Parallel Computing 56 (2016) 18–33

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

The landscape of GPGPU performance modeling tools

Souley Madougou[a,*], Ana Varbanescu[a], Cees de Laat[a], Rob van Nieuwpoort[b]

[a] University of Amsterdam, Science Park 904, Amsterdam 1098XH, The Netherlands
[b] Netherlands eScience Center, Science Park 140, Amsterdam 1098XG, The Netherlands

ARTICLE INFO

Article history:
Received 23 March 2015
Revised 4 February 2016
Accepted 3 April 2016
Available online 6 April 2016

Keywords:
GPGPU
Performance analysis
Performance modeling
State-of-the-art report
Empirical evaluation

ABSTRACT

GPUs are gaining fast adoption as high-performance computing architectures, mainly because of their impressive 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 typical kernel examples with various computation and memory access patterns 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.

1. Introduction

Today's computing landscape is dominated by parallel, heterogeneous systems. This is the resultant of two main phenomena: the *power wall* and the success of graphics cards. The first phenomenon forced CPU manufacturers to consider multicore designs while the second led to the adoption of the GPU as a general-purpose computing resource which can be highly efficient for certain types of applications. Indeed, speedups of up to three orders of magnitude have been reported for GPGPU applications over their CPU counterparts. Even if those numbers are often inflated by poor CPU code [18], the performance gain of using GPUs remains significant, and their popularity increases for more and more applications. Behind this success is the massively parallel architecture of the GPU, which enables tremendous peak performance (*e.g.*, over 1 TFLOP for a regular, consumer GPU card [1,26]).

Attaining the best possible performance on such platforms for real-world applications is very challenging because it typically requires:

(1) the algorithms to be amenable to the data parallel paradigm,
(2) the porting of the applications (kernels) to GPU programming models and especially,
(3) a lot of manual tuning for tailored algorithm implementations to specific hardware, and a large design and optimization space (manual) exploration.

* Corresponding author. Tel.: +31646756870.
E-mail addresses: S.Madougou@uva.nl, smadougou@gmail.com (S. Madougou), A.L.Varbanescu@uva.nl (A. Varbanescu), delaat@uva.nl (C. de Laat), R.vanNieuwpoort@esciencecenter.nl (R. van Nieuwpoort).

http://dx.doi.org/10.1016/j.parco.2016.04.002
0167-8191/© 2016 Elsevier B.V. All rights reserved.

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 theoretical peak performance of the given hardware system [3,15]. Furthermore, the whole process needs to be repeated when moving to a new platform, if severe performance degradation is to be avoided.

We believe this trial-and-error practice for performance searching needs to change. Instead of using word-of-mouth and primitive, in-house profiling techniques and empirical program tuning, we advocate a systematic approach through performance modeling tools. Although GPU hardware vendors propose profilers and various performance analysis tools [5,27], the latter are less flexible, often vendor-specific and only cover a limited domain of use-cases compared with (fully-fledged) performance models. So are the approaches based on application case-studies [25] and heterogeneous programming systems [7,21]. Models are used in a myriad of domains (*e.g.*, mathematics, physics, finance) to capture the salient features of phenomena with clarity and the right degree of accuracy to facilitate analysis and prediction. Just as performance modeling has been found useful in the design of efficient algorithms on traditional parallel systems [30], the "affordability" of high performance promised by the ubiquity of heterogeneous architectures can only be enjoyed with the support of the appropriate performance modeling tools and approaches. We also believe they may provide the first step towards performance portability.

Despite the lack of a unique standard model for heterogeneous computation similar to the Random Access Memory (RAM) model for sequential computation, interesting and promising work is being done to model GPGPU computation for performance analysis and prediction [22]. This work, mostly coming from academia, adopts several methodologies and follows a few goals. Approaches include statistical regression and machine learning based [11,12,28,37,39], compiler-based methods[2], analytical [10,14,32,33], visual [19,31] and quantitative models [38]. Their goals vary from simple execution time prediction [12,14,32] to performance bottlenecks identification [10,33,39] and insights for fixing them [31,38], to software and architecture improvement advice [38], to optimization space exploration [2,11,29], to power-performance efficiency [33,39]. Finally, we are aware of two popular simulators for GPUs [3,6]. However, we purposely discard them from this study as they are primarily designed not for predicting GPU performance, but for understanding the inner workings of the processors (because of the non-disclosure of the instruction set architectures by the vendors).

To help application programmers and hardware architects alike in their quest for performance understanding, we provide a clear picture of the performance modeling tools available today. The work presented here builds upon and extends our previous work [23]. It makes the following new contributions:

- First, we sketch the performance modeling landscape for heterogeneous systems and deduce taxonomies useful for model organization and as common vocabularies in the performance analysis community.
- Second, we provide a thorough description of 12 different approaches to performance modeling specific to GPUs. For each one of them, we highlight the goals and methodology, the design and/or optimization space it explores, its validation conditions and its specific features.
- Third, we empirically evaluate, the best we can, the performance of the models using three different and well-known kernels on at most four platforms from two generations of NVIDIA GPUs (GTX480, Tesla C2050, GTX-Titan, and/or K20) and an AMD GPU (Radeon HD 7970), for which we report and discuss the results.

2. Background

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 will further use the terminology defined here for the in-depth analysis of our set of GPGPU performance models from Sections 5 to 7 and we rely on those features to assess the adequacy of the model to current GPUs. Without losing generality, we will use NVIDIA's Compute Unified Device Architecture (CUDA) terminology for concepts pertaining to GPUs and their programming.

2.1. GPGPU execution model

The processing power of GPUs resides in their array of Streaming Multiprocessors (SMs). When a GPGPU program invokes a kernel grid, the thread blocks (TBs) of the grid are distributed to the SMs. 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*. Warps are 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 enough 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 often 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 highest 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 overview of the landscape

In this section, we describe the methodology adopted for the presentation of the studied performance models. Each study consists of two parts, a *description* and an *evaluation*. We first detail the features used for the description and then the criteria and benchmarks retained for the evaluation.

3.1. Features

For each studied model, we clearly state what the goal of the model is from the authors’ perspective along with the approach taken. We then briefly describe the steps followed to build the model mentioning on the way the structure of the modeling tool if any. We also explicit the inputs to the tool and its output(s) and if the authors provide any software to help, *e.g.*, in collecting the inputs. We will report whether the model separately characterizes the workload and the hardware or not. The (micro-)benchmarks used for validation of the model, the type of kernels supported (*e.g.*, CUDA, OpenCL) along with the hardware used are also reported.

3.2. Evaluation criteria

Our empirical evaluation starts from the assumption that performance modeling is based on a smart combination of workload and architecture models. A performance modeling tool must be (1) *accurate*, (2) *fast* to evaluate or *tunable* for accuracy *versus* speed and (3) *simple* to build. Another desirable feature is the *reusability* of the models; when this is not possible, only little effort should be needed to adapt models to new workloads and hardware. Workload 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 execution time. Instead, it should also highlight *performance bottlenecks* both from the application and the hardware, and eventually provide hints to fix them.

