

Leveraging Binary Translation for Heterogeneous Profiling

Dan Upton and Kim Hazelwood
Department of Computer Science
University of Virginia

Abstract—Heterogeneous systems, such as those including a graphics processor for general computation, are becoming increasingly common. While this increases the potential computing power that can be leveraged, it also increases the complexity of the system. This in turn increases the complexity of understanding behavior of the system, which is important when developing new software as well as when designing a new heterogeneous platform. Most current approaches profile individual computational resources in isolation, making it difficult to understand the characteristics of the full system. In this paper, we present an approach based on binary instrumentation that simplifies the task of profiling the behavior of heterogeneous systems by enabling a user to easily combine profiling data from multiple resources. We accomplish this by using communication and computation functions as markers to align the profiles. By using a lightweight method, we can achieve an average overhead of 35% overall and only 1% overhead on the compute kernels.

I. INTRODUCTION

Heterogeneous systems have become increasingly common over the past several years. While heterogeneous systems-on-chip with multiple IP blocks have long been common, heterogeneity has more recently moved into general-purpose desktops, laptops, and servers. In such settings, there are two main versions of heterogeneity. The first group consists of systems such as the Cell Broadband Engine [1], which includes multiple kinds of processors on a single package. The second group, which is our primary focus in this paper, consists of systems that combine one or more general purpose processors with a highly-parallel accelerator, such as a GPU. GPUs are used for graphics applications as well as highly-parallel scientific computation, which is known as general-purpose GPU or GPGPU computing.

While such systems have greater potential computing power, this power comes at a cost of increased complexity in both the system and the software. This increased complexity makes profiling the system to understand the performance characteristics more important; however, it

also increases the complexity of collecting profiling data to analyze the performance.

Most profiling approaches for heterogeneous systems analyze only one computational resource at a time. A developer can profile the CPU code using any number of preexisting techniques, such as using hardware performance counters. A variety of tools and APIs have been introduced for profiling GPUs for both graphics applications and computation applications [2], [3], [4], [5], [6], [7], [8]. Two of these tools, GPUView [5] and Graphics Performance Analyzers [7], [8] are notable in that they do allow some cross-resource analysis, primarily by determining whether a workload is CPU- or GPU-bound and making determinations about the GPU based on CPU-level information.

The issue with profiling in isolation is that it is not always possible to determine whether or how the code running on one resource affects the code running on another resource. There are also intermediate software layers, such as the driver code for interfacing between the resources, and the GPU driver commonly performs just-in-time compilation to target the specific device. As a result, application performance is derived from many levels: the application code on the CPU, the execution of the driver on the CPU, the code generated for the GPU, and the execution of that code on the GPU. Understanding performance across all layers is particularly critical when developing a new platform, when many of these layers involve new hardware or software.

Our approach to heterogeneous profiling seeks to leverage existing techniques for profiling the platforms in isolation and extend that to improve the ability of a developer to make connections between those profiles. We begin with a straightforward approach based on sampling on the CPU that is unable to provide enough context to combine the various profiles. We then extend this approach using binary instrumentation to selectively insert markers that allow us to match CPU performance to GPU activities. We compare performance of this

technique with two different methods of instrumentation and show that we are able to align the profiles at a cost of 35% overhead on average. Furthermore, we show that the instrumentation primarily affects the CPU code, with an overhead of less than 1% on the computational kernel.

The rest of this paper is organized as follows. Section II discusses a basic sampling-based approach and its shortcomings. We then describe our method of combining the sampled profiles by inserting function markers using binary instrumentation in Section III. Section IV describes an example use of our approach. Section V compares the performance impacts of two different implementations of our method. Section VI discusses related work, and Section VII concludes and discusses future work.

II. SAMPLE-BASED PROFILING

We begin with a sampling-based approach to profiling both the CPU and GPU portions of the code. Sampling simply involves periodically collecting information about the activity of the running system. Interrupt-based sampling, triggered by either a timer or an overflow in a hardware performance counter, can provide profiling information with low overhead. However, sampling in this sense trades overhead for accuracy, in that determinations about execution frequency and time are statistical and are dependent upon the sampling frequency.

