A practical performance model for compute and memory bound GPU kernels

Elias Konstantinidis
Department of Informatics and Telecommunications, University of Athens, Athens, Greece
e-mail: ekondis@di.uoa.gr

Yiannis Cotronis
Department of Informatics and Telecommunications, University of Athens, Athens, Greece
e-mail: cotronis@di.uoa.gr

Abstract—Performance prediction of GPU kernels is generally a tedious procedure with unpredictable results. In this paper, we provide a practical model for estimating performance of CUDA kernels on GPU hardware in an automated manner.

First, we propose the quadrant-split model, an alternative of the roofline visual performance model, which provides insight on the performance limiting factors of multiple devices with different compute-memory bandwidth ratios with respect to a particular kernel. We elaborate on the compute-memory bound characteristic of kernels. In addition, a micro-benchmark program was developed exposing the peak compute and memory transfer performance using variable operation intensity. Experimental results of executions on different GPUs are presented.

In the proposed performance prediction procedure, a set of kernel features is extracted through an automated profiling execution which records a set of significant kernel metrics. Additionally, a small set of device features for the target GPU is generated using micro-benchmarking and architecture specifications. In conjunction of kernel and device features we determine the performance limiting factor and we generate an estimation of the kernel's execution time. We performed experiments on DAXPY, DGEMM, FFT and stencil computation kernels using 4 GPUs and we showed an absolute error in predictions of 10.1% in the average case and 25.8% in the worst case.

Keywords-GPU kernels; performance prediction; performance model; micro-benchmarks;

I. INTRODUCTION

With the rise of programmability of GPUs for general purpose computation they were rapidly adopted by the scientific community mostly due to their high compute performance and energy efficiency. Compute environments like CUDA[17] and OpenCL[8] enabled their use as compute accelerators.

The primary goal in use of GPUs for computing is the significant improvement in performance they are able to achieve in problems which fit to their peculiarities. However, it is not always easy to predict the benefits of migrating to a GPU accelerator or moving from one type of GPU to another. On GPUs, memory throughput plays a major role in the procedure of “feeding” data to its compute resources. The roofline visual model [22] is capable to provide essential insight to the limiting factor of performance and determine a performance bound for the application. We propose an alternative visualization which we call quadrant-split performance model and it allows a better focus on a particular problem while inspecting the behaviour of multiple processors.

In addition, we conduct a quantitative prediction of performance on GPUs. We collect some features of our kernel through profiling on a CUDA enabled system. These kernel features constitute a minimal set of characteristics used to estimate the performance limiting factor and the effective compute performance bound for the particular kernel. The combination of these features with the GPU specification data are used to predict the expected performance of the kernel on another GPU without actually executing it on the real hardware. As such we are able to achieve more accurate results than using the pure visual model.

Furthermore, we developed a mixed micro-benchmark program, which allows inspecting the attained compute (GFLOPS) and memory (GB/sec) performance using a configurable operation intensity. As such, we executed this micro-benchmark on a wide range of operation intensity values. This allows to investigate the behaviour of various GPUs on a range of different operation intensity values.

The usefulness of our model is validated by applying it on a number of kernels for popular problems. These are the DAXPY vector operation, matrix multiplication (DGEMM), FFT and a set of stencil computation kernels (LMSOR[3]). In general our model provided an adequate prediction on the applied experiments with an average absolute error of 10.14% though in the worst case the absolute error reached to 25.8%.

The rest of this paper is structured as follows. In the next section the roofline model along with the quadrant-split model is described. In section 3 the applied method for performance prediction is presented along with the mixed micro-benchmark results. In section 4 the experimental results are presented and justified. Related work is referred in section 5 and the conclusions and future work follow in section 6.

II. THE QUADRANT-SPLIT VISUAL PERFORMANCE MODEL

In [22] authors present a visual model for identifying the critical performance factor of a program with regard to its operational intensity. The operational intensity is the compute to memory transfer ratio, typically measured in flops/byte. By exposing the limiting performance factor this model can lead to valid optimization decisions.

