A Tool for Bottleneck Analysis and Performance Prediction for GPU-accelerated Applications

Souley Madougou,
Ana Lucia Varbanescu, Cees de Laat
University of Amsterdam, The Netherlands
{S.Madougou,A.L.Varbanescu,delaat}@uva.nl

Rob van Nieuwpoort
Netherlands eScience Center, The Netherlands
R.vanNieuwpoort@esciencecenter.nl

Abstract

High-level tools for analyzing and predicting the performance GPU-accelerated applications are scarce, at best. Although performance modeling approaches for GPUs exist, their complexity makes them virtually impossible to use to quickly analyze the performance of real life applications and obtain easy-to-use, readable feedback. This is why, although GPUs are significant performance boosters in many HPC domains, performance prediction is still based on extensive benchmarking, and performance bottleneck analysis remains a nonsystematic, experience-driven process. In this context, we propose a tool for bottleneck analysis and performance prediction for GPU-accelerated applications. Based on random forest modeling, and using hardware performance counters data, our method can be used to quickly and accurately evaluate application performance on GPU-based systems for different problem characteristics and different hardware generations. We illustrate the benefits of our approach with three detailed use cases: a simple step-by-step example on a parallel reduction kernel, and two classical benchmarks (matrix multiplication and sequence alignment). Our results so far indicate that our statistical modeling is a quick, easy-to-use method to grasp the performance characteristics of applications running on GPUs. Our current work focuses on tackling some of its applicability limitations (more applications, more platforms) and improving its usability (full automation from input to user feedback).

1. Introduction

GPUs are popular platforms for many parallel applications in need for performance, but we are far from using them efficiently and analyzing their performance critically. For example, we lack easy-to-use performance analysis and prediction tools, which should help programmers identify critical performance bottlenecks and assess the match between applications and architectures. Unfortunately, existing attempts for performance modeling require detailed knowledge of the hardware architecture, use complex methodologies, or are built to be architecture- or application-specific [11, 12]. Even architecture-specific tools cannot keep up with the newer generations of the same architecture, or require cumbersome calibration and additional tuning. This level of complexity is ultimately equivalent to lack of usability: most users will prefer performance debugging by trial-and-error instead of performance modeling through a complex procedure.

To address the need for simplicity, we propose in this work a performance analysis approach, called BlackForest (BF), that can be used to analyze GPGPU application performance. BF is a statistical method, which relies on the ubiquity of hardware performance counters and uses machine learning techniques to build performance models. We define the performance model as a random forest (RF) object, built from a collection of experimental data consisting of the values of GPU performance counters (acquired by profiling during the training phase), together with application and machine characteristics set as independent variables (i.e., *predictors*). Execution time is the response variable of interest. We have specifically selected random forest, because it usually outperforms the more traditional classification and regression algorithms, such as support vector machine and neural networks [10], especially for scarce training data.

The RF object learns from the training data how the various predictors relate to the response value, and builds a collection of regression trees which are used to predict responses for unseen inputs. To simplify the statistical model and its interpretation (i.e., reduce the number of variables), we use additional analyses, including linear and nonlinear regressions, and principal component analysis (PCA).

While building the regression forest, the most important predictors in determining the response are identified. This feature, called *variable importance*, combined with the partial dependence plot of individual predictors vis-à-vis the response, shows how a predictor affects the response, both qualitatively and quantitatively. Consequently, variable importance can be correlated to performance patterns [21], enabling us to provide systematic bottleneck detection and analysis, as well as suggest potential elimination strategies.

While hardware counters have been successfully used in the past to attempt performance prediction and energy consumption modeling, there is little work specifically targetting bottleneck analysis of GPUs. To the best of our knowledge, ours is the first work that uses this combination of statistical methods to build an accurate performance model, which in turn can be implemented as a tool for bottleneck analysis and performance prediction on both new data points and new hardware. Thus, the contribution of this paper is twofold. First, we introduce BlackForest, the first *easy-to-use tool* based on a mix of random forest modeling, regressions, and PCA, and able to predict performance of GPGPU applications for both data and hardware scaling. Second, we demonstrate how BlackForest is used to detect performance bottlenecks, resulting in useful insight into kernel-platform performance behavior.

The paper is structured as follows. We first discuss alternative performance modeling and analysis approaches in Section 2. Section 3 provides a brief introduction to GPU computing and GPU hardware performance counters, both necessary to further understand the details of the proposed method and its limitations. The core of our method is presented in Section 4, which also includes the architecture of the toolchain. Sections 5 and 6 showcase BlackForest at work. We conclude the paper by discussing the advantages and limitations of our method in Section 7.

2. Related work

There is a large body of research on performance analysis of both traditional parallel architectures [14, 16] and recent GPU-based systems [11, 12]. We only discuss here those contributions that either are similar to or had impact on ours, and we restrict ourselves to statistical models.

Many approaches have combined GPU hardware performance counters with machine learning for performance and/or power modeling. For example, he closest related work is by Zhang *et al.* [21], and describes a statistical approach aiming 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. The authors use random forest to find the most influential variables for the achieved throughput on the target GPU. The method collects performance data (from the ATI profiler), feeds them to the random forest modeling, and derives insightful principles. Input variables consist of a set of 23 performance counters. 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%. Besides minor differences in the used hardware and performance counters, our work also has a different goal: whereas the focus in [21] is put on the platform, our work analyzes the performance behavior of the application-platform combination. Furthermore, we are able to perform both bottleneck analysis and hardware scaling, which is not the case for this previous work. Other works, like [13, 18, 20], use less powerful statistical models in designing their approach, which fundamentally lack the ability to determine performance bottleneck analysis. Futhermore, none of them explores the idea of hardware scaling.

On the tools side, Stargazer (STAtistical Regression-based GPU Architecture analyZER) [7] 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. Stargazer aims at GPU design space exploration by pruning, and, as such, only considers hardware characteristics. Furthermore, those characteristics do not cover the entire feature space of current GPU architectures. In addition, as Stargazer relies on a simulator for experimental data collection, this process can take days, making it difficult to use as aid to application developers. Our approach, focusing also on application performance, is much quicker, although limited to existing hardware (i.e., no simulator is being used).

Finally, a more general tool example is Eiger [8], an automated statistical methodology for modeling program behavior on different architectures. To discover and synthesize performance models, the methodology has 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. 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 methodology is promising due to its high accuracy and automation, but it is complex and requires C++ programming for experimental data collection, in contrast to our work requiring only profiler runs and simple problem and machine characteristics.

