Fast Computational GPU Design with GT-Pin

Melanie Kambadur*, Sunpyo Hong†, Juan Cabral†, Harish Patil†, Chi-Keung Luk†, Sohaib Sajid†, Martha A. Kim*

*Columbia University, New York, NY
{melanie,martha}@cs.columbia.edu

†Intel Corporation, Hudson, MA
{sunpyo.hong,juan.r.cabral,harish.patil,chi-keung.luk,sohaib.m.sajid}@intel.com

***Abstract*—As computational applications become common for graphics processing units, new hardware designs must be developed to meet the unique needs of these workloads. Performance simulation is an important step in appraising how well a candidate design will serve these needs, but unfortunately, computational GPU programs are so large that simulating them in detail is prohibitively slow.**

This work addresses the need to understand very large computational GPU programs in three ways. First, it introduces a fast tracing tool that uses binary instrumentation for in-depth analyses of native executions on existing architectures. Second, it characterizes 25 commercial and benchmark OpenCL applications, which average 308 billion GPU instructions apiece and are by far the largest benchmarks that have been natively profiled at this level of detail. Third, it accelerates simulation of future hardware by pinpointing small subsets of OpenCL applications that can be simulated as representative surrogates in lieu of full-length programs. Our fast selection method requires no simulation itself and allows the user to navigate the accuracy/simulation speed trade-off space, from extremely accurate with reasonable speedups (35X increase in simulation speed for 0.3% error) to reasonably accurate with extreme speedups (223X simulation speedup for 3.0% error).

***Keywords*-computer simulation; graphics processing unit; performance analysis**

I. Introduction

"Graphics processing unit" is now an inadequate term to describe a piece of hardware with a domain extending well beyond graphics applications. As programmers realize the unique advantages of GPUs (e.g., wide availability on commodity machines, extremely high throughput on parallel tasks, fast memory accesses), many non-graphics applications are being ported from their original CPU implementations to GPU versions. Such *computational GPU* applications are now commonplace in a range of fields including scientific computing [24], computer vision [7], finance [27], and data mining [17].

GPU architects must deliver improved hardware designs to meet the computational needs of these varied applications. A major barrier in achieving this is the massive overheads associated with detailed micro-architectural performance simulations. Simulators execute a program up to 2 million times slower than native execution [4], [14], depending on the simulator and the level of detail in the information recorded. These slowdowns are further compounded when hardware designers need to repeatedly re-run applications to test thousands of design space choices.

These prohibitively large simulation times force architects to focus their evaluation on graphics kernels (potentially neglecting important computational workloads) or to evaluate computational workloads using only kernels rather than full applications. Thus, there is a need in the computer architecture community for detailed analyses of commercially-sized computational GPU applications without the overheads of full-program simulation.

This work addresses that need in three ways. First, it provides **a fast profiling tool that measures performance statistics as applications run natively on existing hardware (Section III)**. This new, industrial-grade tool, called *GT-Pin*, can collect a variety of instruction-level data to inform hardware design. Profiling with GT-Pin typically takes 2-10 times as long as normal execution, does not perturb program execution, and requires no source code modifications or recompilation. In our second contribution, we use GT-Pin to conduct **a characterization study of very large OpenCL programs, averaging 308 billion dynamic GPU instructions apiece (Section IV)**. The commercial and benchmark applications studied are substantially larger than any OpenCL programs that have been characterized publicly. The statistics reported include dynamic instruction counts, breakdowns of memory, control, computation, and logic instructions, kernel and basic block execution counts, SIMD lengths, and memory access information. This characterization reveals a breadth of computational GPU workloads that implies an even greater need for comprehensive simulation when evaluating future GPU designs.

Finally, we **demonstrate how to select small, representative subsets of OpenCL programs to accelerate the simulation of future GPU architectures (Section V)**. These small subsets can be simulated in lieu of full programs in a fraction of the time, while still providing an accurate evaluation of the applications' performance on future hardware. The selection process uses GT-Pin profiling and a little post-processing, but itself requires no simulation. This is a key contrast to prior work in CPU subset selection [2], [25] that allows us to make selections *even for applications that are prohibitively expensive to simulate in full a single time*. Developing this methodology required several innovations including how best to break GPU execution into intervals, how best to characterize those intervals, and how to rapidly find the best combination of interval and characterization for any given application. The resulting methodology offers an exploitable trade-off between simulation accuracy and speed, for example speeding simulation by 35X for 0.3% error or speeding simulation by 223X for 3% error.

These new means of exploring large computational applications enable computer architects to rethink and optimize GPU designs for the burgeoning diversity of workloads now being targeted to GPUs.

II. Background on OpenCL

The GT-Pin tool, the benchmark analyses, and the simulation speedup methodology are all based on OpenCL programs and programming concepts. Unlike other GPU languages, such as CUDA which is specific to NVIDIA, OpenCL programs can run on any heterogeneous architecture from any vendor. This paper uses a number of OpenCL keywords which we briefly introduce here; a more comprehensive discussion of OpenCL can be found elsewhere [19].

OpenCL programs consist of two parts. A *host*, which uses *API calls* to manage the program's execution, and *kernels*, which are procedures that define computational work for OpenCL *devices*. OpenCL devices can be any mix of processing units, for example multiple GPUs and CPUs, but in this paper the device is always a GPU. To manage OpenCL kernels, the host must determine the available devices, set up device-specific memory, create kernels on the host, pass arguments to and run kernels on target devices, and organize any results returned by the kernels. Each of these tasks is completed via built-in OpenCL API calls. For example, one named `clSetKernelArg`, as its name implies, sets an argument to an upcoming kernel.

The API call `clEnqueueNDKernelRange` is particularly important in this work. This call dispatches a kernel to a device, signaling that GPU computation is commencing. Also relevant to this project are a set of API calls that manage *synchronization*. Synchronization calls constrain the order of other API calls, enforcing the desired sequences of events. Until a synchronization call forces coordination, for example to make a memory transfer, kernels execute on the devices asynchronously to the host program. OpenCL has seven synchronization calls:

- `clFinish`,
- `clEnqueueCopyImageToBuffer`,
- `clWaitForEvents`,
- `clFlush`,
- `clEnqueueReadImage`,
- `clEnqueueCopyBuffer`, and
- `clEnqueueReadBuffer`.

Because these calls are the only points where host and device work are guaranteed to align, they constitute a natural and necessary point to start and stop device (in our case, GPU) simulation. Thus, in Section V-B, we will use synchronization calls as one potential means to divide a program's execution into intervals.

Another OpenCL concept relevant to this work is the notion of *global work size*. Supplied as an argument to `clEnqueueNDKernelRange` calls, the global work size defines the total amount of work to be done on a given device, so that larger global work sizes take more execution time.

III. Tracing GPU Programs with GT-Pin

GT-Pin serves a community need for a fast, accurate, flexible, and detailed tool to profile commercial-scale native OpenCL GPU applications. This section describes how GT-Pin collects profiles within OpenCL execution environment and discusses the kinds of profiling data it can collect.

