Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling

Abstract—Optimization, portability and development of GPGPU applications are not trivial tasks, since the capabilities and organization of GPU processing elements and memory subsystem greatly differ from the traditional CPU concepts, as well as among different GPU architectures. This work goes a step further in aiding this process by delivering a set of visual models that can be used by GPU programmers to analyze and improve application performance and energy-efficiency across a range of different GPU devices. For the first time in this paper, the state-of-the-art Cache-aware Roofline Modeling principles are applied for insightful modeling of GPU upper-bounds for performance, power consumption and energy-efficiency. The proposed models are developed by relying on extensive GPU micro-benchmarking aimed at fully exercising the capabilities of GPU functional units and memory hierarchy levels. The models are experimentally validated across 8 GPU devices from 3 different NVIDIA generations, and their benefits are explored when characterizing the behavior of 23 real-world applications from 5 different benchmark suites. Furthermore, the DVFS effects on GPU performance upper-bounds are also analyzed by scaling both core and memory frequencies.

1. Introduction

To satisfy current market demands, computer architectures need to sustain high energy-efficiency, while delivering high performance. This trend is evidenced in almost every domain of the electronics market, spanning from embedded systems (e.g., mobile platforms, smartphones and tablets) to high performance computing environments (such as desktop and server systems) [1], [2]. However, energy-efficiency and performance requirements typically lie in opposite sides of the optimization spectrum, thus simultaneously satisfying them is an open challenge for both industry and academia.

Since disruptive semi-conductor technologies are only expected to address these challenges in a couple of decades, modern computing systems rely on heterogeneity to pursue a balance between performance and energy-efficiency [5]. Current trends in heterogenous computing aim at merging the execution across several devices with different architectures, typically general-purpose CPUs (with only a dozen of cores on a single chip) and programmable accelerators (usually, with hundreds of specialized cores). Due to their intrinsic availability in almost every computing environment and potential to deliver very high performance gains, Graphics Processing Units (GPUs) have emerged as one of the most widely used programmable accelerators [6].

The major breakthrough in General-Purpose computing on Graphics Processing Units (GPGPU) came with the introduction of programming frameworks that ease the development of GPU applications, such as NVIDIA Compute Unified Device Architecture (CUDA) [7] and OpenCL [8]. However, simplification of the programming interface does not imply any guarantees on the efficiency of the developed GPU codes. In fact, fully exploiting the GPU capabilities for general-purpose applications is far from being a trivial task. Typically, the developers are required to hand-tune their codes in order to avoid main execution bottlenecks and further improve the code efficiency.

The process of optimizing the GPU applications is not only time consuming, but it also requires a very deep knowledge regarding the specific features of the GPU architecture for which the code is developed (especially, its realistically achievable upper-bounds). However, GPU architectures adhere to significantly different design and execution principles when compared to the traditional CPU microarchitectures. These differences do not only affect the disposition and capabilities of hundreds of processing elements and functional units, but also the overall organization of the memory subsystem. For example, besides the “traditional” organization that includes a set of private and shared cache levels and the main memory (i.e., the GPU device or global memory), GPUs also contain additional memory levels, such as shared (“scratchpad”), local and texture memory [7].

Besides these obstacles when optimizing the codes for a specific GPU, additional challenges arise when ensuring the efficient execution and code portability across different GPU devices/architectures. In contrast to CPU designs, GPUs are susceptible to significant architectural changes across different micro-architecture generations [7]. Hence, the increasing diversity of modern GPU architectures makes the GPU code optimization even more challenging and complex, especially when the optimization efforts are shifted from the hardware designers to the software developers.

In general, aiding the application optimization and development in accelerator-based ecosystems via performance modeling and simulation frameworks is still an open challenge [9]. Therefore, the main objective of this work is to deliver to the GPU programmers a set of simple, but complete, visual models that can be used to analyze and improve application performance and energy-efficiency across a range of different GPU devices. For this purpose, in this work, the insightful modeling of GPU architectures is deeply exploited based on the state-of-the-art Cache-aware Roofline Modeling principles, which, until this date, are only
applied for multi-core CPUs [3], [4]. Hence, for the first time, this type of modeling is applied herein to model the performance, power and energy-efficiency upper-bounds of many-core architectures (specifically, GPU devices), in order to help programmers when analyzing their applications and guide them in the optimization process.

