicted by (1) and calibrated from actual run times

3.2 Eiger Framework [13]

Eiger is an automated statistical methodology for modeling program behavior on different architectures. To discover and synthesize performance models, the methodology goes trough 4 steps: *1*) experimental data acquisition and database construction, *2*) a series of data analysis passes over the database, *3*) model selection and, finally, *4*) model construction. For the training phase (step 1), the application is executed on the target processor, facilitating the measurement of execution time and the collection of multiple parameter values (characterizing both processors and applications, 47 in total). In step 2, principal component analysis (PCA) is performed on the collected data. The generated principal components are post-processed so they have either strong or weak relationships with the initial metrics. Finally, an analytical performance model is constructed using parametric regression analysis. The approach is validated on 12 applications from different GPU benchmark suites and the CUDA SDK.

Evaluation. Eiger is a promising approach with support for GPUs and desirable features such as independent application and hardware characterization, allowing one to generically include major performance features from both the application and the hardware into the modeling. The first column in Table 11 shows some of the metrics used in a GPGPU use case. Besides static code analysis and instrumentation, the experimental data collection stage leverages Ocelot [14], a PTX simulator, for features necessitating code emulation. A software package[1] for interacting with Ocelot is provided to ease the modeling process, but this requires significant improvements on documentation and usability, especially on using Ocelot to extract relevant metric values. Therefore, we ended up using the NVIDIA profiler to build the database based on the correspondences shown in Table 11. We have trained the model on a Tesla C2050 and predicted the runtime on a Geforce GTX580. Even though both GPUs correspond to Fermi architecture, the predictions accuracy shown in Fig. 13 is not ideal. Observe that the more accurate the model, the closer the data points to the $y = x$ line.

Summary. Eiger is a promising approach which already has the desirable features for a usable system. However, its complexity and the poor documentation of the provided software make it hard to use. To use the framework efficiently, one must have good knowledge of many statistical methods, appropriate tools for those, and C++ programming.

3.3 STARGAZER Framework [15]

The STAtistical Regression-based GPU Architecture analyZER is an automated GPU performance exploration framework based on stepwise regression modeling. STARGAZER sparsely and randomly samples the parameter values of the full GPU design space. Simulation or measurement is then performed for each sample. Finally, stepwise linear regression is performed on the simulations or measurements to find the most influential (architectural) parameters to the performance of the application. The linear regression basis is enhanced with spline

[1] `https://bitbucket.org/eanger/eiger`

Table 1. Metric correspondence. For performance counter definition, see `http://docs.nvidia.com/cuda/profiler-users-`

Eiger metric	Counter
memory efficiency	(gld_efficiency+gst_efficiency)/
memory intensity	ldst_executed / inst_executed
memory sharing	code analysis
activity factor	CUDA occupancy
SIMD/MIMD	execution configuration
DMA size	code analysis

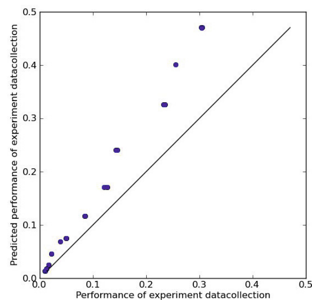

Fig. 3. Prediction accuracy

functions to allow nonlinearity between independent variables of the regression as interactions between architectural parameters usually affect performance in a nonlinear manner. Analysis of the relationships between the application runtime and the most influential architectural parameters allows the discovery of performance bottlenecks. Several CUDA applications from benchmark suites are used to test the system with good average accuracy (around 99%).

Evaluation. STARGAZER aims at GPU design space pruning and, as such, only considers hardware characteristics. It captures parallel execution behavior through metrics like block concurrency and SIMD width. There are also metrics to capture intra- and inter-warp memory coalescing. However, there are no metrics for estimating shared memory bank conflicts or control flow divergence. As a statistical framework, STARGAZER needs to collect data for all considered hardware characteristics. Python and R scripts are provided to perform this task by using the GPGPU-Sim [5] simulator. Depending on the space to explore and the dataset (e.g., the matrix size for MM), it can take days to generate training data. One alternative is to use actual measurements, but extracting all design space parameter values may not be straightforward or even possible. We ran our MM kernel in GPGPU-Sim with Tesla C2050 GPU settings to collect data and build a model which predicts the run time as measured by GPGPU-Sim reasonably well. However, run times obtained by GPGPU-Sim are orders of magnitude higher than those of the actual hardware for the same matrix size.

Summary. STARGAZER is usable for design space exploration, but the proposed metrics do not cover all the current GPU performance features, leading in turn to low accuracy for some applications. We believe that adding more relevant hardware features and application characteristics, and especially providing a faster way of acquiring experimental data, would make STARGAZER one of the few tools actually usable in practice.