For our empirical evaluation, we apply each model to three kernels (MM, HS, and BFS – see Section 3.3) running on popular GPU architectures at the time of writing. We then report the accuracy as computed from the experiments. As we build

Table 1
Execution times in milliseconds for typical data set sizes of used benchmarks on NVIDIA Tesla C2050.

Benchmark	Data sets	Time
MM	512×512	11.093
	768×768	21.966
	1024×1024	43.336
BFS	graph4K	0.080
	graph64K	0.554
	graph1M	11.790
HS	64×64	0.092
	512×512	0.310
	1024×1024	1.235

the model ourselves, we can appreciate the effort and the level of hardware knowledge required for model construction. We can further evaluate whether the model captures the performance factors of recent GPU architectures. Once the model is constructed, we can also appreciate its complexity and its level of workload or hardware abstraction. Finally, by evaluating the model, we estimate how fast it is to evaluate and what insights we get from it.

3.3. Evaluation benchmarks

Matrix Multiply (MM) is a popular kernel, used for many performance modeling tools as an example. We selected the tiled dense matrix multiplication (MM) version from the CUDA SDK as our first benchmark. To compute the product C of two $n \times n$ matrices A and B, the kernel uses $b \times b$ block matrices (or tiles), with b a divisor of n. A grid of $\frac{n}{b} \times \frac{n}{b}$ 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. MM, which is characterized by regular access patterns, performs $O(n^3)$ computations and $O(n^2)$ data accesses. The kernel is optimized by loading both tiles of A and B into shared memory to avoid redundant transfers from global memory.

HotSpot (HS) is a structured grid thermal simulation tool used for estimating processor temperature. The kernel iteratively solves a series of differential equations for block temperatures. Each element is computed by gathering a square neighborhood of elements (2D stencil pattern) from the input grid. Tiling is used to partition the stencil loops for parallel processing. HS uses a ghost zone of redundant data around each tile to reduce the frequency of expensive data exchanges with neighboring tiles.

Breadth-First Search (BFS) is a graph traversal algorithm. Graph traversal algorithms are notoriously hard to parallelize because of their irregular, data-driven access patterns. The implemented kernel, which traverses all the connected components of a graph, does not use any shared memory (too difficult to determine a caching policy due to limited data locality). BFS is an irregular application, limited in performance by the off-chip memory bandwidth.

Table 1 lists the execution times of the typical data sets of the three benchmarks, just to illustrate their magnitude. Main inputs to MM and HS are matrices and grids of floating point numbers, respectively, which sizes are given as $n \times n$. BFS's inputs are synthetic graphs of 4K-, 64K-, and 1M-vertices. The execution times of MM are as reported in the CUDA SDK, whereas those from BFS and HS are those reported by the profiler as kernels cumulated execution times.

3.4. A high-level taxonomy

We propose a first classification of the approaches used for performance analysis and prediction of GPGPU applications into four categories. We briefly describe each category and highlight the differences among the categories. The categories are: analytical, statistical regression or machine learning (ML) based, quantitative, and compiler-based methods. In the remainder of the paper, we will discuss representative members of each class in detail.

Analytical methods. These meth ods provide an abstract view of the workload and/or the hardware, and use equations to estimate the execution time (or number of cycles) of a given kernel. Characteristics of the computation, the memory accesses and the hardware are captured by different parameters and further used in these equations. The models in this category are tedious to build, as they often require intimate knowledge of the workload and the hardware being modeled. Once the models are built, they are fast to evaluate, often without requiring application execution. Examples are the PMAC framework [32], MWP–CWP and GPUPerf [10,31], GPU à la QRQW PRAM [14] and Components [33]. Following the success of the Roofline model [36], some models such as GPUPerf and Transit [19] also provide visual interpretation mechanisms.

Statistical and machine learning methods. As the name suggests, these methods use statics or machine learning (ML) techniques to sample the design or optimization space (set of parameters characterizing either the workload or the hardware or both) and extract, often using regression, the most influential parameters to the behavior of a system (*e.g.*, performance)

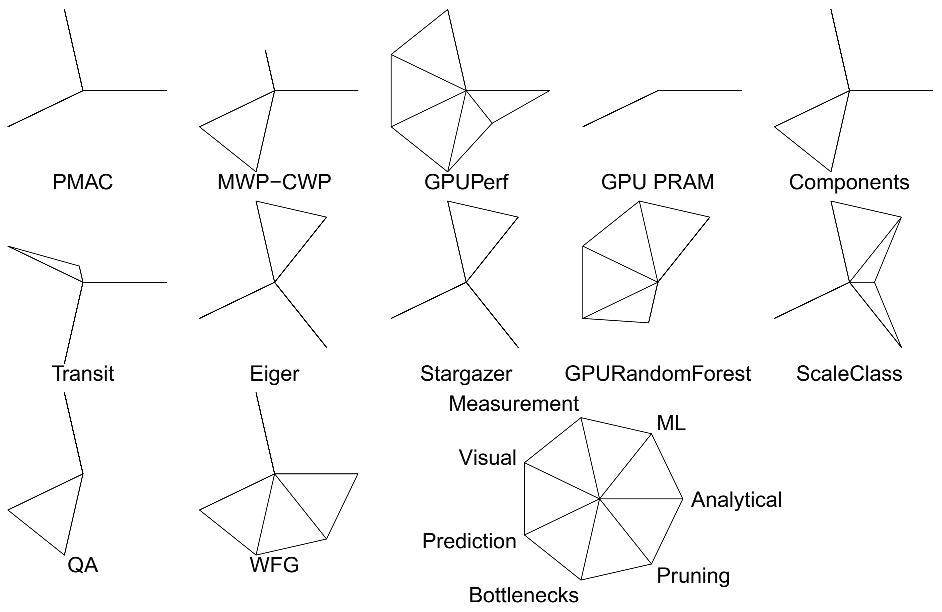


Fig. 1. Methodology and goal profiles of the studied performance models.

from which a (set of) model(s) of the behavior is built. All these methods require training, sometimes extensive and expensive, but they require less user intervention are often automated. Therefore, such methods are often used for design and optimization space exploration, and performance bottlenecks highlighting. Examples include the Eiger framework [12], Stargazer [11] and GPURandomForest [39]. Recent approaches, including [28] and ScaleClass [37], adopt more complex methodologies combining several statistical and ML techniques.

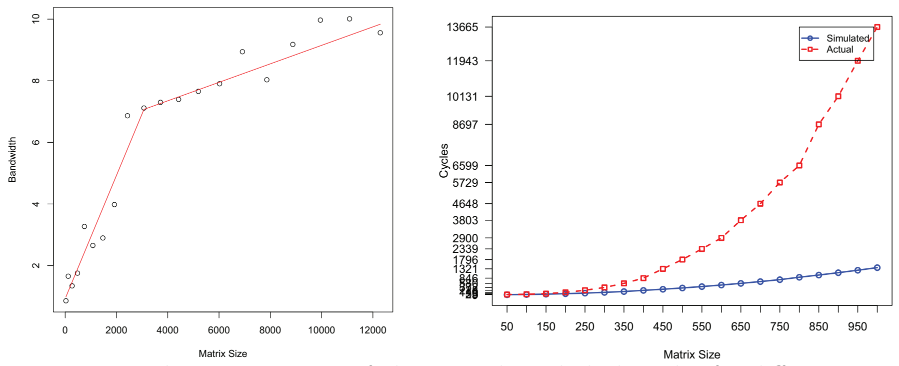
Quantitative methods. The methods in this class use measurements to derive a (set of) model(s) for the performance of a kernel on a given architecture. Custom microbenchmarks are built to exercise different components of the target GPU (*e.g.*, instruction pipeline or global memory) and measure their performance (*e.g.*, throughput). The workload model is a detailed breakdown of the instructions in the kernel code. An advantage of these methods is that they can identify performance bottlenecks and help in program optimization. However, as the microbenchmarks are specific to a GPU generation, they lack generality and need rewriting to adapt to newer GPU generations. A representative example of this category is Quantitative Analysis [38].