In general a GPU kernel can either be memory bound, compute bound or latency bound. It is memory bound when
the memory bus is congested by limiting the rate of execution of compute instructions. It is compute bound primarily when the ALUs (arithmetic logic units) of the processor are fully utilized and unable to provide additional throughput. In case the pipeline or memory latencies are the primary reason that limits performance the kernel is considered as latency bound.

The operational intensity is a characteristic that identifies the compute requirements of an application with regard to the DRAM traffic. Some known applications are illustrated with their operational intensities in the roofline model [22] (fig.1).

In the quadrant-split model we propose presenting the same information by using a slightly different data mapping in the respective graph. Instead of using the operational intensity on the horizontal axis we use the DRAM bandwidth of the device. On the vertical axis we keep using the compute performance unit (GFLOP). With this change we can describe every GPU (or CPU) with single point as specified by the peak compute and memory throughput specifications of the device.

The application is represented as a half-line with a slope determined by the compute to memory traffic requirements of the application. The same applications described on figure 1 as vertical lines are presented on figure 2 as half-lines all crossing the intersection of the axes with various slopes. Typical memory bound problems tend to have a small slope whereas compute bound problems have high slope. The half-line splits the quadrant into two half-quadrants where device points residing into the upper one have higher compute resources than memory traffic potential with respect to the application’s requirements whereas the device points residing on the lower one have lower compute potential. Simply put, the half-line is the visual bound for the distinction of the devices into two groups where the kernel is expected to behave as memory bound for the devices in the upper half-quadrant and as compute bound for the others instead. In this regard it is clear that a problem can be considered as memory bound for some devices and as compute bound for others. In other words the limiting factor is a relative term which is dependent on both the problem and the device specifications.

For instance, regarding the 3D-FFT problem as depicted on figure 2, the Intel Xeon and both NVidia Tesla GPUs reside on the upper half-quadrant which entails that the problem is memory bound with respect to these devices. In contrast, the GTX-480 GPU is compute bound as it resides on the lower half-quadrant. In order to determine the performance of a memory bound problem we mark a vertical line from the device point straight down to the problem half-line (shown as a dotted line). Similarly, in order to determine the performance of a compute bound problem we mark a horizontal line from the device point moving to the left till the application half-line is crossed. The intersection of these lines mark the anticipated performance on this device.

In the quadrant-split model more devices than one can be naturally applied on a single graph. Roofline model is device-centric as it is convenient for applying multiple problems on a single device whilst the quadrant-split model is application centric depicting better one application with many devices having different characteristics.

In this work we do not account for either latency bound or CPU-GPU transfer bound cases. We presume that the kernels under consideration are either compute or memory bound. In latency bound cases the kernel programmer should focus on eliminating the latency bottleneck. Additionally, all data are assumed to reside in the GPU memory so CPU-GPU transfer considerations are not applicable.

### III. Performance prediction

Apart from designing a visual model in this work we proceed to estimate the performance of GPU kernels following a quantitative approach. However, in order to get realistic results three adjustments had to be applied on the theoretical peak device performance which reduce them by a factor. The first one adjusts both compute and memory transfer peaks whereas the rest two adjust the compute peak under the particular kernel.
A. Measured compute and transfer peaks

The theoretical specifications set a good base of the performance that can be achieved on a device. However this might not always be adequately realistic. In some cases the measured performance is a fraction of the theoretical values. In order to estimate the practical peaks in both compute and memory transfer performance we developed some micro-benchmark kernels through which the real performance of the devices under investigation is evaluated.

