Efficient Instrumentation of GPGPU Applications Using Information Flow Analysis and Symbolic Execution

Naila Farooqui, Karsten Schwan, and Sudhakar Yalamanchili
College of Computing, School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
naila@cc.gatech.edu, schwan@cc.gatech.edu, sudha@ece.gatech.edu

ABSTRACT

Dynamic instrumentation of GPGPU binaries makes possible real-time introspection methods for performance debugging, correctness checks, workload characterization, and runtime optimization. Such instrumentation involves inserting code at the instruction level of an application, while the application is running, thereby able to accurately profile data-dependent application behavior. Runtime overheads seen from instrumentation, however, can obviate its utility. This paper shows how a combination of information flow analysis and symbolic execution can be used to alleviate these overheads. The methods and their effectiveness are demonstrated for a variety of GPGPU codes written in OpenCL that run on AMD GPU target backends. Kernels that can be analyzed entirely via symbolic execution need not be instrumented, thus eliminating kernel runtime overheads altogether. For the remaining GPU kernels, our results show 5-38% improvements in kernel runtime overheads.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Heterogeneous (hybrid) systems; D.3.4 [**Programming Languages**]: Retargetable compilers; D.3.4 [**Programming Languages**]: Run-time environments; D.4.8 [**Operating Systems**]: Measurements

General Terms

GPU Computing, Instrumentation, Dynamic Binary Compilation

Keywords

OpenCL, CUDA, GPGPU, Rodinia

1. INTRODUCTION

Dynamic binary instrumentation is a technique that inserts additional code into the existing execution path of an application, by modifying the application's binaries via an instrumentation tool. Such instrumentation offers opportunities to observe and/or improve an application's behavior at runtime. For example, code can be inserted to perform correctness checks, such as memory bounds checking, improve an application's performance via runtime code specialization, and/or characterize the application's memory and compute usage to drive runtime resource management policies.

Recent research has focused on dynamic instrumentation techniques for data-parallel applications on GPGPU execution environments [11, 12]. Our previous work, GPU Lynx [12], for instance, is a dynamic instrumentation engine that enables the construction of customizable program analysis methods for data-parallel applications executing on GPU backend targets. It uses just-in-time (JIT) compilation to construct control-flow graphs of GPU kernels on-the-fly, enabling sophisticated dataflow analyses and insertion of customized instrumentation code at the kernel, basic-block, and/or instruction levels.

Although dynamic instrumentation is known to be a powerful program analysis technique, its drawback is the potential overhead incurred in the program's critical execution path. Depending on the complexity of the instrumentation and the nature of the program being profiled, these overheads may obviate the utility of instrumentation and/or may sufficiently perturb the program to mask its undesirable or erroneous behavior. It is important, therefore, to reduce these overheads as much as possible.

GPU Lynx presently supports instrumentation of CUDA [21] applications on the NVIDIA GPU backend. This paper extends Lynx to enable instrumentation of OpenCL [15] applications running on AMD GPU backends, and it uses two techniques to substantially reduce the runtime overheads associated with dynamic instrumentation: (i) information flow analysis and (ii) symbolic execution. Specifically, information flow analysis is used to track causal dependencies introduced through control and data flow, prior to a GPGPU kernel's execution. We then use these dependencies to identify program points that exhibit divergent control flow. This helps us isolate the parts of the kernel that potentially exhibit thread divergence, and thus require runtime monitoring to precisely characterize the kernel's execution. For the parts of the program where there is no thread divergence, we can instead proceed to analyze the kernel with symbolic execution, removing the need for instrumentation and thus, eliminating its runtime overheads. In summary, using a combination of these program analysis techniques, we can selectively apply instrumentation at finer-grained program points, achieving significant gains in the overall runtime performance of instrumented kernels.

Performance improvements obtained with the use of these techniques are substantial. For some applications, kernel runtime overheads are eliminated altogether, while for oth-

GPGPU-7, March 01 2014, Salt Lake City, UT, USA

ers, such overheads are reduced from 5-38%, depending on the extent of divergent code present in the kernels.

The remainder of the paper is organized as follows. Section 2 provides background on GPU instrumentation. Section 3 describes the design and implementation of our instrumentation framework, as well as how it uses information flow analysis and symbolic execution to instrument GPU programs. Experimental results appear in Section 4, future work is discussed in Section 5, related work is summarized in Section 6, and conclusions are presented in Section 7.