Compiler-based methods. These methods combine complex program (kernel) analysis with analytical modeling to provide a framework to both predict execution time and identify performance bottlenecks. One advantage of these methods is that they capture major performance factors thanks to the complex program analysis. Unfortunately, this complexity also constitutes a weakness as the analysis tool, often implemented as compiler frond-end, and thus unavailable or hard to use. Furthermore, they inherit advantages and drawbacks of the analytical modeling approach they adopt. A representative example is WFG [2].

Fig. 1 represents a summary of the GPGPU performance models we study further along seven dimensions which are Analytical, machine learning (ML), Measurement, Visual, Prediction, Bottlenecks and Pruning. The first three dimensions pertain to the methodology used by the model and the last three to their stated goals. The visual dimension is an additional feature for facilitating insight visualization. None of the subgroups is exclusive. For instance, we observe that most of the models include some form of measurement or microbenchmarking, yet some models are entirely based on that. In this figure, we did not retain the compiler-based approach as a dimension as only a tool uses it and happens to use also analytical modeling. Finally, the goals, which are abbreviated for the sake of space, deserve some explanation. The dimension Prediction specifies whether the model can be used to predict execution time. Similarly, Bottlenecks and Pruning dimensions state that the model is usable for performance bottlenecks identification (with or without insight on fixing them) and for optimization space pruning, respectively. One can already notice the outlines of our classification. For instance, pure analytical models (*e.g.*, MWP–CWP) or hybrid models (*e.g.*, WFG) with analytical components all include the horizontal line representing the Analytical dimension. Likewise, all ML approaches include the ML dimension and have similar profiles.

4. Analytical methods

In this section, we describe in detail five analytical methods.



(a) Piecewise linear regression of the memory throughput versus matrix size data of MM kernel on NVIDIA Tesla C2050.

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

Fig. 2. Regression and prediction of the PMAC framework.

4.1. PMAC framework [24]

Description: A relatively dated framework for distributed systems performance analysis, PMAC [32], is extended with diverse tools [4,16,34] 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 [4]. 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 [16]. PMAC needs to know the list of basic blocks pertaining to the idiom along with their memory references and a model for the memory bandwidth of the idiom on the accelerator of interest. Consequently, PMAC separately characterizes both the idiom and the hardware. The outcome is execution time prediction. Gather/Scatter and stream idioms in two HPC applications (Milc and HYCOM) have been tested on two types of accelerators, NVIDIA 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. Eq. (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.

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

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 execution times. A model for $MemBW_{\text{stream}}$ is built from these data *via* linear regression as done by the authors, and shown in Fig. 2(a). 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. 2(b). 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. We observed similar behavior for the HS kernel.

In summary, PMAC is complex and tedious to use, and only characterizes an application by idioms. For instance, we cannot use PMAC to evaluate the performance of BFS kernels as they do not contain any known idiom. Generally, the observed accuracy is low. Finally, (1) only works for memory-bound kernels.

4.2. The MWP–CWP model and GPUPerf [10,31]

Description: In the MWP–CWP model, the characteristics of a CUDA application are ultimately 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 metrics computation requires 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 (pre-Fermi generations) and shows good results.

To alleviate the limitations of the original MWP–CWP model, as well as to update the model for the Fermi generation of GPUs, the authors have proposed (in [31]) GPUPerf. Using similar, in-depth knowledge of the GPU and its parameters, the model not only predicts the performance of current application implementations, but attempts to help programmers decide, with higher accuracy than Roofline [36], the impact of different optimizations that could be applied. The model remains very difficult to calibrate, parameter-wise. Furthermore, the use of the optimization-guiding indicators (the B^* indicators) is not clear: a systematic interpretation guide for these numbers is missing, making the evaluation of their general usability impossible.

Evaluation: MWP–CWP: 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 [10], the other 5 parameters required micro-benchmarking, impossible to do because the authors have deprecated the micro-benchmarking suite which *became obsolete for the new generations of GPUs.*[1] We attempted to approximate the results of these benchmarks by using platform similarities, but failed: the predicted performance was far off from the observed performance on the GTX480. The recalibration of the model for newer generation required more effort than a simple update (especially given caching and different synchronization opportunities). Adding to these findings the newer GPUPerf method, we consider this model deprecated for Fermi and post-Fermi GPU generations.

Evaluation: GPUPerf: This tool is based on the same ideas as MWP–CWP: the authors exploit in-depth knowledge of the GPU hardware and scheduling, identify contention points, and estimate this contention as close as possible to reality. This approach requires no less than 28 parameters (dependent on the application and platform) to be precisely estimated. They are used in 25 equations to predict the execution time for the kernel and the potential impact of certain optimizations. The parameters are estimated using multiple tools (microbenchmarking, the (visual) profiler, an instruction analyzer, and a static analyzer). Precise estimations of the instruction mix of the application and the warp scheduling effects are required for an accurate prediction, any attempt to visually estimate these quantities has failed. Furthermore, installing, configuring, and using the recommended tools proved to be a whole new challenge. We were unable to reproduce the numbers presented in the paper, probably due to different configurations of the necessary tools. After three days of attempts we were forced to drop this evaluation due to time constraints.[2]

In summary, because of the complexity in both modeling and calibration, this model is not directly usable for application design and tuning. Further, it seems difficult to believe that non-expert users (*i.e.*, beginner and intermediate GPU programmers) will be able to determine the parameters needed for computing all the metrics. This process was very cumbersome for us, and it requires additional tuning for any new hardware platform. It is more likely to believe that such a model will be designed and tuned by a performance engineer. For GPUPerf, the authors recognize this difficulty and discuss a possible toolchain for simplifying parameter determination and calibration, but until this toolchain is automated, the results will raise more questions than provide answers.

4.3. *GPU* à la *(QRQW) PRAM [14]*

Description: 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 [35], PRAM [8], and QRQW PRAM [9] models and get a good approximation of the execution times. Compared with the MWP–CWP model (see Section 4.2), 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. As long as determining this crucial parameter of the model is not somehow automated, the model is difficult to use for large applications. To alleviate this challenge, we envision a two-stage estimation: a *portable* microbenchmarking approach to determine the duration of different operations for different platforms, and a (profiler-assisted) estimation of the numbers of operations of different kinds. We further note that the model is not modeling in anyway the caching behavior of applications on the GPU (as it has been designed for pre-Fermi architectures) and it has no support for estimating the effects of synchronization. While compensating for these lapses can probably be done by more extensive microbenchmarking, there is no evidence such approximation (*i.e.* including the effects of these elements as part of the cycle estimation for different operations) can preserve the accuracy of the model.