The family of GPUs used for the experiments of this work comprises four NVidia GPUs (GTX-660, GTX-480, Tesla S2050 and Tesla K20c). The Teslas are professional GPUs targeted to GPU computing sector [13] and the GTX (GeForce) GPUs are consumer oriented cards primarily for gaming. The peak performance of these GPUs as measured is depicted on table I. It is evident that the compute performance exposed in the micro-benchmark is very close to the theoretical one provided by the manufacturer. The GTX-660 especially performs better than the theoretical performance in 64bit floating point operations. After further investigation with profiling metrics it was evident that the particular card boosted the GPU’s frequency when run pure double precision intensive code to about 1123MHz instead of the 1033MHz base clock frequency justifying our measurement. In contrast, the performance of memory bandwidth does not reach to the same point the theoretical peak. Especially, the Tesla GPUs perform bellow 70% of their theoretical peaks due to the enabled ECC protection[14]. The measured values were later used for correcting the GPU performance points in the model.

It should also be noted that the theoretical bandwidth given by vendors use the assumption that 1GB = 10^9 bytes which is inaccurate. We chose to use the 1GB = 1024^3 bytes convention which is more compliant with the multiples of byte. Thus all the rated values are given using the latter convention.

B. Floating point operation mix efficiency

Another factor that has to be taken into account is the floating point operation mix. GPU vendors tend to provide the peak performance achieved using multiply-add operations. These operations fuse a multiplication and an addition operation into a single instruction (a×b+c). These instructions are typically optimized to be executed in just one shader cycle (for single precision). The theoretical peaks provided by vendors assume a perfectly balanced stream of floating point multiplications and additions. If the stream of executed instructions is not perfectly balanced the performance drops. For instance, having a pure stream of addition instructions would reduce the floating point performance to a half as the addition instructions tend to be executed as fast as the multiply-add instructions though they perform just one operation instead of two.

In order to evaluate the operation mix we perform a profiling execution of the kernel with the NVidia profiler[18] and record the metrics flops_dp, flops_dp_fma. flops_dp is the total double precision floating point operations executed and the flops_dp_fma is the count of the multiply-add executed instructions. The efficiency of Flop mix is evaluated as follows:

\[ E_{fp\_mix} = \frac{flops_{dp}}{2 \times (flops_{dp} - flops_{dp\_fma})} \]

where \( E_{fp\_mix} \) is the efficiency of the floating point operation mix which ranges from 50% to 100% depending on the usage of multiply-add operations. The performance which should be taken into account should be adjusted as follows:

\[ P_{est(1)} = P_{measured} \times E_{fp\_mix} \]

where \( P_{measured} \) is the peak compute performance of the device as it was measured and the \( P_{est(1)} \) is the adjusted peak performance under the consideration of the floating point operation mix.

In this stage we also evaluate the operation intensity (\( I_{kernel} \)) of the kernel as described bellow:

\[ I_{kernel} = \frac{flops_{dp}}{32 \times dram\_trans} \]

where \( dram\_trans \) is the sum of the dram_read_transactions and dram_write_transactions, and it is used to express the kernel DRAM traffic requirements. It is worth noting that the total amount of memory transactions does not strictly reflect the actual memory access needs of a kernel. In kernels with poor irregular memory accesses each access of a thread warp causes multiple memory transactions which leads to an order of magnitude higher DRAM traffic.

In our experiments instead of using flops_dp in (1) and (3) as a metric for executed 64bit flops, we applied the expression inst_fp_64 + flops_dp_fma which seems to include more actual floating point operations than the regular ones (addition, multiplication, multiply-add). It is assumed that all instructions perform one floating point operation except the multiply-add which performs two.