3.4 WFG Modeling Tool [16]

A GPU analytical performance modeling tool is defined by abstracting a GPU kernel as a Work Flow Graph (WFG), based on which the kernel execution time

is estimated. The kernel is represented by its dependence graph (both control flow and data dependence graph). The nodes of the WFG are instructions which are either computation or memory (global or shared) operations, the edges are either transition arcs from the control flow or data dependence arcs. Transition arcs are labeled by the average number of cycles required to execute the source node, while data dependence arcs are labeled with the portion of GPU load latency that is not covered by interleaving execution of different warps. The tool aims to accurately identify performance bottlenecks in the kernel. Symbolic evaluation of certain fragments of code is used to determine loop bounds, data access patterns and other program characteristics. The effects of SIMD pipeline latency, memory bank conflicts, and uncoalesced memory accesses are highlighted in the WFG. The tool is validated on 4 kernels and shows reasonable accuracy.

Evaluation. The model considers major performance factors such as TLP and MLP, memory access patterns and different latencies. However, as an analytical performance modeling tool, WFG lacks runtime information such as achieved occupancy or memory and instruction efficiency, which limits its potential prediction accuracy. The crux of the model is the program dependence graph representation of the kernel, generated by a compiler front-end - which is not publicly available, making it impossible for us to directly evaluate this model for our MM application. However, we managed to build an approximate WFG in a different way. Equation 4 in [16], $Latency_{\mathrm{BW}} = \max(0, \frac{CYC_{\mathrm{mem}} - CYC_{\mathrm{compute}}}{NUM_{\mathrm{mem}}}) + \frac{SIMD_{\mathrm{work}}}{SIMD_{\mathrm{engine}}}$, estimates memory stalls due to lack of available bandwidth per warp. CYC_{compute}, CYC_{mem}, and NUM_{mem} represent the average number of compute cycles, global memory cycles, and operations per warp, respectively. $SIMD_{\mathrm{work}}$ is the warp size and $SIMD_{\mathrm{engine}}$ the number of SPs in a SM. For a memory-bound kernel like MM, this will significantly contribute to the total runtime. Using code analysis and the profiler, we can determine the values of the relevant metrics to evaluate the equation. Assuming correspondences between metrics in Equation 4 and combinations of hardware performance counters measured by the CUDA profiler, as shown in Table 13, we plot the values of LatencyBW from both the profiler and the model in Fig. 14. We observe that global memory latency as seen by the application (thus Equation 4) and by the hardware do not always agree.

Summary. WFG captures most of the major performance factors on GPUs, except caching. However, it is complex and aimed at consumption by optimizing compilers. We have struggled to build WFG for the MM kernel, and found it inaccurate for the cases, where matrices are small, and, probably fit in caches.

3.5 The MWP-CWP Analytical Model [17]

In this model, the characteristics of a CUDA application are captured in two different metrics: MWP (memory warp parallelism), the amount of memory requests that can be serviced in parallel, and CWP (computation warp parallelism) or how much computation can be done while a warp is waiting for the memory. Using only these two metrics, the application performance can be assessed. The

Table 3. The correspondence between WFG metrics and combinations of performance counters and hardware specification. WordSize is determined from the application; Cycles per instruction (CPI) and BWPerSM are determined from the hardware specifications; all the other metrics are extracted from the CUDA profiler.

WFG metric	Counter
$Latency_{\mathrm{BW}}$	1-sm_efficiency / warps x stall_data_request x elapsed_cycles_sm
CYC_{compute}	inst_per_warp x CPI
NUM_{mem}	gld_request + gst_request
CYC_{mem}	NUM_{mem} / warps x WordSize x BWPerSM

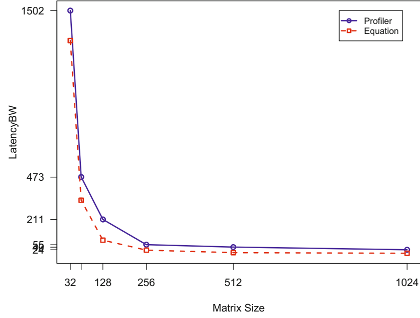

Fig. 4. Comparison of LatencyBW from the profiler (measurement) and from the model (predicted) for different matrix sizes

metrics computations require 17 hardware and application parameters obtained from either hardware specification or micro-benchmarking for the former and extracted from either the source or the PTX code for the latter. Furthermore, the model is static, so no run of the application is necessary. Instead, parameters of the application are estimated and fed into the model, which calculates the expected execution time. The model is validated using two NVIDIA GPUs and shows good results.