[1] This statement was made in a direct discussion with one of the authors.
[2] Note to reviewers: we will include the evaluation results for this particular system in the next version of the paper.

Evaluation: To assess the usability of this model, we have attempted to predict the runtime of our three benchmarks – MM (as implemented in the original paper [14]), HS, and BFS, using the C2050 architecture. Note that C2050 features a newer generation of GPU architecture compared with the ones discussed in the paper. For MM, we used the same estimations for cycles per thread as provided in the original paper. This approach lead to a 50% prediction error for all three matrix sizes. This is due to a systematic error in the estimation of the cycles per instruction for the computation. By tuning the number of compute cycles per operation to a more accurate number for C2050, we were able to reduce the prediction error to about 3%, an excellent result. For HS, however, the accuracy of the prediction was 2%, -65%, and 15% for the three datasets. Given the memory-intensive nature of the application, we conclude that the differences in the performance of the caches appearing with increasing the size of the input dataset is responsible for this large variation. For BFS, the results were even more difficult to estimate: determining the number of compute cycles and memory cycles spent by every thread is highly challenging, given the data-dependent nature of the computation. While code instrumentation can be used to count the performed operations, we believe this is beyond the scope of the model. Hence we conclude that this model is not suitable for irregular applications, where the per-thread workloads vary significantly.

In summary, this model is promising for high-level approximation of the execution times, and it can be very useful, once calibrated, to assess different geometries and/or application parameters (*e.g.*, the tile sizes in the matrix multiplication). The calibration of the model for a given application remains challenging, requiring significant microbenchmarking effort. Finally, the model cannot be used to analyze applications where the mapping of data-items to threads is non-trivial or where the per-thread workloads are data-dependent.

4.4. Components [33]

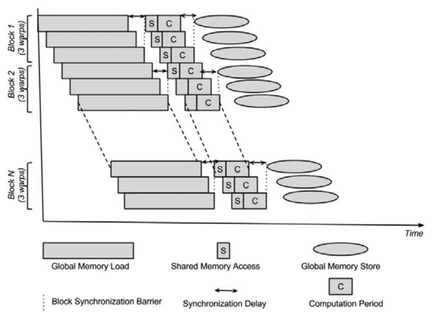
Description: The contribution's goal is the prediction of optimal power-performance efficiency on emerging GPU architectures. To this aim, it builds three analytical models for power, performance and energy and identifies power-performance bottlenecks using performance counters and machine learning. In this paper, we only consider the performance modeling aspects. As of the performance model, it uses selected performance events to isolate the time spent on each major GPU architecture component (global memory, shared memory, computation and texture) and derive from cross-component analysis the potential performance bottlenecks accordingly (largest time). The approach uses 13 counters and more than 30 parameters whose values are acquired from code analysis, performance counters, microbenchmarks and hardware specifications. Thus, it characterizes both the kernel and the GPU. The output is total execution cycles prediction and bottleneck identification. Experiments are conducted on an NVIDIA Tesla C2075 GPU for 12 CUDA kernels from the SDK and GEM (bimolecular electrostatic analysis application) with an average absolute prediction error rate of 6.7%.

Evaluation: Although this approach considers major performance factors, one of its weaknesses is that it does not define a systematic way of finding the total execution cycles of the kernel based on that of the architectural components. Instead, the user needs to figure out what the execution time pipeline of each specific kernel is (using a paper-and-pencil drawing) and, analogously to a case study from the paper, determine the total execution cycles. This exercise, already complex for small kernels, is practically impossible for larger ones. We have evaluated this model using our three kernels on an NVIDIA Fermi GPU (Tesla C2050), just as the authors. As we could not get the microbenchmarks used by the authors to determine some of the parameter values, we had to estimate them from other work. One of those parameters is the average delay between two global memory transactions which corresponds to departure delay as defined in [31] and evaluated on a Tesla C2050. Fig. 3(a) shows the execution time pipeline of the MM kernel, and Fig. 3(b) reports the accuracy of the model for that same kernel. We observed that the model is inaccurate for small matrices and relatively accurate as the matrix becomes larger. We also illustrate the performance for the other two benchmarks in Fig. 3(c) and (d). The very bad performance of the model for BFS is to be linked with the fact the model only supports balanced kernels. The poor performance on HS could mean the model may not capture well either occupancy or synchronization.

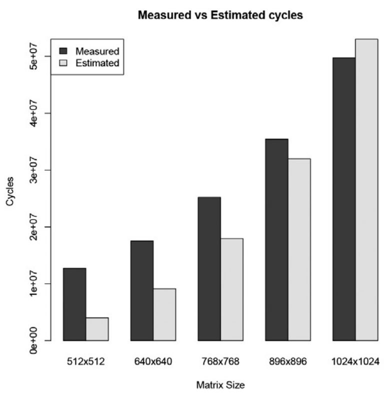
In summary, Components tool captures most performance factors and the component-wise approach allows the discovery of performance bottlenecks with component granularity. However, the lack of a systematic way to estimate the overall kernel time is a serious drawback of the tool.

4.5. Transit [19]

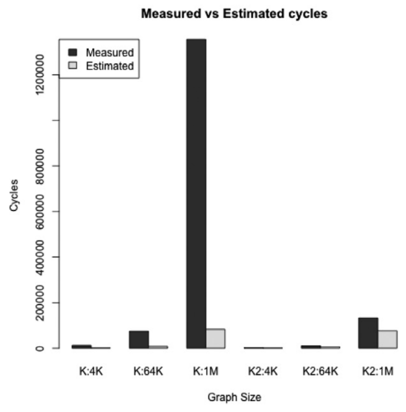
Description: Transit is a high-level, visual and throughput-oriented analytical model for general multithreaded machines. Based on flow balancing between service demand and supply of the memory system, it is essentially a throughput analysis tool. Transit sees a multithreaded machine as composed of a computation system (CS) and a memory system (MS). The machine and its components are all modeled using the queuing networks formalism. The machine itself is modeled as an interactive queuing network, MS an aggregate queuing system and CS a single-queue-multiple-server where each server represents an execution lane or thread slot. Threads represent the users of the queuing network. The intersection of CS service demand (to MS) throughput function and MS service supply throughput function represents the equilibrium between service demand and supply. Combining both function curves into one plot (called transit figure), reveals the bottlenecks of the machines and the optimization directions to overcome them. The model seems easy to use as it requires only a few parameters from the platform and workload. The model is validated on 16 kernels of the Rodinia benchmark suite using NVIDIA Fermi and Kepler GPUs, achieving prediction accuracy of 90% and 83%, respectively.