C. Beneficial instruction mix efficiency

Another factor that further lowers peak floating point performance of a kernel is the density of floating point instructions in the executed instruction stream. The beneficial instructions on scientific problems are the floating point instructions which perform the actual computations as required by the algorithm. The rest of the instructions can be control flow, pointer arithmetic, etc. These operations consume valuable resources of the GPU and they stress the peak floating point performance to significantly lower levels. In order to take into account these operations we measure the inst_fp_32, inst_fp_64, inst_integer, inst_bit_convert, inst_control, inst_compute_ld_st, inst_misc, inst_inter_thread_communication metrics which accumulate all the instructions executed for various types. Two derived features are used, the density of 64bit float instructions (\( D_{fp64} \)) and the density of the load/store instructions (\( D_{ldst} \)) in the whole instruction stream executed. These are derived as follows:

\[ D_{fp64} = \frac{inst\_fp\_64}{inst\_total}, D_{ldst} = \frac{inst\_ld\_st}{inst\_total} \]
The total computational cost can be estimated by formula 8 and the relative efficiency of floating point instructions in the whole instruction mix is given by formula 9. The new adjusted peak performance can be estimated with formula 10.

\[ C_{\text{other}} = (1 - D_{\text{f64}} - D_{\text{ldst}}) \times 1 \]  
\[ C_{\text{total}} = C_{\text{other}} + C_{\text{f64}} + C_{\text{ldst}} \]  
\[ E_{\text{inst\_mix}} = \frac{C_{\text{f64}}}{C_{\text{total}}} \]  
\[ P_{\text{est}(2)} = P_{\text{est}(1)} \times E_{\text{inst\_mix}} \]  

**D. Mixed micro-benchmarks**

If a kernel is purely compute bound then compute performance reaches close to the theoretical peaks of the device as previous micro-benchmarks showed. However, in real applications the instruction mix consists of additional instructions like load/stores, integer arithmetic as address computations, controls instructions, etc. All these instructions reduce the performance peak in productive instructions (i.e. flops).

In order to experiment with performance in a mixed kernel which features both computation and memory traffic we developed a kernel containing artificial mixed operations with a configurable balance. As such we can investigate the behaviour of various GPUs in mixed instruction streams and the degree the compute and memory traffic peak values differentiate. We focused to keep all other instructions to a minimum in order to keep extra overhead as low as possible. Template variables have been used where possible including access strides, block size and the loops were unrolled[16]. A significant workload has been assigned to each thread in order to eliminate the impact of the initialization overhead.

In figure 3 the compute throughput and the memory bandwidth are shown as the operational intensity of the kernel is increased by moving from a pure memory bound to a pure compute bound kernel. The kernel was executed on the GTX-480 GPU. In fact, the GFLOPS line of the chart is an experimental graphical representation of the roofline graph model as it was generated by the micro-benchmark and the similarities with figure 1 are apparent.
The same results are illustrated in figure 4 which relates the compute throughput with the memory bandwidth. The generated graph is almost ideal as the lines forming are straight, persistently bounded by fixed limits. Essentially, it is the quadrant-split visual model chart in which the corner formed by the vertical edges in the middle represents the effective device performance point.

On the Tesla GPUs the same graph appears with more disturbances (figures 5 and 6) as it is not as flat as the previous one. This is partially justified by the different throughput ratio of double precision operations in relation to other instructions. As Tesla’s throughput of 64bit floating point instructions is close to the performance of integer throughput, the latter plays an important role in the additional overhead. In the consumer GPUs where the integer instructions are much faster than the 64bit floating point ones, their impact is not so important and the additional overhead is smaller. The fluctuations evident for the memory bound samples of the graph (more apparent on the K20c case) are caused by the fact that as the operational intensity of the particular kernel is increased by one step, the amount of memory access operations increase by either one read or one write operation in its main loop and write operations seem to allow higher bandwidth in this case.

IV. EXPERIMENTAL RESULTS

In order to prove the usefulness of the described adjustments we performed experiments with 6 kernels of different types. These were the vector operation DAXPY, the DGEMM (matrix multiplication), the FFT and 3 variations of the LMSOR stencil computation we had developed in a previous work [4], [3]. All experiments were conducted with double precision arithmetic. For the DGEMM and FFT problems the implementations of Nvidia libraries have been used (CUBLAS, CUFFT libraries). For the rest code custom kernels were used.

