Online Performance Projection for Clusters with Heterogeneous GPUs

Lokendra S. Panwar*, Ashwin M. Aji*, Jiayuan Meng‡, Pavan Balaji†, Wu-chun Feng*
*Dept. of Computer Science, Virginia Tech. {lokendra, aaji, feng}@cs.vt.edu
†Mathematics and Computer Science Div., Argonne National Lab. balaji@mcs.anl.gov
‡Argonne Leadership Computing Facility, Argonne National Lab. jmeng@alcf.anl.gov

Abstract—**We present a fully automated approach to project the relative performance of an OpenCL program over different GPUs. Performance projections can be made within a small amount of time, and the projection overhead stays relatively constant with the input data size. As a result, the technique can help runtime tools make dynamic decisions about which GPU would run faster for a given kernel. Usage cases of this technique include scheduling or migrating GPU workloads over a heterogeneous cluster with different types of GPUs.**

I. Introduction

Accelerators are being increasingly adopted in today's high-performance computing (HPC) clusters. A diverse set of accelerators exist, including graphics processing units (GPUs) from NVIDIA and AMD and the Xeon Phi coprocessor from Intel. In particular, GPUs have accelerated production codes for many scientific applications, including computational fluid dynamics, cosmology, and data analytics. To enable programmers to develop portable code across these accelerator devices from various vendors and architecture families, general-purpose parallel programming models, such as the Open Computing Language (OpenCL) [11] , have been developed and adopted.

Today, large-scale heterogeneous clusters predominantly consist of identical nodes in order to ease the burden of installation, configuration, and programming such systems; however, we are increasingly seeing heterogeneous clusters with different nodes on the path toward 10x10 [4], which envisions the paradigm of mapping the right task to the right processor at the right time. An early example of such a system is Darwin at LANL, which consists of both AMD and NVIDIA GPUs.

In order to improve the programmer productivity on such heterogeneous systems, virtual GPU platforms, such as Virtual OpenCL (VOCL) [24] and Remote CUDA (rCUDA) [9], have been developed to decouple the GPU programmer's view of the accelerator resource from the physical hardware itself. For example, a user of the VOCL platform writes an OpenCL program without worrying about the physical location and other architectural details of the device. The VOCL runtime system in turn schedules and migrates application kernels among GPU devices in the backend [23].

It is critical for such runtime systems to assign the optimal GPU for a given kernel. Our experiments indicate that the peak performance ratios of the GPUs do not always translate to the optimal kernel-to-GPU mapping scheme. GPUs have different hardware features and capabilities with respect to computational power, memory bandwidth, and caching abilities. As a result, different kernels may achieve their best performance or efficiency on different GPU architectures.

However, no clear strategy exists that can help runtime systems automatically choose the best GPU device for an incoming kernel. GPU performance analysis tools have recently been developed to either predict the kernel's performance on a particular device [10] or to identify critical performance bottlenecks in the kernel [16], [25]. While these techniques provide insights into the kernel and device characteristics, they are often tied to particular GPU families and require static analysis or offline workload profiling, which makes them infeasible to be used with runtime systems.

To address this issue, we propose a technique for online performance projection, whose goal is to automatically select the best-performing GPU for a given kernel from a pool of devices. Our online projection requires neither source-code analysis nor offline workload profiling, and its overhead in casting projections has little to no effect with increase in input sizes. We first build a static database of various device profiles; the device profiles are obtained from hardware specifications and microbenchmark characterizations and are collected just once for each target GPU.

When projecting the performance of a GPU kernel, we first obtain a workload's runtime profile using a downscaled emulation with minimal overhead, and then apply a performance model to project the full kernel's performance from the device and workload profiles. This hybrid approach of emulation and modeling is named as *mini-emulation*. The emulation code is invoked by intercepting GPU kernel launch calls, an action that is transparent from the user's perspective. Our technique is particularly suitable for projecting the relative performance among GPU devices for kernels with large data sets.

Our specific contributions are as follows:

- An online kernel characterization technique that retrieves key performance characteristics with relatively low overhead by emulating a downscaled kernel execution.
- An online performance model that projects the performance of full kernel execution from statistics collected from the downscaled emulation.
- An end-to-end implementation of our technique that covers multiple architectural families from both AMD and NVIDIA GPUs.

We evaluate our online performance projection technique by ranking four GPUs that belong to various generations and architecture families from both AMD and NVIDIA. Our workload consists of a variety of memory- and compute-bound benchmarks from the AMD APP SDK [3]. We show that the

Panwar and Aji have contributed equally to this work.

device selection recommended by our technique is the optimal device in most cases.

The rest of the paper is organized as follows. Section II discusses related work regarding GPU performance modeling and analysis. Section III describes the OpenCL programming model. Our performance projection framework is introduced and explained in Section IV. In Section V we evaluate our framework, and in Section VI we conclude the paper with a brief summary.

II. Related Work

Several techniques for understanding and projecting GPU performance have been proposed. These techniques are often used for performance analysis and tuning. We classify the prior work into two categories: performance modeling and performance analysis and tuning.