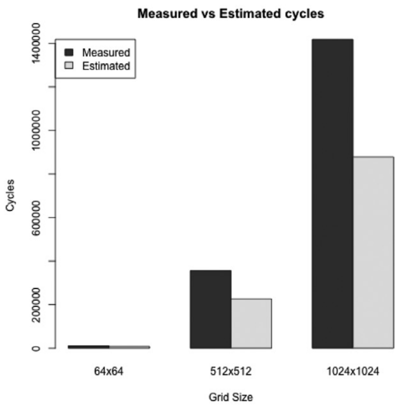
(a) Execution time pipeline of MM kernel



(b) Accuracy of the Components model for the MM kernel



(c) Accuracy of the Components model for the two kernels (K and K2) of BFS



(d) Accuracy of the Components model for the HS kernel

Fig. 3. Results for the Components model.

Evaluation: As an analytical visual model, Transit is relatively laborious to build despite the small number of parameters, but once built it is easy to use for bottleneck analysis. The accuracy is good, although it seems to deteriorate on newer architectures with more subtle performance factors. Indeed, Transit only considers the number of execution lanes in the architecture, ignoring other important factors such as cache levels (including shared memory) and SIMD width. The available parallelism in the workload is captured by n, the number of active threads. Memory access patterns and divergence behavior, which are important performance factors on current GPUs, are left out. Furthermore, Transit is clearly designed for throughput bottleneck analysis, and consequently, cannot be used, for instance, for execution time prediction.

In summary, Transit is an elegant and insightful approach for throughput analysis of multithreaded machines, in the lineage of the Roofline model, but it does not cover all performance factors of current parallel multi-threaded architectures such as GPUs.

5. Statistical methods

In this section, we describe in detail four methods from this class.

5.1. *Eiger framework [12]*

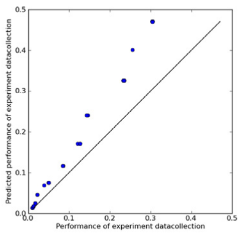
Description: Eiger is an automated statistical methodology for modeling program behavior on different architectures. To discover and synthesize performance models, the methodology goes through 4 steps:

(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.

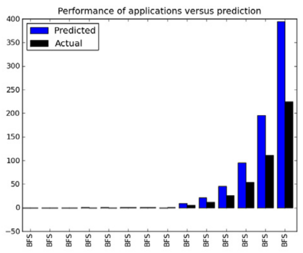
For the training phase (step 1), the application is executed on the target processor, facilitating the measurement of execution time and the collection of multiple parameter values (characterizing both processors and applications, 47 in total). In

Eiger metric	Counter
memory efficiency	(gld_efficiency+gst_efficiency)/2
memory intensity	ldst_executed / inst_executed
memory sharing	code analysis
activity factor	CUDA occupancy
SIMD/MIMD	execution configuration
DMA size	code analysis

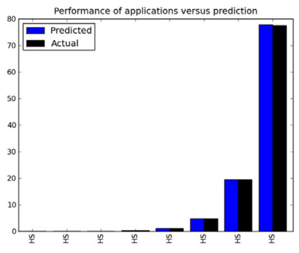
(a) Metric correspondence. For performance counter definition, see `http://docs.nvidia.com/cuda/profiler-users-guide`



(b) Accuracy of Eiger for the MM kernel



(c) Accuracy of Eiger for the BFS kernels



(d) Accuracy of Eiger for the HS kernel

Fig. 4. Results for the Eiger model.

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 over training data and a model pool consisting of basis functions. Model evaluation produces execution time prediction. The approach is validated on 12 applications from different GPU benchmark suites and the CUDA SDK on three NVIDIA Fermi GPUs.

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 of the table in Fig. 4(a) shows some of the metrics used in a GPGPU use case. Besides static code analysis and instrumentation, the experimental data collection stage leverages Ocelot [13], a PTX simulator, for features necessitating code emulation. A software package[3] 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 Fig. 4(a). Consequently, some part of inaccuracy may be accountable to this fall-back. We have trained the model on a Tesla C2050 and predicted the execution time on a Geforce GTX580. Even though both GPUs correspond to Fermi architecture, the predictions accuracy shown in Fig. 4(b) is not ideal. Observe that the more accurate the model, the closer the data points to the $y = x$ line. The accuracy of the model for the other two benchmarks is shown in Fig. 4(c) and (d). For those two benchmarks, the test data is acquired on a GTX 480. The low accuracy is confirmed on the BFS kernel whereas Eiger shows high accuracy on HS. A peculiarity of HS is that we have only a few data points (8) which might not reach adequate statistical significance. Furthermore, the execution times are quite close for the C2050 and the GTX480. *In 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.

5.2. Stargazer framework [11]

Description: 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 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. The model consumes parameter values in input and produces execution time.

[3] https://bitbucket.org/eanger/eiger.

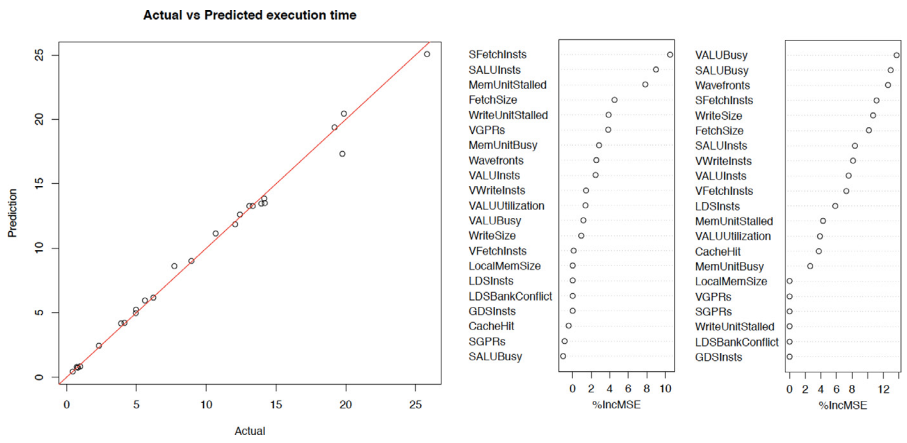


Fig. 5. Accuracy and variable importance of the random forest approach. The plots are for the MM, BFS and HS kernels, respectively from left to right.

Several CUDA applications from benchmark suites are used to validate the approach on an NVIDIA GPU 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 such as 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 [3] 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. We do not show figures for this tool because of the uncertainty about the measurement units.

In 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.

5.3. GPURandomForest [39]

Description: This approach aims at capturing relationships between execution characteristics of a kernel on a GPU and the achieved performance (and its power consumption), based on which the authors derive instructive guidance to software designers and hardware architects for building more power-efficient high performance systems. They make use of a statistical tool, a random forest, to find the most influential variables to the execution performance (throughputs) of the target GPU. A random forest is a set of regression trees providing interpretation tools for visualizing the relative importance of each input variable and its relation to the GPU performance. The approach consists in collecting performance data (from ATI profiler), in feeding these data to the random forest modeling and in deriving insightful principles from the modeling results. Input variables consist of 23 performance counters. The random forest can be used both for execution time prediction and variable importance visualization. The approach is validated on 22 kernels from ATI Stream SDK on a Radeon HD5870 with a coefficient of determination of 79.7% and a median absolute error of 13.1%.