The existing approaches for insightful GPU modeling are based on a fundamentally different concept, i.e., the Original Roofline Model (ORM) [10], which considers the data traffic at a single memory level (device memory) as the potential execution bottleneck [11]–[14]. In contrast, the herein proposed GPU Cache-Aware Roofline Model (CARM) allows visualizing the GPU architecture upper-bounds (in a single plot) by not only considering the peak throughput of computational units, but also the attainable bandwidth of different GPU memory levels, i.e., by explicitly considering GPU device memory, all cache levels and shared memory. Hence, in contrast to ORM-based approaches, the proposed GPU CARM observes the complete memory traffic from the perspective of the load/store units within the processing elements, thus it provides additional insights when characterizing the application behavior.

The proposed GPU CARMs for performance, power consumption and energy-efficiency are constructed and experimentally validated across a range of 8 GPU devices from 3 different NVIDIA architectures. In order to determine the GPU upper-bounds, an extensive experimental evaluation is conducted via a set of specifically developed micro-benchmarks, which are tailored to fully exercise the capabilities of different functional units and memory hierarchy levels. Furthermore, the insighfulness of the proposed GPU CARMs is extensively validated when characterizing the behavior of 23 real-world applications from 5 different standard benchmark suites. By applying a range of different core and device memory frequencies, the effects of Dynamic Voltage and Frequency Scaling (DVFS) on the GPU performance upper-bounds are also analyzed in the proposed models from both architecture and application perspectives.

2. Background: Cache-aware Roofline Model

The recently proposed Cache-Aware Roofline Model (CARM) provides an insightful approach for modeling the attainable micro-architecture upper bounds for performance, power, energy and energy-efficiency [3], [4]. From the micro-architecture perspective, it is based on the observation that the overall execution can be limited either by the processor compute capabilities or by the memory subsystem capabilities. For a single memory level, the CARM has two distinct modeling regions that intersect at a single point, i.e., the ridge point. The area on the left side of the ridge point marks the memory-bound region, while the compute-bound region is enclosed on the right side of the ridge point.

As presented in Fig. 1, CARM explicitly considers all memory hierarchy levels in a single plot, thus it includes several lines (slopes) that mark different memory-bound regions, each corresponding to a specific cache level or DRAM. The x-axis refers to the Arithmetic Intensity (AI), expressed as the ratio between compute and memory operations, i.e., \( \text{flops/bytes} \), where \( \text{bytes} \) reflect data traffic at the memory ports of the processor pipeline (as seen by the cores) [3], [4]. These are notable distinctions over the fundamentally different “classic” ORM-based approaches [10], [11], which focus only on a single memory level, thus the x-axis in ORMs refers to the Operational Intensity (OI) (e.g., \( \text{flops/DRAMbytes} \), where \( \text{DRAMbytes} \) is the amount of data traffic between the last level cache and DRAM).

The CARMs are typically used to simplify detection of main application bottlenecks (characterization) and to provide optimization guidelines. For this purpose, the application is typically plotted with a single point in the respective CARM (see points 1–6 in Fig. 1), from where one can derive: i) if the application is memory- or compute-bound – by observing the relative position of the application AI (on the x-axis) in respect to the ridge point; and ii) how far is the application from fully exploiting the architecture capabilities – by observing the point position (on y-axis) in respect to the modeled maximum (rooflines). Due to CARMs single-plot nature, easiness to assess the application AI (both experimentally and theoretically), and the possibility to provide consistent and more intuitive application characterization when compared to ORMs, the performance CARM is currently being integrated in the Intel Advisor (the main application development framework for Intel microarchitectures) [15].

As it can be observed in Fig. 1 for a quad-core Intel 3770K processor, the CARMs for different modeling domains maintain the same x-axis, i.e., AI in \( \text{flops/bytes} \). However, the y-axis differs across modeling domains. The performance CARM considers the peak Floating Point (FP) performance of computational units in \( \text{flops/s} \) and the realistically achievable bandwidth of different memory levels.
to the core (e.g., $L2 \rightarrow C$) [3], while the power and energy-efficiency CARMs reflect the impact of accessing different memory levels and exercising different functional units to the micro-architecture upper-bounds in watts ($W$) and $\text{flops}$ per unit of energy ($\text{flops/joule}$), respectively [4].

### 2.1. GPU Roofline Modeling

The existing approaches only rely on the ORM principles for GPU Roofline modeling. In [12], the performance ORM was explored for a range of GPU devices and different optimization roofs were defined by only focusing on the device (global) memory. The ORM principles were also applied for GPU power consumption and energy-efficiency modeling in [11], while simulation-based DVFS effects were analyzed in [14]. In [13], an application-centric approach was proposed, which relies in ORM to determine the most suitable GPU hardware to satisfy the demands of a specific application. ORM was also used for GPU application characterization and optimization in [16]–[18].