While we can use sampling on the CPU, current GPUs can not have their performance sampled in the same way. This is because execution on a GPU generally can not be interrupted in the same way. On graphical applications, a driver for sharing the graphics card and display may queue commands and can interrupt an application between commands. On computational applications, computational kernels may be queued and an application may be interrupted between sending kernels. However, once an operation or kernel has been sent to the device, the command runs to completion. Since reading the performance counters generally requires taking control of the processor, we can not sample the GPU state when processing a CPU sample.

Instead of sampling the GPU, we use counting mode for the GPU performance counters. In this case, the performance counters are initialized prior to performing a series of actions and then read at a later time. In the GPU case, each performance sample records performance indicators for a GPU function or series of functions. Both of the tools we use to access GPU performance counters—NVIDIA’s Compute Visual Profiler [6] (CVP) and AMD’s GPUPerfAPI [4]—support

reading performance counters in counting mode. CVP automatically records performance at a granularity of each API call, while the GPUPerfAPI allows the developer to include any number of functions within a performance sample. In this work, we actually also use a combination of sampling and counting on the CPU: we read multiple counters during an execution, so one performance counter acts as a sample trigger and the others are read at each sample point in counting mode.

A natural way to combine the two profiles—sampled CPU performance and GPU counts by function—is to annotate each CPU sample with the function executing at that sample, and then use those function names to align the two data sets. In practice, this is generally insufficient on both graphics applications and GPGPU applications. On 3DMarkMobileES, a graphics benchmark to test OpenGL performance on embedded platforms, samples in OpenGL API functions were rare. There were more samples within the graphics driver, which provides some idea of where the driver is spending its time, but it does not provide any information about what the main application is doing that may be leading to greater amounts of time being spent in a particular driver path. On MUMmerGPU++ [9], a GPGPU application for computing sequence alignment written in CUDA, zero of the more than 16,000 samples occurred in any CUDA API functions. Approximately 20% of the samples occurred elsewhere in `libcuda`, primarily in `cuMemGetAttribute`, which is never called directly from the user application. Again, while this tells the developer that some amount of time is being spent in library calls or waiting on results, it does not help determine what application code is specifically responsible for that activity.

Although this straightforward method did not provide sufficient information for combining profiles across heterogeneous resources, it did provide insight to the pitfalls that can arise from simply collecting the results in isolation. An example is trying to determine the cause of a lower frame rate than expected on 3DMarkMobileES. Our initial sampling collection showed more than 50% of the CPU samples occurred in `memcpy` when collecting samples every millisecond. We changed the system to use a different version of `libc` with an optimized version of `memcpy`, which reduced its percentage of execution time from 50% to 10%. However, this had essentially zero impact on the application’s frame rate. This is interesting because it implies that on heterogeneous applications, statistically significant information such as execution frequency may have no bearing on

other performance indicators such as execution time.

III. COMBINING SAMPLING AND INSTRUMENTATION

While the two sampled profiles provide useful information about each resource, the usefulness is limited due to the difficulty of aligning the two streams of data. Since this difficulty arises due to the infrequent occurrences of samples in functions that provide a good point for alignment, one possible solution is to force samples to occur in functions of interest. However, that would create an inconsistent sense of time between samples. Instead, another straightforward solution is to augment the data in a way that maintains the sampling rate while also providing useful points to align the two profiles.

We accomplish this by first defining a list of functions of interest that we can use to correspond GPU and CPU behavior. This list primarily consists of functions from the APIs for interacting with the GPU, such as CUDA library calls, OpenCL library calls, and OpenGL library calls. Such a list corresponds particularly well to the GPU performance information presented by NVIDIA’s Compute Visual Profiler, which automatically provides a GPU performance sample for each such call.

In some cases, an extra translation step is required to match the CPU markers to the functions reported in the GPU profile. For instance, the CUDA API only has a function `cudaMemcpy`, which is used to initiate transfers between main system memory and GPU memory. One parameter to that function is an enumeration that specifies the source and destination of the memory copy. When collecting the GPU profile from CVP, the original function name is replaced with a name specific to the transfer direction, such as `memcpyHtoD` when copying from the host to the device, or `memcpyDtoH` when copying from the device back to the host. This is shown for a CUDA “Hello World” program in Figure 1.