A. Instrumenting within the OpenCL Runtime

GT-Pin, which was inspired by the CPU tool, Pin [15], collects profiling data via dynamic binary instrumentation. Instrumentation, which involves injecting profiling instructions into program code, allows for much faster profiling than simulation. GT-Pin uses binary instrumentation, which while harder to implement than static compiler instrumentation (because it necessitates GPU driver modifications), has the additional benefit of not requiring program recompilation. At a high level, GT-Pin's instrumentation injects instructions into binaries' assembly code as the they are just-in-time (JIT) compiled. These insertions later output profiling results as the program executes natively on the GPU.

Before describing how GT-Pin inserts profiling code into OpenCL programs, we first describe how a normal OpenCL execution works. The left side of Figure 1 illustrates an uninstrumented OpenCL application's execution. First, the application communicates with the OpenCL Runtime by making API calls. Then, when `clEnqueueNDKernelRange` calls are made, the OpenCL Runtime passes the associated kernel source and arguments to the appropriate device driver, in our case a GPU driver. The GPU driver JIT-compiles the kernel source, typically when a `clBuildProgram()` API call is issued. Finally, the compiled, machine-specific binary code is passed along to the GPU for execution.

GT-Pin modifies this process at two points, as shown in the middle and right sides of Figure 1. First, when the OpenCL runtime is initially called by the application, GT-Pin intercepts the call and inserts a GT-Pin initialization routine, which notifies the GPU driver that GT-Pin has been invoked. At this time, a memory space called a *trace buffer* is allocated using malloc. The trace buffer is accessible by both the CPU and GPU and will be used to hold profiling data.

The GPU driver is the second point where GT-Pin must make modifications. After the driver compiles the kernel source code into machine-specific assembly, rather than allowing the driver to send the binary directly to the GPU for execution, the binary is diverted to a GT-Pin *binary rewriter*. The binary re-writer inserts profiling instructions into the program's assembly code. The injected instrumentation differs depending on the profiling data GT-Pin's users wish

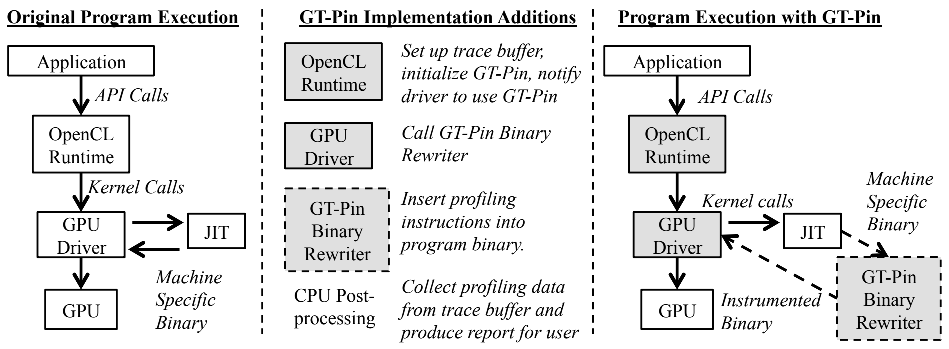
Original Program Execution
Application
API Calls
OpenCL Runtime
Kernel Calls
GPU Driver
JIT
GPU
Machine Specific Binary
GT-Pin Implementation Additions
OpenCL Runtime
Set up trace buffer, initialize GT-Pin, notify driver to use GT-Pin
GPU Driver
Call GT-Pin Binary Rewriter
GT-Pin Binary Rewriter
Insert profiling instructions into program binary.
CPU Post-processing
Collect profiling data from trace buffer and produce report for user
Program Execution with GT-Pin
Application
API Calls
OpenCL Runtime
Kernel calls
GPU Driver
JIT
Machine Specific Binary
GT-Pin Binary Rewriter
GPU
Instrumented Binary

Fig. 1. **The GT-Pin Implementation** makes multiple changes to the OpenCL runtime and the GPU driver, and adds a new GT-Pin binary re-writer and a CPU post-processor. From a user perspective, however, the tool is easy to use and non-intrusive, with low overheads, no perturbation, and no source code modifications or recompilation required.

to collect. For example, to track dynamic basic block counts, GT-Pin adds instructions to initialize a basic block counter at the program's start, to update a counter at each block, and to write the final counter value to the trace buffer at the program's end. Once the re-writer finishes inserting profiling instructions, the GPU driver passes the instrumented binary to the GPU. Then, as the program executes, profiling data is sent to the trace buffer.

Finally, when GPU execution concludes, GT-Pin has the CPU read the profiling results from the trace buffer to post-process the data and generate a user report.

B. *Types of information that GT-Pin can collect*

GT-Pin can observe everything that is happening at the level of both the kernel source and machine-specific binary, so it is able to capture many kinds of profiling data, including:

- static and dynamic instruction execution counts for the source and assembly;
- static and dynamic distributions of opcodes;
- static and dynamic SIMD width counts;
- static and dynamic basic block counts;
- thread cycles in kernel and non-inlined functions;
- latency for memory instructions per thread;
- cache simulation through the use of memory traces;
- memory bytes read and written per instruction; and
- utilization rates of per execution unit SIMD channels.

To reduce overheads, users may collect only the desired subset of these statistics by writing custom profiling tools. For example, for the simulation subset selection in Section V, we wrote a custom GT-Pin tool that collected only instruction counts and opcodes, basic block counts, and memory bytes read and written per instruction.

C. *Overheads and limitations*

Like Pin, GT-Pin guarantees that the side-effects of inserting instructions do not perturb program execution. During instrumentation, GT-Pin minimizes the number of inserted instructions. For example, when counting dynamic instructions, GT-Pin inserts counter increments only once per basic block rather than per instruction. To profile timing events (e.g., thread cycles spent in kernels), GT-Pin inserts a simple timer call, which reads the event timer register. For this type of tracking, we observed timer read overheads of fewer than 10 cycles. From a user perspective, GT-Pin profiling runs take only a little longer than uninstrumented executions. While collecting data for the benchmark characterization study in Section IV, we observed 2-10X overheads. These overheads are very small when compared to the up to 2,000,000X slowdowns required to collect the same information through simulation.

The current version of GT-Pin works only on Intel architectures and for OpenCL programs, although the design concepts could be applied to GPU architectures from other vendors. This would require a new driver implementation and a new ISA specific binary re-writer for each architecture.

IV. A Study of Large OpenCL Applications

This section presents performance data relevant to GPU design for 25 commercial and benchmark applications shown in Table I. All of the programs are written in OpenCL, and come from three sources. First, there are 15 applications from the CompuBench CL 1.2 desktop and mobile suites [12] spanning the domains of computer vision, physics, image processing, throughput, and graphics. Next, there are three applications from the SiSoftware Sandra 2014 suite [26], including two cryptography benchmarks and a GPU performance benchmark. Finally, there are seven video rendering benchmarks from the Sony Vegas Pro Test Project [29]. Sony Vegas Pro 2013 is a video editing tool [28], and the seven benchmarks are pieces of a press release project, each demonstrating different kinds of video attributes such as crossfades and Gaussian blurs.

