A GPU performance estimation model based on micro-benchmarks and black-box kernel profiling

Elias Konstantinidis *

National and Kapodistrian University of Athens
Department of Informatics and Telecommunications
ekondis@di.uoa.gr

Abstract. Over the last decade GPUs have been established as compute accelerators. However, GPU performance is highly sensitive to many factors, e.g. memory access patterns, branch divergence, the degree of parallelism and potential latencies. Consequently, the execution time on GPUs is a difficult to predict measure. Unless the kernel is latency bound, a rough estimate of the execution time on a particular GPU could be provided by applying the roofline model. Though this approach is straightforward, it cannot not provide accurate prediction results. In this thesis, after validating the roofline principle on GPUs by employing a micro-benchmark, an analytical performance model is proposed. In particular, this improves on the roofline model following a quantitative approach and a completely automated GPU performance prediction technique is presented. In this respect, the proposed model utilizes micro-benchmarking and profiling in a "*black-box*" fashion as no inspection of source/binary code is required. It combines GPU and kernel parameters in order to characterize the performance limiting factor and to predict the execution time, by taking into account the efficiency of beneficial computational instructions. In addition, the "*quadrant-split*' visual representation is proposed, which captures the characteristics of multiple processors in relation to a particular kernel. The experimental evaluation combines test executions on stencil computations, matrix multiplication and a total of 28 kernels of the Rodinia benchmark suite. The observed absolute error in predictions was 27.66% in the average case. Special cases of mispredicted results were investigated and justified. Moreover, the aforementioned micro-benchmark was used as a subject for performance prediction and the exhibited results were very accurate. Furthermore, the performance model was also examined in a cross vendor configuration by applying the prediction method on the AMD HIP/ROCm programming environment. Prediction errors were comparable to CUDA experiments despite the significant architectural differences of different vendor GPUs.

Keywords: performance model, GPU, roofline model

* Dissertation Advisor: Yiannis Cotronis, Associate Professor

1 Dissertation Summary

1.1 Introduction

The main focus of GPU computing is about performance so it would be of great significance to be able to predict performance of GPU applications on a wide range of hardware. Performance modeling information is particularly important that can be exploited for either the consideration of a hardware upgrade or even on taking important optimization decisions. However, performance impact of migrating to a GPU accelerator or moving from one type of GPU to another can be a puzzling process to predict. Performance bottlenecks can be different due to architectural differences or variations on the balance of processor resources between different types of processors.

CPUs do not require a vast amount of parallelism in order to yield decent performance. They utilize large cache memory hierarchies that are able to alleviate the long access latencies of main memory. In addition, they employ advanced techniques in order to maximize the single threaded performance, e.g. aggressive speculative execution, register renaming, result value forwarding, etc. All these features potentially eliminate pipeline and memory bottlenecks, leading to more predictable execution results.

On the other hand, GPUs are significantly more performance sensitive to supplied parallelism, resource usage and memory access patterns. They are considered as massively parallel compute devices as they practically need thousands of active threads in order to keep them occupied. This fact poses large problems with abundant parallelism as a requirement. The GPUs feature much smaller cache memories which in conjunction with the large amount of active threads allows only limited use, mostly for exploiting the spatial locality between sibling threads. The miss of large cache hierarchies forces programmers to effectively use main memory. However, GPUs require regular memory accesses with specific requirements in order to apply coalescing, which is a mandatory requirement for efficient memory accessing. All reasons above induce potential bottlenecks for GPU performance. Practical experience has proven that GPU performance is sensitive to design decisions and fine tuning. In general, GPUs tend to be less tolerant to naive programming practices in regard to performance. Overall, though GPUs provide great compute performance, this can only be achieved on problems that match their characteristics.