As it can be observed, the GPU Roofline modeling is an important and yet to be fully explored topic. In contrast to the state-of-the-art ORM-based approaches, in this work, for the first time, the CARM principles are applied for insightful modeling of GPU architectures for performance, power consumption and energy-efficiency. Furthermore, an in-depth experimental evaluation is conducted for a set of GPU devices from different architectures, which considers different compute units and several memory levels. This is also one of the first works that provides an extensive characterization of a range of real applications from standard benchmark suites and it also thoroughly analyzes the DVFS effects when scaling both GPU core and memory frequency.

### 3. Reaching the GPU upper-bounds via micro-benchmarking

As previously referred, the process of optimizing general-purpose applications for GPU devices is not an easy task, especially when considering the significant differences among GPU architecture generations. Programming frameworks and APIs, such as NVIDIA CUDA [7], include model abstractions of the architectures that facilitate program development and porting between devices, but without providing guarantees regarding the code efficiency.

In particular, the CUDA programming model relies on thread and memory hierarchies, which map the way how the GPU architecture is organized. The first defines a multi-dimensional structure of kernels, thread blocks and threads, while the latter provides a set of memory spaces with different scopes. Different memory spaces are usually optimized for different purposes (e.g., different addressing modes), and consist of a global memory (visible to all threads), a shared memory (visible within a thread block), a private local memory (per thread), and two read-only memory spaces (constant and texture) [7].

Overall, NVIDIA GPU architectures are constituted by multiple Streaming Multiprocessor (SM) (that may be grouped into clusters) and a Global Scheduler (which assigns thread blocks to the SMs). The total pool of threads is organized in thread blocks, and within a thread block the threads are issued in groups called warps, where each thread executes the same instruction (also known as SIMT) [7]. Nevertheless, the exact details of the SM size, how they are interconnected with memory, the memory organization itself (e.g., L1 and L2 implementations), and other aspects of the microarchitecture can substantially vary between different GPU generations, thus they have an important impact in performance and in the programming approach thereof.

The analysis presented in this paper relies on four different generations of NVIDIA GPU architectures: Fermi, Kepler, Maxwell and Pascal\(^1\). In order to analyze and evaluate the different microarchitectural aspects of these GPUs

### Table 1. Specifications of the analyzed GPUs

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Fermi</th>
<th>Kepler</th>
<th>Maxwell</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>G80</td>
<td>K20</td>
<td>K40</td>
<td>K80</td>
</tr>
<tr>
<td>Cores per SM</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Memory Frequency (MHz)</td>
<td>1058</td>
<td>750</td>
<td>750</td>
<td>750</td>
</tr>
<tr>
<td>SP/DP units per SM</td>
<td>192/192</td>
<td>192/192</td>
<td>192/192</td>
<td>192/192</td>
</tr>
<tr>
<td>FP Performance (FMA)</td>
<td>1607</td>
<td>1536</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/s)</td>
<td>4729.75</td>
<td>2843.90</td>
<td>3072</td>
<td>3072</td>
</tr>
<tr>
<td>Max Core Frequency (MHz)</td>
<td>1280</td>
<td>1280</td>
<td>1280</td>
<td>1280</td>
</tr>
</tbody>
</table>

---

1. Since a Pascal GPU was not available for benchmarking, the results for this architecture are not provided. Nevertheless, due to the high similarity of Maxwell and Pascal architectures, the presented experimental methodology is fully applicable to Pascal GPUs, where a better energy-efficiency, higher FP performance and higher device memory bandwidth are expected.
and to extract the necessary metrics to construct CARM, we rely on micro-benchmarks specifically designed to reach the performance upper bounds for FP operations and memory bandwidth at different levels (shared memory, L1 cache, L2 cache and device memory). The micro-benchmarks are also tailored, such that specific microarchitectural features of different GPU generations are explicitly taken into account.

Table 1 presents the specifications of GPU devices analyzed and evaluated in this paper. To conduct highly accurate experimental evaluation, the micro-benchmarks are enhanced with the CUPTI toolkit, which allows accessing built-in microarchitecture counters and obtain very precise metrics. As shown in Table 1, the main differences between GPU architectures reside in the number and size of SMs (that vary even across devices with the same chip model) and their organization. The SMs typically include an instruction cache, warp scheduler, instruction dispatch unit, CUDA cores, Special Function Units (SFUs), Load/Store Units (LSUs), register file, shared memory and L1 cache [7]. The Global Scheduler is responsible for assigning thread blocks to SMs, which execution is steered by the warp scheduler. This unit fetches a thread warp (32 threads), decodes the instruction and verifies the availability of the operands and operation units. If the conditions are favorable, the dispatch unit issues a warp to the requested unit.