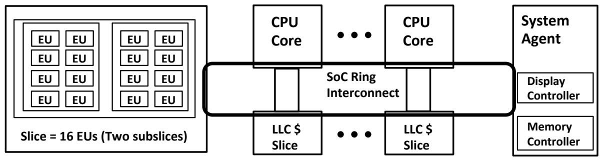

Fig. 2. **The Processor Architecture** of our test system, which has an Intel Core i7-3770 CPU and HD 4000 GPU.

A. *Experimental system*

All applications and benchmarks were run on a machine with an Intel Core i7-3770 CPU and an Intel HD 4000 GPU,

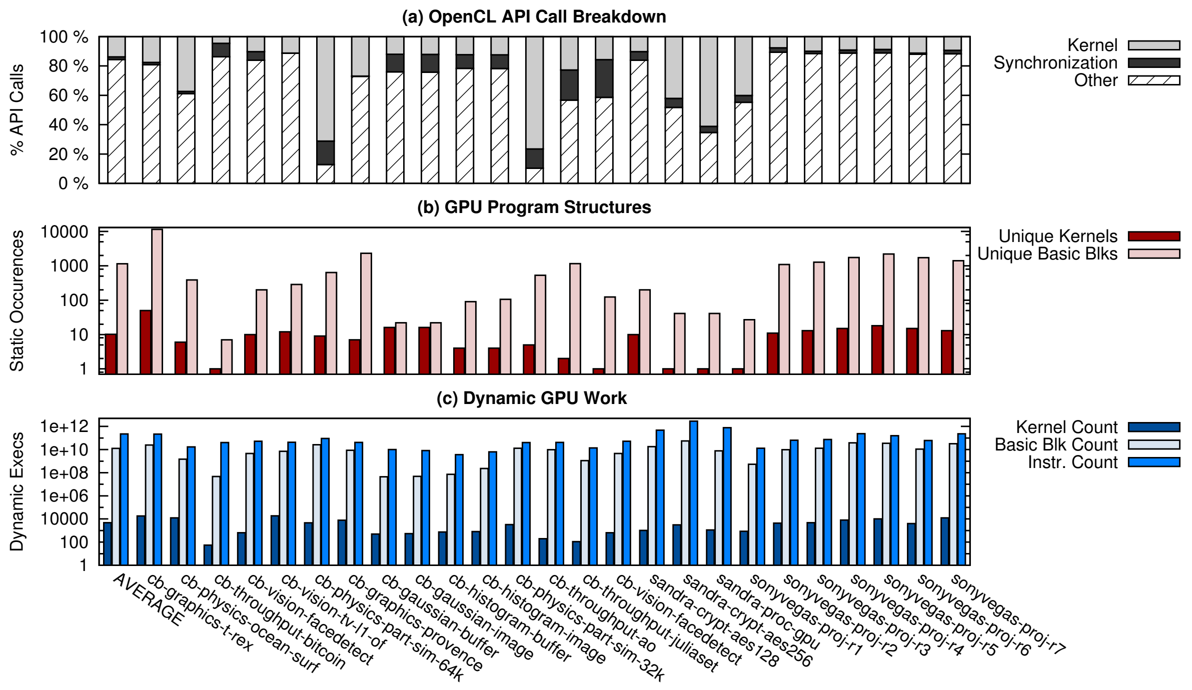

Fig. 3. **Benchmark Characterization.** OpenCL call breakdowns (% synchronization, kernel, and other API calls) were measured on the CPU *host* using CoFluent; program structure counts (unique kernels and static basic blocks) and dynamic work counts (executions of kernels, basic blocks, and instructions) were measured on the GPU *device* using GT-Pin.

TABLE I
BENCHMARKS USED IN THIS STUDY.

Source	Applications
CompuBench CL 1.2 Desktop [12]	Graphics T-Rex, Physics Ocean Surf, Physics Part Sim 64K, Throughput Bitcoin, Vision Facedetect, Vision Tv-l1-of
CompuBench CL 1.2 Mobile [12]	Graphics Provence, Gaussian Buffer, Gaussian Image, Histogram Buffer, Histogram Image, Physics Part Sim 32K, Throughput Ao, Throughput Juliaset, Vision Face Detect
SiSoftware Sandra 2014 [26]	Crypto Aes128, Crypto Aes256, Processor GPU
Sony Vegas Pro 2013 [28]	Press Project Region 1, Region 2, Region 3, Region 4, Region 5, Region 6, Region 7

both of the "Ivy Bridge" generation. As depicted in Figure 2, the HD 4000 has 16 execution units (EUs) organized into two subslices. The EUs are simultaneous multi-threaded (SMT) processor units, which are highly optimized for floating point and integer computations. Each EU has 8 hardware threads per core for a total of 128 simultaneously executing hardware threads. The GPU can perform at a peak rate of 332.8 GFLOPS and has a maximum frequency of 1150 MHz. The system has 16 GB RAM and runs the Windows 7 64-bit operating system. OpenCL Version 1.2 is used for the runtime, and the GPU driver version is 15.33.30.64.3958.

B. *Profiling results*

At program execution time, the CPU was specified as the OpenCL *host*, and the GPU was specified as the *device*. As a convention, data reported at granularities smaller than a kernel invocation (i.e., one execution of a `clEnqueueNDKernelRange`) are aggregate counts across hardware threads.

Calls between the CPU and GPU (Figure 3a). First, we examine how the CPU and GPU communicate through the OpenCL API. GT-Pin tracks only GPU instructions, so we used the Intel CoFluent CPR API tracing tool [3] to count and categorize OpenCL API calls made by the CPU. To collect the name and arguments of every runtime API call, CoFluent intercepts the calls at execution time just before they application passes them to the OpenCL driver. Application performance is unaffected by this capture. Figure 3a divides the API calls made by our 25 applications into three types: *kernel invocations* (i.e., `clEnqueueNDKernelRange` calls), *synchronization* calls (those previously listed in Section II), and *other API* calls, which include program setup, post-processing, and cleanup and supply arguments to kernels. Since the results are reported as percentages, the figure does not show that the 25 applications vary significantly in terms of the total number of OpenCL API calls, from just over 700 calls to over 160,000 calls. The applications are somewhat more consistent in terms of their usage of synchronization and kernel calls. Most applications initiate GPU work through kernel calls with about 15% of the total API calls, though in the case of `throughput bitcoin` and `physics part-sim 32K`, use as few or as many as 4.5% and 76.5%, respectively. Synchronization calls unsurprisingly tend to comprise only a small percentage of the total calls, on average 6.8%, and for the majority of applications less than 3%. The application that uses the highest proportion of synchronization calls (`throughput juliaset` at 25.7%) has the fewest total API calls of any program at 703.