For all the reasons above this thesis is focused on proposing a performance model that provides the necessary abstraction in order to be applicable on a wide range hardware, yet it provides decent prediction accuracy, is quick and straightforward to apply and can be fully automated based on *black-box* kernel inspection. In addition, this model was developed as roofline based and as such it is able to indicate an upper bound on performance, which can be fairly useful to the programmer as a guidance, providing performance feedback for further optimizations. The ultimate goal was to provide a tool that runs automatically the whole performance prediction process by utilizing an existing GPU program and producing the final results without user's intervention.

1.2 Related work

The roofline model [12] introduced by Williams and Patterson, is a visual model that provides insight on the maximum expected performance of a kernel by considering both pure computation and DRAM memory transfer requirements. It is based on the assumption that performance is either bound on the compute potential or the memory bandwidth of the underlying processor. The performance bound is either one depending on the relative requirements of operations of the application. Operational intensity is measured in *flop/byte* units and is used to determine the limiting performance factor on a particular processor. This can be applied by estimating the program's operational intensity which is determined by the program's requirements as formula (1) indicates:

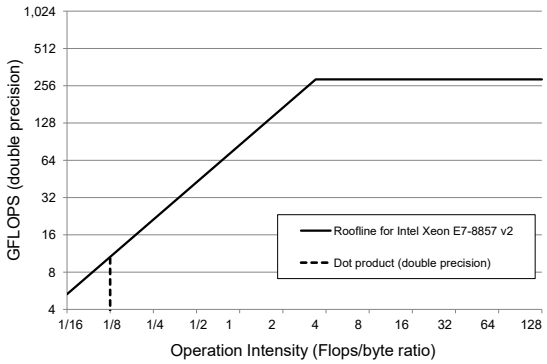

Fig. 1. The roofline visual model for Intel Xeon E7-8857 v2.

$$O_{kernel} = \frac{Operations_{(compute)}}{Traffic_{(memory)}} \tag{1}$$

The operational intensity is measured in *flop/byte* units and it is dependent on the application characteristics. Depending on the whether $O_{kernel} > \frac{Throughput_{dev}}{Bandwidth_{dev}}$ the kernel is considered as compute bound or memory bound. The graphical representation of the roofline model is able to provide a quick and insightful visual representation of the device theoretical peak performance. In figure 1 the solid line represents the theoretical peak performance of an Intel Intel Xeon E7-8857 v2 CPU depending on the program's operational intensity. In this example for program operational intensities up to 3.39 flop/byte the program is considered as memory bound. Compute bound programs must exhibit higher compute intensity.

1.3 Results

The proposed performance model, which is the primary contribution presented within this thesis, is an analytical GPU performance model based on the roofline

model [12]. An early foundation of the proposed model was presented in a preliminary stage as a regular conference paper [7] and subsequently it was extended and published as an elaborate work in the form of a journal article [11]. The first paper contribution [7] presented an initial form of the method along with a limited number of experimental results. The relevant journal publication [11] extended the method to a fully automated prediction process. The experimental results included executions on a wide range of different real world kernels and a micro-benchmark. The hardware used for the experiments included 4 consumer and 2 professional GPUs. Furthermore, the proposed model was extended to the experimental use on a cross-vendor GPU environment by employing an AMD GPU and the exhibited results were quite promising.

Other contributions that have been used in this thesis include an implementation of a red-black SOR stencil computation method [9, 10] which has been utilized in the experiments and it poses as a proof of concept case study in this thesis. The reordering by color strategy was the primary contribution of this published work. A theoretical performance analysis of the algorithm was provided and the implementations included various kernels, each utilizing a different memory caching approach. Additionally, a set of developed LMSOR stencil computations [4–6] were also developed which served to investigate the re-computation strategy as an optimization. In this respect various implementations were investigated characterized by different operational intensities due to the different degree of re-computation applied. Implementations of this work were also applied on the performance model in this thesis. Last, a set of micro-benchmarks [8] was presented that serves to the purpose of better understanding of the hardware capabilities regarding the GPU's fast on-chip memories. The micro-benchmarks assess the fast on-chip memories which include shared memory, L1 & L2 cache, texture cache and constant memory cache.