2. GPU INSTRUMENTATION

Given the Single Instruction Multiple Data (SIMD) execution model of GPU computing, instrumenting GPU applications has its own unique considerations. This section offers a brief overview of the GPU execution model, the overall goals of instrumenting GPU kernels using our framework, and a description of a specific class of GPU metrics useful in characterizing GPU utilization and efficiency. For this study, we have implemented two such metrics: activity factor and memory intensity, defined below.

2.1 GPU Execution Model

A GPU primarily consists of a scalable array of SIMD cores with distinct on-chip and off-chip memory spaces. Computations are performed by a tiered hierarchy of threads. The smallest schedulable unit is called a *wavefront* (or a *warp* in NVIDIA's terminology). Threads within a wavefront execute the same instruction in lockstep fashion. Wavefronts are further divided into *thread blocks*. Threads belonging to the same block are mapped to a single SIMD core, and can communicate through a faster access, on-chip, local memory. Collection of blocks form a *grid*, and threads from different blocks can only communicate via global memory, which is off-chip and has the highest latency in the memory hierarchy.

2.2 Goals of Instrumentation

The overall goal of our instrumentation framework is to enable user-defined, custom instrumentation of GPGPU applications on various backends. As noted earlier, the existing GPU Lynx instrumentation engine enables the instrumentation of CUDA applications on the NVIDIA GPU backend. In this work, GPU Lynx extends support for instrumenting OpenCL applications on the AMD GPU backend, providing the unique and desirable capability to allow user-defined instrumentation routines to be inserted at runtime for AMD backends. The current implementation supports this functionality by allowing end-users to write instrumentation code in AMD's intermediate representation, known as AMD IL [3], with the plan to provide a higher-level programming abstraction for HSA IL [16], the next generation intermediate representation for future AMD GPU systems.

2.3 Wavefront-Level Instrumentation

Wavefront-level instrumentation focuses on the behavior of a wavefront rather than the independent behavior of each of its threads. Since a wavefront is the smallest schedulable unit on a GPU, such instrumentation is useful in characterizing SIMD utilization and overall GPU efficiency. For instance, when threads within a wavefront diverge, the wavefront serially executes each branch path taken, disabling threads that are not on that path. This imposes a significant penalty on GPU kernels with heavy non-uniform control-flow. Wavefront-level instrumentation enables the characterization of control-flow as well as memory irregularity in GPU applications. Additionally, many counting-based metrics, such as the total number of dynamic instructions, the ratio of memory operations to total instructions, and the ratio of arithmetic operations to total instructions, can be implemented as wavefront-level instrumentations. In this work, we focus on the use of program analysis techniques to reduce the overheads of this class of GPU metrics. Note that this does not limit the scope of metrics that can be defined using the GPU Lynx engine, but the techniques for reducing overheads via symbolic execution described in this work are applicable to the specific class of warp-level, counting-based GPU metrics.

2.4 GPU-Specific Metrics

Two wavefront-level GPU metrics that we have implemented are *activity factor* and *memory intensity* [17].

Activity Factor: This metric attempts to characterize how well an application is utilizing the GPU SIMD parallel execution model. It is computed as the ratio of the actual number of active threads to the maximum possible number of active threads within an application. If an application has no branch divergence and/or uniform control flow, all instructions in the application will be executed by all of the threads, and the application will exhibit 100% activity factor. Applications with low activity factor are considered to exhibit a higher degree of control-flow irregularity. Note that a 100% activity factor does not indicate the non-existence of control-flow instructions, but instead indicates that the decision to take or not take a data-dependent branch is uniform for all threads within a wavefront.

The method for computing the activity factor metric is shown in Figure 1. The method maintains two global memory counters per wavefront, one to keep track of dynamic instructions executed by only *active* threads in a given wavefront, and the other to keep track of dynamic instructions executed by *all* threads in a given wavefront. Upon each basic block entry, it computes the number of active threads in a given wavefront. In order to ensure that a single thread in each wavefront updates the two global memory counters, it compares the current thread index with the smallest index of all active threads. This thread then increments the two global counters described above. After kernel execution, it then aggregates this data for all wavefronts executing a particular GPU kernel, thus determining the final activity factor as the ratio of dynamic instructions executed by all active threads over the dynamic instructions executed by all threads. If this ratio is one, the kernel has uniform control flow and therefore, exhibits no thread divergence.