In summary, statistical models and performance counters are already known to be useful for performance analysis, and various approaches have targeted different GPU architectures and performance analysis scenarios, ranging from hardware design to power modeling [7, 8, 13, 18, 20]. In this context, our work has *its specific use of counters* for *performance bottleneck analysis*, *hardware scaling*, and *performance prediction*. By combining hardware performance counters, random forest, and principal component analysis, BlackForest is built with existing, accessible tools, and enables automated performance prediction, complemented by performance bottleneck analysis.

3. Background

The background information in this section covers the GPU architecture basics and performance factors, the basics of performance counters, and elements of machine learning techniques used in this paper. Without losing generality, we will use NVIDIA's CUDA (Compute Unified Device Architecture) terminology to describe the GPU architecture and performance factors.

3.1 GPUs and Performance

GPUs are massively parallel, multi-threaded architectures with hundreds to thousands of processing cores and very high bandwidth between those cores and the off-chip device memory. From a hardware perspective, a GPU consists of one or more streaming multiprocessors (SM) and one or more L2 cache units. Each SM consists of schedulers, dispatch units, streaming processors (SP), register file and on-chip SRAM memory used as shared memory and L1 cache (for the more recent cards). Each SP contains CUDA cores for integer and floating-point operations, memory load and store units, and special function units. From a software perspective, a kernel running on the GPU corresponds to multiple threads executing in *warps* (groups of 32 threads on current NVIDIA cards). Warps are organized into thread blocks (TB), themselves structured in a grid. During execution, TBs are distributed to the SMs. The warp schedulers chose warps for execution from active TBs. The instructions in a warp are dispatched to functional units over several cycles. As TBs terminate, new blocks are launched on vacated SMs.

To efficiently utilize the hardware, applications must expose sufficient parallelism to fill the platform and optimize memory accesses and instruction execution for maximum throughput. To expose sufficient parallelism, there should be sufficient independent work within a thread, either independent arithmetic instructions or memory accesses, and sufficient concurrent threads (or warps). This is captured by the well-known occupancy metric, which is kernel-dependent. To maximize memory throughput, there should be sufficient concurrent memory accesses in flight and their address patterns must meet memory *coalescing* rules on the target architecture. To maximize instruction throughput, one has to take into account instruction mix as different instructions have different throughput and also serialization due to shared memory bank conflicts, uncoalesced memory accesses, and control flow divergence within warps.

3.2 Performance counters

Hardware performance counters (or, performance counters, PCs) are special-purpose registers built into modern microprocessors to store the counts of *hardware events*, or occurrences of specific signals related to the processor's function. Monitoring these events allows the programmer to establish correlations between code and its mapping to the underlying architecture. Initially proposed for CPUs, PCs are nowadays found on most GPUs, too. Popular vendors such as AMD and NVIDIA both provide PC interfaces for their products. For the NVIDIA products used in this work, PCs are accessible through APIs and tools such as profilers. In this study, we use **nvprof** to collect metrics and events of CUDA kernels running on NVIDIA GPUs. Without loss of generality, this paper uses the counters of NVIDIA Fermi GPUs (compute capability (CC) 2.0) and Kepler GPUs (CC 3.5).

Ideally, these metrics and events should be linked to the performance factors laid out in the above section. When faced with kernel optimization, it is important to first identify the performance-limiting factor. Several PCs can be used for that purpose. For instance, for architectures with different cache levels, the number of instructions issued (`inst_issued`) can be compared to the combined number of store transactions and of last level cache misses, `global_store_transaction` and `l1_global_load_miss`, respectively. Based on this, the user can then deduce whether the kernel is arithmetic-, memory- or latency-bound. To assess the attained level of parallelism, the classical occupancy metric is available as counter, `achieved_occupancy`. Several metrics are also available for throughput analysis. For memory, if the number of memory requests made by the kernel (`gld_request+gst_request`) is significantly lower than the number of actual memory transactions (`l1_global_load_miss+l1_global_load_hit +global_store_transaction`), this may indicate issues about memory access patterns. The same goes for arithmetic throughput, directly available as `ipc`, which one can compare to hardware specifications provided that the instruction mix is known. Furthermore, comparing the number of actual instructions issued (which include replays) to the number of instructions executed (`inst_executed`, which does not include replays) tells something about potential serialization issues if the former is significantly larger than the latter. The same conclusion can be drawn from analysis of the counter `inst_replay_overhead`. Other metrics directly measure well-known performance issues such as warp divergence through `branch` and `divergent_branch`, shared memory bank conflicts through `l1_shared_bank_conflict` and `shared_replay_overhead`. Based on these considerations, we have built a list of metrics and events, of which a sample is shown in table 1(the tools' guide[1] contains the full list).

4. Random Forest Modeling

For the proposed modeling, we make use of several machine learning tools to get insight into the performance behavior of GPGPU applications. The core of the approach is built around random forest modeling [2], a popular and very efficient algorithm for classification and regression problems [3, 21]. It belongs to the family of *ensemble learning* methods, *i.e.*, it generates and uses many classifier or regression trees and aggregates their results. Additionally, we use multivariate adaptive regression splines - MARS [5], and PCA. In this section, we give a brief overview of these methods, and we discuss how we use them, step by step, to build our performance model.

4.1 Statistical data analysis

This section provides the required background to understand the virtues and limitations of random forests, MARS, and PCA in the context of performance modeling.

[1] `http://docs.nvidia.com/cuda/profiler-users-guide/#axzz3ZYPyss4Z`

4.1.1 Regression trees and forest

Given a *learning set* (X, Y) consisting of p inputs and a numerical response, for each of N observations, *i.e.*, $(X, Y) = \{(x_i, y_i), i = 1, \ldots, N\}$, with $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$. A regression problem consists in finding a so-called *regression function* r such that $Y = r(X) + \epsilon$, where ϵ is zero-mean noise.

A regression tree, as a tree-based method, partitions the feature space into a set of regions, and then fits a simple model (such as a constant) to each region [6]. Recursive binary trees are used to represent the partitioning. The full dataset sits at the top of the tree. Observations satisfying the condition at each junction are assigned to the left branch, and the rest to the right branch. The terminal nodes or leaves of the tree correspond to the regions.

A regression tree is grown by letting the algorithm automatically decide on the variables and the points to split in order to achieve best fit. Suppose we have a partition into M regions $R_1, R_2, \ldots, R_M$, and we model the response as constant c_m in each region, the regression function would be:

$$r(x) = \sum_{m=1}^{M} c_m I(x \in R_m) \tag{1}$$

where I is the indicator or characteristic function over the regions. Using minimization of sum of squares $\sum(y_i - r(x_i))^2$ as criterion, it is easy to see that the best $\hat{c}_m$ is just the average of y_i in region R_m.