Performance Modeling: GPU performance models have been proposed to help understand the runtime behavior of GPU workloads [10], [25]. Some studies also use microbenchmarks to reveal architectural details [10], [21], [22], [25]. Although these tools provide in-depth performance insights, they often require static analysis or offline profiling of the code by either running or emulating the program.

Several cross-platform, performance-projection techniques can be used to compare multiple systems, many of them employ machine learning [13], [19]. However, the relationship between tunable parameters and application performance is gleaned from profiled or simulated data.

Performance Analysis and Tuning: Researchers have proposed several techniques to analyze GPU performance from various aspects, including branching, degree of coalescing, race conditions, bank conflict, and partition camping [2], [5], [18]. They provide helpful information for the user to identify potential bottlenecks.

Several tools have also been developed to explore different transformations of a GPU code [12], [14]. Moreover, Ryoo et al. proposed additional metrics (efficiency and utilization) [17] to help prune the transformation space. Furthermore, several autotuning techniques have been devised for specific application domains [6], [15].

The various techniques either require the source code for static analysis or rely on the user to manually model the source-code behavior; both approaches are infeasible for application to a runtime system. Moreover, they often require offline profiling, which may take even longer than the execution time of the original workload. To our knowledge, no GPU performance models have been developed that are suitable for online device selection.

III. Background

Here we describe the Open Computing Language (OpenCL) and the Virtual OpenCL (VOCL) runtime system.

A. OpenCL and Its Performance Optimizations

OpenCL [11] is an open standard and parallel programming model for programming a variety of accelerator platforms, including NVIDIA and AMD GPUs, FPGAs, the Intel Xeon Phi coprocessor, and conventional multicore CPUs. OpenCL follows a kernel-offload model, where the data-parallel, compute-intensive portions of the application are offloaded from the CPU host to the device. The OpenCL kernel is a high-level abstraction with hierarchically grouped tasks or *workgroups* that contain multiple *workitems* or worker threads.

OpenCL assumes a hierarchical memory model where the bulk of the data is assumed to be in the device's global memory and accessible by all the workitems in the kernel. Each workgroup also has its own local memory, which the OpenCL implementation can map to the on-chip memory for faster accesses. The local memory is typically used to share and reuse data among threads within a workgroup and cannot be accessed by any other workgroup.

Depending on the computational intensity and the memory access patterns, the performance of an OpenCL kernel can be limited by compute instructions, local memory accesses, or global memory accesses.

B. VOCL

The VOCL platform is a realization of the OpenCL programming model that enables the programmer to utilize both local and remote accelerators through device virtualization. The VOCL user can write a kernel for a generic OpenCL device without worrying about the physical location and other architectural details of the device. With virtual GPU frameworks such as VOCL, all the OpenCL-capable devices in a cluster can be used as if they were installed locally. The VOCL runtime system internally manages scheduling workloads to the devices in a cluster, forwarding the OpenCL calls and transferring data to and from the remote GPU. However, no clear strategy has been developed that can help such systems automatically choose the *best* GPU device for an incoming OpenCL kernel.

IV. Performance Projection

The goal of online performance projection is twofold: (1) to accurately rank the GPU devices according to their computational capabilities and (2) to do so reasonably quickly in support of dynamic runtime scheduling of GPUs. The actual execution of the kernel on the target GPUs serves as the baseline to evaluate both the accuracy and the performance of any performance projection technique. However, for a runtime system in cluster environments, it is infeasible to always run the kernel on all the potential devices before choosing the best device, because of the additional data transfer costs. Below we discuss the accuracy vs. performance tradeoffs of potential online performance projection techniques for GPUs.

Cycle-accurate emulation, with emulators such as GPGPU-Sim [1] and Multi2Sim [20], can be used predict the minimum number of cycles required to execute the kernel on the target device. The accuracy of the projection and the device support directly depends on the maturity of the emulator. Moreover, the overhead of cycle-accurate emulation is too high to be used in a runtime system.

Static kernel analysis and projection can be done at (1) the *OpenCL code* level, (2) an *intermediate GPU language* level (e.g., PTX or AMD-IL), or (3) the *device instruction* level (e.g., cubin). Performance projection from static analysis will be inaccurate because it does not take into account the dynamic nature of the kernel, including memory access patterns and input data dependence. The performance projection will, however, not experience much overhead and is feasible to be used at runtime.

Dynamic kernel analysis and projection involve a tradeoff between the above approaches. The dynamic kernel characteristics, such as instruction distribution, instruction count, and memory access patterns, can be recorded by using functional emulators, such as GPU-Ocelot [8] or the functional emulator mode of GPGPU-Sim and Multi2Sim. The dynamic kernel profile, in conjunction with the target device profile, can be used to develop a performance projection model. While the accuracy of this approach is better than that of static code analysis, the emulation overhead of this approach will be greater. On the other hand, the overhead will be much smaller than with cycle-accurate emulation.

A. Approach

We realize a variant of the dynamic kernel analysis for performance projection while significantly limiting the emulation overhead with minimal loss in accuracy of projection. Our online projection technique requires that all the target GPU devices are known and accessible and that the workload's input data is available. However, we can intercept OpenCL's kernel setup and launch calls to obtain the required OpenCL kernel configuration parameters.