2 Results and Discussion

2.1 The *quadrant-split* visual representation

The roofline visual model is a valuable abstract representation of the compute device capability. As an alternative representation, the *quadrant-split* is proposed where in the horizontal axis the memory bandwidth is used instead of the operational intensity. In this respect, a device can be represented by a single point on the chart determined by its memory bandwidth and compute throughput peak rates. A program can be represented by a half-line crossing the intersection of the axes with a slope equal to its operational intensity. The half-line is the visual bound for the distinction of the area into two parts where the kernel is expected to behave as memory bound for the devices residing in the upper half-quadrant and as compute bound for the others instead. For instance, figure 2 represents the LBMHD problem with respect to 4 GPUs and a CPU. The dashed arrow lines point to the estimated roofline performance points for the each device on the particular problem.

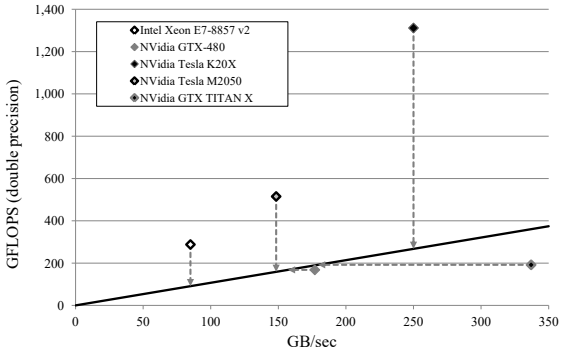

Fig. 2. The *quadrant-split* representation of the LBMHD problem using 5 CPU/GPUs.

2.2 The proposed performance prediction method

A profiling approach on a reference GPU is employed by extracting kernel execution information without requiring any internal knowledge of the kernel characteristics. The parameters used for the GPU device that is targeted for performance prediction are extracted by running a set of micro-benchmarks. The combination of both sets of parameters are employed for the performance prediction procedure. The whole process involves the steps described in figure 3.

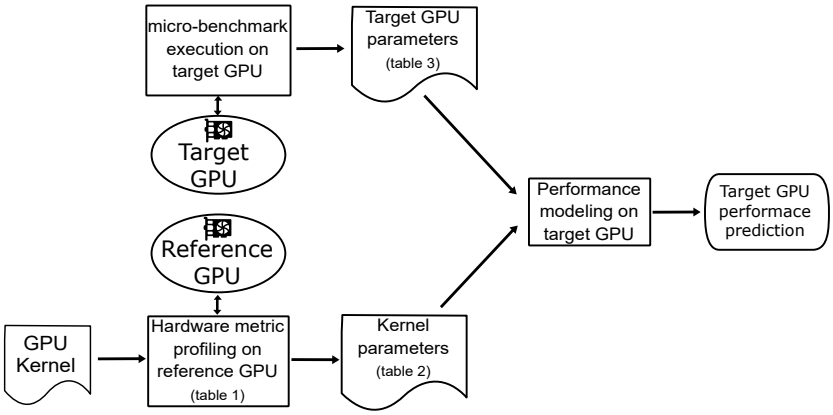

Fig. 3. The performance prediction methodology flow diagram.

In general, the approach for performance estimation of GPU kernels can be summarized in three aspects:

– Modeling compute and memory parameters of GPU kernels, largely independently of GPU architectural details, obtained by using a "*black box*" ap-

proach based exclusively on profiling measures (figure 3: "Hardware metric profiling on reference GPU")
- Modeling the GPU generic peak performance ratings on various operations, obtained by micro-benchmarking the target GPU (figure 3: "micro-benchmark execution on target GPU")
- Estimation of the target GPU performance (figure 3: "Performance modeling on target GPU") on the particular kernel according to:
  - the estimated maximum rate of executed compute operations on the target GPU for the particular kernel, and
  - the compute and memory demands of the given kernel (i.e. operational intensity) determining whether its performance is limited by the compute or memory throughput when executed on the target GPU

Kernel parameter extraction The required kernel parameters are extracted by profiling the execution of the subject kernel on a reference GPU. The list of the required kernel metrics is shown in table 1 and the provided notation will be used for reference.