Evaluation. We have attempted to predict the performance of the MM kernel (also used in the paper) on a newer GPU, a GTX480. While we were able to preserve 12 parameters from the original paper [17], the rest of 5 parameters required micro-benchmarking. However, the authors have decided to deprecate the benchmarking suite as it *became obsolete for the new generation of GPUs*[2]. Our attempts of approximating the results of these benchmarks using platform similarities have failed: the predicted performance was far off from the observed performance on the GTX480. We conclude from this that the recalibration of this model for new generations of GPUs is very time consuming, if at all possible. We add to this the perceived difficulty of modeling new applications, which require very detailed inspection of the code (be it CUDA or PTX) to determine, at instruction level, the characteristics of operations and memory accesses, and that is cumbersome even for relatively experienced CUDA programmers. Finally, we argue that the model does capture the inner workings of a GPU, but requires too much effort for the provided result: it only predicts the execution time, without really pointing out what the problems of the implementation are. This questions its usability for real applications: modeling requires a PTX version of the running code and a calibration of the model for the specific GPU. As

[2] This statement was made in a direct discussion with one of the authors.

both these elements must be available, why isn't a simple run on the GPU not sufficient?

Summary. Because of its complexity in both modeling and calibration, we believe this model is not directly usable for application design and tuning. Further, we were unable to assess the usability of this model, and we could not obtain any relevant predictions by using it on a new GPU platform.

3.6 GPU *a la* (QRQW) PRAM [18]

The authors of this contribution focus on a high level model of the GPU from the perspective of its computation model. Specifically, they base their analytical modeling on a mix of the BSP [19], PRAM [20], and QRQW PRAM [21] models and get a good approximation of the execution times. Compared with the MWP-CWP model (see Section 3.5), this is a much more lightweight modeling approach, in which only 7 platform parameters are used (5 from the specifications and 2 from the occupancy calculator). The other 6 parameters are approximating the application behavior: the geometry of the kernel execution, the computational load of a thread, and the data size. The model computes a predicted execution time by "mapping" the dataset on the threads and approximating a cumulated number of cycles of the execution. In the case of this model, the difficulty lies in assessing the cycles per thread, *i.e.* characterizing the application. This is an approximation step that can be done by either calibration (e.g., micro-benchmarking) or source code analysis, but should be portable between different platforms. Thus, a model of the application can be reused for performance prediction on multiple platforms.

Evaluation. To assess the usability of this model, we have attempted to predict the runtime of the matrix multiplication (as implemented in the original paper [18]) on the GTX480 and GTX-Titan, with mixed results. For the GTX480 (the Fermi architecture), we were able to predict the execution time for multiple execution scenarios with accuracies ranging between 3% and 10% (in the order of milliseconds). However, for the GTX-Titan (the Kepler architecture) we were unable to get any good predictions: the measurements were consistently pessimistic, with errors ranging from 30% up to 70%. This indicates that the model might not capture some of the new features of the Kepler architecture.

Summary. We believe this model is promising for high-level approximations of the execution times, and it is probably very useful, once calibrated, to assess different geometries and/or application parameters (e.g., the tile sizes in the matrix multiplication). However, the calibration of the model for a given application remains challenging, and we are skeptical if the model can analyze applications where the mapping of data-items to threads is non-trivial.

3.7 Quantitative Analysis [22]

The authors of this model have an ingenious solution for analyzing the performance of a GPU application: they first measure "everything" about the target GPU, express an application model in terms of consumption of these resources,

and finally detect the application bottlenecks by checking which of these resources is dominating the application runtime. Once the bottleneck is found, the execution time is estimated using the tabulated benchmarking results which are the achievable numbers for (1) instruction throughput (per class of instructions), (2) shared memory bandwidth, and (3) global memory bandwidth in conditions of variable occupancy of the platform (i.e., when varying the number of blocks and threads). The application model is a detailed breakdown of the instructions in the code - computation and memory accesses alike. The observed accuracy of this model is within 15%.

Evaluation. In order to evaluate this model for a new GPU, the micro-benchmarking needs to be reapplied. As the suite is not available, we were unable to model another GPU, hence the lack of prediction for the GTX480 and GTX-Titan. Moreover, the analysis of the shared and global memory, as presented in the original paper, was performed on non-cache architectures. It is unclear whether the code instrumentation can detect caching operations and treat them separately. If that is not possible, architectures such as Fermi or Kepler will get very inaccurate predictions for memory-intensive applications.

Summary. The model is interesting because it provides more insight into the causes of the performance behavior of applications. However, the micro-benchmarking suite is necessary for calibration, and we are not convinced that the approach will work when caches play a significant role. We were unable to perform any predictions on the GTX480 or GTX-Titan because we did not have access to the benchmarking suite.