Using a greedy approach, the algorithm proceeds in finding the best binary partition in terms of minimum sum of squares. Starting from all data, consider a splitting variable j and split point s, and define the pair of half-planes

$$R_1(j, s) = \{x | x_j \leq s\} \quad \text{and} \quad R_2(j, s) = \{x | x_j > s\} \tag{2}$$

The algorithm seeks for j and s that solve

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right] \tag{3}$$

For any choice of j and s, the inner minimization is solved by taking the average of each region.

After the best split is found, the data is partitioned into two regions and the splitting process is recursively repeated on each of the two regions. The tree must be grown to the "right size" as a very large tree might over-fit the data, while a small tree might not capture all the important features. A possible, well-known strategy is to grow a large tree, stopping the splitting process only when some minimum node size (*e.g.*, 5) is reached. Then this large tree is pruned using a cost function to trade off between the tree size and its goodness of fit to the data [6].

One major problem with trees is their high variance: a small change in the data may result in a very different series of splits, making interpretation difficult. A solution around this issue is *bagging*, which averages trees grown from bootstrapped training sets, to reduce this variance. The principle of *random forest* is to *combine* many binary *decision trees* built from several bootstrap samples from the training set and choosing randomly at each node a subset of predictors at that node. In fact, RF adds some randomness to bagging, leading to better predictions compared to other approaches including discriminant analysis, support vector machines, and neural networks [10]. The forest is constructed using the following algorithm: *1*) compose n_t bootstrap samples from the original data, *2*) for each of those samples, grow an unpruned regression tree, and *3*) predict new data by averaging the predictions of the n_t trees.

RF provides two interpretation tools convenient for performance analysis: variable importance and partial dependence plots.

counter	meaning
shared_replay_overhead	*average number of replays due to shared memory conflicts for each instruction executed*
shared_load|store	*number of executed shared load (store) instructions, increments per warp on a multiprocessor*
inst_replay_overhead	*average number of replays for each instruction executed*
l1_global_load_hit	*number of cache lines that hit in L1 for global memory load accesses*
l1_global_load_miss	*number of cache lines that miss in L1 for global memory load accesses*
gld_request	*number of executed global load instructions increments per warp on a multiprocessor*
gst_request	*similar to* gld_request *for store instructions*
global_store_transaction	*number of global store transactions increments per transaction which ca be 32,64,96 or 128 bytes*
gld_requested_throughput	*requested global memory load throughput*
achieved_occupancy	*ratio of average active warps per active cycle to the maximum number of warps per SM*
l2_read_throughput	*memory read throughput at L2 cache*
l2_write_transactions	*memory write transactions at L2 cache*
ipc	*number of instructions executed per cycle*
issue_slot_utilization	*percentage of issue slots that issued at least one instruction, averaged across all cycles*
warp_execution_efficiency	*ratio of the average active threads per warp to the maximum number of threads per warp supported by the multiprocessor*

Table 1: Performance counters used in this study

Variable importance is estimated by looking at how much the prediction error increases when the values for that variable in the OOB sample are permuted while all others are left unchanged; the necessary calculations are carried out tree by tree as the forest is constructed. The second tool, the partial dependence plot, shows how the response changes as a predictor or a set of predictors change(s), which gives more interpretability to important variables.

4.1.2 Principal Component Analysis (PCA)

When possible, only the first few important variables (<10) are retained in the final model. Indeed, keeping a statistical model small is very important as it helps mitigate the risk of over-fitting and reduce the required number of samples for training [9]. To avoid any potential problems due to highly correlated variables ([19]), we use PCA, a dimensionality reduction method in which variables that are correlated (and thus measure the same property) are associated within higher dimensional variables. Specifically, PCA produces new variables, called principal components, from linear combinations of the original (potentially correlated) variables such that there is *no correlation between these components*. As a side effect, PCA's *factor loadings analysis*, i.e., the analysis of the coefficients of the variables in the principal components, may complement the insight given by the partial dependence plot from random forest modeling. We use these factor loadings to interpret the correlations, as seen in Section 5.

4.1.3 Multivariate adaptive regression splines

Finally, we also use several regression techniques to model the counters behavior in terms of problem and/or platform characteristics to make the use of the approach even easier. In simpler cases, (generalized) linear models are sufficient. In other cases, more elaborate approaches such as MARS (multivariate adaptive regression splines) [5] are used. MARS is a non-parametric method that automatically models nonlinearities and interactions between variables. MARS builds models of the form

$$\hat{f}(x) = \sum_{i=1}^{k} c_i B_i(x) \tag{4}$$

where c_i are constants and B_i are basis functions over $\mathbb{R}$. Each B_i is either a constant (there is only one such term called the *intercept*), a *hinge* function (of the form $\max(x-c)$ or $\max(c-x)$, $c \in \mathbb{R}$), or a product of hinge functions to model interactions between variables.

4.2 Our modeling approach

The core of our modeling approach is random forest modeling, which is a five-stage process: (1) data collection, (2) random forest construction and validation, (3) variable importance analysis, (4) refinement with PCA (optional, but recommended), and (5) results interpretation. We describe these phases in detail.

Data collection

We perform data collection by running the application multiple times (typically, tens to hundreds) on the architecture of interest, with different problem characteristics. We collect the characteristics, record the values of the performance counters of interest, as described in section 3, and measure the response variable (i.e., execution time, in this case). Performance counter data are collected using nvprof[2]. Problem characteristics are application-dependent: they typically include different input parameters, such as input size of a matrix multiply problem, or image size and pixel color information for a image processing kernel. The size of the data collection is important for the accuracy of the algorithm. So far, we have found that 100 samples are more than sufficient for 1-D problems, but finding a less empirical way to determine the ideal size is still work in progress. All the results in this paper have been obtained with collections of less than 100 data samples (training and test set, combined).

Random forest construction and validation

After random sampling of the collected data into a training set (80%) and a test set (20%), we construct a random forest using the first set. In most cases, this will result in a handful of most influential predictors (less than 10). We validate this set on the test data by checking that it retains most of the predictive power of the entire set using the OOB error rate and the variance explained.

Variable importance analysis

In this particular scenario, the response variable is execution time. Thus, we have just identified the most important variables, which

[2] NVIDIA's profiler user guide: http://docs.nvidia.com/cuda/profiler-users-guide

are the most influential predictors that can uniquely determine the execution time; the remaining, unimportant variables contribute only marginally or act as noise. The variable importance does not tell in which way a certain variable affects performance. For that, we resort to the partial dependence plot of the variable vis-à-vis the execution time, which shows the relative change of both variables on the entire range of runs. Monotonic variation over the entire range reveals strong correlation with the response, either positively or negatively. Variables strongly and negatively correlated to the execution time are identified as performance bottlenecks. In some cases, variables are strongly correlated only for part of the range and hence cannot explain the time variation elsewhere.