Table 1. The NVidia profiler metrics required for the derivation of kernel parameters.

Metric	Notation	Description
flop_count_sp_fma	M_{fma32}	Number of single-precision floating-point multiply-accumulate operations executed by non-predicated threads
flop_count_dp_fma	M_{fma64}	Number of double-precision floating-point multiply-accumulate operations executed by non-predicated threads
inst_compute_ld_st	M_{ldst}	Number of compute load/store instructions executed by non-predicated threads
inst_executed	M_{inst}	The number of instructions executed
inst_fp_32	M_{fp32}	Number of single-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)
inst_fp_64	M_{fp64}	Number of double-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)
inst_integer	M_{int}	Number of integer instructions executed by non-predicated threads
dram_read_transactions	$M_{tran\text{-}r}$	Device memory read transactions
dram_write_transactions	$M_{tran\text{-}w}$	Device memory write transactions

The produced parameter set is provided in table 2. K_{type} parameter determines the type of beneficial operations within the kernel. It can be either *fp64*, *fp32* or *int*. A simple rule based approach in order to avoid user interaction is a function selecting *fp64* if the M_{fp64} metric is non zero, *fp32* if the M_{fp32} is non zero or *int* otherwise. The W_{comp} parameter represents the total beneficial compute operations performed by the kernel. It is evaluated by formula (2).

Table 2. The set of required *kernel parameters* in the proposed performance model.

Parameter	Description	Obtained
K_{type}	Dominant ops (fp64, fp32 or int)	rule based function
W_{comp}	Compute operations	formula (2)
W_{traf}	DRAM bytes accessed	formula (3)
E_{mix}	Operation mix efficiency (%)	formula (4)
D_{ops}	Operation instruction density (%)	formula (5)
D_{ldst}	Ld/St instruction density (%)	formula (6)
D_{other}	Other instruction density (%)	formula (7)

$$W_{comp} = \begin{cases} M_{fp32} + M_{fma32}, & \text{if } K_{type} = \text{fp32} \\ M_{fp64} + M_{fma64}, & \text{if } K_{type} = \text{fp64} \\ M_{int}, & \text{if } K_{type} = \text{int} \end{cases} \tag{2}$$

The parameter regarding the conducted memory traffic is the W_{traf} and it is estimated by using the DRAM transaction count metrics as shown in formula (3):

$$W_{traf} = 32 \times (M_{tran\text{-}r} + M_{tran\text{-}w}) \tag{3}$$

The efficiency of compute instructions E_{mix} is defined as shown in formula (4) which involves the type of compute instructions executed.

$$E_{mix} = \begin{cases} \frac{M_{fp32}+M_{fma32}}{2\times M_{fp32}} \times 100\%, & \text{if } K_{type} = \text{fp32} \\ \frac{M_{fp64}+M_{fma64}}{2\times M_{fp64}} \times 100\%, & \text{if } K_{type} = \text{fp64} \\ 50\%, & \text{if } K_{type} = \text{int} \end{cases} \tag{4}$$

Finally, the instructions executed are classified in 3 different types (compute, load/store and other instructions) and the individual density of each type in the instruction stream is determined by formulae (5), (6) and (7):

$$D_{ops} = \frac{I_{ops}}{I_{total}} \times 100\% \tag{5}$$

$$D_{ldst} = \frac{M_{ldst}}{I_{total}} \times 100\% \tag{6}$$

$$D_{other} = 100\% - D_{ops} - D_{ldst} \tag{7}$$

where I_{ops} and I_{total} are estimated by formulae (8) and (9):