Evaluation: We evaluated this model using our three benchmarks on a Radeon HD7970 GPU. We ran the kernels inside the profiler version 2.5 and let it collect all available performance counters. For each benchmark, we performed several executions to gather as much data as possible using different input parameters. Then, using the R randomForest package [20], we built the random forest object by specifying all the collected counters as the independent variables and the execution time, that we also collected, as the response variable. Finally, we used that object to assess the prediction accuracy and to visualize the most influential counters in the performance of the benchmark on the GPU. From the leftmost image in Fig. 5, we observe that the predictions are reasonably accurate on this particular case of the MM kernel. The same holds for the other (not shown) kernels in a lesser extent. The remaining plots show the variable importance for BFS and HS kernels. For BFS, the most important counters pertain to global memory operations (*e.g.*, SFetchInsts, MemUnitStalled, WriteUnitStalled), corroborating the global memory intensiveness of the kernel. However, ALU instructions dominate HS execution, which is in contradiction with the memory intensive nature of most stencil codes.

In summary, GPURandomForest adopts a very interesting statistical approach but its total reliance on performance counters limits somehow the usability of the solution (only GPUs for which performance counters can collected) and the current counters do not cover all performance factors (*e.g.*, there is no counters pertaining to memory coalescing).

5.4. ScaleClass [37]

Description: This approach focuses on rapidly estimating the performance and power of GPUs across a range of hardware configurations using statistics and ML. First, a training set is gathered on a collection of OpenCL kernels that are run on a real GPU at various core frequencies, memory bandwidths and compute unit counts. For the training set, execution time and power are measured, and hardware performance counters are collected. Then, *scaling surfaces* describing how the performance and power of these kernels change across hardware configurations are built from the measurements. Later, when analyzing the scaling behavior of a new kernel, performance and power data are first acquired from a single configuration. That data is then passed to the model along with the desired target hardware configuration. The model will output predicted performance and power for the target hardware configuration. The model is constructed in two major phases. In the first phase, the training kernels are clustered to form groups of kernels with similar performance and power scaling behaviors across hardware configurations. In the second phase, a neural network classifier is constructed to predict which cluster's scaling behavior best describes a new kernel based on its performance counters. To evaluate their approach, the authors use a large number of OpenCL kernels of which 20% serve as validation set on an AMD GPU used as base hardware configuration. The performance and power estimates are fairly accurate: within 15% and 10%, respectively, of the measured data.

Evaluation: ScaleClass is ML-based, so it requires less user intervention and can be automated, leading to very easy model construction. Similarly, the model usage is relatively simple as it only requires the user to run the target kernel once on the base architecture. However, the accuracy for performance prediction is below typical values published elsewhere, *e.g.*, Stargazer and GPURandomForest previously described. Probably because regression-based approaches usually perform better than neural networks in terms of accuracy [17]. However, this can be acceptable, as ScaleClass's main goal is scaling behavior analysis. Furthermore, although the model integrates major performance factors, it ignores caching from both the analyzed hardware configurations and the performance counters capturing the performance signature of the studied workloads.

In summary, this approach is useful for scalability analysis, and has the potential to also be useful for accurate predictions, provided that accuracy can be improved and lacking performance factors such as caching can be considered.

6. Quantitative methods

In this section, we describe in detail the most representative quantitative analysis (QA) method, by Zhang et al. [38].

Description: 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.

In 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.

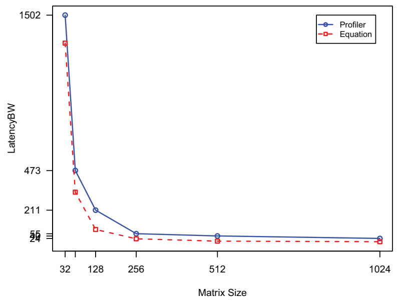
7. Compiler-based methods

In this section, we describe the WFG Modeling Tool [2], the representative solution for compiler-based methods.

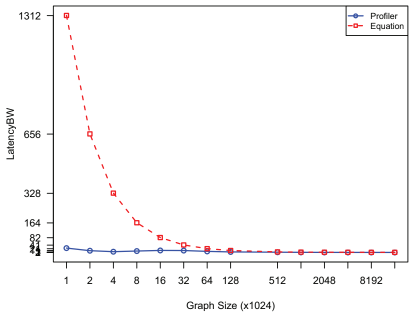
Description: A GPU analytical performance modeling tool is defined by abstracting a GPU kernel as a Work Flow Graph (WFG), based on which the kernel execution time is estimated. The kernel is represented by its dependence graph (both control flow and data dependence graph). The nodes of the WFG are instructions 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

WFG metric	Counter
$Latency_{BW}$	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

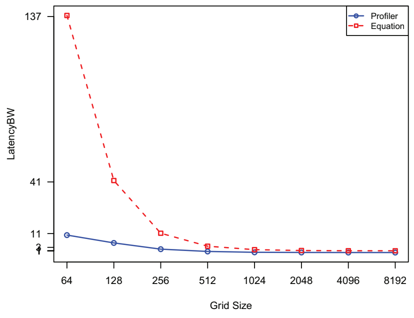
(a) Correspondence between WFG metrics and combinations of performance counters and hardware specification. WordSize is determined from the application; Cycles per instruction (CPI) and BWPerSM are determined from the hardware specifications; all the other metrics are extracted from the CUDA profiler



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



(c) Accuracy of the WFG prediction for BFS on NVIDIA Tesla C2050



(d) Accuracy of the WFG prediction for HS on NVIDIA Tesla C2050

Fig. 6. Accuracy of the WFG prediction.

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 CUDA kernels on NVIDIA GPUs 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 caching performance, 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 we could not find despite our inquiry to the authors, making it impossible for us to directly evaluate this model for our benchmarks. However, we managed to build an approximate WFG in a different way for the MM kernel. Eq. (4) in [2],

$$Latency_{BW} = \max(0, \frac{CYC_{mem} - CYC_{compute}}{NUM_{mem}}) + \frac{SIMD_{work}}{SIMD_{engine}}, \quad (3)$$

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. $SIMD_{work}$ is the warp size and $SIMD_{engine}$ the number of SPs in a SM. For a memory-bound kernel like MM, this latency 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 Eq. (3) and combinations of hardware performance counters measured by the CUDA profiler, as shown the table in Fig. 6(a), we plot the values of LatencyBW from both the profiler and the model in Fig. 6(b)–(d). We observe that global memory latency as seen by the application (thus Eq. (3)) and by the hardware do not always agree.

In 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.

Table 2
Key findings of our study of the selected GPGPU performance models. For brevity, we use the following abbreviations: V means variable, ETP execution time prediction, PBA performance bottleneck analysis and OSE optimization space exploration.