GPU program structures (Figure 3b). Using GT-Pin, we next profiled the static program structures created within the kernels. The *unique kernels* counted in the first set of bars in Figure 3b are the GPU's analogue of CPU procedures. Applications vary widely in the number of unique kernel programs they contain, ranging from 1 to 50 kernels, with a mean of 10.2. Looking at a smaller granularity, we found that each program has at least 7 and at most 11,500 unique basic blocks within these kernels, with a mean of 1139.

Dynamic GPU work (Figure 3c). The number of unique kernels has little correlation with the number of kernel invocations (initiated by `clEnqueueNDKernelRange` calls), which range from 55 to over 18,000, with a mean of 4764. Inside the kernels, 3.7 billion to 2.9 trillion GPU instructions were executed depending on the application (with a mean of 227 billion), within 44 million to 180 billion total basic block executions (on average, 13 billion).

Dynamic instruction mixes (Figure 4a). Figure 4a shows the percentage of opcodes in five categories including *logic*, *control*, *computation*, *send*, and *move* instructions. The logic instructions, which include `and`, `or`, `xor`, `shift`, and `compare` instructions among others, are heavily used, as are the `mov` instructions. This is to support vector operations, such as loading vectors and arithmetic operations within vectors. The *control* instructions account for a smaller overall proportion, at an average of 7.3% of total instructions, and computation instructions account for 36.2% of the total instructions. The `proc gpu` application stands out with a relatively large proportion of computation instructions (91%), because it is designed to stress-test GPU performance. In GEN ISA, Intel GPU's instruction set architecture [10], *send* instructions make up all of the memory communications between hardware threads and execution units. In our applications they account for 5.1% of the overall instructions across applications.

SIMD vector lengths (Figure 4b). In general, the applications take reasonable advantage of data-parallelism. All use a large proportion of 16- and 8-wide SIMD vectors: they comprise 52% and 45% of the instructions, respectively, across applications. Single-width instructions are just 4% of the instructions on average, 4-wide instructions are much less common ($<$0.1% across all applications, and 0.3% of the 6 applications that do use them), and 2-wide instructions are never used.

Memory operations (Figure 4c). Finally, we tracked the cumulative bytes read and written to memory across all GPU hardware threads. The two cryptography applications read the most, at 624 and 2174 GB apiece. The seven Sony video rendering applications were on the high end of writes, and tended to write many more bytes (up to 525X more for `proj-r5`) than they read. On average across all applications, however, the opposite was true: an average of 105 GB were written and 1110 GB were read.

V. Selecting GPU Simulation Subsets

As we just saw, computational GPU benchmarks can be extremely large. Simulating such large benchmarks to determine their performance on future architectures is a problem that until now has been unaddressed. GT-Pin profiling can be used to speed simulation by providing the information necessary to choose small, representative program subsets to simulate from within the large benchmarks. Unlike prior work in simulation subset selection [2], [25], this selection process does not require simulation, allowing us to target extremely large applications that may be prohibitively long to simulate even once in full.

This section describes our GT-Pin-enabled GPU simulation subset selection methodology. GPUs pose a number of unique challenges to address relative to existing CPU selection methodologies (Section V-A). In the experiments that follow, we explore how computational GPU programs can be represented as temporal intervals and architectural features (Section V-B), how to rapidly identify the best interval and feature set for a given application (Section V-C), and how to trade simulation time for accuracy (Section V-D). Finally, we validate that the selections made based on one profiled execution are accurate across multiple execution trials on different processor architecture generations (Section V-E).

A. CPU vs. GPU subset selection

First presented over ten years ago [25], simulation subset selection is a well-studied field in the CPU domain [2], [13], [21]–[23], [25]. In most of these works and in industry, a standard procedure is used to select simulation subsets. The procedure is as follows: **(1)** Profile the program. **(2)** Divide the program trace into *intervals* that serve the dual purpose of encapsulating periodic program behavior and marking the starting and stopping points of the simulation subsets (that will be selected in future steps). **(3)** For each interval, construct a unique *feature vector* that reflects the interval's architectural features. The feature vector's entries count the dynamic occurrences of select runtime events such as the execution of a particular basic block or procedure. **(4)** Group similar feature vectors into a small number of *clusters* (e.g., 10) using machine learning. **(5)** Choose a representative feature vector per cluster, typically the centroid. Additionally, compute a *representation ratio* per cluster, by dividing the number of total dynamic instructions across intervals in the cluster by the number of total dynamic instructions in the whole program. This metric gauges the impact a given cluster has on overall program performance. **(6)** The small number of intervals to which the chosen feature vectors belong make up the selected simulation subset. Simulate this subset of program intervals in detail, while ignoring the remainder of the program by fast-forwarding or check pointing. **(7)** Extrapolate the full-program performance from the results of simulating the representative subset. To do this, simply take the average of each interval's simulated performance, weighted by the representation ratio.

Adapting the procedure for GPUs. To adapt the above procedure to GPUs, we had to answer several open-ended questions. First, *how to build a GPU selection methodology that is architecturally independent* and not tied to a specific GPU platform or ISA. To achieve architectural independence,