The microarchitectures listed in Table 1 embody significant differences. For example, while on Fermi there are two schedulers and one dispatch unit per scheduler that supports issuing of two distinct warp instructions per clock, Kepler and Maxwell have two dispatch units per scheduler and four schedulers per SM, which enables issuing eight independent warp instructions from four distinct warps simultaneously. A significant change in the SM organization can also be observed between Kepler and Maxwell/Pascal architectures, i.e., from 192 cores per SM in Kepler to 128 in Maxwell and Pascal. Other major differences lie in the number and latency of SP/DP units, and several aspects of the memory hierarchy, such as the number of transaction buses connecting the SMs to the higher memory levels (thus changing the available bandwidth) and the type of L1 cache memory implementation, as discussed below.

**Execution Units:** The dispatched instructions are issued, according to their type, to the CUDA cores, Double Precision (DP) FP units, SFUs and LSUs. CUDA cores have two available pipeline lanes: the FP Unit (FPU) and the Arithmetic and Logic Units (ALUs). The first delivers one Single Precision (SP) Fused Multiply-Add (FMA) per clock across all architectures (i.e., 2 fops/clk). The latter can execute a set of integer and logical operations. All GPUs, apart from GK110, provide limited DP performance due to the reduced number of DP units per SM. GK110 GPUs have a specific set of DP units that allow executing DP FMA in one clock cycle. Finally, SFU is used for transcendental and other special arithmetic operations like sine, cosine, square root and exponential, while the purpose of LSUs is to steer the memory requests and decouple the execution pipeline from the memory pipeline. Data requests can be coalesced and are redirected from/to global, local or shared memory.

As shown in Fig. 2(B), a specific micro-benchmark was developed in order to reach the peak FP performance across different GPUs, which favors the highest throughput operations, i.e., SP/DP FMA. To maximize the code efficiency and pipeline utilization, all FP operations use architectural registers and a minimum number of dependencies between instructions is guaranteed to reduce the required memory transactions and pipeline hazards. Nevertheless, for compilation purposes, the registers are initialized from the shared memory. The results presented in Fig. 2(A) for the SP and DP FP benchmarks rely on an array of N = 32768 and ≈32768 thread blocks of 1024 threads to guarantee high level of pipeline utilization. The number of thread blocks (32768) is rounded to a multiple of available SMs in the specific device. As can be seen in Fig. 2(A), a near theoretical peak FP performance was achieved (on average 95.3% across all devices). In general, lower SP FP performance was achieved on Kepler devices, which may be attributed to the misbalance between the number of SP units per SM (192) and the number of available warp schedulers (4), thus creating a potential bottleneck at the issue stage. For DP FP performance, the small number of units remains the main bottleneck, although an average utilization higher than 99% was experimentally attained across all tested devices.

**Shared Memory/L1 cache:** Shared memory is a low-latency and very high bandwidth “scratchpad” memory block used for efficient data sharing between threads within the same thread block, thus it helps reducing the off-chip traffic. Across all GPU architectures, it is composed by 32 banks, where each thread from the same warp should access different banks to achieve the highest efficiency. However, the number of bytes accessed per memory transaction differs
among microarchitectures. For example, in Fermi, it is only possible to access 64 bytes per clock cycle (4 bytes per bank), while Kepler provides two different access modes: 32-bit (4 bytes) and 64-bit (8 bytes), thus sustaining a bandwidth of 128 bytes (32-bit mode) or 256 bytes (64-bit mode) per clock cycle. Finally, Maxwell and Pascal allow accessing 128 bytes per clock cycle. The total GPU shared memory throughput for each device is presented in Table 1.

As shown in Fig. 2(C), a specific micro-benchmark was developed in order to reach the maximum throughput, where each thread in a warp performs two shared memory transactions (load and store) to consecutive memory addresses, thus enforcing accesses to different shared-memory banks and avoiding any conflicts. The results, using a data set with $N = 32768$ and $\approx 32768$ thread blocks with 1024 threads, are presented in Fig. 2(D), where for both SP and DP tests a utilization higher than 98% was achieved across all GPUs.