Refinement with PCA

There are pathological cases where the amount of variance explained by the forest is low, or the most influential variables do not retain enough prediction power, or they only explain portions of response variation over the entire range (see Section 6). In these cases, we combine the random forest with PCA for the selection of the most effective variables. For that, we use the factor loadings of the variable with relation to the retained principal components.

Results interpretation

After the most influential variables are known, and because we are aiming at ease of use, we model those parameters in terms of typical characteristics of either the problem in hand or both the problem and hardware type, so that predictions can be made solely based on the latter. For instance, we can use the models to generate values for the most influential variables from an unseen problem size for which the execution time will be predicted by the random forest. To be able to predict kernel performance on different hardware is slightly more complicated and requires modeling both the problem and architectural typical features. Because the accuracy of the final predictions depends on these models, the latter must be carefully implemented. Hence, unless confronted with trivial cases (*e.g.*, single problem characteristics such as matrix size in matrix multiply) for which (generalized) linear models are adequate, we use MARS regressions to take into account nonlinearities and parameter interactions.

4.3 Putting it all together: the toolchain

BlackForest is implemented as a toolchain that closely matches the stages of the methodology, as illustrated in figure 1. For data collection, we use specific architecture tools - i..e, nvprof for NVIDIA GPUs. The output of this stage is a collection of values for hardware performance counters (HWPC) and/or metrics derived from them. The data are stored in either a database or a structured repository (we used the latter), and further used for model building. We use tools from the R statistical software environment[3]: `randomForest` for RF and clustering, `prcomp` for PCA, `varimax` for varimax rotation of principal components, `glm` and `MARS` for counter modeling using regressions, along with various plotting packages. Once built, the model can be used for performance bottleneck analysis and/or performance prediction, enhanced with visualization and reporting capabilities.

5. Performance bottlenecks analysis

In this section we present the usability of our method for performance bottleneck analysis. Specifically, we show, through three well-known kernels from the CUDA SDK, how the variable importance feature of RF can be used to identify performance bottlenecks.

[3] `www.r-project.org`

5.1 Parallel reduction

Reduction consists in applying a binary associative operation to an array of values to produce a single value. Common binary operations are addition, maximization, and averaging. Parallel reduction is an important primitive commonly used by many parallel algorithms. The implementation in CUDA SDK uses a tree-based approach within TBs. In order to reduce large arrays with optimal performance, multiple TBs need to be launched. As each TB computes only a portion of the array, there should be multiple kernel launches to serve as synchronization points. CUDA parallel reduction is an educational example to showcase various CUDA optimization techniques. As reduction has low arithmetic intensity, the optimization goal is to maximize the bandwidth utilization. The benchmark consists of six kernels, each implementing a specific optimization technique addressing specific performance bottlenecks. In this section, we apply two statistical analyzes to three of those kernels: kernel 1 (`reduce1`), kernel 2 (`reduce2`) and kernel 6 (`reduce6`). For each kernel, we perform a fixed number of runs to collect data for the analysis. That data is first used to train a random forest where the execution time is the dependent variable and the other counters are the independent variables. After validating the model by predicting the execution time for unseen array lengths, we evaluate the importance of the different independent variables in the prediction along with their marginal effect using partial dependence graphs. When none of the variables covers the full range of time, we apply principal component analysis to correlate and associate appropriate variables and gain insight into their specific contributions to the performance.

5.2 Kernel 1 (reduce1)

`reduce1` substitutes strided accesses to shared memory to expensive modulo operations from original code. This introduces shared memory bank conflicts leading to overhead due to replays (execution of warp instructions in sequence instead of being run in parallel). From figure 2(a), we see that the random forest approach captures well this behavior. The x-axis in that figure represents the increase in MSE_{OOB}. Figure 2(b) shows the marginal effect of the `shared_replay_overhead` counter, the most important variable for the execution time prediction, as partial dependence plot. Replay overhead due to shared memory conflicts strongly and negatively affects the average predicted execution time, although this influence fades for short runtimes. So, we turn to PCA to get more insight. PCA produces four principal components (PC1, PC2, PC3 and PC4) accounting for more than 97% of the variance in the data. PC1 is related to memory intensity of `reduce1`, PC2 to MIMD and ILP parallelism, PC3 to SIMD efficiency, and PC4 to memory subsystem throughput. We observe that the most relevant counters (*e.g.*, `shared_replay_overhead` and `inst_replay_overhead`) are positively and strongly connected to PC2 and also negatively connected to PC4. While replays contribute to improve hardware utilization (through IPC for instance), they negatively impact the memory subsystem performance in the form of wasted bandwidth, especially replays due to bank conflicts.

5.3 Kernel 2 (reduce2)

`reduce2` improves over `reduce1` by replacing the strided memory accesses with a reversed loop and thread index-based access leading to sequential addressing. From figure 3(a), we see that now the most relevant counters all pertain to the memory subsystem performance. Observe how the most important counter for `reduce1` is the least important for `reduce2`. The partial dependence graph 3(b) of the most influential variable shows a strong positive relationship with the average predicted execution time, although on a rather limited range, before becoming mild. Once again, PCA produces four principal components, pictured in 3(c), and accounting for more

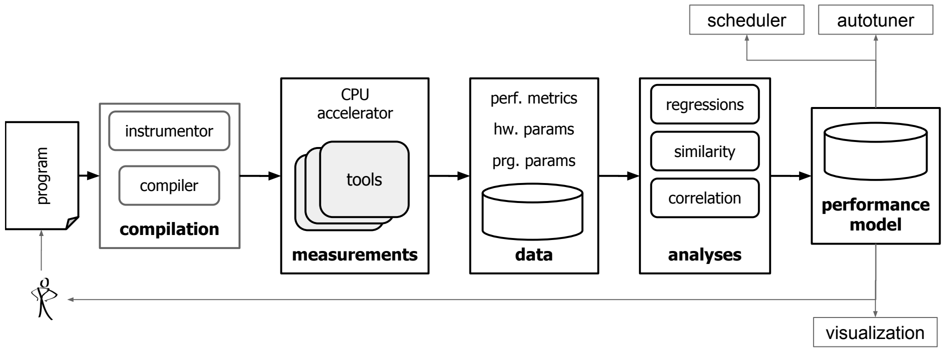


Figure 1: RFPM framework architecture.

than 96% variance. We observe that optimal GPU performance, both in term of instructions and memory, is precluded by low resource utilization. Indeed, whereas variables in the first component (*e.g.*, `gld_request`, `shared_load` and `l2_read_transactions`) have positive loadings, most in the other components have negative loadings (*e.g.*, `achieved_occupancy`, `ipc` and `ldst_fu_utilization`) meaning they have negative effect on optimal utilization. Furthermore, having `gld_requested_throughput` and `gst_requested_throughput` strongly and negatively correlated with PC3, again relative to memory subsystem throughput, means that the memory throughput requested by the kernel is not enough to fully utilize the hardware. This corresponds to the idling progressively experienced by an important number of threads in `reduce2`. It is worth noting how the metric measuring overhead due to shared memory bank conflicts also vanishes from PCA outcome.