The DAXPY kernel is the addition of a scalar multiple of a vector with another vector. It is clearly a memory bound kernel on all recent architectures. The vector constituted of 48M double precision floats (384MB global memory per vector). We applied a 2048x2048 matrix multiplication in CUBLAS and a $32 \times 1024^3$ element vector in CUFFT. The matrix multiplication is clearly a compute bound problem even for GPUs. The CUBLAS and CUFFT libraries provide optimized kernels for each architecture. As a result a different kernel is chosen per GPU architecture. Therefore we used the appropriate kernel data for each applied GPU.

The LMSOR kernel is a red/black stencil computation with applied memory reordering by color optimization [10], [9].
In addition, 3 variations of the kernel had been developed on a previous work in which the recomputation strategy was applied [4], [3]. The motivation was the significantly increased computation capability of GPUs compared to their memory access potential, a trend which is expected to widen. The recomputation aims to reduce memory accesses at the expense of extra computations and thus affecting the \textit{flops per byte access} ratio. The variations are the following:

1) \textbf{Kernel #1 - No recomputations}
   7 read accesses per computed element

2) \textbf{Kernel #2 - Minor recomputations}
   5 read accesses per computed element (4 read accesses were replaced by 2 read accesses plus 4 recomputations)

3) \textbf{Kernel #3 - Aggressive recomputations}
   4 read accesses per computed element (an additional read access was replaced by one intensive recomputation)

We performed the stencil computation experiment on a \(3842 \times 3842\) mesh for a total of 26 iterations. All experiments were conducted on a 64bit Linux environment with CUDA versions ranging from 5.0 to 6.5 as multiple systems were used with different installations.

First we profiled all applications in order to extract the required metric values for the adjustments. These values were used to adjust the compute peak derived from micro-benchmarks to the peak we expect under the investigated problem. All results are depicted on table III.

In figure 7 we provide an illustration of the adjustment performed on the peak values of the Tesla S2050 under the LMSOR - kernel #3 problem. First the theoretical point is moved to left because of the peak measured effective bandwidth with micro-benchmarks (102.66 GB/sec). The vertical line corresponds to the compute throughput reduction as it is adjusted due to the operation mix efficiency and instruction mix efficiency considerations (72.22% and 72.91% giving overall 52.66%). Therefore, the peak compute throughput reaches to 268.1 GFLOPS. Eventually, from the last point we cross the application line by moving horizontally towards the vertical axis. Seemingly the first adjustment moved the point to the memory bound region by crossing the application line and the last adjustment moved it back to the compute bound region. Therefore the expected behaviour of the problem is compute bound. The "X" point represents the actual performance as measured on this GPU.

In figure 8 the performance estimation for the 4 GPUs is depicted for the FFT kernel after the adjustments were applied. The two Teslas seem to be memory bound where the GeForce GPUs are compute bound.

After performing the series of experimental executions we compared the real with the estimated executions times on all kernels. The results are depicted on figure 9. The memory limited DAXPY exhibits very good prediction with an average absolute difference 3.76%. The CUDA library kernels DGEMM and FFT also exhibit good predictions with absolute differences of 10.37% and 9.50%, respectively. LMSOR kernels #1 and #3 exhibit absolute differences 8.88% and 15.32% respectively. After disassembling the last kernel it was identified that a high number of binary shifts instructions were evident which are not as fast as integer addition instructions and thus the adjusted compute throughput is actually lower which is crucial for a compute limited kernel.

In some cases the predicted times are larger than the measured times. This could be caused by the fact that some instruction executions of different types could be actually overlapped. For instance, the Kepler based SMs consist of 4 schedulers each able to issue 2 instructions per cycle [17]. Thus it is possible to issue 6 instructions on the 6 ALU units and 2 on the 2 load/store units in one cycle. The same issue could be true for the Fermi GeForce GPUs when executing double precision floating point instructions, though this is not documented and needs to be further investigated. The perfection of figures 3 and 4 also supports this argument.