As Figure 1 shows, our online projection consists of three steps. First, we obtain the hardware characteristics through offline profiling. The capability of the device may vary according to the *occupancy*[1] or the *device utilization* level. We use microbenchmarks to profile the devices and their efficiency in response to different occupancy levels.

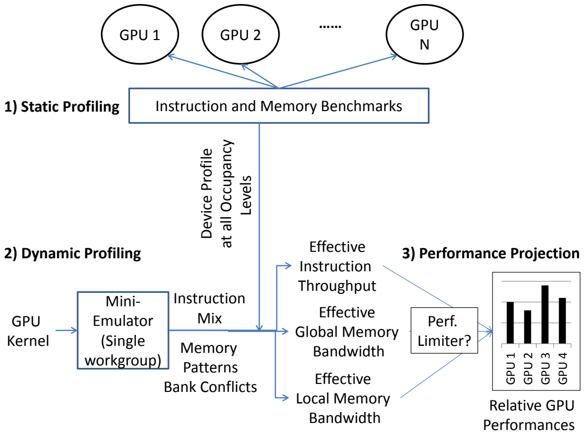

GPU 1
GPU 2
......
GPU N
1) Static Profiling
Instruction and Memory Benchmarks
Device Profile at all Occupancy Levels
2) Dynamic Profiling
GPU Kernel
Mini-Emulator (Single workgroup)
Instruction Mix
Memory Patterns
Bank Conflicts
Effective Instruction Throughput
Effective Global Memory Bandwidth
Effective Local Memory Bandwidth
3) Performance Projection
Perf. Limiter?
GPU 1
GPU 2
GPU 3
GPU 4
Relative GPU Performances

Fig. 1: The Performance Projection Methodology.

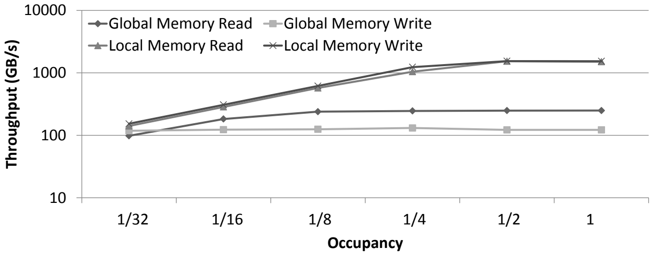


Fig. 2: Memory Throughput on the AMD HD 7970.

Second, we collect the dynamic characteristics of an incoming kernel at runtime. We leverage existing GPU emulators and develop a miniature emulator that emulates only one workgroup of the kernel. With such an approach, we can obtain dynamic workload characteristics including instruction mix, instruction count, local and global memory accesses, and the coalescing degree. The code for our mini-emulator is invoked by transparently intercepting GPU kernel launch calls.

Third, with the per-workgroup characteristics and the per-device hardware profile, we project the runtime of the full kernel execution. Our projection also takes into account various potential performance limiting factors and compares the tradeoffs among devices. GPU runtime systems, such as VOCL, can then select the ideal performing device for the purposes of migration of an already running workload for consolidation or scheduling subsequent invocations in case of repeated execution.

B. Offline Device Characterization

Our device characterization focuses on three major components of GPU performance: instruction throughput, local memory throughput, and global memory throughput. The instruction throughput of a device (or peak flop rate) can be obtained from hardware specifications, and the memory throughput under various occupancy levels and access patterns are measured through offline profiling. We use microbenchmarks derived from the SHOC benchmark suite [7] to measure the hardware's dynamic memory performance under different runtime scenarios, such as occupancy, type of the memory accessed, and word sizes. The hardware characteristics are collected only once per device.

For the global memory accesses, the microbenchmarks measure the peak throughput of coalesced accesses for read and write operations at various occupancy levels. For example, Figure 2 shows the coalesced memory throughput behavior for the AMD HD 7970 GPU. The throughput for uncoalesced memory accesses are derived by analyzing the coalesced throughputs along with the workload characteristics that are obtained from the emulator, as described in Section IV-D.

Similar to global memory, the local memory benchmarks measure the throughput of local memory at varying occupancy levels of GPU. Our local memory throughput microbenchmarks do not account for the effect of bank conflicts, but our model deduces the number of bank conflicts from the emula-

[1] Occupancy refers to the ratio of active wavefronts to the maximum active wavefronts supported by the GPU.

tor's memory traces and adjusts the projected performance as described in Section IV-D.

C. Online Workload Characterization

This subsection describes our fully automated approach for dynamically obtaining a workload's characteristics—in particular, statistics for both dynamic instructions and dynamic memory accessess— in order to cast performance projections.

Statistical measures about dynamic instructions include the instruction count, branch divergence intensity, instruction mixes, and composition of the very long instructions. The dynamic memory access statistics include the local and global memory transaction count, bank conflict count, and the distribution of coalesced and uncoalesced memory accesses. The above runtime characteristics can impact the actual kernel performance in different ways on different GPUs. For example, the HD 5870 is more sensitive to branching than the HD 7970 [3]. Similarly, the NVIDIA C1060 has fewer shared memory banks and is more sensitive to bank conflicts than the C2050. Emulators are useful tools to obtain detailed workload characteristics without extensive source code analysis. However, the time to emulate the entire kernel is usually orders of magnitude larger than the time to execute the kernel itself. Therefore, off-the-shelf emulators are not suitable for online projection.