Model	Accuracy	Construction effort	Evaluation speed	Abstraction level	Hardware knowledge	Insight
PMAC	Low	Major	High	Medium	Limited	ETP
MWP–CWP	Low	Major	High	Fine	High	ETP
GPUPerf	?	Major	High	Fine	High	ETP, PBA
GPU à la PRAM	V	Moderate	High	Coarse	None	ETP
Components	V	Major	High	Coarse	Good	ETP, PBA
Transit	?	Moderate	High	Coarse	Limited	PBA
Eiger	Low	Moderate	Low	Coarse	Good	ETP, OSE
Stargazer	Low	Minor	Low	Fine	Good	ETP, PBA, OSE
GPURandomForest	V	Minor	High	Fine	None	ETP, PBA, OSE
ScaleClass	Low	Minor	Moderate	Coarse	None	ETP, PBA, OSE
QA	?	Moderate	High	Coarse	None	ETP, PBA
WFG	V	Major	High	Fine	Good	ETP, PBA

8. Discussion

In this section we discuss the overall results of our evaluation, focusing on comparing the results across the different classes of methods and tools. Specifically, Table 2 presents an overview of the features and evaluation results for all the tools. Besides ‘Accuracy’ and ‘Construction Effort’, which qualify the gain *versus* the effort a model requires, we also estimate ‘Evaluation Speed’ (speed of evaluating the model once constructed), ‘Abstraction Level’ (level at which the model abstracts the hardware), ‘Hardware Knowledge’ (amount of architectural details assumed known to the user), and ‘Insight’ (performance information deduced from the model), providing a more in-depth comparison of these models. In the following paragraphs, we emphasize the results and trends this table exposes. Finally, we propose a more in-depth, goal-based taxonomy for these models.

Generally, analytical models are fairly accurate for the hardware family and, to some extent, for the workload they are built for. They are less accurate for different hardware and kernels. We specify their accuracy as variable, except for PMAC, which has very limited applicability. Their evaluation is fast once they are built, but their construction requires major effort, especially when their abstraction level is fine. The insights they provide are execution time prediction and performance bottlenecks highlighting.

We found two of the ML-based models to be relatively inaccurate, partially because of imprecisions in the underlying measurement tools (*e.g.*, GPGPU-Sim) and because we used performance counters as replacement in the case of Eiger. In contrast to those models, GPURandomForest can be accurate in some cases (the MM kernel). The relative superiority of GPURandomForest is explained by the regression techniques used, superior to the linear or step-wise linear approach used in the other models. The model construction here requires little effort, assuming the tools function correctly. When the model construction and evaluation are done in one go, the evaluation is slow; otherwise, it is fast. Provided that the model is counter-based and the GPU to be worked with is available, no specific hardware knowledge is required. For other cases, good knowledge of architectural details is necessary to create the experimental data. The insights such models provide are prediction and optimization space exploration, with some additional capability of identifying performance bottlenecks.

The only purely quantitative approach, QA, could not be evaluated because of missing microbenchmarks. Building such benchmarks would require important initial effort which is amortized by the speed of the evaluation consisting of simple table lookups. Once the benchmarks are available, no hardware knowledge is required. Finally, as this approach is based on measurements and provided that caching is taken into account, we expect good accuracy within device generation boundary.

We could not entirely evaluate the sole compiler-based approach because of the unavailability of the compiler front-end necessary to build the WFG. We ended up using a component-based approach which shows some variability of the accuracy. The combined complexity of the analytic modeling and of the program analysis to determine the WFG, the computation and memory access patterns and loop bounds make its construction very difficult. All the other aspects are inherited from the underlying analytical model.

The examination of the last column of Table 2 also reveals that the advertised features (as in Fig. 1) do not always correspond to the real capabilities of the tools. Indeed, whereas actual goals of tools such as PMAC, GPU à la PRAM and Components do correspond to their advertised goals, we did not find a straightforward way of using MWP–CWP for performance bottlenecks analysis. In contrary, Stargazer and GPURandomForest do not advertise performance bottlenecks analysis and optimization space exploration, yet they can be used for such analysis respectively.

In consistence with Section 3.2, an ideal model would have high accuracy, minor construction effort, fast evaluation, high level abstraction of the hardware requiring no or little knowledge of architectural details and giving as various insights as possible, including execution time prediction, performance bottlenecks highlighting and optimization space exploration. We observe that there is no such a model in Table 2. Instead, every model has its strengths and weaknesses. It will probably be a combination of models, heavily supported by tools, that will manage to sustain the performance prediction and optimization space exploration needs of an ordinary GPU programmer.

Finally, the severe imbalance between the classes of the naive categorization we used so far made us realize it is probably more valuable to consider an alternative taxonomy, based the performance information (hence, usability) of the model. Following this criterion, we identify four overlapping categories: (1) models which only predict execution time (PMAC), (2) models used to explore a design and/or optimization space (Eiger, Stargazer, GPURandomForest, GPUPerf), and (3) models for performance bottlenecks identification (MWP–CWP, Components, GPURandomForest, WFG).

9. Conclusion

GPUs become a natural computation accelerator by being present in almost any modern architecture. The efforts put into simplifying their programming have successfully attracted many regular users and non-expert performance engineers into using GPUs. However, these efforts were not matched in the field of performance modeling and prediction, where the few available tools seem to remain unusable for the ordinary developer. In this work, we provided a clear overview of current effort, mostly academic, of modeling GPGPU performance. In addition to providing an overview that we believe useful, we also *actually evaluate* 10 promising performance models, putting ourselves in the position of the programmer, sometimes the architect, looking for insights into the performance behavior of the application or the hardware. We point out the following findings:

- most of the contributions can be categorized in a few classes based on methodology leading to a simple taxonomy,
- members of each class in that taxonomy exhibit some consistency regarding criteria such as accuracy, construction effort or evaluation speed but can be different with regard to others such as knowledge required from the user or the insight provided,
- the classes in this first taxonomy are very unbalanced, most models being either analytical or statistical, and
- a more useful taxonomy is one based on the insight provided, as the user is primarily interested in “anything” that can help her improve her application performance, and cares very little about the used methodology.

Overall, a lot of interesting work is being done but there is no outstanding model yet. None of the models have such high accuracy or such overwhelming simplicity to become the GPGPU performance model. We believe that, although the complexity of parallel computation and architecture makes it very hard to build such a model for GPGPU computation, performance modeling is on the critical path to the wide-spread adoption of heterogeneous computing.

Acknowledgments

This work is in part funded by the research program of the Netherlands eScience Center (www.nlesc.nl).