Memory Intensity: Memory intensity is the ratio of dynamic global memory instructions to the total dynamic instructions in a given application. The implementation of this metric follows a similar pattern as the activity factor metric, with a single thread in each wavefront updating counters to accumulate the total number of dynamic memory instructions over all dynamic instructions for each wavefront. This metric attempts to characterize the extent to which an application is memory-bound versus compute-bound. Its current implementation focuses only on global memory operations since global memory can be 150x slower than the on-chip, local memory, thus being the most significant contributor to performance degradation in memory-bound GPU codes. Note that memory intensity alone is not a strong predictor of pure bandwidth performance, how-

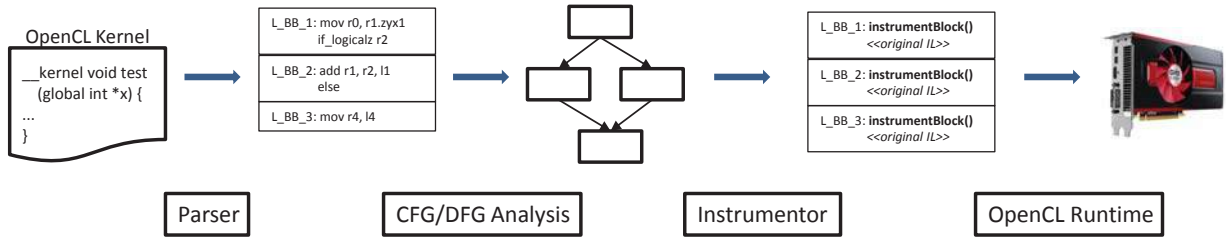

Figure 1: Instrumentation Framework

```
for every basic block do
    if current thread index = smallest index of all active
    threads in wavefront then
        /* update global memory counter for active
        threads in wavefront executing this basic block */
        globalMem[wavefrontId * 2] += (active threads
        in wavefront * instructions per basic block)

        /* update global memory counter for all threads
        in wavefront executing this basic block */
        globalMem[wavefrontId * 2 + 1] +=
        (total threads in wavefront * instructions per
        basic block)
    end
end
```
Figure 2: Activity Factor Algorithm

ever, because it neither takes into account the irregularity in memory access patterns nor the GPU cache behavior. Nonetheless, it is a useful indicator of the extent to which an application is memory- vs. compute-bound.

2.5 IR Considerations

As described earlier, wavefront-level instrumentations focus on the behavior of a wavefront versus independent threads. Therefore, in order to compute such instrumentations, the IR must either expose special instructions to obtain wavefront-level information, or our instrumentation code must construct data structures to accumulate data for all threads within each wavefront. In the case of the AMD IL, no such special instructions are available. As a result, in our instrumentation, we make use of atomic operations on local memory and memory barriers to aggregate this data for each wavefront. Specifically, each thread in a wavefront increments a counter atomically in order to compute the total number of active threads. And we select a single thread in each wavefront by using an atomic *test and set* instruction. We believe that the use of atomics and memory fences, applied on each basic block, significantly increases the overhead associated with this kind of instrumentation. However, this is strictly an implementation-specific overhead. AMD IL is now deprecated and future AMD GPU systems will be HSA (Heterogeneous System Architecture)-compliant. The HSA IL specification [16] includes special cross-lane instructions, which perform work across lanes in a wavefront, to expose this kind of information. These include `countlane`, `countuplane` and `masklane`, where the first one returns the number of active threads in the current wavefront, the second one computes a prefix-sum of the number of executing threads in the current wavefront, and the last one returns a bit mask where all active threads in the current wavefront have a non-zero value. By using these cross-lane primitive operations, we expect to see a significant overall improvement in wavefront-level instrumentation-related overheads.

That said, the program analysis techniques we describe in this paper to reduce overheads are orthogonal to implementation specific instrumentation and IR considerations. The focus of the techniques presented in this paper is to perform instrumentation at finer-grained program points, by symbolically evaluating the GPU kernels prior to inserting instrumentation. Depending on the control-flow structure of the GPU workloads, these techniques reduce the extent of dynamic instrumentation code that needs to be applied to the kernels in the first place. As such, these techniques will apply irrespective of the implementation-specific details of the underlying IR and/or the target backend, equally relevant to both NVIDIA and AMD GPU systems.

3. DESIGN AND IMPLEMENTATION

3.1 Instrumentation Framework

GPU Lynx [12] is a dynamic instrumentation engine for data-parallel applications on GPU architectures. Lynx allows the creation of customized, user-defined instrumentation routines that can be applied transparently at runtime for a variety of purposes, including performance debugging and correctness checking. When built as a library, Lynx can be linked with any runtime. The Lynx framework includes a default implementation of the NVIDIA CUDA [21] runtime to support the execution of CUDA applications on NVIDIA GPU devices. It provides a parser and IR abstraction for extracting and generating NVIDIA's parallel thread execution (PTX) from the compiled CUDA fat binary. An essential feature of GPU Lynx is its flexibility and customizability, enabling the creation of sophisticated control-flow and dataflow analyses from its IR abstraction.