4 Conclusion

In this work, we presented an overview of existing performance analysis and prediction tools dedicated to GPGPU computing. We further selected, based on citations and usage according to our own literature survey, a set of seven popular and/or interesting tools discussed in literature. We evaluated these seven GPU performance modeling tools with slightly disappointing results: only two approaches were able to predict, with a reasonable accuracy, the runtime of a well-known kernel on newer hardware. Three of the seven models were able to provide hints into the performance bottlenecks which could help programmers and designers improve their current solutions.

We believe our study is a good starting point for a much needed, thorough investigation of the stat-of-the-art in performance modeling for GPUs. Our future work aims at transforming this study into a comprehensive survey of the performance modeling for heterogeneous architectures with highlight on what is working and what needs to be improved.

References

1. Saule, E., Kaya, K., Çatalyürek, Ü.V.: Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. CoRR abs/1302.1078 (2013)
2. NVIDIA Corporation: Press release: Nvidia tesla gpu computing processor ushers in the era of personal supercomputing (June 2007)

3. Advanced Micro Devices (AMD) Inc. Press release: Amd delivers enthusiast performance leadership with the introduction of the ati radeon 3870 x2 (January 2008)
4. Asanovic, K., et al.: A view of the parallel computing landscape. Commun. ACM 52(10), 56–67 (2009)
5. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing cuda workloads using a detailed gpu simulator. In: ISPASS, pp. 163–174. IEEE (2009)
6. Mudalige, G.R., Vernon, M.K., Jarvis, S.A.: A plug-and-play model for evaluating wavefront computations on parallel architectures. In: IPDPS, pp. 1–14. IEEE (2008)
7. Diamos, G.F., Yalamanchili, S.: Harmony: An execution model and runtime for heterogeneous many core systems. In: Proceedings of HPDC 2008, pp. 197–200. ACM, New York (2008)
8. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: A programming model for heterogeneous multi-core systems. SIGPLAN Not. 43(3) (March 2008)
9. Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., Purkayastha, A.: A framework for performance modeling and prediction. In: Proceedings of SC 2002, pp. 1–17. IEEE Computer Society Press, Los Alamitos (2002)
10. Tikir, M.M., Laurenzano, M.A., Carrington, L., Snavely, A.: PSINS: An open source event tracer and execution simulator for MPI applications. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 135–148. Springer, Heidelberg (2009)
11. Laurenzano, M., Tikir, M., Carrington, L., Snavely, A.: Pebil: Efficient static binary instrumentation for linux. In: ISPASS 2010, pp. 175–183 (March 2010)
12. Carrington, L., Tikir, M.M., Olschanowsky, C., Laurenzano, M., Peraza, J., Snavely, A., Poole, S.: An idiom-finding tool for increasing productivity of accelerators. In: Proceedings of ICS 2011, pp. 202–212. ACM, New York (2011)
13. Kerr, A., Anger, E., Hendry, G., Yalamanchili, S.: Eiger: A framework for the automated synthesis of statistical performance models. In: Proceedings of WPEA 2012 (2012)
14. Kerr, A., Diamos, G., Yalamanchili, S.: A characterization and analysis of ptx kernels. In: Proceedings of IISWC 2009, Washington, DC, USA, pp. 3–12 (2009)
15. Jia, W., Shaw, K., Martonosi, M.: Stargazer: Automated regression-based gpu design space exploration. In: ISPASS 2012, pp. 2–13 (April 2012)
16. Baghsorkhi, S.S., Delahaye, M., Patel, S.J., Gropp, W.D., Hwu, W.M.W.: An adaptive performance modeling tool for gpu architectures. SIGPLAN Not. 45(5), 105–114 (2010)
17. Hong, S., Kim, H.: An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. SIGARCH Comput. Archit. News 37(3), 152–163 (2009)
18. Kothapalli, K., Mukherjee, R., Rehman, M., Patidar, S., Narayanan, P.J., Srinathan, K.: A performance prediction model for the cuda gpgpu platform. In: HiPC 2009, pp. 463–472 (December 2009)
19. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM 33(8), 103–111 (1990)
20. Fortune, S., Wyllie, J.: Parallelism in random access machines. In: Proceedings of STOC 1978, pp. 114–118. ACM, New York (1978)
21. Gibbons, P.B., Matias, Y., Ramachandran, V.: The queue-read queue-write asynchronous pram model. In: Euro-Par 1996. LNCS, vol. 1124, pp. 279–292. Springer, Heidelberg (1996)
22. Zhang, Y., Owens, J.: A quantitative performance analysis model for gpu architectures. In: HPCA 2011, pp. 382–393 (February 2011)