$$I_{ops} = \begin{cases} M_{fp32}, & \text{if } K_{type} = \text{fp32} \\ M_{fp64}, & \text{if } K_{type} = \text{fp64} \\ M_{int}, & \text{if } K_{type} = \text{int} \end{cases} \tag{8}$$

$$I_{total} = 32 \times M_{inst} \tag{9}$$

Target GPU parameter extraction All required device parameters are collected by using micro-benchmarks and are shown in table 3. All floating point computation throughput parameters (T_{SP} and T_{DP}) concern MAD (Multiply-ADd) operations. The T_{xxx} parameters (T_{SP}, T_{DP}, T_{int}, T_{add}, T_{ldst}) regard the compute throughput of the device in various types of instructions and the B_{mem} parameter which reflects the effective memory bandwidth of the device.

Table 3. The set of *GPU parameters* used in the performance model.

Parameter	Description	Unit
T_{SP}	Single precision floating point operation throughput	GFLOPS
T_{DP}	Double precision floating point operation throughput	GFLOPS
T_{int}	Integer multiply-add operation throughput	GIOPS
T_{add}	Integer addition operation throughput	GIOPS
T_{ldst}	Load/Store instruction throughput on shared memory	GOPS
B_{mem}	Memory bandwidth	GB/sec

Kernel performance estimation In this model the throughput of various instruction types is considered for the efficiency estimation of instruction execution regarding beneficial computation. The purpose is to estimate the attainable peak throughput by considering the portion in which the pipeline is available for the execution of beneficial instructions. In this regard the instruction type densities (D_{ops}, D_{ldst}, D_{other}) should be considered in order to provide an estimation on the overall instruction execution throughput on the particular kernel.

The peak throughput on raw beneficial operations is selected in (10):

$$T_{op} = \begin{cases} T_{SP}, & \text{if } K_{type} = \text{fp32} \\ T_{DP}, & \text{if } K_{type} = \text{fp64} \\ T_{int}, & \text{if } K_{type} = \text{int} \end{cases} \tag{10}$$

For the estimation of the instruction execution efficiency the instruction densities along with the instruction throughput for various types are considered. The instruction types considered correspond to the throughput parameters of the GPU (table 3). The fastest instruction on the GPU typically is the single precision multiply-add instruction, and therefore it is the instruction that potentially is used to execute the most operations per second. So, the single precision multiply-add instructions are used as a point reference. The weight factor of executing a type of instruction is defined as the throughput ratio of fast single precision floating point instructions to the throughput of the particular type of instructions. Thus, weight factor is normalized by setting the weight of single precision instructions to 1. Therefore, the weight of all other instructions is typically greater or equal to 1. In this regard we define the weight factor operators as follows in formulae (11), (12), (13):

$$W_{op} = \frac{T_{SP}}{T_{op}} \quad (11) \qquad W_{ldst} = \frac{1/2 T_{SP}}{T_{ldst}} \quad (12) \qquad W_{other} = \frac{1/2 T_{SP}}{T_{add}} \quad (13)$$

In the estimation of W_{other} the throughput of integer addition is used. This is an arbitrary decision based on the assumption that the rest of the instructions apart from computation and load/store, is constituted mostly of simple integer instructions or instructions that execute roughly with the same cost. The ½ factor in (12) and (13) is applied in order to convert the operation throughput rate T_{SP} to instruction execution rate as each floating point MAD instruction is accounted as 2 operations. All beneficial operations are assumed to be executed using MAD instructions (two operations per instruction) whereas the load/store and integer addition operations are assumed to be implemented with single operation instructions. By taking into account the instruction densities and the respective weight factors the relative execution cost of each instruction type can be defined as shown in (14), (15), (16):

$$C_{op} = D_{ops} \times W_{op} \quad (14)$$
$$C_{ldst} = D_{ldst} \times W_{ldst} \quad (15)$$
$$C_{other} = D_{other} \times W_{other} \quad (16)$$

The estimated instruction efficiency can be estimated by formula (17):

$$E_{instr} = \frac{C_{op}}{C_{op} + C_{ldst} + C_{other}} \times 100\% \quad (17)$$

This cost modeling for the instruction execution assumes that all instructions are executed by the GPU multiprocessor on a single pipeline and therefore the execution of different types of instructions cannot be co-issued in a super-scalar fashion.