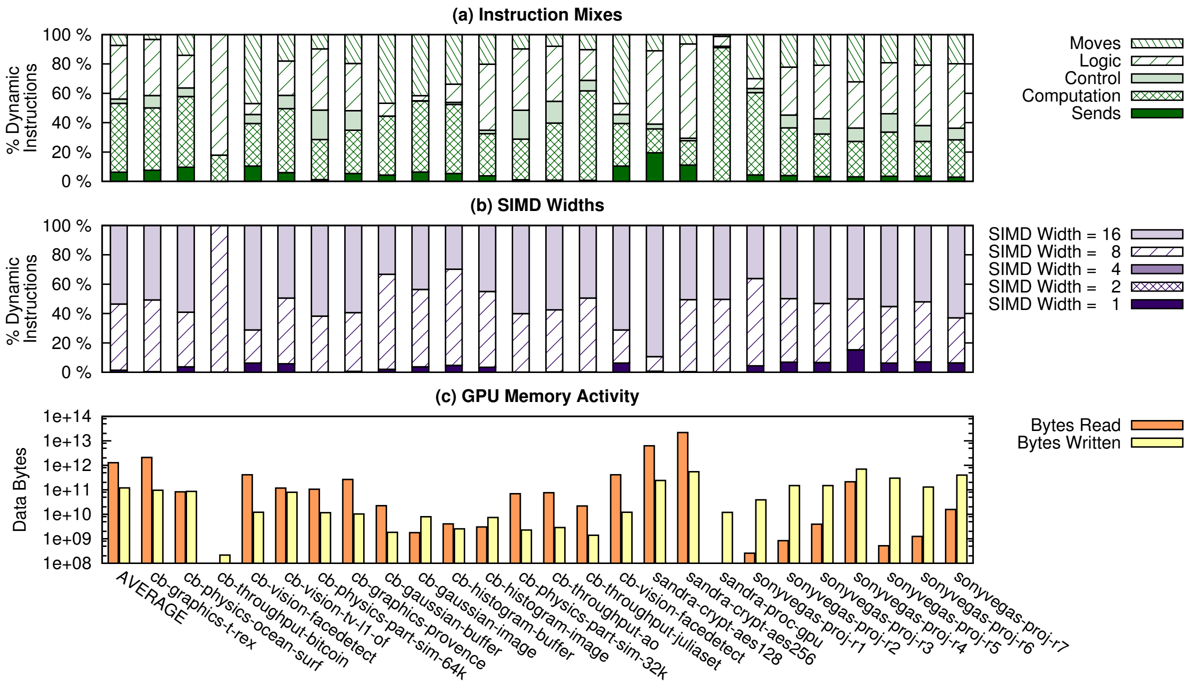

Fig. 4. **GPU Work.** GT-Pin can also measure GPU instruction mixes, the SIMD widths of instructions (i.e., how data-parallel an application is), and the cumulative number of bytes read and written to memory across hardware threads.

TABLE II
THE PROGRAM INTERVAL SPACE EXPLORES THREE DIFFERENT WAYS OF DIVIDING GPU PROGRAM TRACES INTO INTERVALS.

Interval Bound	Relative Size	Intervals per Program Min	Avg	Max
Synchronization calls	*large*	56	545	2115
~100M instructions	*medium*	55	916	3121
Single kernel boundaries	*small*	55	4749	18157

TABLE III
THE PROGRAM FEATURE SPACE EXPLORES TEN FEATURE VECTORS, WITH THE KEYS BELOW AND VALUES THAT COUNT THE DYNAMIC EXECUTION COUNT OF THE RESPECTIVE KEY.

Feature Key	Identifier
Kernel	KN
Kernel, Argument Values	KN-ARGS
Kernel, Global Work Size	KN-GWS
Kernel, Argument Values, Global Work Size	KN-ARGS-GWS
Kernel, # Bytes Read, # Bytes Written	KN-RW
Basic Block	BB
Basic Block, # Bytes Read	BB-R
Basic Block, # Bytes Written	BB-W
Basic Block, # Bytes Read, # Bytes Written	BB-R-W
Basic Block, # Bytes Read + # Bytes Written	BB-(R+W)

we based our methodology around OpenCL programming units and concepts. The next challenging decision was *how to divide the program into intervals*. Interval division 1) must not pose synchronization problems, 2) must strike a balance between being large enough to capture periodic behaviors but not so large as to capture multiple types of behaviors, and 3) must have appropriate boundaries for later simulation, since intervals mark the start and stop points of the selected subsets. According to GPU hardware designers we spoke with, it is a strict limitation that any GPU simulation subset selections be at least a full kernel call in length and that they do not span multiple OpenCL synchronization calls. Another open question was *what feature vectors will accurately summarize the behavior of a GPU execution interval.*

B. GPU interval and feature exploration

To answer these questions, we ran experiments using three types of interval divisions and ten types of feature vectors. In each of these experiments we used the profiling information from GT-Pin and CoFluent to divide the execution into intervals and populate the feature vectors.

Interval space. Previous CPU work divides program traces into uniform intervals of a given number of dynamic instructions, for example 100M instructions [25]. However, such rigid divisions will not work on a GPU as they violate the constraint that GPU intervals should not span kernel boundaries or synchronization calls. Instead, we experiment with three variable length interval sizes summarized in Table II. Synchronization intervals are the largest division, splitting traces at each OpenCL synchronization call. The next smallest intervals further subdivide these into roughly 100M dynamic instruction segments. In order not to split an interval across kernels or a kernel across intervals, this results in some intervals that are slightly larger or smaller than exactly 100M instructions, so we call the division “Approximately 100M instructions”. Finally, we consider each kernel invocation its own interval. While some kernels are larger than 100M instructions, most are not, resulting in the smallest average interval size.

Feature space. Having broken a program into intervals, the second question is which program features to use to characterize that interval for clustering. We experiment with the ten types of feature vectors summarized in Table III. Each feature vector is essentially a set of (key,value) pairs, where the key is a distinct program event such as "calls to kernel foo" or "calls to kernel foo with argument 256", and the values are counts of the number of times this event occurred in a given interval. As Table III shows, our experiments explore whether there is value in increasing the specificity of events to include not only computational information such as kernel or basic block ID, but also data interaction such as the kernel arguments or the number of bytes read or written.

To ensure that these vectors place appropriate value on differently sized kernels and basic blocks, we weight each vector entry by instruction count. For example, if an interval executes block `A` 10 times and block `B` 5 times, these counts alone would suggest that `A` is a more important feature of this interval. However, if `A` were 3 instructions long and `B` were 20, then the weighted score of $5 \times 20 = 100$ for `B` versus $10 \times 3 = 30$ for `A` will better reflect their actual importance. Note that this weighting will also impact the cluster representation ratios that are computed in the next section.

Quantifying simulation error. Once intervals have been divided and feature vectors constructed, any tool can be used to cluster and score them. We used the standard tool from prior CPU work, SimPoint. Specifically, we used SimPoint version 3.0 which can handle variable-sized intervals [8]. SimPoint takes program feature vectors as input, and uses the *k-means* clustering algorithm to group similar feature vectors. It then computes the centroid of each cluster, based on the total element count of each vector, and returns these centroids. We trace these reported feature vector centroids back to their associated intervals to get our simulation subset selections. Along with the cluster centroids, SimPoint also returns representation ratios for each of the selected feature vectors. SimPoint allows users to specify the maximum number of clusters and thus selections, but may return fewer than this maximum if its machine learning algorithm judges it appropriate to do so. The maximum clustering and therefore selection subset count is set to 10 in all the experiments that follow.

Traditionally, detailed simulation of a full program is used to evaluate the representativeness of the selected subsets. However, since we needed to evaluate 30 interval size/feature vector configurations for our 25 large applications, detailed full-program simulation was out of the question. Instead, we developed a heuristic for validating individual selections based on per-kernel timing data, which we collected with the CoFluent CPR tool. The validation heuristic is an error percentage of the measured whole program *seconds per instruction* (SPI) versus the projected whole program SPI, extrapolated from the selections' timings and weights:

$$\text{Error} = \frac{abs(\text{Measured SPI} - \text{Projected SPI})}{\text{Measured SPI}} * 100\% \qquad (1)$$

To get the measured seconds per instruction of the whole program, we divide the combined time in seconds taken by all of the kernel invocations by the total number of dynamic instructions executed by all of the kernel invocations. To get projected SPI, we first find the SPI per selected interval, dividing the sum of CoFluent reported time in seconds of the kernels in the selected interval by the sum of dynamic instruction execution counts reported by GT-Pin for the kernels in the interval. Then, we multiply each selected interval's SPI by its SimPoint ratio, and add these products together to get the projected whole program SPI.