5.4 Kernel 6 (reduce6)

Kernel 6 implements all applicable optimizations to maximize throughput, including complete loop unrolling, and processing multiple elements by each thread, which leads to better latency hiding and less overhead in kernel invocations. In figure 4(a), we observe that memory performance counters are still the most influential in predicting the execution time and that they have a strong correlation with it, as witnessed by figure 4(b) for `gst_request`. Principal components generated by PCA are as described in `reduce1`, with slightly different variables, memory throughput and occupancy counters strongly contributing to the memory throughput component. We observe that there is only a few variables such as `gld_requested_throughput` and `inst_replay_overhead`, seriously precluding optimal utilization, confirming the bandwidth bounded character of the reduction primitive.

6. Performance prediction

In this section, we present empirical evidence on the usability of BF for performance prediction. Specifically, we focus on two other case studies: matrix multiplication and the Needleman-Wunsch algorithm for sequence alignment, and demonstrate the applicability of our method for both *problem scaling* (i.e., predicting performance on unseen problem sizes) and *machine scaling* (i.e., predicting performance on unseen, yet similar hardware).

6.1 Problem scaling

This section puts the predictive power of random forest to work by applying it to two different kernels for predicting execution time for unseen problem characteristics.

6.1.1 Matrix Multiply

Matrix Multiply (MM) is a well-known kernel used in many applications. In this study, we use the tiled version as implemented in CUDA SDK[4]. This version of MM is compute intensive and bandwidth-limited for large matrix sizes. To compute the product C of two $n \times n$ matrices A and B, the kernel uses $b \times b$ tiles, with b a divisor of n. A grid of $\frac{n}{b} \times \frac{n}{b}$ TBs is launched, and each TB computes the elements of a different tile of C from tiles of A and of B, respectively. 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 and to global memory.

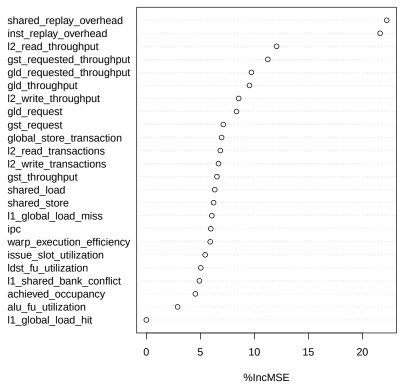
We vary the matrix size from 2^5 to 2^{11} (*i.e.*, 24 runs) to build an experimental dataset that we analyze using random forest. We randomly split the data into training and test sets (with a 80:20 ratio). Predictions are done on the test set. The bandwidth limitation is captured by the variable importance feature shown in figure 5(a). Indeed, the most important variables for the prediction are counters relative to global memory performance and occupancy, especially counters pertaining to global store throughput. This is expected as load and store operations are highly unbalanced. Indeed, for each result sub-matrix element, all elements of the operand sub-matrices have to be loaded in shared memory for computations, resulting in a ratio of block size loads per store. For larger matrix sizes, this means higher memory parallelism for load operations in contrary to stores which then become the bottleneck.

After the most influential variables have been identified, we retain the first few of them (usually, between 6 and 8) for further analysis. We first validate that those variables keep similar predictive power as the initial set. Next, we model the retained variables in terms of the matrix size; the models for MM are shown in figure 5(c) where solid lines represent predicted values and dotted lines measured values. These models, combined with the random forest, allow us to predict the execution times for unseen matrix sizes on the same hardware. This is illustrated in figure 5(b) where we observe that the predicted times mostly match the measured ones, with average MSE of 3.2 and 98% of explained variance. The models of important variables, built as generalized linear models because of their simplicity, have all low residual deviance, between 0 and 2.7, except for `inst_replay_overhead` whose average residual deviance is as large as 203, which is visible in figure 5(c) and affects the predictions.

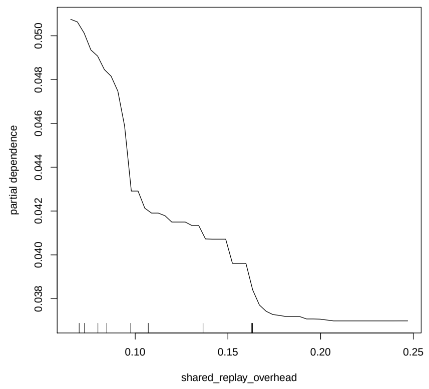
6.1.2 Needleman-Wunsch

Needleman-Wunsch (NW) is a nonlinear global optimization method for DNA sequence alignment. The potential pairs of sequences are organized in a 2D matrix filled from top left to bottom right with scores representing the value of the maximum weighted path ending at each cell. The maximum path is traced back to deduce the optimal alignment. NW is memory intensive and ultimately bandwidth-limited. The Rodinia GPU implementation processes the score matrix in parallel along diagonal strips us-

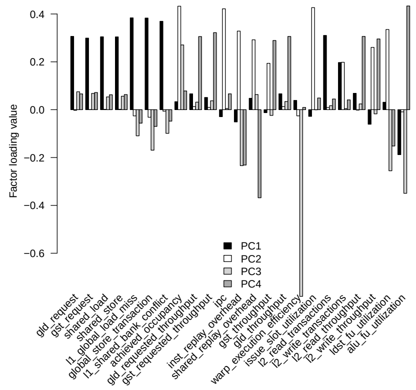
[4] `https://developer.nvidia.com/cuda-toolkit`