The adjusted throughput is estimated by applying both efficiency ratios each decreasing the theoretical instruction throughput by a factor. The adjusted throughput is given in (18):

$$T'_{op} = E_{mix} \times E_{instr} \times T_{op} \quad (18)$$

As such, the kernel's operational intensity is $O_{krn} = W_{comp}/W_{traf}$ and the device's adjusted operational intensity is $O_{dev} = T'_{op}/B_{mem}$. The comparison of the two intensities is used to determine whether the application is considered to behave as memory or compute bound. Thus, the estimated compute throughput is given by (19):

$$T_{predicted} = \begin{cases} T'_{op}, & \text{if } O_{krn} > O_{dev} \\ O_{krn} \times B_{mem}, & \text{if } O_{krn} \leq O_{dev} \end{cases} \quad (19)$$

2.3 Experimental evaluation

The executed experiments include two variants of stencil computations (red/black SOR & LMSOR) [9, 10], a matrix multiplication (SGEMM) kernel and a large subset of the Rodinia benchmark suite[3]. The experiments were applied on 6 different GPUs, characterized by 4 different architectures. The prediction procedure on red/black stencil computations, SGEMM and Rodinia benchmarks exhibited an average APE 3.42%, 15.18% and 28.97%, respectively. By summarizing all prediction results it is concluded that out of all conducted experiments more than half of them exhibited less than 25% APE (Absolute Percentage Error). This is considered a significant achievement given the small set of input that is used by the method.

In order to assess the performance prediction method in a cross vendor environment, the HIP/ROCm platform of AMD was chosen because of its CUDA kernel source code compatibility feature. By porting a CUDA implementation to HIP, the kernel source code effectively remains the same. This allows the profiling procedure to be performed on NVidia hardware by either using the CUDA application or the HIP application itself as HIP provides a compatibility layer for both hardware platforms.

The HIP programming environment was applied as supported by ROCm 1.4.0 release, on Ubuntu 14.04 Linux 64bit, using an AMD R9-Nano GPU. The kernel parameters were extracted on the GTX-480 and used for the performance prediction model on the AMD GPU. The required benchmarks were also ported to HIP platform and they were used to generate the R9-Nano GPU parameters.

The applied problems were the red/black SOR stencil computation, SGEMM and the lavaMD benchmark from the Rodinia suite (*lvmd-krn*). Running the performance model yields the execution times shown in table 4. It is evident that the observed prediction errors are very comparable to the ones produced on NVidia GPUs. Out of the 3 kernels, *lvmd-krn* exhibited slightly higher APE.

Table 4. Prediction results on the R9-Nano GPU for red/black SOR, SGEMM and lvmd-krn kernels

Benchmark	Predicted time (msecs)	Measured time (msecs)	Error (%)
red/black SOR	7.75	8.72	-11.18%
SGEMM	0.83	0.94	-11.45%
lvmd-krn	46.27	54.57	-15.21%

In general, it is expected to observe slightly higher APEs on the AMD platform due to the architectural differences between the two different vendor GPU architectures. These differences could slightly differentiate the extracted kernel parameters between the two architectures. However, this is not expected they change dramatically allowing the use of the performance prediction method in cross architecture environments, in the same way it was applied on this experiment.

3 Conclusions

This thesis presents an analytical performance model that derives from the roofline model [12]. Through a quantitative approach, the proposed model is able to provide timings that approximate actual execution measurements on real hardware. In addition, an alternative visual representation approach was presented, named *quadrant-split*, which is insightful in cases of multiple compute devices being represented along with a single application characterized by a particular operation intensity. The merit of the model's simplicity and its high abstraction characteristic allows providing results, final and intermediate, that can be easily interpreted by the final developer by being more human friendly. The small amount of required parameters pose the method as readily applicable.