Interval and feature exploration results. Figure 5 shows how the 30 types of interval/feature vector combinations fared in terms of selecting representative program subsets. For brevity, the figure presents error and selection size results of just 3 sample applications, but we tested all 30 combinations on all 25 applications. The results of the remaining 22 applications lead to the same conclusion as the 3 shown: no single combination of interval size and feature vector is 'best' in terms of error or selection size across all applications. There are, however, several trends across applications. For example, basic block based features tend to outperform kernel based features, and features with memory access counts improve basic block based features for most applications. The applications with the fewest unique kernels tend to have high error rates when kernel-only features are used.

As for interval size, synchronization-bounded intervals tend to produce the smallest errors, but since they are also the largest division, they produce the largest selection sizes. For basic block based features, interval size tended to have less of an effect on error rate than the effect of interval size on kernel based features. If we were to choose the best average interval size/feature vector combination of the 30 tested, the combination with the smallest error rate would be basic block intervals with no memory features (BB), and synchronization bounded intervals. This combination averages 1.5% error across all 25 applications, and selects subsets that are 1.9% of the total program instructions (corresponding to a 53X simulation speedup). In the worst case, one individual application has an error of 8.8% and another application has a selection containing 24.0% of the total program instructions.

C. Identifying application specific intervals and features

To improve these error and selection size numbers, we can leverage the fact that the combination that works best for one application is not always what works best for another. Rather than choosing one universal interval and set of features, we can choose the best interval and features *for each individual application*. Somewhat counter-intuitively, there is almost no additional overhead for doing so, as we need to profile (natively) each application just once to characterize the error and selection sizes for all 30 interval and feature vector combinations.

Picking the error-minimizing interval and feature combination for each individual application achieves an average error rate of just 0.3%, with the worst case error being 2.1% for the `histogram buffer` application. Figure 6 shows the error-

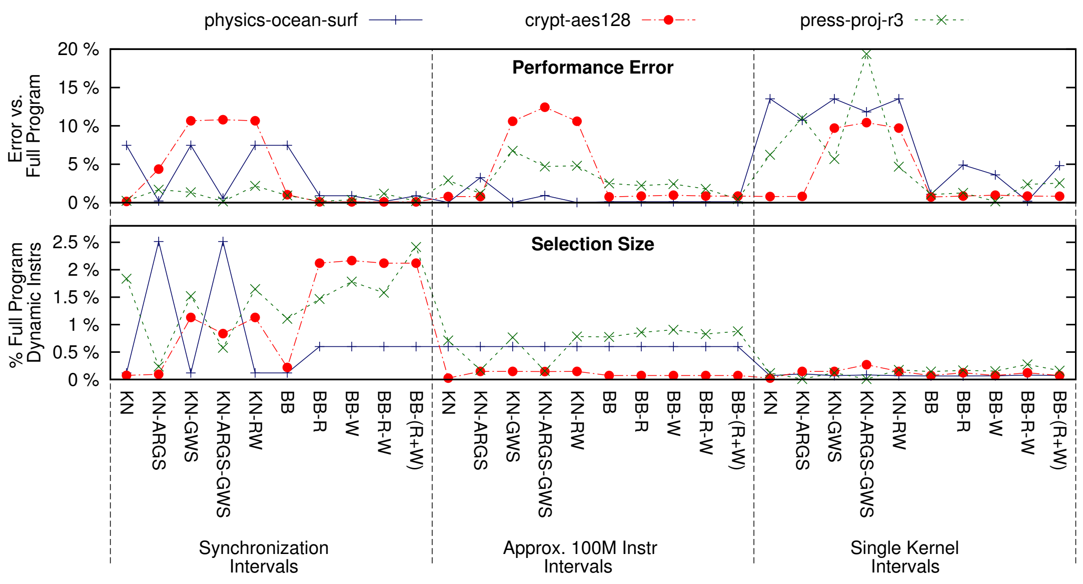

Fig. 5. **Feature and Division Space Exploration.** Applications vary in terms of which of 10 different feature vector choices and 3 interval division sizes are best able to select subsets that match full program performance. Also, the most accurate selection configurations are not always the best at reducing the number of instructions to simulate.

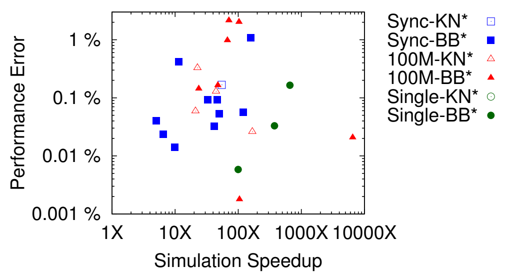

Fig. 6. **Optimizing Selection to Minimize Error** results in individual applications choosing different interval/feature vector configurations. Across the 25 applications, errors average 0.3% and simulation speedups average 35X, ranging from 6X to 6509X.

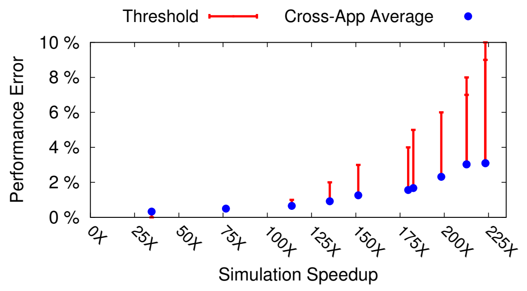

Fig. 7. **Optimizing for Both Error and Selection Size** means choosing the per application configuration that has the smallest selection size with an error below a given *threshold*. For example, with an error threshold of 3%, simulation speedups average 223X.

minimal configuration for each of the 25 applications. Of the 25 applications, only 5 chose kernel-based features while the remainder chose basic block features.

As for interval sizes, 3 applications chose single kernel long slices, 11 chose synchronization bounded slices, and 11 chose 100M instruction slices. Memory-based features were chosen by 20 of the 25 applications. These diverse choices in ‘best’ configuration support our previous observation that no single configuration is suitable for all applications.

D. *Co-optimization of simulation time and error*

Minimizing the error without regard to simulation speedup may still result in subsets that are too large for certain simulation needs. Across applications, this policy resulted in an average simulation speedup of 35X, but just 6X in the worst case. To improve these numbers, we tried jointly-optimizing for error and selection size. By setting an acceptable error *threshold* rather than aiming to minimize error, we can greatly accelerate simulation. Specifically, we choose the per-application configuration with the smallest selection size that also has an error below a given threshold. If no configuration has an error below the specified threshold, we choose the configuration with the smallest error, regardless of selection size. Figure 7 shows the results of applying a range of different thresholds. The furthest left point on the plot shows the cross-application average error and simulation speedup when selection configurations are chosen to minimize error. The remaining points show error thresholds of 0.5% and 1% to 10% at steps of 1%. As error thresholds are relaxed to higher values, the speedups monotonically increase. At the far right end of the graph, when we set the error threshold to 10%, we get an average error across applications of 3.0% and an average simulation speedup of 223X.