The L1 cache keeps the data from individual threads, either from global or local memory accesses (register spilling). Although cacheability of global memory accesses in L1 can be enabled for some devices at compilation time (K40 and GTX 980), this property has been discontinued in recent architectures. In Fermi and Kepler, L1 and shared memory are implemented in the same (polymorphic) memory block, thus the L1 bandwidth is expected to match the one for register spillings. As shown in Fig. 3, the experimental results reveal a round-robin distribution of thread blocks to the SMs. Subsequently, the SMs from the same cluster access the same L2 cache banks (both slices), while Kepler provides two different access modes: 32-bit (4 bytes) and 64-bit (8 bytes), thus sustaining a bandwidth of 128 bytes (32-bit mode) or 256 bytes (128-bit mode) per clock cycle. Finally, Maxwell and Pascal allow accessing 128 bytes per clock cycle. The total GPU shared memory throughput for each device is presented in Table 1.

As a first step, a specific micro-benchmark was developed to uncover how the thread blocks are assigned to the SMs and clusters (group of four successive SMs). The micro-benchmark iteratively launches an incremental number of thread blocks, and the information regarding the banks that were used to service the memory requests was retrieved with CUPTI. To simplify this analysis, Fig. 3 presents the results obtained with thread blocks of 512 threads, each performing 512K SP loads from the global memory. It is worth to emphasize that the same access pattern is obtained for DP operations, as well as when mixing the load and store operations and when changing the number of threads assigned to each thread block (i.e., 256 or 1024). As it can be seen in Fig. 3, the experimental results reveal a round-robin distribution of thread blocks to the SMs. Subsequently, the L2 cache accesses were tested by performing a set of benchmarks with different access patterns, referred as “Standard”, “Consecutive” and “Clustered” in Fig. 3.

Figure 3(A) presents the results obtained with the “Standard” data access pattern, where each thread accesses subsequent memory locations (based on the thread ID). As it can be observed, the SMs from the same cluster access two sets of L2 cache banks, e.g., blocks from cluster 0 (0, 4, 8, 12) access banks 6/7 and 2/3. This access pattern achieves a throughput of 13.51 transactions per clock cycle, i.e., 471.08 GB/s. The “Consecutive” pattern aims at forcing that different groups of four consecutive blocks fetch the data from the same L2 bank, with the objective to enable data prefetching for consecutive blocks. As shown in Fig. 3(B), the “Consecutive” pattern still shows accesses

2. For devices with enabled L1 cacheability by default, the compiler flag

$-Xptxas -dlcm=cg$ was used.
to two different sets of L2 banks (e.g., 6/7 and 2/3 for the first four blocks), although it provided improvements in the achievable throughput over the “Standard” solution by achieving 13.88 transactions/clk (i.e., 483.98 GB/s). Finally, the “Clustered” pattern was examined, with the aim at forcing the blocks that run in the same cluster to access a fixed set of L2 banks, i.e., cluster 0 accesses banks 6/7, cluster 1 accesses banks 4/5 and so on. As it can be seen in Fig. 3(C), the “Clustered” pattern yields the expected results and provides even further throughput improvements by delivering 14.04 transaction/clock (489.57 GB/s), i.e., an improvement of 3.9% over the “Standard” pattern.

This evaluation methodology was adopted for different GPU devices/architectures in order to assess their attainable L2 throughput. Figure 4(A) reports the obtained experimental results for SP and DP accesses by relying on the micro-benchmarks presented in Fig. 4(B). The reported throughput is obtained for a mix of load/store operations, which results in a significantly higher throughput than the one obtained with the read-only benchmark (possibly indicating a two L2 cache). As it can be observed in Fig. 4(A), the L2 throughput has been gradually increasing across GPU micro-architectures, and it scales with the number of SMs.

**Device Memory:** The above-mentioned global, local, constant and texture memory spaces map into the largest memory in the GPU, i.e., the device memory. All the GPUs listed in Table 1 implement a GDDR5 DRAM, with different bus width and memory clock. To reach the maximum bandwidth between the device memory and the cores, it must be ensured that all data requests are forwarded to this memory level. Thereby, as presented in Fig. 4(C), a specific microbenchmark was developed, where each thread accesses the device memory with a stride that guarantees: 1) misses in all above memory levels (e.g., caches), and 2) that memory accesses from consecutive threads are coalesced [7].