One of the key points of the proposed method is the ability to extract the kernel's parameters by exploiting a mere set of profiling metrics as input parameters. This is captured through a kernel profiling procedure in a *black-box* fashion. Any internal knowledge of the kernel structure itself is not required by the developer. Furthermore, the proposed method can be developed as an automated tool which is executed without intervention from the developer. In this regard, the developer can apply the method on kernels and use it as a guidance tool without previous inspection the kernel design itself.

The proposed method achieves a better understanding of both compute and memory workloads compared to a pure theoretical peak approach primarily for two reasons. First, both the execution of non-essential and load/store instructions are considered by modeling their implications in the instruction pipeline and thus, their impact on the effective peak performance on beneficial instructions. Additionally, the type of mix of compute operations is also taken into account, i.e. the proportion of effective multiply-add operations in total amount of compute instructions. Second, the memory traffic requirements are measured by considering the actual traffic, thus any trivial locality and the degree of memory access coalescing are being indirectly accounted. The model provides an adjusted roofline on the peak performance based on these considerations.

The proposed performance model was tested and validated on a wide range of real world kernels. It was applied on stencil computations (red/black SOR and LMSOR), matrix multiplication and a wide range of Rodinia suite kernels. Furthermore, it was also tested for cross-vendor applicability on the HIP programming environment [2, 1] of the ROCm platform which is developed by AMD. The results were quite promising as they were similar to CUDA prediction in terms of absolute errors, despite the broader architectural differences between different vendor GPUs. The exact performance is dependent on special issues as the exact instruction mix, variability in cache behavior, pipeline latencies, the available parallelism and additional latencies which push performance to lower levels than the predicted ones, as the proposed model does not take into account these factors. Nevertheless, in these cases the performance prediction measurements serve as an upper bound performance and they indicate the potential room for improvement with further optimizations.

References

1. AMD. HIP. `https://github.com/GPUOpen-ProfessionalCompute-Tools/HIP`, 2016.
2. AMD. *HIP Data Sheet*, 2016. Rev. 1.7.
3. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.
4. Yiannis Cotronis, Elias Konstantinidis, Maria A. Louka, and Nikolaos M. Missirlis. *Parallel SOR for Solving the Convection Diffusion Equation Using GPUs with CUDA*, pages 575–586. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
5. Yiannis Cotronis, Elias Konstantinidis, Maria A. Louka, and Nikolaos M. Missirlis. A comparison of CPU and GPU implementations for solving the convection diffusion equation using the local modified SOR method. *Parallel Computing*, 40(7):173 – 185, 2014. 7th Workshop on Parallel Matrix Algorithms and Applications.
6. Yiannis Cotronis, Elias Konstantinidis, and Nikolaos M. Missirlis. *A GPU Implementation for Solving the Convection Diffusion Equation Using the Local Modified SOR Method*, pages 207–221. Springer International Publishing, Cham, 2014.
7. E. Konstantinidis and Y. Cotronis. A practical performance model for compute and memory bound GPU kernels. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 651–658, March 2015.
8. E. Konstantinidis and Y. Cotronis. A quantitative performance evaluation of fast on-chip memories of GPUs. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 448–455, Feb 2016.
9. Elias Konstantinidis and Yiannis Cotronis. Accelerating the red/black sor method using gpus with cuda. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics: 9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part I*, pages 589–598, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
10. Elias Konstantinidis and Yiannis Cotronis. Graphics processing unit acceleration of the red/black sor method. *Concurrency and Computation: Practice and Experience*, 25(8):1107–1120, 2013.
11. Elias Konstantinidis and Yiannis Cotronis. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *Journal of Parallel and Distributed Computing*, 107:37 – 56, 2017.
12. Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.