In addition to API-level calls, we can extend the list of interesting functions to include other functions that indicate activity between the CPU and GPU but that are implicitly called. For instance, in the user-level code for the CUDA “Hello World” application, the kernel that computes the message is essentially launched as a regular function with some extra syntax related to mapping the computation to the hardware. While CVP reports the name of the function, the kernel is launched implicitly using `cudaLaunch`. Furthermore, as shown in Figure 1, the kernel invocation is preceded by `cudaConfigureCall` and two invocations of `cudaSetupArgument` for the two arguments to the function that computes the message. An OpenCL “Hello

GPU	CPU
-----	<code>__cudaRegisterFatBinary</code>
-----	<code>__cudaRegisterFunction</code>
-----	<code>cudaMalloc</code>
-----	<code>cudaMalloc</code>
<code>memcpyHtoD</code>	<code>cudaMemcpy</code>
<code>memcpyHtoD</code>	<code>cudaMemcpy</code>
-----	<code>cudaConfigureCall</code>
-----	<code>cudaSetupArgument</code>
-----	<code>cudaSetupArgument</code>
<code>hello</code>	<code>cudaLaunch</code>
<code>memcpyDtoH</code>	<code>cudaMemcpy</code>
-----	<code>cudaFree</code>
-----	<code>__cudaUnregisterFatBinary</code>

Fig. 1. Function calls reported by NVIDIA’s Compute Visual Profiler and using our API-based marking on a CUDA hello world application. Several extra functions are called implicitly in the CPU code when setting the system up to run a kernel on the GPU. The GPU profiler reports more specific names for some function calls, such as to indicate the source and destination of `cudaMemcpy`.

World” application exhibits the same behavior, calling some additional functions and calling other functions multiple times; ultimately, the running application has more than twice as many API calls as the static code. Capturing such additional functions is important because they could impact the overall performance but are not directly reported by monitoring either the GPU activity or the functions called directly by the application.

Simply defining the list of functions is insufficient. While it gives a more explicit list of all the functions that are called and may uncover functions that were not explicit in the application code, it still does not specifically guarantee that we can align those function calls with the CPU samples and therefore does not guarantee that we can use the function list to align CPU performance with GPU performance. To use the function list to align the two profiles, we need a consistent notion of time. We can not use the GPU time because even if the GPU has a readable time stamp, it is unlikely to be synchronized with the CPU or to proceed at the same rate. Instead, we read the CPU’s time stamp counter using the `rdtsc` instruction both at each performance counter sample and each time we encounter a function on our list. This allows us to place the function occurrences within the list of samples, and thus to see the CPU performance leading up to execution on the GPU.

Since we generally can not modify the source of the drivers and libraries to collect these time stamps, we turn

to dynamic binary instrumentation. Binary instrumentation allows us to modify the instruction stream of the running application without needing access to the source and without having to recompile prior to execution.

We use Pin [10] to perform our instrumentation because of its flexible instrumentation API. In particular, its ability to instrument in “probe” mode instead of just-in-time (JIT) compiling the code is important for our approach. JIT compilation in binary translators introduces overhead due to recompiling every instruction encountered. As we will show in the next section, adding too much overhead can potentially affect the relationship between profiled execution and unprofiled execution, especially when multiple execution units are involved. One example of this is if commands are being queued, such as submitting several computational kernels or a long series of graphics commands, where the queue stays non-empty and the GPU stays busy in an unprofiled execution but stalls due to overhead in a profiled execution.

Pin’s probe mode instead primarily executes the native code except for the places instrumentation is added. Instrumentation is performed by relocating a few instructions, replacing them with a jump to the instrumentation, and then the instrumentation is followed by a jump to the relocated code and finally a jump back to the original function for the remainder of its code. This has a much lower overhead, but at the cost of significantly limiting the locations that instrumentation can be added to the beginning and end of functions. However, this is sufficient in practice for our approach, because we only need to instrument the beginning of the function with code to read the time stamp and record the time and function name to a file.

While the main idea of inserting function markers with binary instrumentation is straightforward, there are some issues that arise. One is dealing with multiple executions. As noted previously, GPUs can not be interrupted during execution, and performance counters have to be read once execution of some unit of computation is completed. Another side-effect of this is that we can not multiplex counters—running for a subset of the time with different sets of counters and extrapolating—if we want to collect results from more counters than can be used during a single execution. Both the Compute Visual Profiler for CUDA and OpenCL on NVIDIA GPUs and the GPUPerfAPI for OpenCL on AMD GPUs support the notion of multiple passes to collect all of the counters, although they approach the problem in different ways. CVP automates performance counter collection externally to the program being profiled, and

performs multiple passes by simply executing the application multiple times. On an NVIDIA Tesla C2050, 12 executions are required to collect all of the performance counters, collecting a few counters on each pass. The GPUPerfAPI requires performance counter reading to be managed inside the application and provides API calls to define multiple passes, with the caveat that the user must ensure that the same code is executed during each pass to collect consistent results. On the AMD Radeon 5870, three passes are required to collect all of the performance counters. Alternately, a user collecting a profile can manually run the complete application multiple times rather than making sure certain sections recompute exactly the same data during one execution.