The results, as shown in Fig. 4(D) for SP and DP, were obtained with the described micro-benchmark, constituted by 32K thread blocks (rounded to the nearest multiple of SMs) with 1024 threads, each performing \( N=32768 \) iterations, and a “stride” that varies between 32K/64K across different architectures. The reported percentage is relative to the peak theoretical device memory bandwidth, which represents the bandwidth between the device memory and the L2 cache, typically used in the ORM-based approaches [10]–[12]. In contrast, the experimentally obtained device memory bandwidth, as presented in Fig. 4(D) and used in CARM, represents the realistically attainable bandwidth between the device memory and the cores. This bandwidth is lower than the theoretical peak bandwidth, due to several limiting factors in the pipeline and it also includes the time to traverse the higher memory levels (i.e., caches).

**4. GPU CARM: Construction and Validation**

By adhering to the CARM construction methodology [3], [4], the experimentally obtained micro-architecture upper-bounds from Section 3 were relied upon when building the herein proposed performance, power and energy-efficiency GPU CARMs. For all modeling domains, different CARM performance roofs, power hills and energy-efficiency slopes were obtained for different memory hierarchy levels. For example, in the performance GPU CARM, there are several slopes corresponding to the bandwidth upper-bounds of different memory levels that intersect the peak FP performance at different ridge-points.

Besides constructing the proposed GPU CARMs, it is also necessary to experimentally validate them on the GPUs. Therefore, an extensive set of experiments was performed based on custom micro-benchmarks, which interleave a different number of FP operations and specific memory operations in order to attain the desired AI. Similarly to the experiments described in Section 3, the results presented herein were also collected by relying on the CUPITI toolkit.

It is worth to emphasize that the complete validation of all proposed GPU CARMs (i.e., for performance, power and energy-efficiency) can only be achieved on four GPUs (Tesla K20, Tesla K40, Titan X and GTX 980), since these are the only ones providing built-in power sensors. Due to the limited space, we mainly focus on presenting the results for the Titan X, for which an extensive characterization of real benchmarks and DVFS analysis were also performed.

Snippets of the source code used to test each level of the memory hierarchy are presented in Fig. 5(B). As it can be observed, within two loops the FP fma operations are mixed with different ld/st memory operations, being the overall AI of the kernel defined by the AI of each loop iteration. For example, with eight FP operations (four FMA) and two memory operations of 4 bytes (SP), we target \( AI=1 \). Although the codes in Fig. 5(B) represent the baseline versions, other variations were also used, which
exercise a variable number of FP operations and registers. Hence, by manipulating the FP and memory operation mixes to achieve different AIs, it was possible to obtain a range of performance and power samples across three different levels of the memory hierarchy as presented herein, namely shared memory, L2 cache and device memory.

As shown in Fig. 5(A), for the performance GPU CARM, the experimentally obtained samples allowed achieving the peak attainable performance for each memory level. However, as the AI moves closer to the ridge point the obtained values slightly diverge from the modeled performance. This is due to the challenge (impracticability) of creating a mixture of instructions that exactly matches the ridge-point AI. Although shared memory line is presented together with L2 cache and device memory, it should not be interpreted as the \( L_1 \rightarrow C \) cache line in the CPU CARM [3], since the GPU shared memory is a “scratchpad” memory and not part of the cache memory hierarchy.

When considering the GPU power CARM, as presented in Fig. 5(C), the power curves were obtained by relying on the equation presented in [4], where the constant power was selected for the most accurate fit. The GPU power CARM presents similar properties as the CPU power CARM [4]: i) the “hill-tops” coincide with the respective memory level ridge-points and result from the partial superposition of the power consumption from executing FP and memory operations; ii) the power lines for each memory level meet as \( A_1 \rightarrow +\infty \), i.e., the FP power value; iii) when \( A_1 \rightarrow 0 \), the lines tend to the respective memory power value; and iv) the device memory has lower peak power consumption than the L2 cache, due to the fact that the L2 cache is “idle” waiting for the requests to be serviced by the device memory.