The DAXPY kernel was identified as memory bound and the DGEMM as compute bound on all GPUs as expected. The FFT kernel was identified as memory bound on the Tesla GPUs and as compute bound on GeForce GPUs instead. All results are depicted on table IV.
TABLE III
KERNEL FEATURES AS FORMED FOR THE INVESTIGATED KERNELS.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Total flops</th>
<th>Balance Flops/byte</th>
<th>Floating point operation mix efficiency (50%-100%)</th>
<th>Instruction percentages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>fp_64</td>
<td>load/store</td>
</tr>
<tr>
<td>DAXPY</td>
<td>100,663,296</td>
<td>0.083</td>
<td>100.00%</td>
<td></td>
</tr>
<tr>
<td>DGEMM (Fermi)</td>
<td>17,188,519,936</td>
<td>9.969</td>
<td>100.00%</td>
<td>65.58%</td>
</tr>
<tr>
<td>DGEMM (Kepler, GTX660)</td>
<td>17,188,519,936</td>
<td>4.822</td>
<td>100.00%</td>
<td>52.85%</td>
</tr>
<tr>
<td>DGEMM (Kepler, K20c)</td>
<td>17,184,129,024</td>
<td>22.703</td>
<td>99.98%</td>
<td>41.32%</td>
</tr>
<tr>
<td>FFT (Fermi)</td>
<td>2,283,798,528</td>
<td>1.038</td>
<td>65.37%</td>
<td>45.53%</td>
</tr>
<tr>
<td>FFT (Kepler, GTX660)</td>
<td>2,279,604,224</td>
<td>1.044</td>
<td>65.40%</td>
<td>38.56%</td>
</tr>
<tr>
<td>FFT (Kepler, K20c)</td>
<td>2,283,798,528</td>
<td>0.866</td>
<td>65.37%</td>
<td>10.15%</td>
</tr>
<tr>
<td>LMSOR (kernel #1)</td>
<td>2,976,335,232</td>
<td>0.238</td>
<td>73.75%</td>
<td>27.45%</td>
</tr>
<tr>
<td>LMSOR (kernel #2)</td>
<td>4,509,877,632</td>
<td>0.482</td>
<td>63.49%</td>
<td>35.98%</td>
</tr>
<tr>
<td>LMSOR (kernel #3)</td>
<td>33,892,699,776</td>
<td>4.193</td>
<td>72.22%</td>
<td>24.15%</td>
</tr>
</tbody>
</table>

In the overall the experiments exhibited an average difference of $\approx 10.14\%$ between the real and measured performance which can be considered as an adequate approximation. In the worst case however the performance of K20c was predicted with an absolute error of $\approx 25.8\%$. In order to make more accurate predictions we have to take into account other factors that complicate the analysis. These can be serialization factors as low occupancy or ILP (instruction level parallelism which is more important for Kepler architectures), shared memory bank conflicts, latency bound issues, saturation of atomic operations etc. For the sake of simplicity and the lack of more detailed profiling information we left other performance factors out of scope of this work.

V. RELATED WORK

As the GPUs were gaining the interest of the scientific community many researchers began to focus on building performance models for them [21], [6]. In [23] authors create a performance model based on low level GPU components, as the pipeline, shared memory and global memory data. They used the native GPU instruction set of the rather old now GeForce 200 architecture. On [7] Karami et al. present a regression model in which they predict the execution time of OpenCL kernels. In [5] Goswami et al. use statistical methods to characterize various CUDA kernels. They employ PCA (Principal Component Analysis) and cluster analysis. Baghsorkhi et al. used a type of analytical approach to perform performance prediction on GPU kernels and they validate their model with matrix multiplication and FFT kernels[1]. Sim et al. proposed a full framework for performing performance predictions and guiding the programmer to beneficial optimizations in order to improve performance[19]. Another

Fig. 9. Comparison of the estimated execution times with the actual measured execution times and the respective difference percentage (brackets imply a negative error percentage).