In either case, multiple executions leads to multiple CPU profiles that must also be aligned. One simple approach in the case of running the application multiple times is to use some consistent notion of time for sampling such that the CPU samples will naturally fall in the same place, as long as the CPU code is deterministic. We accomplish this by sampling based on the instruction count rather than the cycle count or a timer. Although CPU overflow sampling is imperfect and the number of instructions between samples will vary within a few instructions of the target overflow amount, this provides a more stable metric than CPU cycles and ensures that approximately the same work happened during a given sample across multiple executions. We also normalize the time stamps based on `main` as a fixed point in execution. In the case of multiple passes within a single execution when using the GPUPerfAPI, we can take advantage of the fact that we must modify the code and manually turn CPU sampling off and on around extra passes.

IV. EXAMPLE COMBINED PROFILES

We show as a simple example a combined profile from an OpenCL application. We use a Monte Carlo simulation for Asian option pricing, included with the ATI Stream SDK. Each iteration of application’s main loop enqueues 10 kernels, where in each kernel each GPU thread calculates a series of values based on different possible stock parameters. For each kernel invocation, the computational kernel is enqueued, the CPU waits for completion, a read command is enqueued to read the results, and the CPU again waits for completion before beginning the next iteration.

Figure 2 shows the CPU performance and a set of markers for the Monte Carlo simulation with the default parameters, which executes one iteration of the main

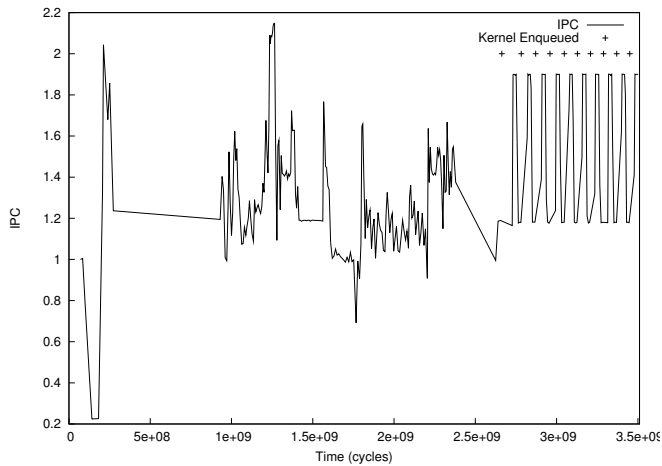


Fig. 2. Sample combined profile for an OpenCL Monte Carlo simulation. Roughly 70% of execution is spent in startup in this case, and performance is very regular while enqueueing kernels to execute on the GPU.

loop on the CPU and therefore launches 10 simulations on the GPU. Here we only include markers for calls to `clEnqueueNDRangeKernel`. We enabled GPU counters for the entire loop, so our GPU profile represents all 10 executions of the computational kernel. For our GPU profile, we read the performance counter that measures the percentage of time the GPU is busy. From this visualization, we can see approximately 70% of the execution time is spent in startup before any computational kernels are executed. In the performance results section, we extend the execution of our benchmarks so that most of the execution time reflects the actual benchmark and not startup time.

The GPU profile shows that the GPU is only busy 29% of the time across the 10 executions of the kernel. However, we see that CPU performance is very regular between kernel invocations, with a lower IPC section when enqueueing the kernel and waiting on completion and a higher IPC section when reading the results and initializing the next execution. Since no one iteration of the loop is significantly slower, this implies that a performance improvement would require either improving the performance of the loop as a whole, or performing more work per kernel invocation and decreasing the number of times the kernel needs to be executed.

V. PERFORMANCE ANALYSIS

Now that we have discussed the issues relating to the implementation of aligning profiles for multiple resources in a heterogeneous system, we consider the performance impact of our technique. We will compare

the overhead of both JIT and probe mode instrumentation in terms of wall-clock time for the entire application as well as the impact on GPU kernel performance.