1) Mini-Emulation: To alleviate the emulation overhead problem, we propose a technique named "mini-emulation", which employs a modified emulator that functionally emulates just a single workgroup when invoked. Our assumption is that workgroups often exhibit similar behavior and share similar runtime statistics, which is typical of data-parallel workloads. Subsequently, the aforementioned runtime statistics of the full kernel can be computed from the number of workgroups and the statistics of a single workgroup, thereby significantly reducing the emulation overhead.

To emulate both NVIDIA and AMD devices, we adopt and modify two third party emulators: GPGPU-Sim [1] for NVIDIA devices and Multi2sim [20] for AMD devices. We note that our technique can employ other emulators as long as they can generate the necessary runtime characteristics and support the OpenCL frontend. Our modified mini-emulators accept a kernel binary and a set of parameters as input, emulates only the first workgroup, ignores the remaining workgroups and outputs the appropriate statistics. We do not change the task assignment logic in the emulator, i.e. the single emulated workgroup still performs the same amount of work as if it were part of the full kernel having many workgroups.

2) Deriving Full Kernel Characteristics: The mini-emulator outputs characteristics of only a single workgroup. To cast performance projections, however, we need to obtain the characteristics of the full kernel. The scaling factor between the characteristics of a single workgroup and that of a full kernel depends on the device occupancy, which in turn depends on the per-thread register usage and local memory usage, which can be obtained by inspecting the kernel binary.

Using device occupancy as the scaling factor, we linearly extrapolate statistics about dynamic instructions and memory accesses of a single workgroup to that of the full kernel. The derived characteristics of the full kernel can then be used to project the kernel's performance.

D. Online Relative Performance Projection

The execution time of a kernel on a GPU is primarily spent executing compute instructions and reading and writing to the global and the local memory. Hence, we follow an approach similar to [25] in modeling three relevant GPU components for a given kernel: compute instructions, global memory accesses and local memory accesses. Moreover, GPUs are designed to be throughput-oriented devices that aggressively try to hide memory access latencies, instruction stalls and bank or channel conflicts by scheduling new work. So, we assume that the execution on each of the GPU's components will be completely overlapped by the execution on its other components, and the kernel will be bound only by the longest running component. We then determine the bounds of a given kernel for all the desired GPUs and project the relative performances. While our three component based model is sufficiently accurate for relative performance projection, it is easily extensible to other components such as synchronization and atomics for higher levels of desired accuracy. We will now describe the approach to independently project the execution times of the three GPU components.

Compute Instructions ($t_{compute}$): When the given OpenCL workload is run through the emulator, our model obtains the total number of compute instructions and calculates the distribution of instruction types from the instruction traces. The throughput for each type of instruction can be found in the GPU vendor manuals. We model the total time taken by the compute instructions as $\sum_i(\frac{instructions_i}{throughput_i})$ where i is the instruction type.

Global Memory Accesses (t_{global}): The global memory performance of a GPU kernel can be affected by the memory access patterns within a wavefront, because the coalescing factor can influence the total number of read and write transactions made to the global memory. For example, in an NVIDIA Fermi GPU, a wavefront (containing 32 threads) can generate up to thirty two 128B transactions for completely uncoalesced accesses, but as low as a single transaction if all the accesses are coalesced and aligned. Hence, there can be up to a 32-fold difference in the bandwidth depending on the coalescing factor on the Fermi. From the memory access traces generated from the emulators, we can deduce the coalescing factor and the number of memory transactions generated per wavefront. Since the memory transaction size and coalescing policies vary with each GPU, we calculate the number of transactions using device-specific formulas. Since the throughput of global memory also varies with the device occupancy, we inspect our per-device characteristics database and use the throughput value at the given kernel's occupancy. We model the time taken by global memory accesses as $\frac{trasactions \times transaction\_size}{throughput_{occupancy}}$. We calculate the above memory access times separately for

TABLE I: Summary of GPU Devices.

GPU	Arch. Name	Compute Units	Peak Perf. (GFlops)	Peak Mem. BW (GB/s)	Mem. Transaction Sizes (B)	Shared Mem. Banks
HD5870	Evergreen	20	2720	264	64	32
HD7970	Southern Islands	32	3790	154	64	32
C1060	Tesla	30	933	102	32,64,128	16
C2050	Fermi	14	1030	144	128	32

read and write transactions and sum them to obtain the total global memory access time.

Local Memory Accesses (t_{local}): The local memory module is typically divided into banks and accesses made to same banks are serialized. On the other hand, accesses made to different memory banks are serviced in parallel to minimize the number of transactions. We inspect the GPU emulator's local memory traces and calculate the degree of bank conflicts. Also, we use the local memory throughput at the given kernel's occupancy for our calculations. We calculate the total time taken by the local memory accesses similar to that of global memory, where we model the read and write transactions separately and sum them to get the total local memory access time.

The boundedness of the given kernel is determined by the GPU component that our model estimates to be the most time consuming, i.e. $max(t_{compute}, t_{global}, t_{local})$.