By combining the performance and power models it is possible to build the energy-efficiency GPU CARM, as presented in Fig. 6 for the Titan X and K20 devices. In this model, as the AI increases, the energy-efficiency lines asymptotically converge towards the maximum energy-efficiency of the GPU architecture, which is higher for Titan X than for K20. In particular, Titan X is capable of providing an additional 12.2 GFlops per unit of energy relative to K20. Finally, the vertical lines in Fig. 6 represent the minimum AIs to achieve the 99% of peak energy-efficiency for each memory level, thus defining the high-efficiency regions of the architecture. As it can be seen, for both GPUs, the shared memory requires a lower AI to reach the high efficiency region, while the device memory needs the highest AI. When comparing Titan X and K20, it can be observed that the high energy-efficiency region for the shared memory starts at approximately the same AI. Therefore, for an application bound by the shared memory accesses and an application optimized for energy-efficiency in Titan X (i.e., an application bound by the shared memory accesses and with an AI higher than the 99% mark) an high energy-efficiency in K20 may be implied. However, for L2 and device memory, Titan X reaches the respective high energy-efficiency regions with lower AIs than K20. Therefore, for the L2/device memory bound applications optimized for energy-efficiency on Titan X, the high energy-efficiency may not be implied in K20 (see dots in Fig. 6).

5. Characterization of real benchmarks

In order to showcase the full capabilities of the proposed GPU CARM we have characterized the behavior of 20 real-world applications from four different benchmark suites,
namely from Parboil [22], SHOC [23], Rodinia [24] and Polybench [25]. Furthermore, some of the benchmarks and libraries provided by NVIDIA were also characterized, such as cuBLAS, cuFFT and BlackScholes [7]. The number of kernel instructions was also sampled (FP, DP, load/store, integer and control-flow) using mprof and the compute/memory utilization was evaluated by relying on the NVIDIA Visual Profiler [7]. In particular, Fig. 7 shows the benchmark classifications as provided by the NVIDIA Visual Profiler, namely: compute bound (COMP), L2 bound (L2), bound by the device memory or L2 (DM/L2), bound by the device memory (DM), or bound by instruction latency (LAT).

In addition to the GPU CARM characterization the applications were also characterized using the Original Roofline Modeling principles (see Fig. 8(a) and 8(b), respectively). The benchmarks from different benchmark suites are also identified in Fig. 8 as pb (Parboil), sh (SHOC), r (Rodinia), p (Polybench) and cu (CUDA). In order to facilitate the analysis of the results in Fig. 8, the benchmarks were also separated into five groups and plotted with distinct symbols, namely: compute-bound (+), completely L2 bound (□), limited by L2 and device memory (△), device memory bound (○) and instruction latency bound (●).

Moreover, in order to provide a better insight for some applications, an additional ceiling was included in the performance GPU CARM that corresponds to the device memory instruction latency. This ceiling was constructed by relying on a specifically created micro-benchmark that performs load/store operations to/from the global memory by forcing a non-coalesced memory access pattern, i.e., where multiple threads access to the same memory position. This benchmark reached a bandwidth of 47.08 GB/s, which was used for constructing the “Instruction Latency” ceiling, i.e., the dashed line in the GPU CARM in Fig. 8(a).

As it can be seen in Fig. 8(a), cuGEMM is characterized in the GPU CARM as a compute-bound application, since it is bound by SP arithmetic instructions and balanced in memory utilization (shared memory and L2 cache). pbSGEMM represents a case of an L2 bound application, which falls in the memory bound region and on top of the L2 bandwidth roof. Although pbSGEMM also has a high shared memory utilization, the L2 cache accesses still represent the main application execution bottleneck. Being limited by both device memory and L2 cache accesses is a characteristic present in applications that have a strong mix of L2/device memory transactions and, although characterized as memory bound, they appear between the L2 and the Device Memory roofs (e.g., rSRAD, cuGEMV, shFFT). Also, we can clearly identify the benchmarks that are bound by the device memory as these hover around the Device Memory bandwidth roof, such as pGESUMMV, pSYRK or rN. Finally, the latency bound applications are characterized close to the instruction

Figure 7. Characterization of tested applications from five benchmark suites regarding instruction mixes, utilization and NVIDIA Profiler characterization.

Figure 8. Characterization of 23 tested applications in the proposed performance GPU CARM and ORM.
latency roof, e.g., cuGETRF, pCOVAR and pCORR. It is worth noting that although these applications may have different characterization of instructions and utilization, they are also marked by the NVIDIA Proﬁler as latency bound.

As it can be observed in Fig. 8(a), the characterization of 23 tested applications in the performance GPU CARM closely matches the one provided by the NVIDIA Visual Proﬁler, as well as the characterization based on instructions and utilization. However, this is not the general case for the performance ORM [10], [12], whose characterization of certain applications may signiﬁcantly vary when compared to the NVIDIA Proﬁler and GPU CARM. As shown in Fig. 8, some of the applications are characterized as memory-bound in the GPU CARM and NVIDIA Proﬁler, while they are designated as strictly compute-bound in the performance ORM, such as: p3MM, pGEMM and pbSGEMM (all L2 bound), shFFT (device memory and L2 bound), and rNN (device memory bound). Furthermore, the substantial discrepancy between the two models resides in the latency bound applications, such as rLUD and cuFFT, which are both classiﬁed as compute-bound in ORM.