We use two different machines for our experiments to cover two main vendors of GPU hardware and the two main ways to write heterogeneous GPGPU applications. For CUDA applications, we use a 3.2GHz Core 2 Quad with an NVIDIA Tesla C2050 and the CUDA toolkit version 3.2. For OpenCL applications, we use a 2.6GHz quad-core Core i7 with an AMD Radeon 5870 using the AMD Stream SDK version 2.3. We collect performance counters on the Tesla C2050 using the Compute Visual Profiler version 3.2, and we collect performance counters on the Radeon using GPUPerfAPI version 2.5. Function markers are inserted by instrumenting our application with Pin [10] version 2.9, kit 39599. CPU performance counters are accessed using `perf events` [11], available in the Linux kernel since 2.6.32, and `libpfm4` [12].

We insert additional code to read CPU performance counters as necessary, and also insert code to read GPU performance counters on OpenCL applications being profiled with the GPUPerfAPI. In many CPU-only cases, tools based on `libpfm4` can fork and read performance counters for another application without requiring recompilation of the profiled application. However, this is not currently possible when profiling CUDA applications with CVP, because CVP will not follow the fork and thus does not ever see any CUDA calls to monitor. Using GPUPerfAPI requires modifying the application to determine where to start and stop counters, but more importantly, it requires a handle to the OpenCL device being used by the application, which can not be determined prior to execution.

A. OpenCL Applications

We test performance on several sample computational kernels included with the Stream SDK. These include Fast Fourier Transform (FFT), Floyd-Warshall for calculating the shortest path over a randomly-generated graph, Mersenne Twister for generating Gaussian random numbers, single- and double-precision Monte Carlo simulations for Asian Option pricing, a recursive Gaussian image filter, and a simple convolution filter. Because many of these kernels are short and most of the time for execution of a single iteration is spent in the CPU code setting up execution, we configure them to use enough iterations to have an approximate native run time of 30 seconds.

Figure 3 shows the overhead of collecting the time

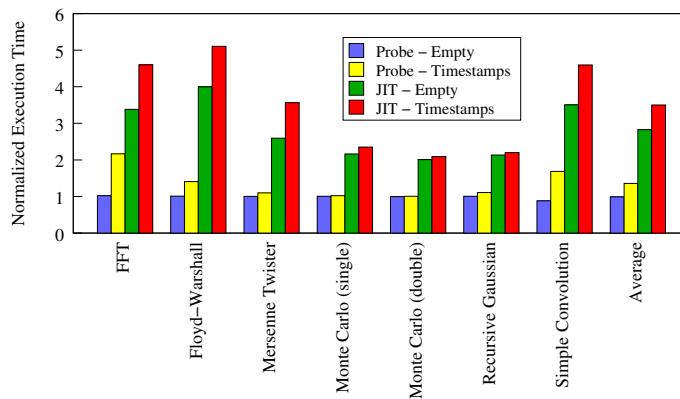


Fig. 3. Profiling overheads on OpenCL kernels, averaged over three executions and normalized to native. Lower is better. In general, probe mode has a low overhead, while JIT mode has a more noticeable performance impact. Collecting the timestamps leads to a larger overhead; although the two versions collect the same information, performance degradation is higher with JIT mode.

stamps on our OpenCL applications, using both methods of instrumentation. We also present just the overhead of using the binary translator as a point of reference. The overheads are reported as the average of three executions and then normalized to native. Probe mode on its own gives us less than 2% overhead, which is essentially within the noise of different execution times natively. The overheads for actually collecting the time stamps range depending on the number of iterations required and the number of API calls being used. For instance, for a native run time of 30 seconds, FFT, which has the highest overhead, requires 50,000 iterations and makes approximately 1.4 million calls to OpenCL API functions. Compare this to the single-precision Monte Carlo, which requires only 80 iterations and just under 17,000 API calls. JIT mode has a much higher baseline overhead. Both versions record instrumentation in the same way—by checking each function name when a binary object is loaded—but JIT mode also has a greater increase in overhead for recording the time stamps due to other details of JIT mode instrumentation.