V. Experimental Methodology

In this section, we describe our experimental setup and present the evaluation of our online performance projection model.

A. System Setup

Our experimental setup consists of four GPUs: two from AMD and two from NVIDIA. We used the AMD driver v9.1.1 (fglrx) for the AMD GPUs and the CUDA driver v285.05.23 for the NVIDIA GPUs. The host machines of each GPU were running 64-bit Linux. We used Multi2sim v4.0.1 for simulating the OpenCL kernels on AMD devices and GPGPU-Sim v3 for simulating the NVIDIA GPUs.

Table I presents the architectural details of the GPUs. Besides the differences in the number of computation units and memory modules, these devices also represent a variety of GPU architectures. While the AMD HD 5870 is based on the previous VLIW-based 'Evergreen' architecture, the AMD HD 7970 belongs to the new Graphics Core Next (GCN) 'Southern Islands' architecture, where it moves away from the VLIW-based processing elements (PEs) to scalar SIMD units. The architecture of HD 7970 closely resembles the NVIDIA C2050 (Fermi) architecture in terms of the scalar SIMD units and the presence of a hardware cache hierarchy. The key differences between the NVIDIA C1060 'Tesla' and the NVIDIA C2050 'Fermi' architectures are the hardware cache support, improved double-precision support, and dual-wavefront scheduling capabilities on the newer Fermi GPU.

TABLE II: Summary of Applications.

FloydWarshall	FastWalshTrans.	MatMul (gmem)	MatMul (lmem)
Nodes = 192	Size = 1048576	[1024,1024]	[1024,1024]

Reduction	NBody	AESEncryptDecr.	Matrix Transpose
Size = 1048576	Particles = 32768	W=1536, H=512	[1024,1024]

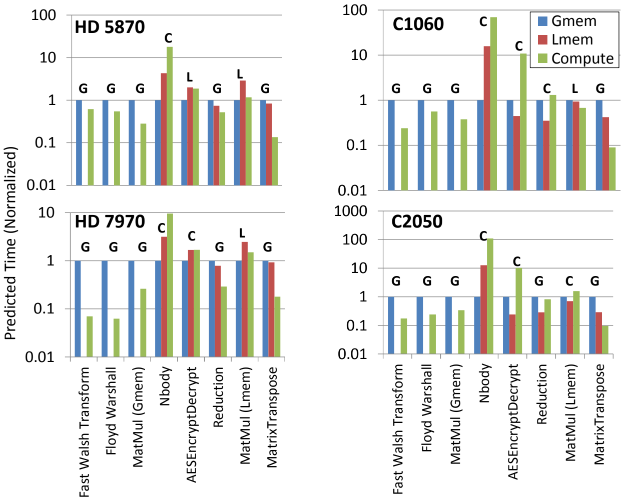
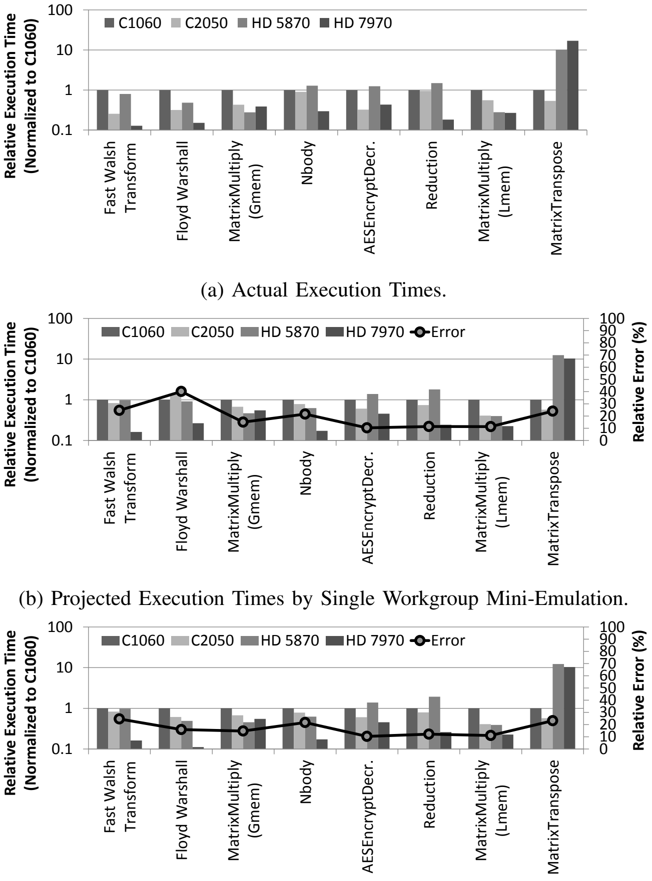


Fig. 3: Analysis of the Performance Limiting Factor. The performance limiter of an application is denoted at the top of each bar: G for Gmem, L for Lmem and C for Compute.

Table II summarizes our chosen set of eight benchmarks from the AMD APP SDK v2.7, which are all written in OpenCL v1.1. We chose benchmarks that exhibited varying computation and memory requirements. We apply our model-based characterization, as described in Section IV-D, to identify the performance bottlenecks in computation, global memory throughput, and local memory throughput. Figure 3 presents the projected normalized execution times for the AMD and NVIDIA devices and characterizes the applications by the performance limiting components.