E. *Validating the selections for future architectures*

We have already seen that the selected subsets can accurately predict the performance of full-programs executed on the same hardware. Here, we test whether the selections

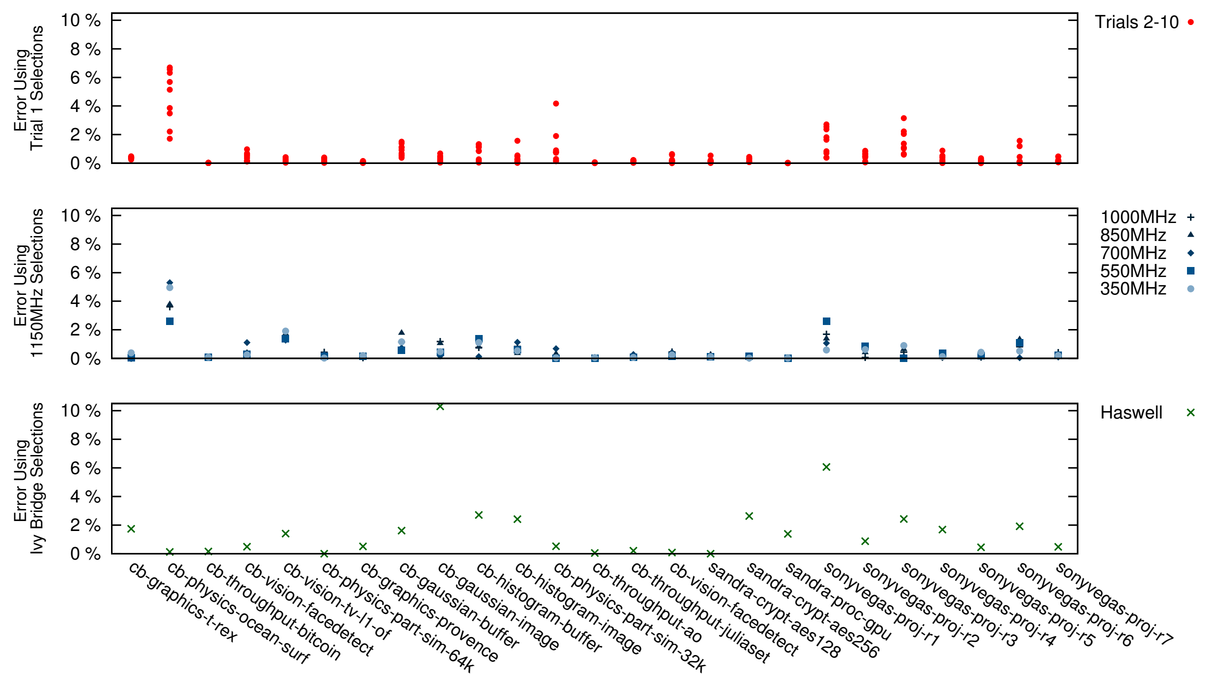

Fig. 8. **Timed Validation.** One trial's selections are still accurate across trials, frequencies, and architecture generations.

built from one set of profiling data can predict full program execution across multiple trials run on different architectures.

Quantifying cross-trial and cross-architecture accuracy. To test a single set of selections across trials and architectures, we first need to guarantee that the kernel calls contained in the selected intervals will be present and findable in future executions. Thanks to non-determinism, this is not automatically the case, but we can force it to be so with a *record* and *replay* feature of the previously discussed CoFluent tool. CoFluent's record mechanism captures API call data as it passes between the application and the OpenCL runtime. In addition to call names, the recorder captures configuration parameters, memory buffers and images, and OpenCL kernel code and binaries. This recorded information can later be replayed and runs just as a normal executable on native hardware would, with the only difference being a consistent and repeatable ordering of API calls.

We generate just one *original* set of selections and representation ratios per application using a CoFluent recording. We next verify this selection against measured SPIs computed from *new* replayed trials' timing and instruction data. Then, we compute the error of the original selection on the new trial.

Cross trial accuracy. Our first experiments tested the selection of one trial against multiple future trials on the same machine. The top plot of Figure 8 shows the resulting error rates for the new Trials 2-9 versus the original Trial 1, for each of the 25 benchmark applications. Most of the error rates are below 3% (with many below 1%), indicating that a single trial's selections can be successfully used to predict the whole program performance of other trials.

Cross frequency accuracy. To see how the selections hold up for future architectures with different processing rates, we next validated the original set of selections against timing data for new trials executed at varying GPU frequencies. All of the data previously reported in this paper use the GPU's maximum frequency of 1150MHz, so the new frequency tests use lower frequencies, specifically at 1000, 850, 700, 550, and 350MHz. The middle plot of Figure 8 shows the resulting error rates. Again, most are less than 3%, indicating that the selections of a single frequency can be used to predict the whole program performance of executions at other frequencies.

Cross architecture generation accuracy. As a final experiment, we tested whether our selections could predict whole program performance across different GPU architecture generations. Specifically we used selections collected on our Ivy Bridge HD4000 GPU to predict program performance on a newer Intel GPU, the HD4600 Haswell processor. The primary difference between the two processors is the number of execution units (EUs) within each GPU: the HD4000 has 16 EUs whereas the HD4600 has 20 EUs. To compare the two processors' raw performance, we ran LuxMark [16] on both machines. LuxMark is a popular cross-platform benchmarking tool, which scores GPUs on their ability to render different test scenes of varying complexity. The results (higher scores are better) were 269 for the HD4000 and 351 for HD4600, demonstrating the performance increases due to parallelism on the HD4600.

The bottom plot of Figure 8 shows the error rates of using HD4000 selections to predict HD4600 performance. Once again, most of the error rates are less than 3%, and the worst case application (`gaussian-image`, one of the shortest benchmarks in terms of kernel invocations) has 11% error. These results show that a single set of selections can predict the performance even on architectures with very different performance characteristics.

VI. Related Work

This is the first work to characterize large OpenCL programs, and one of the first works to explore accelerating GPU simulation through the selection of representative subsets.

GPU application analysis. There are two related profiling tools from the Georgia Institute of Technology: Ocelot [5], [11] and Lynx [6]. Ocelot is a GPU compiler that instruments programs at compile time to measure various performance statistics. Unlike our work Ocelot emulates programs rather than running them on native hardware, it also does not yet fully support OpenCL compilation. Lynx, a binary instrumentation tool that stems from Ocelot does support OpenCL execution on native hardware, but unlike GT-Pin, Lynx has only been demonstrated to work on small programs (their tested applications averaged 2 million times fewer dynamic GPU instructions than ours). Lynx also instruments the NVIDIA PTX instruction set rather than the GEN ISA, and does not offer any solutions for selecting simulation subsets as we do.