This paper describes new functionality in the GPU Lynx framework that adds efficient instrumentation support for OpenCL [15] applications running on AMD GPU backends. Efficiency is obtained by combining instrumentation with static data flow analysis and symbolic execution, to instrument only where instrumentation is actually needed to capture dynamic behavior. Instrumentation, then, is carried out using a parser and an IR abstraction for the AMD IL [3]. Further, such instrumentation is made *transparent* to the application, without the need to modify application code, via an implementation of the OpenCL runtime linked with the AMD GPU Lynx library.

The current GPU programming model necessitates explicit data transfer between the CPU and GPU, since GPU applications are hosted by CPU threads. This model applies to instrumentation, as well. Not only must instrumentation

code be inserted into GPU kernels, but the instrumentation data structures must also be copied over to the GPU prior to kernel execution, and results must be copied back after execution. The GPU Lynx library, therefore, exposes three APIs to the end-user: `instrument`, `initializeInstrumentation`, and `finalizeInstrumentation`. The `instrument` method parses the AMD binary, extracts the AMD IL, constructs a control-flow graph from the IL, and optionally performs dataflow analyses on the control-flow graph. It then applies an instrumentation pass to the control-flow graph, inserting instrumentation code into the original GPU kernel. Finally, it generates an updated AMD IL binary from the instrumented code, which is executed on the AMD backend. Figure 2 illustrates the interaction between various components of our framework to perform the `instrument` step. GPU Lynx has two additional APIs, `initializeInstrumentation` and `finalizeInstrumentation`, which are used for the runtime management of instrumentation data. The `initializeInstrumentation` method is responsible for allocating necessary instrumentation data structures on the GPU device prior to kernel execution, and the `finalizeInstrumentation` call extracts instrumentation results and cleans up such data structures after kernel execution.

3.2 Dataflow Analysis & Symbolic Execution

This section describes the dataflow analysis used in our framework to reduce the runtime overheads associated with instrumentation, specifically with wavefront-level instrumentation. The primary insight used is that with wavefront-level metrics, it is critical to identify the program points where thread divergence occurs. This is because in general, basic blocks that exhibit uniform control flow can be easily and more efficiently analyzed via symbolic execution, thus obviating the need to instrument them. Note, however, that the ability to replace instrumentation with symbolic execution depends on the nature of the instrumentation being performed in the first place. For instance, if a particular instrumentation depends on the values loaded from memory, which may differ across threads within the same warp and are not known during the just-in-time analysis phase, symbolic execution cannot be used in place of instrumentation. However, for a large class of counting-based metrics, particularly instruction type counts, these techniques can be applied successfully. By identifying divergent basic blocks, one can apply instrumentation selectively, thus incurring overheads only where needed. The outcome is an overall reduction in the runtime overheads associated with instrumentation.

Note that in a GPU kernel, control-flow logic does not necessarily depend on the thread index. There are many applications that exhibit uniform control-flow and thus can be analyzed entirely via symbolic execution. Examples of uniform control-flow include conditional code that is based on the value of an input kernel parameter, which is shared across all threads, or a backwards branch in a loop conditional, where the number of loop iterations is either known statically or is once again a function of an input kernel parameter shared across all threads. Code listing 1 shows the GPU kernel code for the *NBody* application. The kernel consists of three loop conditionals, each of which are based either on a kernel input parameter or a statically known value.

The Lynx framework uses a form of dataflow analysis, known as taint analysis [20], to track causal dependencies between control-flow instructions and the special registers that keep information about the current thread index.