Figure 3 shows that our set of benchmarks exhibits a good mix of application characteristics, where the benchmarks are bounded by different GPU components. Our model establishes that the benchmarks FloydWarshall, FastWalsh, MatrixTranspose, MatrixMultiply (global memory version), and NBody retain their boundedness across all the GPUs, while the boundedness of other applications changes across some of the devices. Figure 3 also suggests that the performance limiting components of AESEncryptDecrypt and MatrixMultiply (local memory version) are not definitive because the projected times are very close and within the error threshold. For example, AESEncryptDecrypt can be classified as either local memory bound or compute bound for the AMD devices, and MatrixMultiply (local memory version) can be local memory or global memory bound for the NVIDIA C1060. However, the ambiguity in boundedness does not affect the relative ranking of the GPUs because our model just picks one of the competing components as the performance limiter.



(a) Actual Execution Times.

(b) Projected Execution Times by Single Workgroup Mini-Emulation.

(c) Projected Execution Times by Full Kernel Emulation.

Fig. 4: Accuracy of the Performance Projection Model.

B. Evaluation

Our online performance projection technique needs to estimate the relative execution time among devices within a small amount of time. Therefore, we evaluate our technique from two aspects: modeling accuracy and modeling overhead.

1) Accuracy of Performance Projection: In this section, we evaluate the ability of our technique to project the relative performance among target GPUs. Figure 4a shows the actual execution time of all benchmarks on our four test GPUs, and Figures 4b and 4c show the projected execution times by the single workgroup mini-emulation and the full kernel emulation, respectively. All the numbers are normalized to the performance of NVIDIA C1060.

Optimal Device Selection: The main purpose of our performance projection model is to help runtime systems choose the GPU that executes a given kernel in the smallest amount of time. We define the *device selection penalty* to be the $\frac{|T(B)-T(A)|}{T(A)} \times 100$, where $T(A)$ is the runtime of the kernel over its best performing GPU and $T(B)$ is the runtime of the kernel over the recommended GPU. Figure 4 shows that our model picks the most optimal device for all cases except one: the AES-Encrypt application. In this case, our model picks the AMD HD 7970, whereas the most optimal device is C2050, with an optimal device selection penalty of 33.72%.

Relative Performance among Devices: In some cases, the runtime system may want to perform a global optimization to schedule multiple kernels over a limited number of GPUs. In those circumstances, the runtime may need information about the kernel's relative performance among the GPUs in addition to the optimal GPU for each kernel. This helps the runtime evaluate the tradeoffs of various task-device mappings and make judicious decisions in scheduling multiple kernels.

We measure the error of the relative performance as follows. Let us consider the kernel's actual performance on the four GPUs as one 4D vector, T_{actual}. Similarly, the kernel's projected performance on the four GPUs can then be represented as another 4D vector, $T_{projected}$. T'_{actual} and $T'_{projected}$ are the normalized, unit-length vectors for T_{actual} and $T_{projected}$, respectively. They reflect the relative performance among the GPUs. We then formulate the error metric of our relative performance projection to be $\frac{||T'_{actual}-T'_{projected}||}{\sqrt{2}} \times 100\%$, which ranges from 0% to 100% and correlates with the Euclidean distance between T'_{actual} and $T'_{projected}$. Note that $\sqrt{2}$ is the maximum possible Euclidean distance between unit vectors with non-negative coordinates. For the single workgroup mini-emulation mode, the average relative error of our model across all kernels in our benchmark suite is 19.8%, with the relative errors for the individual applications ranging from 10% to at most 40%. On the other hand, if the full-kernel emulation mode is used, then the average relative error becomes 16.7%.

Limitation of Our Performance Projection Model: We note that the single workgroup mini-emulation mode does not change the individual application-specific relative errors from the full kernel emulation for most of the applications, with the exception of Floyd Warshall. A key requirement for the mini-emulation mode is that all the workgroups of the kernel must be independent of each other and that all the workgroups will execute in approximately the same amount of time with the same number of memory transactions and instructions. The Floyd Warshall application comprises a series of converging kernels, where the input of one kernel is dependent on the output of the previous kernel; that is, there is data dependence between iterations. Since only a single workgroup is being emulated in the mini-emulation mode, all the data elements are not guaranteed to be updated by the program iterations, thereby causing the memory and compute transactions to change over iterations. Since Floyd Warshall is a global memory bound application, we compared the projected global memory transactions of the mini- emulation mode and the full kernel emulation modes for similar global memory bound kernels (Figure 5). We see that for the Floyd Warshall application, the projected global memory transactions using the mini-emulation mode is $2.5\times$ less than the projected transactions from the full kernel emulation mode for the C1060 GPU. For the C2050 GPU, this difference is less: 10%. The data-dependent and iterative nature of the Floyd Warshall application introduces errors into our mini-emulation–based projection model, which may cause our model to pick the

TABLE III: Performance Model Overhead Reduction – Ratio of Full-Kernel Emulation Time to Single Workgroup Mini-Emulation Time.