(a) variable importance

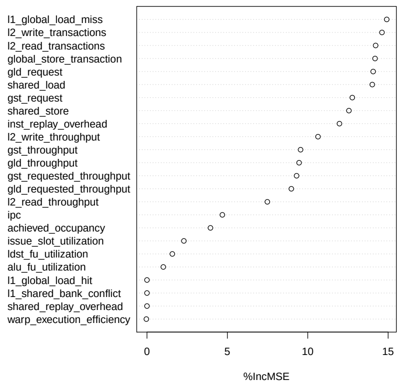


(b) Partial dependence

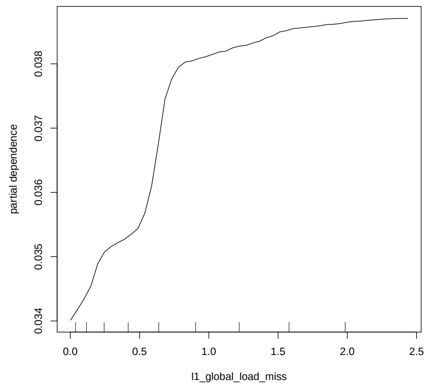


(c) Factor loadings

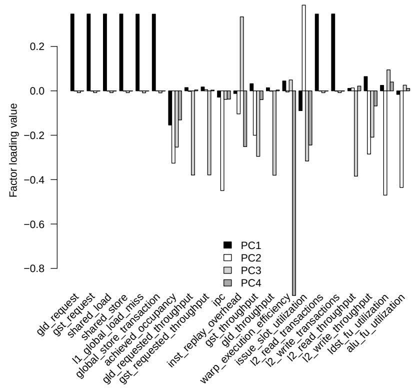
Figure 2: Counters affecting the performance of reduce1. Top three features, in order :`shared_reply_overhead, inst_reply_overhead, and l2_read_throughput.`



(a) variable importance

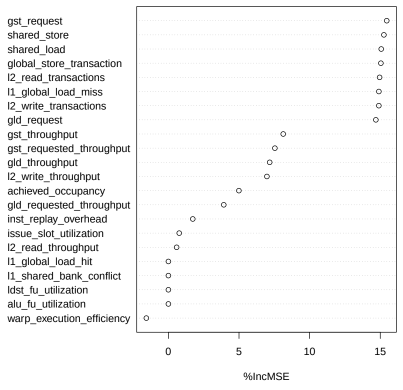


(b) Partial dependence

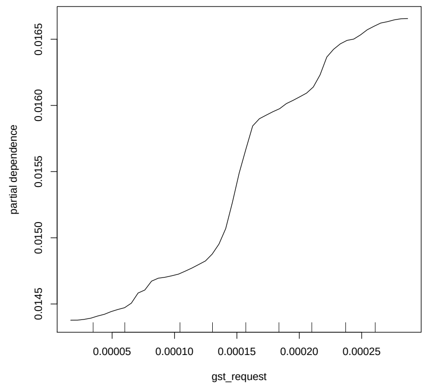


(c) Factor loadings

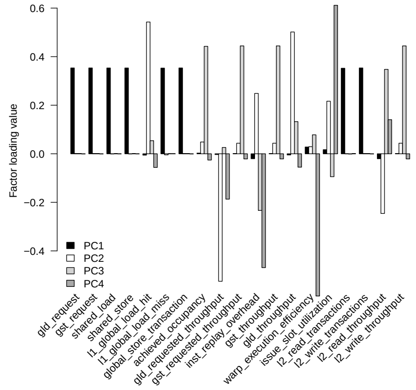
Figure 3: Counters affecting the performance of reduce2. Top three has changed due to optimizations, and features, in order :`l1_global_load_miss, l2_write_transactions, and l2_read_transactions.`



(a) variable importance

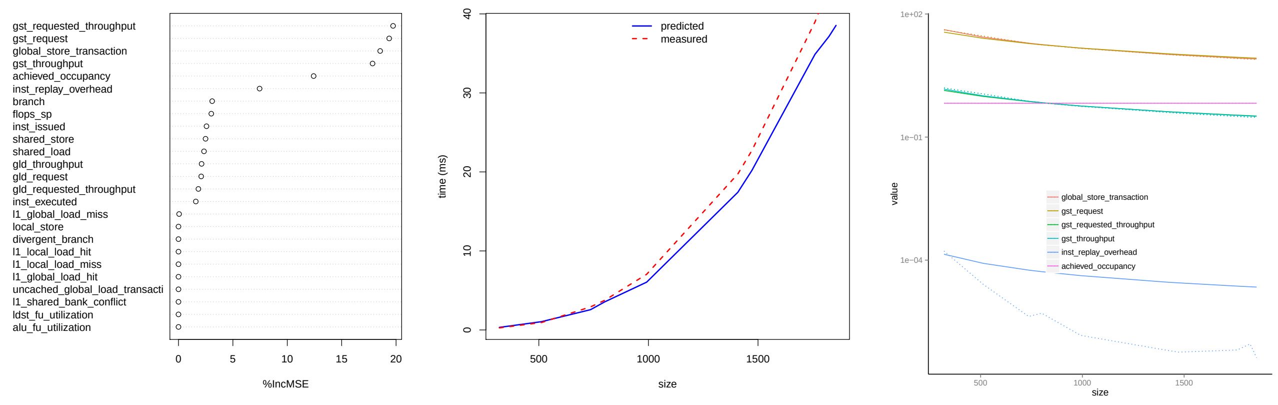


(b) Partial dependence



(c) Factor loadings

Figure 4: Counters affecting the performance of reduce6. Top three has changed due to optimizations, and features, in order :`gst_request, shared_store, and shared_load.`



(a) variable importance (b) predicting unseen matrix sizes (c) variable modeling

Figure 5: Characterization and prediction of MM.

ing hierarchical parallelism (at grid-level and TB-level) and shared memory to reduce global memory accesses and kernel launch overhead. For maximum occupancy, each TB only has 16 threads. This leads to idling of some threads in the warps. Also, as each thread must access values of its north-, northwest- and west-neighbors, this pattern may result in shared memory bank conflicts.

NW consists of 2 kernels used for the top left and bottom right traversals of the score matrix. We measure the contribution of each kernel in the overall execution time and use that to compute an average behavior for both kernels which we use for the analysis. We vary the sequence length from 64 to 8192 with a pitch of 64, generating 129 trials. We randomly split the data in a training set and a test set, with a 80:20 ratio.

As illustrated in figure 6(a), `achieved_occupancy` and `size` are the most influential variables for the time prediction, followed by a bunch of predictors of similar importance among which various memory throughput metrics. The lack of locality from the diagonal strip memory accesses leads to the presence of both `l1_global_load_miss` and `l1_shared_bank_conflict`. First, we show that a random forest built from the first 6 important variables has the same predictive power as one built with all variables, both having good accuracy. Models of those variables, shown in figure 6(c), are used to predict the execution times of unseen sequence lengths in figure 6(b). This time, the models are built using *earth*, an R [15] MARS implementation, with average R-squared of 0.99. The average MSE and explained variance of the random forest are around 0 and 99%, respectively. Which explains why the predictions are so accurate in figure 6(b).