6. GPU CARM DVFS Effects

Besides being an assistance tool for programmers when optimizing their code, CARM can also be used for DVFS analysis, emphasizing the advantages of frequency scaling for certain GPU applications. Titan X was used for this analysis since it provides more core and memory frequency levels than the other GPUs, namely ﬁve different core frequencies ($C_i = \{1164, 1101, 986, 785, 595\}_{i=1}^{5}$ in MHz), and three available memory frequencies ($M_i = \{3505, 3300, 810\}_{i=1}^{3}$ in MHz) were selected. As presented in Fig. 9, the respective performance GPU CARMs were constructed according to the preselected frequency levels, where the roofs scale with the core and memory frequency (dark to light color). It can be observed that the shared memory and the L2 lines also scale with the core frequencies (see C1-C5 in Fig. 9), while the M1, M2 and M3 lines in Fig. 9 refer to the device memory lines for the three memory frequencies considered.

In addition, three applications with different characteristics were selected (based on the analysis performed in Section 5), namely: cuGEMM (compute bound), p3MM (L2 bound) and cuBlackScholes (device memory bound). Each application was executed for every combination of core and memory frequency, i.e., $5 \times 3$ combinations per application. The overall obtained results are presented in Fig. 9 for all three applications. To further facilitate the analysis, the behavior of each application is separately presented for each considered memory frequency level (i.e., from M1 to M3), as shown in Fig. 10. The application markers with the darkest color correspond to the highest core frequency, while the lightest color corresponds to the lowest core frequency.

For the cuGEMM benchmark, it can be observed that the application points scale with the core frequency, while maintaining the same performance through memory scaling.

Hence, the obtained results match the assigned characterization, since the cuGEMM is bound by the GPU computational resources, which are controlled by the core frequency.

On the other hand, cuBlackScholes presents minimal performance decrease from M1 to M2, due to a small difference between those frequencies. Although device memory bound, the cuBlackScholes has the maximum L2 cache utilization, thus its performance decreases with the decrease in core frequency at the M1 and M2 levels, since the attainable L2 bandwidth scales with the $C_i$. Nevertheless, when the memory frequency is set to M3, the points overlap at the same performance, due to the application becoming device memory bound, i.e., the decrease in the device memory bandwidth has a stronger influence on the cuBlackScholes performance than the L2 cache bandwidth.

Finally, the p3MM performance scales with both core and memory frequencies. At M1 and M2, it is possible to observe that the points are always on top of their respective L2 lines. Although not clearly visible, there is a 1% performance variation between the points at M1 and M2 (for the same core frequency). When the memory frequency is set to M3, the points drop from the L2 bandwidth line, due to the higher influence of the decreased device memory bandwidth. However, the application is still characterized as L2 bound.

Besides confirming the application characterization, the DVFS analysis can also be useful to infer possible energy savings. Since power consumption directly depends on the clock frequency, energy savings may occur when the application does not signiﬁcantly change its performance across frequency ranges. For instance, cuBlackScholes delivers the same performance for different core frequencies, when the memory frequency is set to M3. Therefore, at M3, by setting the lowest core frequency, the cuBlackScholes can achieve energy savings without any performance loss, thus being more energy-efﬁcient.

7. Conclusions

In this paper, a set of visual and insightful GPU models are proposed based on the state-of-the-art Cache-aware Roofline Modeling. The proposed models allow describing the GPU upper-bounds for performance, power consumption and energy-efficiency and they aim at easing the devel-
opment, portability and optimization of GPGPU applications. Extensive GPU micro-benchmarking was conducted in order to assess the maximum capabilities of different functional units and memory hierarchy levels across 8 GPU devices from 3 different NVIDIA generations, upon which the proposed models are constructed and experimentally validated. In the proposed models, we further analyzed how different architectures may provide different energy-efficiency insights depending on the application bottlenecks, and we additionally characterized the behavior of 23 real-world applications from 5 different benchmark suites. The characterization in the proposed models closely matches the low-level NVIDIA profiling, which shows the capability of the proposed models to provide more insightful information than the currently existing approaches. Finally, the DVFS effects on GPU performance upper-bounds were also analyzed from architecture and application aspects by scaling both core and memory frequency.

References