Another limitation of JIT mode, beyond the larger overhead, is its impact on sampling the CPU performance counters. We enable the CPU performance counters within the application. Pin delivers the interrupt when the performance counter triggers a sample, but there are two potential issues with using this sample to determine application performance. First, because Pin in JIT mode executes a large number of instructions, many of the samples may be within Pin itself rather than within the application. Second, Pin does not translate the instruction pointer returned with the sample, which means even if the sample occurs during application

TABLE I
COMPARISON OF APPLICATION AND KERNEL EXECUTION TIMES FOR 45000 ITERATIONS OF MATRIX MULTIPLICATION. ALL TIMES ARE IN SECONDS. ALTHOUGH JIT MODE INTRODUCES A 5% OVERALL OVERHEAD, TOTAL KERNEL PERFORMANCE IS DEGRADED BY LESS THAN 1%.

	Native	Probe	JIT
App time	400.95	401.84	424.09
Kernel time	399.76	400.14	401.39

code instead of Pin code, the sample IP refers to the JITted version and not the original and so we can not map the sample back to the original application code. Since we are sampling based on the instruction count, one potential solution would be to put the performance counters in counting mode and have Pin keep track of the application instruction count and read the performance counters when appropriate. However, this is unsuitable because the performance counters values would then reflect n application instructions plus an unknown number of instructions executed by Pin.

Overhead in terms of wall-clock time does not necessarily preclude us from using JIT mode, which has benefits of more flexibility in the instrumentation. If the overhead is only affecting the CPU code but does not degrade performance on the GPU, then we can invent a multi-pass method to overcome the limitations profiling the CPU portion of the code in JIT mode. On some applications, the difference in kernel execution time and GPU performance is negligible. For instance, Table I compares the application and kernel execution times for a simple matrix multiplication kernel. Although the wall-clock time degrades by 5% for JIT mode, the kernel time

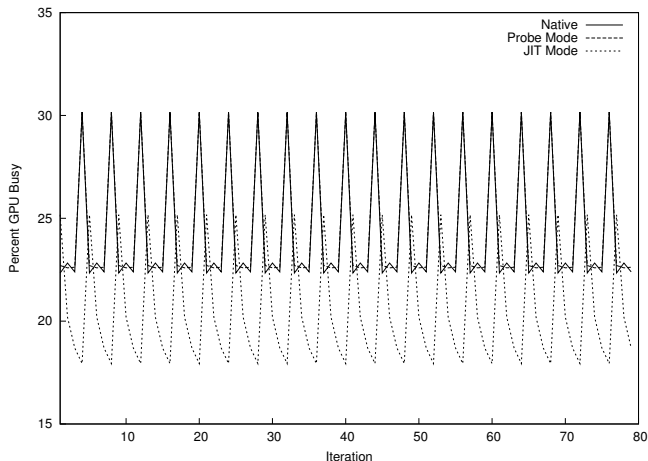


Fig. 4. Comparison of GPU usage on the single-precision Monte Carlo simulation natively, with time stamps collected in probe mode, and time stamps collected in JIT mode. JIT mode on the CPU degrades kernel performance on the GPU by about 5%, as compared to 0.2% for probe mode.

degrades by less than 1%, and the percentage of time the GPU is busy based on GPU performance counters is essentially unchanged.

However, on other applications and particularly in the case of fewer iterations, JIT mode has a higher impact. On the same matrix multiplication kernel, with only 30 iterations, the percentage of busy GPU time across all iterations decreases from 98% to just 60%. As another example, Figure 4 shows the percentage of time the GPU is busy for each of 80 iterations on the single-precision Monte Carlo kernel. Recall from Figure 3 that probe mode only slowed down performance by 2% on this benchmark, while JIT mode led to a 2.3x overhead. We see in this figure that in addition to any overhead on the user-level code, JIT mode is decreasing the GPU utilization by roughly 5%, while probe mode only decreases GPU usage by 0.2%. JIT mode affects the transfer of data to the card, which leads to lower utilization.

B. CUDA Applications

For CUDA, we use a selection of benchmarks from the Rodinia benchmark suite [13]. We use Kmeans, a data mining clustering algorithm, Heart Wall and Leukocyte, medical imaging algorithms, SRAD, an image processing algorithm, backprop, an artificial neural network, and Hotspot, a physics simulation. With the exception of Heart Wall, we again increased the iteration count to achieve native run times in the range of 30 seconds. Heart Wall, which tracks the movement of a mouse heart

in a video, is limited to the 104 frames in the input set and has a native execution time of roughly five seconds.