6.2 Hardware scaling

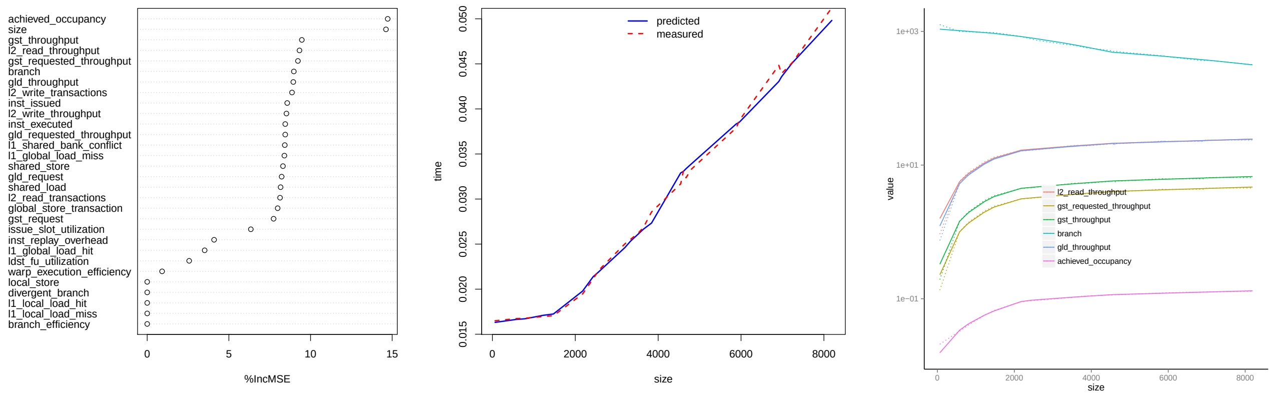
In this section, we characterize both the application and the training hardware, and we further use *both* to predict execution time on a different platform (of similar nature). We define ”sufficiently similar hardware” as hardware where the variable importance would be similar (e.g., same ranking). In this case, only a little calibration is necessary for the target GPU, and our method leads to correct predictions - i.e., a correct estimate of the performance behavior, and reasonable time predictions (due to the calibration). For example, when using two different cards with the same architecture (Fermi or Kepler), but different numbers of SMs, the prediction will be correct. However, for hardware that is not similar enough - *e.g.*, a Kepler GPU and a Fermi GPU, the prediction may become less accurate, as the variable importance features of the same kernel on these GPUs may be very different.

To demonstrate this issue, we use MM and NW as use cases, and the same GTX580 as training GPU. For prediction on different architecture, we use the NVIDIA Tesla K20m, which is based on the Kepler architecture. We use the same experimental data scheme as above, that is, by uniform random sampling of the runs into training and test sets accounting for 80% and 20%, respectively. We inject into this data values of machine characteristics, listed in Table 2, for different GPU architectures. The training set is used to train the random forest. We also run the benchmarks on the test GPU with the same set of application parameters, which are matrix size and sequence length, respectively. The resulting dataset to which we add values of machine parameters specific to the K20m is similarly cut into training and test sets and the test set is used to assess the random forest trained on the GTX580. So, our goal is to predict unknown problem sizes on *partially* known architecture.

Table 2: GPU hardware metrics

metric	**meaning**	**GTX480**	**K20m**
`wsched`	number of warp schedulers	2	4
`freq`	clock rate (GHz)	1.4	0.71
`smp`	number of MPs	15	13
`rco`	cores per MP	32	192
`mbw`	memory bandwidth (BG/s)	177.4	208
`l1c`	registers	63	255
`l2c`	L2 size (KB)	768	1280

The approach works straightforwardly on MM as shown in Figure 7 where we observe that the predictions mostly match the measured execution times, the inaccuracies at the edges coming from interpolation. From the calibration on the K20m, we notice that the most important variables are almost the same on both architectures, which guarantees the good accuracy of the predictions. Things are different for NW where the most important predictors are very different for the GTX580 and the K20m. Indeed, caching related variables such as `l2_read_transactions` and `l1_global_load_miss` are among the most influential predictors for the GTX580 as pictured in figure 8(a) whereas these same variables are less important (`l2_read_transactions`), or even totally unimportant (l1_global_load_miss) for K20m as visualized in Figure 8(b). The consequences of this issue is that we couldn't get decent predictions using the straightforward manner. The workaround is to use a mixture of important variables from both architectures to train the random forest. The predictions shown in Figure 8(c) are obtained with the following variables: `inst_issued`, `global_store_transaction`, `size`, `achieved_occupancy`, `issue_slot_utilization`, and



(a) variable importance (b) predicting unseen sequence lengths (c) model validation

Figure 6: Characterization and prediction of NW.

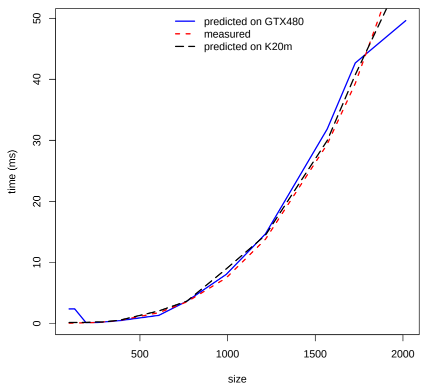


Figure 7: K20m predictions for MM from GTX580.

`gld_throughput`. Yet, predictions are less accurate, especially for small sequence lengths. Different memory throughput counters become dominant on the Kepler in comparison to the Fermi where caching performance metrics were important. Indeed, the bigger caches of the Kepler in both L1 and L2 make caching performance less pertinent to the kernel execution, at least for shorter sequences. Figure 8(c) shows that while the prediction accuracy is bad for sequence sizes up until around 3700, it slightly improves as the size increases.