TABLE IV
THE DESIGNATION OF THE LIMITING FACTOR OF EACH KERNEL PER GPU AS DERIVED FROM THE MODEL.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>GTX660</th>
<th>GTX480</th>
<th>Tesla S2050</th>
<th>Tesla K20c</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAXPY</td>
<td>Memory</td>
<td>Memory</td>
<td>Memory</td>
<td>Memory</td>
</tr>
<tr>
<td>DGEMM</td>
<td>Compute</td>
<td>Compute</td>
<td>Compute</td>
<td>Memory</td>
</tr>
<tr>
<td>FFT</td>
<td>Compute</td>
<td>Compute</td>
<td>Memory</td>
<td>Memory</td>
</tr>
<tr>
<td>LMSOR (ker.#1)</td>
<td>Memory</td>
<td>Memory</td>
<td>Memory</td>
<td>Memory</td>
</tr>
<tr>
<td>LMSOR (ker.#2)</td>
<td>Compute</td>
<td>Compute</td>
<td>Memory</td>
<td>Memory</td>
</tr>
<tr>
<td>LMSOR (ker.#3)</td>
<td>Compute</td>
<td>Compute</td>
<td>Memory</td>
<td>Memory</td>
</tr>
</tbody>
</table>
performance prediction model was proposed by Kothapalli et al. [11]. They took into account of various special GPU characteristics and they experimented with matrix multiplication, link ranking and histogram generation. In [2] authors built the Grophecy++ framework which they use to predict speedups of GPU kernels with a particular focus on the CPU-GPU data transfer cost. A theoretical model for describing performance of GPUs was proposed by Ma[12]. This model is called Threaded Many-core Memory and it is regarded as an improvement of the PRAM model. Volkov and Demmel developed a variety of micro-benchmarks for GPUs in their work on developing dense linear algebra implementations for GPUs such as GEMM, SYRK and matrix factorizations[20]. These include benchmarking kernel launch overheads, CPU-GPU data transfers, GPU memory subsystem and pipeline latencies, memory bandwidth and compute throughput.

This work is focused on the performance prediction of developed GPU kernels on GPU architectures as an automated procedure. A knowledge of the internal design of the kernel is not a requirement as all essential parameter values are acquired by the profiling procedure.

VI. CONCLUSIONS AND FUTURE WORK

In this work we describe a practical performance prediction method for GPU kernels, in conjunction with the quadrant-split model as a visual representation. We employed data supplied by executing the subject kernel with profiling in order to capture a set of metrics through which a set of kernel features is constructed. These features are later on used to predict the kernel’s performance of other GPUs. Our model provides a practical method to predict the performance of a kernel on other GPUs without requiring any knowledge of the kernel design itself. The procedure can be easily automated and thus it can provide useful insight without significant effort. In our experiments we exhibited adequate results.

However, the exact performance is dependent on details of the instruction mix, pipeline latencies, the available parallelism and other serialization factors which are difficult to extract. Nevertheless, the exhibited measurements are significantly closer to the actual measurements due to the peak adjustments.

As future work we consider the inclusion of shared memory bank conflicts and other serialization factors e.g. reduced occupancy or atomic operations. Cache effects could also be considered which potentially differentiate the total amount of DRAM accesses between various types of GPUs. A future goal could also be the separation of the GPU performance characteristics from the applied kernel as in this work the FLOPS adjustment is dependent on the kernel’s features.

ACKNOWLEDGMENTS

We would like to acknowledge the kind permissions of the Innovative Computing Laboratory at the University of Tennessee and Dr. Gizopoulos Dimitris at the University of Athens to use their NVidia Tesla S2050 and NVidia Tesla K20c installations respectively for the purpose of this work.

This research was partially funded by the University of Athens Special Account of Research Grants no 10812.