Figure 5 compares the overhead of collecting profiles and inserting markers on our CUDA applications. As on the OpenCL benchmarks, probe mode tends to have a smaller impact on performance than JIT mode. Due to differences in the two application sets and implementations, the average overhead on JIT mode is lower on these benchmarks than on the OpenCL benchmarks. The faster-than-native execution times on Kmeans and the lower overhead for collecting time stamps on Backprop in probe mode are a result of variations in the short execution times.

Comparing the relationship between the bars, the clear outliers are SRAD and Hotspot. The trends are the same on these two benchmarks, in that collecting the function time stamps is slower than the “empty” version, and JIT mode is slower than probe mode. The overhead of collecting function time stamps on those two benchmarks is much higher because they result in several orders of magnitude more API calls than the other benchmarks; for instance, SRAD has over 5.2 million API calls, compared to just over 19,000 on Leukocyte.

In addition to the decreased impact on wall clock time, JIT mode also appears to have a decreased impact on kernel performance on CUDA applications. CVP does not offer a way to directly calculate the percentage of time the GPU is busy, but it does record the total GPU time for all invocations of a particular kernel or memory transfer function. Comparing these numbers, the execution times on probe mode and JIT mode both vary within 1% of the native time.

VI. RELATED WORK

Tools and interfaces designed for profiling GPU and GPGPU applications generally focus on profiling resources in isolation, specifically on profiling the GPU portions of the application and assuming the CPU portion will be profiled using other tools. Examples include Graphic Remedy’s gDebugger [2], NVIDIA’s nvPerf [3] and Compute Visual Profiler [6], and AMD’s GPUPerfAPI [4]. Coutinho *et al.* instrumented the PTX bytecode for CUDA to collect additional GPU profiling information [14], [15].

Some recent works consider the issue of providing CPU context when profiling the GPU. Gordon *et al.* discuss using VTune to profile the CPU part of a GPGPU application but do not specifically align the profiles [16]. Graphics analysis tools such as GPUView [5] and Intel’s Graphics Performance Analyzers [7], [8]

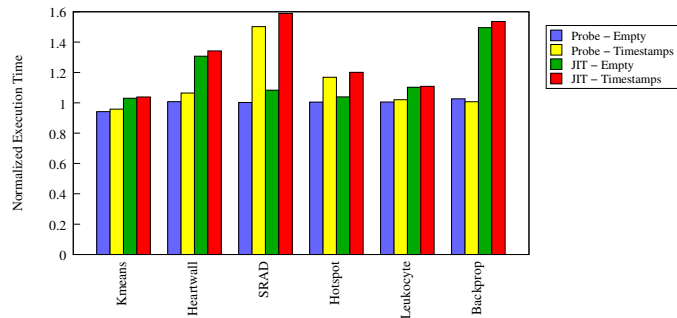


Fig. 5. Profiling overheads on CUDA kernels, averaged across three executions and normalized to native. Lower is better. As on OpenCL benchmarks, probe mode tends to have a lower overhead than JIT mode.

provide some cross-resource analysis by determining whether the workload is CPU- or GPU-bound and then making determinations of GPU performance based on CPU-level information, rather than combining directly-obtained GPU performance information with the CPU-level information.

There are relatively few heterogeneous mainstream heterogeneous systems other than systems combining a CPU and GPU. One other notable heterogeneous system is the Cell Broadband Engine, which consists of a main PowerPC core and several special SIMD cores. Several methods are suggested for profiling Cell applications, including time-based studies [17], observing performance at the instruction level [18], using `gprof` [19], and using various tools in the Cell SDK that focus on individual resource types [20]. These methods are all either high-level or focus on an individual resource, as opposed to providing detailed information about the interactions of resources in the system as a whole.

Another example of profiling on heterogeneous systems is a profiling system for a heterogeneous FPGA multicore system developed by Nunes [21]. The profiling in that work focused on MPI calls between the cores, without considering the rest of the application, and used additional hardware to reduce the overhead of data storage.