References

[1] Advanced Micro Devices (AMD) Inc., Press Release: AMD Delivers Enthusiast Performance Leadership With the Introduction of the ATI Radeon 3870 X2, Advanced Micro Devices (AMD) Inc., 2008.
[2] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, W.-m.W. Hwu, An adaptive performance modeling tool for GPU architectures, SIGPLAN Not. 45 (5) (2010) 105–114, doi:10.1145/1837853.1693470.
[3] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, T.M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator, in: Proceedings of ISPASS'09, IEEE, 2009, pp. 163–174.
[4] L. Carrington, M.M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, S. Poole, An idiom-finding tool for increasing productivity of accelerators, in: Proceedings of ICS'11, ACM, New York, NY, USA, 2011, pp. 202–212, doi:10.1145/1995896.1995928.
[5] AMD CodeXL. http://developer.amd.com/tools-and-sdks/opencl-zone/codexl (accessed 19.10.14).
[6] S. Collange, D. Defour, D. Parello, Barra, a modular functional GPU simulator for GPGPU, techreport, UPVD, France, 2009.
[7] G.F. Diamos, S. Yalamanchili, Harmony: an execution model and runtime for heterogeneous many core systems, in: Proceedings of HPDC'08, ACM, New York, NY, USA, 2008, pp. 197–200, doi:10.1145/1383422.1383447.
[8] S. Fortune, J. Wyllie, Parallelism in random access machines, in: Proceedings of STOC'78, ACM, New York, NY, USA, 1978, pp. 114–118, doi:10.1145/800133.804339.
[9] P.B. Gibbons, Y. Matias, V. Ramachandran, The queue-read queue-write asynchronous PRAM model, in: Proceedings of Euro-Par '96, Springer-Verlag, London, UK, UK, 1996, pp. 279–292.
[10] S. Hong, H. Kim, An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, SIGARCH Comput. Archit. News 37 (3) (2009) 152–163, doi:10.1145/1555815.1555775.
[11] W. Jia, K. Shaw, M. Martonosi, Stargazer: automated regression-based GPU design space exploration, in: Proceedings of ISPASS 2012, 2012, pp. 2–13, doi:10.1109/ISPASS.2012.6189201.
[12] A. Kerr, E. Anger, G. Hendry, S. Yalamanchili, Eiger: a framework for the automated synthesis of statistical performance models, in: Proceedings of WPEA 2012, 2012.
[13] A. Kerr, G. Diamos, S. Yalamanchili, A characterization and analysis of PTX kernels, in: Proceedings of IISWC'09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 3–12, doi:10.1109/IISWC.2009.5306801.
[14] K. Kothapalli, R. Mukherjee, M. Rehman, S. Patidar, P.J. Narayanan, K. Srinathan, A performance prediction model for the CUDA GPGPU platform, in: Proceedings of HiPC 2009, 2009, pp. 463–472, doi:10.1109/HIPC.2009.5433179.
[15] K. Asanovic, et al., A view of the parallel computing landscape, Commun. ACM 52 (10) (2009) 56–67, doi:10.1145/1562764.1562783.
[16] M. Laurenzano, M. Tikir, L. Carrington, A. Snavely, PEBIL: efficient static binary instrumentation for Linux, in: Proceedings of ISPASS 2010, 2010, pp. 175–183, doi:10.1109/ISPASS.2010.5452024.
[17] B.C. Lee, D.M. Brooks, B.R. de Supinski, M. Schulz, K. Singh, S.A. McKee, Methods of inference and learning for performance modeling of parallel applications, in: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'07, ACM, New York, NY, USA, 2007, pp. 249–258, doi:10.1145/1229428.1229479.
[18] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, SIGARCH Comput. Archit. News 38 (3) (2010) 451–460, doi:10.1145/1816038.1816021.

[19] A. Li, Y. Tay, A. Kumar, H. Corporaal, Transit: a visual analytical model for multithreaded machines, in: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, PDC'15, ACM, New York, NY, USA, 2015, pp. 101–106, doi:10.1145/2749246.2749265.

[20] A. Liaw, M. Wiener, Classification and regression by randomforest, R News 2 (3) (2002) 18–22.

[21] M.D. Linderman, J.D. Collins, H. Wang, T.H. Meng, Merge: a programming model for heterogeneous multi-core systems, SIGPLAN Not. 43 (3) (2008) 287–296, doi:10.1145/1353536.1346318.

[22] U. Lopez-Novoa, A. Mendiburu, J. Miguel-Alonso, A survey of performance modeling and simulation techniques for accelerator-based computing, IEEE Trans. Parallel Distrib. Syst. 26 (2014) 272–281, doi:10.1109/TPDS.2014.2308216.

[23] S. Madougou, A. Varbanescu, C. de Laat, R. van Nieuwpoort, An empirical evaluation of GPGPU performance models, in: Proceedings of Euro-Par 2014: Parallel Processing Workshops, in: Volume 8805 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 165–176, doi:10.1007/978-3-319-14325-5_15.

[24] M.R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, S. Poole, Modeling and predicting performance of high performance computing applications on hardware accelerators, IJHPCA 27 (2) (2013) 89–108, doi:10.1177/1094342012468180.

[25] G.R. Mudalige, M.K. Vernon, S.A. Jarvis, A plug-and-play model for evaluating wavefront computations on parallel architectures, in: Proceedings of IPDPS'08, IEEE, pp. 1–14.

[26] NVIDIA Corporation, Press Release: NVIDIA Tesla GPU Computing Processor Ushers in the Era of Personal Supercomputing, NVIDIA Corporation, 2007.

[27] NVIDIA Nsight, Visual profiler, CUPTI. https://developer.nvidia.com/performance-analysis-tools. (accessed 19.10.14).

[28] V.K. Pallipuram, M.C. Smith, N. Raut, X. Ren, A regression-based performance prediction framework for synchronous iterative algorithms on general purpose graphical processing unit clusters, Concurr. Comput.: Pract. Exp. 26 (2) (2014) 532–560, doi:10.1002/cpe.3017.

[29] S. Ryoo, C.I. Rodrigues, S.S. Stone, S.S. Baghsorkhi, S.-Z. Ueng, J.A. Stratton, W.-m.W. Hwu, Program optimization space pruning for a multithreaded GPU, in: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO'08, ACM, New York, NY, USA, 2008, pp. 195–204, doi:10.1145/1356058.1356084.

[30] R.H. Saavedra, A.J. Smith, Analysis of benchmark characteristics and benchmark performance prediction, ACM Trans. Comput. Syst. 14 (4) (1996) 344–384, doi:10.1145/235543.235545.

[31] J. Sim, A. Dasgupta, H. Kim, R. Vuduc, A performance analysis framework for identifying potential benefits in GPGPU applications, SIGPLAN Not. 47 (8) (2012) 11–22, doi:10.1145/2370036.2145819.

[32] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, A. Purkayastha, A framework for performance modeling and prediction, in: Proceedings of SC'02, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 1–17.

[33] S. Song, C. Su, B. Rountree, K.W. Cameron, A simplified and accurate model of power-performance efficiency on emergent GPU architectures, in: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS'13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 673–686, doi:10.1109/IPDPS.2013.73.

[34] M.M. Tikir, M.A. Laurenzano, L. Carrington, A. Snavely, PSINS: an open source event tracer and execution simulator for MPI applications, in: Proceedings of Euro-Par'09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 135–148, doi:10.1007/978-3-642-03869-3_16.

[35] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111, doi:10.1145/79173.79181.

[36] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76, doi:10.1145/1498765.1498785.

[37] G. Wu, J. Greathouse, A. Lyashevsky, N. Jayasena, D. Chiou, GPGPU performance and power estimation using machine learning, in: Proceedings of IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015, pp. 564–576, doi:10.1109/HPCA.2015.7056063.

[38] Y. Zhang, J. Owens, A quantitative performance analysis model for GPU architectures, in: Proceedings of HPCA 2011, 2011, pp. 382–393, doi:10.1109/HPCA.2011.5749745.

[39] Y. Zhang, Y. Hu, B. Li, L. Peng, Performance and power analysis of ATI GPU: a statistical approach, in: Proceedings of the 2011 IEEE Sixth International Conference on Networking, Architecture, and Storage, NAS'11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 149–158, doi:10.1109/NAS.2011.51.