Several additional works also characterize GPU programs, although most study much smaller applications and focus on CUDA workloads, which are NVIDIA specific, as opposed to architecture-independent OpenCL workloads. Zhang et al. [31] model the instruction pipeline, shared memory access, and global memory accesses of GPUs to accurately predict — and eventually improve — the performance of overlying applications. Their tested application has 6500 times fewer instructions than the average application studied in this work. Mistry et al. [18] use built-in OpenCL API calls rather than an external profiler to analyze a computer vision algorithm for kernel call durations, average and variations in time spent processing video frames, and GPU command queue activity. Their API-based profiling is much more limited in terms of the types of data it can collect versus GT-Pin's instrumentation-based profiling. Bakhoda et al. [1] use an instrumented version of the GPGPU-Sim simulator to collect a variety of data including instruction mixes, memory and branching statistics, and parallel execution activity for a large collection of benchmarks, but unlike GT-Pin their tool has hefty overheads, on the order of a million times the original program execution time.

Finally, there are commercial tools that monitor program performance (e.g., [20]), but they do not measure instruction-level metrics as we do.

Simulation region selection. There are just two other works in the area of GPU simulation subset selection. The first, by Huang et al. [9], finds representative GPU simulation subsets using a similar overall methodology to our work, with single kernel invocation intervals and compound feature vectors that include a metric analogous to our global work size, a memory request count, and measures of intra-kernel parallelism. Besides the differences in feature vector construction, the work differs from ours in two significant ways. First, they only demonstrate that their feature vector construction works for 12 very small applications, with an average of just 34 kernels invoked per application (versus our applications that average 4749 kernel invocations a piece). Second, while our simulation time savings come entirely from skipping *whole* kernel invocations, their savings come primarily from skipping *parts* of kernel invocations. The second GPU subset selection paper is a work by Yu et al. [30]. This work also reduces simulation sizes by choosing partial kernel invocations, but rather than having the simulator execute intra-kernel samples, they reconstruct reduced-loop count micro-kernels that can be simulated in full. It is possible that such an partial selection method could be combined with our method of skipping whole invocations for improved simulation speedups.

Since the first proposal of CPU simulation subset selection [25], there have been dozens of papers published in the area. Here we address only those most relevant to this paper, such as the PinPoints paper by Patil et al. [21]. Like our work, PinPoints uses dynamic instrumentation to find representative simulation subsets, but it does so only for CPU programs. Follow-up works by the same authors address a repeatability problem that arose between profiling and tracing runs [22] (we avoid this through the use of CoFluent recordings), and a toolkit for finding representative subsets deterministically and check-pointing them for Pin-based simulation of x86 programs [23]. As we do for GPUs, Lau et al. [13] explore a variety of appropriate feature vectors for CPU simulation, finding that basic blocks, loop frequency counts, and register reuse counts work best to encapsulate interval behavior. Finally, the recent BarrierPoint work by Carlson et al. [2] finds representative subsets in parallel OpenMP programs by aligning their interval divisions with synchronization points, much as we do by restricting our GPU programs to kernel invocation boundaries or greater.

VII. Summary

This paper took three steps towards speeding up the design of GPUs for computational workloads. First, it introduced a new, fast GPU profiling tool called GT-Pin, which measures a variety of instruction-level performance factors of applications as they run natively on existing GPUs. Next, it used GT-Pin to characterize 25 very large OpenCL benchmarks, exploring several features relevant to GPU design. Finally, it demonstrated that representative subset selection can successfully accelerate GPU design, by finding small but representative program subsets for GPU developers to simulate in lieu of full programs. These advances enable designers to optimize for the diverse set of computational workloads that are currently being developed for use on GPUs.

References

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[2] T. E. Carlson, W. Heirman, K. V. Craeynest, and L. Eeckhout, "BarrierPoint: sampled simulation of multi-threaded applications," in *ISPASS*, 2014.

[3] CoFluent CPR., "Intel System Modeling and Simulation," http://www.intel.com/content/www/us/en/cofluent/intel-cofluent-studio.html.

[4] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for GPGPU," in *MASCOTS*, 2010.

[5] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *PACT*, 2010.

[6] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili, "Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures," in *ISPASS*, 2012.

[7] J. Fung and S. Mann, "Computer vision signal processing on graphics processing units," in *Acoustics, Speech, and Signal Processing, 2004. ICASSP '04.*, vol. 5, 2004.

[8] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[9] J.-C. Huang, L. Nai, H. Kim, and H.-H. S. Lee, "TBPoint: reducing simulation time for large-scale GPGPU kernels," in *IPDPS*, 2014.

[10] Intel Corp., "Intel OpenSource HD Graphics programmers reference manual," https://01.org/linuxgraphics/sites/default/files/documentation/snb_ihd_os_vol1_part1.pdf, 2011.

[11] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of PTX kernels," in *IISWC*, 2009.

[12] Kishonti, "CompuBench CL for OpenCL and CompuBench RS for RenderScript," http://compubench.com, 2014.

[13] J. Lau, S. Schoemackers, and B. Calder, "Structures for phase classification," in *ISPASS*, 2004.

[14] S. Lee and W. W. Ro, "Parallel GPU architecture simulation framework exploiting work allocation unit parallelism," in *ISPASS*, 2013.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[16] LuxMark., "OpenCL Benchmarking Tool for GPUs," http://www.luxrender.net/wiki/LuxMark.

[17] W. Ma and G. Agrawal, "A translation system for enabling data mining applications on GPUs," in *SC*, 2009.

[18] P. Mistry, C. Gregg, N. Rubin, D. Kaeli, and K. Hazelwood, "Analyzing program flow within a many-kernel OpenCL application," in *GPGPU Workshop*, 2011.

[19] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*, 1st ed. Addison-Wesley Professional, 2011.

[20] NVIDIA Corporation, "NVIDIA visual profiler," http://developer.nvidia.com/nvidia-visual-profiler, 2014.

[21] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *MICRO-37*, 2004.

[22] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs," in *CGO*.

[23] H. Patil and M. Stallcup, "PinPoints: Simulation Region Selection with PinPlay and Sniper," in *ISCA tutorial*, 2014.

[24] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU accelerated monte carlo simulation of the 2D and 3D Ising model," *Journal of Computational Physics*, vol. 228, no. 12, 2009.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS*, 2002.

[26] SiSoftware, "SiSoftware: Sandra 2014," http://www.sisoftware.net, 2014.

[27] S. Solomon, R. Thulasiram, and P. Thulasiraman, "Option pricing on the GPU," in *HPCC*, 2010.

[28] Sony Creative Software, Inc., "Sony Vegas Pro," http://www.sonycreativesoftware.com/vegaspro, 2014.

[29] Sony Creative Software, Inc., "Sony Vegas Pro Test Project," http://download.sonymediasoftware.com/whitepapers/vp11_benchmark.zip, 2014.

[30] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. John, H.-J. Jin, C. Xu, and J. Wu, "GPGPU-MiniBench: Accelerating GPGPU micro-architecture simulation," *IEEE Transactions on Computers*, 2014.

[31] Y. Zhang and J. Owens, "A quantitative performance analysis model for GPU architectures," in *HPCA*, 2011.