VII. CONCLUSIONS AND FUTURE WORK

This paper has described a method for leveraging binary instrumentation to combine profiles across multiple resources on a heterogeneous system. We first showed that simply collecting CPU and GPU data in isolation will not necessarily lead to data that is easy to combine, and that performance indicators useful for CPU-only code such as percentage of execution time do not always act as performance indicators in heterogeneous systems. We then described using binary instrumentation to insert

markers for function names and time stamps that allow the user to create specific points for aligning multiple profiles. We described several issues that need to be addressed with this technique, such as dealing with overhead constraints and the need for multiple passes to collect GPU performance counters. We then compared the overhead on both OpenCL and CUDA benchmarks, showing that with lightweight instrumentation based on Pin’s probe mode, we can achieve roughly 40% overhead on the application as a whole and 1% degradation on the GPU kernels.

Future work consists of looking at other profile data that can be collected and combined to continue to provide a more complete picture of performance on heterogeneous systems. This may require further work on combining profile data from multiple passes and accounting for differences in overhead. For instance, a data flow analysis to show how data moves through CPU and GPU code could provide more information about how the two computational resources impact one another, but probe mode is ill-suited for such fine-grained analysis. Further work in providing full-system profiles on heterogeneous applications will enable developers to maximize the potential of modern heterogeneous systems.

REFERENCES

- [1] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, “Cell Broadband Engine Architecture and its first implementation—A performance view,” *IBM Journal of Research and Development*, vol. 51, no. 5, September 2007.
- [2] Graphic Remedy, “gdebugger,” <http://www.gremedy.com>.
- [3] J. Kiel and D. Cornish, “Optimize Your GPU with the Latest NVIDIA Performance Tools,” SIGGRAPH 2006 Tutorial, Boston, MA, Boston, MA, August 2006.
- [4] Advanced Micro Devices, Inc., *AMD GPU Performance API*. Advanced Micro Devices, Inc., October 2010.
- [5] M. Fisher and S. Pronovost, “Gpview,” <http://graphics.stanford.edu/mdfisher/GPUView.html>.

- [6] NVIDIA, *Compute Visual Profiler*. NVIDIA Corporation DU-05162-001_v02, October 2010.
- [7] C. Cormack, "Optimizing Game Engines with the New Intel®Graphic Performance Analyzers 3.0 Platform View," White Paper, Intel Corporation, 2010.
- [8] S. Guo, P. Gerasimov, and B. Aona, "Practical Game Performance Analysis Using Intel®Graphics Performance Analyzers," White Paper, Intel Corporation, 2011.
- [9] A. Gharaibeh and M. Ripeanu, "Size matters: Space/time trade-offs to improve gpgpu applications performance," in *Conference for High Performance Computing, Network, Storage, and Analysis*, New Orleans, LA, November 2010.
- [10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Conference on Programming Language Design and Implementation*, Chicago, IL, USA, 2005.
- [11] Linux, "perf - Performance analysis tools for Linux," 2009.
- [12] S. Eranian, "libpfm4," <http://perfmon2.sourceforge.net/>.
- [13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Symposium on Workload Characterization*, Austin, TX, October 2009.
- [14] B. Coutinho, G. Teodoro, R. Sachetto, D. Guedes, and R. Ferreira, "Profiling general purpose gpu application," in *Symposium on Computer Architecture and High Performance Computing*, Sao Paulo, Brazil, October 2009.
- [15] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr., "Performance debugging of gpgpu applications with the divergence map," in *Symposium on Computer Architecture and High Performance Computing*, Rio de Janeiro, Brazil, October 2010.
- [16] B. Gordon, S. Sohoni, and D. Chandler, "Data handling inefficiencies between cuda, 3d rendering, and system memory," in *Symposium on Workload Characterization*, Atlanta, GA, December 2010.
- [17] V. Sachdeva, M. Kistler, E. Speight, and T.-H. K. Tzeng, "Exploring the Viability of the Cell Broadband Engine for Bioinformatics Applications," *Parallel Computing*, vol. 34, no. 11, 2008.
- [18] D. A. Bader and V. Agarwal, "FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine," in *Conference on High Performance Computing*, Goa, India, December 2007.
- [19] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos, "Dynamic Multigrain Parallelization on the Cell Broadband Engine," in *Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, March 2007.
- [20] S. Koranne, *Practical Computing on the Cell Broadband Engine*. Springer US, 2009.
- [21] D. P. Nunes, "A Profiler for a Heterogeneous Multi-Core Multi-FPGA System," Master's thesis, University of Toronto, Toronto, Ontario, 2008.