Application	Fast Walsh Transform	Floyd Warshall	MatMul (Gmem)	Nbody	AES Encrypt	Reduction	MatMul (Lmem)	Matrix Transpose
C1060	2882.1	172.7	267.9	3.9	710.4	1710.7	240.7	554.2
C2050	2772.7	182.6	337.4	4.1	784.0	1709.0	196.6	556.1
HD 5870	2761.0	248.5	219.1	4.7	623.4	2629.1	201.5	469.2
HD 7970	2606.9	229.3	217.7	4.0	581.4	2700.0	209.6	457.1

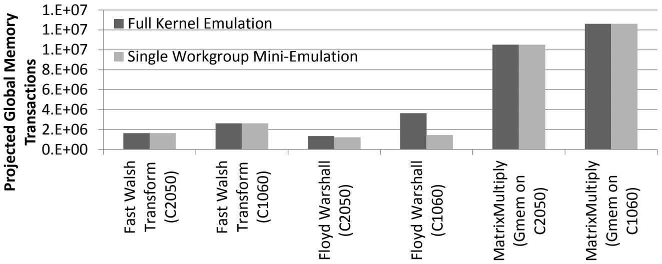


Fig. 5: Global Memory Transactions for Select Applications.

wrong GPU in some cases.

2) Overhead of Performance Projection: The overhead of our online projection includes time spent in online workload characterization as well as casting the projected performance for each device. Because hardware characterization is done offline, once for each hardware, it does not incur any overhead at the time of projection. Among the two sources of overhead, casting performance projection need only calculate a few scalar equations; it has a constant and negligible overhead. The major source of overhead comes from the online workload characterization using mini-emulation, which functionally emulates one workgroup to collect kernel statistics.

The state-of-the-art technique to obtain detailed runtime statistics of a kernel is full kernel emulation. As Table III shows, our mini-emulation approach reduces the emulation overhead by orders of magnitude. Meanwhile, it obtains the same level of details about runtime characteristics. In fact, the mini-emulation overhead is often comparable to kernel execution time with small or moderate-sized inputs and will be further dwarfed if the kernel operates over a large data set, as is often the case for systems with virtual GPU platforms. Such a low overhead makes it worthwhile to employ our technique to schedule kernels with large data sets; it also allows the runtime system to evaluate the task-device mapping in parallel with the workload execution, so that it can migrate a long running workload in time. Below we further study the relationship between input data size and the overhead of mini-emulation.

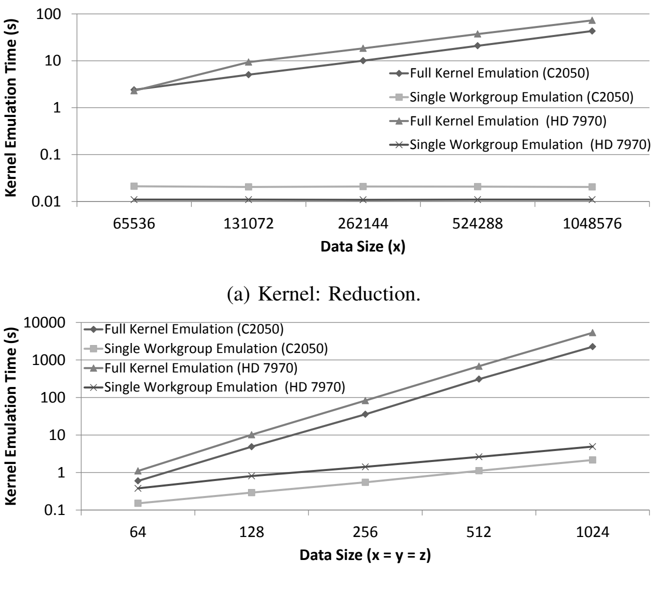
Impact of Input Data Sizes: Figure 6 shows the performance impact of the input data size on the full-kernel emulation and single workgroup mini-emulation overheads. Figure 6a shows that the full-kernel emulation overhead for the reduction kernel increases with data size. The reduction kernel launches as many workitems as the data elements, thereby having a one-to-one mapping between the workitem and the data element. As the data size grows, the number of workitems also increases, so that each workitem and workgroup has a constant amount of computation. Since each workitem of the kernel is simulated sequentially on the CPU, the overhead of the full-kernel emulation also increases for larger data sizes. On the other hand, our mini-emulation scheme simulates just a single workgroup to collect the kernel profile irrespective of the number of data elements. That is why we see a constant overhead for the mini-emulation scheme for reduction kernel.

Figure 6b shows that both the full kernel emulation overhead and the mini-emulation overhead for the matrix multiplication kernel increases with data size. However, we observe that the rate of increase of overhead (slope of the line) is linear for the mini-emulation mode, while it is cubic for the full kernel emulation mode. The matrix multiply kernel launches as many workitems as the output matrix size, but unlike the reduction kernel, each workitem and workgroup do not have a constant amount of computation. The load on each workitem increases linearly with the matrix length (we choose only square matrices for the sake of simplicity). This is why we see that the mini-emulation overhead increases linearly with the matrix length for the matrix multiplication kernel.