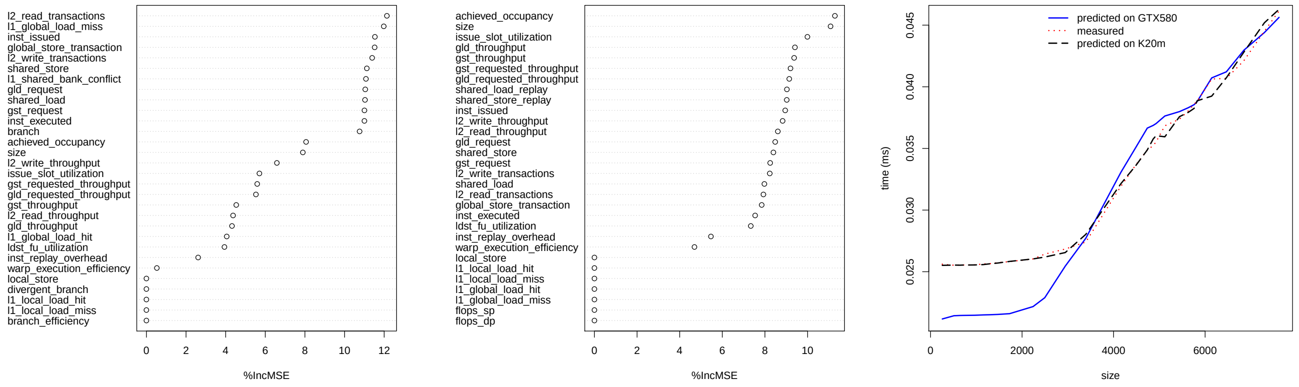
7. Discussion and conclusion

GPU performance analysis remains a challenging task, and tool support is mandatory for the large adoption of systematic performance engineering. This work provides empirical evidence that an easy-to-use tool can be developed, based on statistical modeling and hardware performance counters. Specifically, we demonstrate the successful use of random forest for performance analysis and prediction. Our experimental results illustrate both the performance bottleneck analysis and prediction potential of BF. Being a statistical approach, its major limitations are the applicability restricted to applications where enough training data is/can be available, and the black-box modeling (as no knowledge can be directly inferred from the model itself). Its overhead is as large as the size of the training set. Additional studies need to be made to determine the minimal training set, thus limiting the overhead to a minimum. We note, however, that, when compared with analytical modeling, where the burden of modeling and calibration are on the modeler's creativity, the training overhead of BlackForest is simply computational time - *i.e.*, automated collection of data over multiple runs.

Result interpretation and user feedback remain important aspects to be improved. For example, the additional interpretation tools provided, i.e., variable importance and partial dependence plots, are still perceived as "difficult" to understand for a performance engineer, and therefore we must continue working to improve user-feedback based on these data. Integrating confidence intervals into the partial dependence plots would help interpretation and confidence in the outcome, but their usage paradoxically requires statistical knowledge. We plan to experiment with first applying PCA onto the data to both remove correlated variables and reduce dimensionality, potentially uncovering hidden structure, thus leading to easy interpretation of random forest outcome. This would also help dealing with pathological cases, such as in Figure 6(a) where a large number of variables have similar importance making them equally eligible for inclusion into the subset used for further analysis. The modeling of that subset used by the approach may not meet quality requirement as observed with `inst_replay_overhead` in Figure 5(c). For automation purpose, these cases have to be dealt with appropriately.

Another challenge for our approach is that performance counters evolve, i.e., they may differ from one architecture to another. The consequences are relevant for hardware scalability, where important variables on one architecture may have different names or not exist at all on the architecture we want to predict for. An example of this issue is the absence of the Fermi metric `l1_shared_bank_conflict` on Kepler, which in turn, has `shared_load_replay` and `shared_store_replay` unknown to Fermi. We plan to tackle this problem by designing a "similarity" test to determine platforms that can be used for hardware scalability. When hardware scalability is not feasible, the only downside is that training must be repeated on the new target platform.

We also note that our method is not limited to predicting execution time - one could use other metrics of interest, such as power, as response variable. For instance, on the Kepler architecture, power draw can be directly read using the system management interface. Using BF, one can then both assess the power consumption behavior of the different functional units and of the application, and predict that for unseen problem sizes, or simply evaluate computing



(a) variable importance on GTX580 (b) variable importance on K20m (c) K20m predictions for NW

Figure 8: K20m predictions for NW from GTX580.

efficiency in terms of performance per watt. Finally, we believe our approach is very useful in the context of emerging CPU+GPUs heterogeneous systems, where performance modeling is key to determine workload partitioning [1, 4, 17]. As BF is equally applicable for all processing units in the platform, we can provide a unified modeling approach for heterogeneous platforms. We plan to empirically validate this assumption, by first proving BF's usability on CPUs, and further by automating the analysis.

8. Acknowledgments

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

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *CCPE*, 23(2), Feb. 2011.

[2] L. Breiman. Random forests. *Machine Learning*, 45(1), 2001.

[3] R. Dìaz-Uriarte and S. A. de Andrès. Gene selection and classification of microarray data using random forest. *BMC Bioinformatics*, 7:3, 2006.

[4] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A proposal for programming heterogeneous multi-core architectures. 21(2):173193, 2011.

[5] J. H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1), March 1991.

[6] T. J. Hastie, R. J. Tibshirani, and J. H. Friedman. *The elements of statistical learning : data mining, inference, and prediction*. Springer series in statistics. Springer, New York, 2009.

[7] W. Jia, K. Shaw, and M. Martonosi. Stargazer: Automated regression-based GPU design space exploration. In *ISPASS 2012*, pages 2–13, April 2012.

[8] A. Kerr, E. Anger, G. Hendry, and S. Yalamanchili. Eiger: A framework for the automated synthesis of statistical performance models. In *Proceedings of WPEA 2012*, 2012.

[9] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and 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, pages 249–258, New York, NY, USA, 2007. ACM.

[10] A. Liaw and M. Wiener. Classification and Regression by randomForest. *R News*, 2(3):18–22, 2002.

[11] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso. A Survey of Performance Modeling and Simulation Techniques for Accelerator-based Computing. *IEEE TPDS*, 2014.

[12] S. Madougou, A. Varbanescu, C. de Laat, and R. van Nieuwpoort. An empirical evaluation of GPGPU performance models. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014.

[13] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka. Statistical power modeling of gpu kernels using performance counters. In *Green Computing Conference, 2010 International*, pages 115–122, Aug 2010.

[14] S. Pllana, I. Brandic, and S. Benkner. Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art. In *CISIS'07*. IEEE Computer Society, 2007.

[15] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[16] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM TOCS*, 14(4), Nov. 1996.

[17] J. Shen, A. L. Varbanescu, H. Sips, M. Arntzen, and D. G. Simons. Glinda: A framework for accelerating imbalanced applications on heterogeneous platforms. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13. ACM, 2013.

[18] S. Song, C. Su, B. Rountree, and K. Cameron. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 673–686, May 2013.

[19] C. Strobl, A. laure Boulesteix, A. Zeileis, and T. Hothorn. Bias in random forest variable importance measures: illustrations, sources and a solution. bmc bioinformatics, accepted for publication, 2006.

[20] G. Wu, J. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. Gpgpu performance and power estimation using machine learning. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 564–576, Feb 2015.

[21] Y. Zhang, Y. Hu, B. Li, and L. Peng. Performance and power analysis of ATI GPU: A statistical approach. In *NAS'11*. IEEE Computer Society, 2011.