However, the actual GPU execution time itself increases in a cubic manner with the matrix length; thus, our mini-emulation mode is asymptotically better and will take less time than running the kernel on the device for larger data sizes. Figure 7 shows that as the matrix length increases, the GPU execution time approaches the kernel mini-emulation time for the NVIDIA C2050 GPU. We were unable to store even larger matrices on the GPU's 3 GB global memory; but we can infer that for even larger matrix sizes, our mini-emulation technique will outperform the actual GPU execution. On the other hand, if the mini-emulation time remains constant, as with the reduction kernel, then it is obvious that the mini-emulation approach will cause the least overhead for larger data sizes, thereby making our model amenable to dynamic decision-making runtime systems.

VI. Conclusion

We proposed, implemented, and evaluated an online performance projection framework for optimal GPU device selection. The applications of our framework include runtime systems and virtual GPU environments that dynamically schedule and migrate GPU workloads in cluster environments. Our technique is based on offline device profiling and online kernel characterization. To automatically obtain runtime kernel statistics with an asymptotically lower overhead, we propose the mini-emulation technique that functionally simulates a single workgroup to collect per-workgroup statistics, which can then be used to calculate full-kernel statistics. Our technique is especially suitable for online performance projection for kernels with large data sets. Our experiments with GPU



(a) Kernel: Reduction.

(b) Kernel: Matrix Multiplication (Using Local Memory).

Fig. 6: Kernel Emulation Overhead – Full Kernel Emulation vs. Single Workgroup Mini-Emulation.

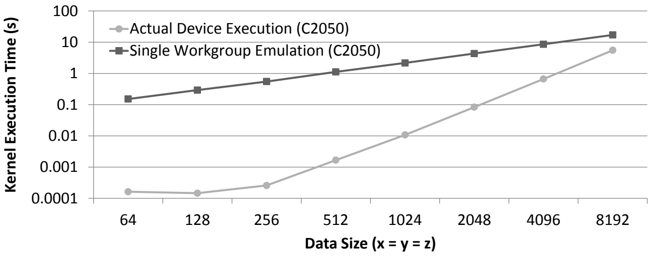


Fig. 7: Kernel Emulation Overhead – Single Workgroup Mini-Emulation vs. Actual Device Execution.

devices of different vendors show our technique is able to select the optimal device in most cases.

Acknowledgments

This work was supported in part by the U.S. DOE, Office of Science, Advanced Scientific Computing Research, under contract DE-AC02-06CH11357, DE-AC02-06CH11357, by the U.S. DOE contract DE-EE0002758 via the American Recovery and Reinvestment Act of 2009, by the VT College of Engineering SCHEV grant, by the NVIDIA CUDA Research Center Program, a NVIDIA Professor Partnership and a NVIDIA Graduate Fellowship.

References

[1] T. Aamodt *et al.*, "GPGPU-Sim 3.x manual," 2012.

[2] A. M. Aji, M. Daga, and W. Feng, "Bounding the effect of partition camping in GPU kernels," in *International Conference on Computing Frontiers (CF)*. ACM, 2011.

[3] AMD. APP SDK v2.7. [Online]. Available: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/

[4] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, 2011.

[5] M. Boyer, K. Skadron, and W. Weimer, "Automated dynamic analysis of CUDA programs," in *Proceedings of 3rd Workshop on Software Tools for MultiCore Systems*, 2010.

[6] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[7] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010.

[8] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2010.

[9] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. Quintana-Orti, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *International Conference on High Performance Computing and Simulation (HPCS)*. IEEE, 2010.

[10] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[11] Khronos Group Std., "The OpenCL Specification, Version 1.0," http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf, 2009.

[12] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, A. Sarma, D. Nanongkai, G. Pandurangan, P. Tetali *et al.*, "Pycuda: Gpu run-time code generation for high-performance computing," *Arxiv preprint arXiv*, vol. 911, 2009.

[13] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[14] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "GROPHECY: GPU performance projection from CPU code skeletons," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[15] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *International Conference on Supercomputing*. ACM, 2009.

[16] NVIDIA Corporation, "NVIDIA Corporation. NVIDIA Visual Profiler," http://developer.nvidia.com/content/nvidiavisualprofiler.

[17] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded GPU," in *IEEE/ACM international symposium on Code Generation and Optimization (CGO)*. ACM, 2008.

[18] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "GPUPerf: A performance analysis framework for identifying potential benefits in GPGPU applications," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[19] V. Taylor, X. Wu, and R. Stevens, "Prophesy: An infrastructure for performance analysis and modeling of parallel and grid applications," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 4, 2003.

[20] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, " Multi2Sim: A simulation framework for CPU-GPU computing ," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[21] V. Volkov and J. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *ACM/IEEE Conference on Supercomputing (ICS)*, 2008.

[22] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2010.

[23] S. Xiao, P. Balaji, J. Dinan, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W.-c. Feng, "Transparent accelerator migration in a virtualized GPU environment," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.

[24] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W.-c. Feng, "VOCL: An optimized environment for transparent virtualization of graphics processing units," in *IEEE Innovative Parallel Computing (InPar)*, 2012.

[25] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2011.