Listing 1: NBody OpenCL Kernel

```
#define UNROLL_FACTOR 8
__kernel
void nbody_sim(__global float4* pos,
        __global float4* vel, int numBodies,
        float deltaTime, float epsSqr,
        __global float4* newPosition,
        __global float4* newVelocity) {

    unsigned int gid = get_global_id(0);
    float4 myPos = pos[gid];
    float4 acc = (float4)0.0f;

    int i = 0;
    for (; (i+UNROLL_FACTOR) < numBodies;)
    {
        #pragma unroll UNROLL_FACTOR
        for(int j = 0; j < UNROLL_FACTOR;
         j++, i++)
        {
            float4 p = pos[i];
            float4 r;
            r.xyz = p.xyz - myPos.xyz;
            float distSqr = r.x * r.x +
                r.y * r.y + r.z * r.z;
            float invDist = 1.0f /
                 sqrt(distSqr + epsSqr);
            float invDistCube = invDist *
                invDist * invDist;
            float s = p.w * invDistCube;
         acc.xyz += s * r.xyz;
        }
    }
    for (; i < numBodies; i++) {
        float4 p = pos[i];
        float4 r;
        r.xyz = p.xyz - myPos.xyz;
        float distSqr = r.x * r.x +
         r.y * r.y + r.z * r.z;
        float invDist = 1.0f /
        sqrt(distSqr + epsSqr);
        float invDistCube = invDist *
        invDist * invDist;
        float s = p.w * invDistCube;
        acc.xyz += s * r.xyz;
    }

    float4 oldVel = vel[gid];
    float4 newPos;
    newPos.xyz = myPos.xyz +
        oldVel.xyz * deltaTime
         + acc.xyz * 0.5f *
        deltaTime * deltaTime;
    newPos.w = myPos.w;

    float4 newVel;
    newVel.xyz = oldVel.xyz +
        acc.xyz * deltaTime;
    newVel.w = oldVel.w;

    newPosition[gid] = newPos;
    newVelocity[gid] = newVel;
}
```

In this analysis, all basic blocks that are branch targets to conditions dependent on the current thread index are marked as tainted. These basic blocks are identified as possibly exhibiting thread divergence, and thus require instrumentation to ensure precise results. All other basic blocks may be evaluated symbolically.

Symbolic execution is a technique for evaluating a program path-by-path, given its inputs. If all inputs to a program are known prior to execution, it is possible to evaluate each path in the program using concrete input values and determine the precise number of times each basic block is executed. This technique is called concolic (concrete and symbolic) execution. Since the GPU programming model requires kernel arguments and data to be explicitly set on the GPU prior to kernel execution, one can use concolic execution to analyze GPU kernels. However, in the presence of hundreds of threads and possible thread divergence, concolic execution becomes infeasible. Therefore, by identifying sections of the kernel that exhibit uniform control-flow, one can symbolically execute such sections under the assumption that a single thread is executing the code, and simply multiply the total number of executed instructions by the number of threads in the grid. This is far more efficient than dynamically instrumenting such code, which incurs per-wavefront overhead in terms of both memory bandwidth and computation.

The process of identifying the execution paths of a program operates as follows. We first construct a computation tree from the information flow analysis. In a computation tree, each node represents the execution of a control-flow statement, such as an *if* statement or the condition of a loop construct, and each edge represents the execution of a sequence of non-conditional statements. As such, a computation tree's edges are representative of its basic blocks. Once a computation tree has been constructed, it is then possible to inspect each node to determine whether it has a reference to the current thread index. If this is the case, the basic blocks represented by the outgoing edges of that node are marked as tainted, as they may exhibit divergent control flow.

To illustrate the process, Figure 3 depicts the IL code snippet and the corresponding computation tree for the *NBody* kernel. For purposes of clarity, we have replaced the identifiers of the special registers holding the current thread index with a `tid` variable.

As can be seen from *NBody*'s computation tree, none of the nodes have references to the `tid` variable, thus indicating that this kernel has uniform control flow and can be analyzed entirely via symbolic execution. In Figure 4, we show the computation tree of another kernel, the *LUD diagonal* kernel from Rodinia's LUD implementation [7]. In the case of this kernel, certain parts of the computation tree are dependent on the `tid` variable, while others are not. The basic blocks dependent on `tid` (outlined in the figure) are analyzed via instrumentation, while the remaining ones are analyzed symbolically. We show in our evaluation that our selective instrumentation approach gives us significant reductions in kernel runtime overheads.

```
func 1028 ; nbody_sim
mov r65, tid
ishl r65.x___, r65.x, l12
iadd r66.x___, r1.x, r65.x
. . .
ilt r67.x___, r1.z, l13
if_logicalnz r67.x
mov r67, l14
mov r68.x___, l14
else
mov r67, l14
mov r68.x___, l14
whileloop
ishl r69.x___, r68.x, l12
iadd r70.x___, r1.x, r69.x
. . .
iadd r67, r67.xy0w, r69.00z0
iadd r69.x___, r68.x, l15
ilt r69.x___, r69.x, r1.z
iadd r68.x___, r68.x, l25
break_logicalz r69.x
endloop
endif
ilt r69.x___, r68.x, r1.z
if_logicalnz r69.x
mov r74.x___, l18
whileloop
ishl r69.x___, r68.x, l12
. . .
iadd r67, r67.xy0w, r69.00z0
iadd r68.x___, r68.x, l26
ilt r69.x___, r68.x, r1.z
break_logicalz r69.x
endloop
else
endif
mov r68, r1.w
. . .
ret_dyn
ret
endfunc
```

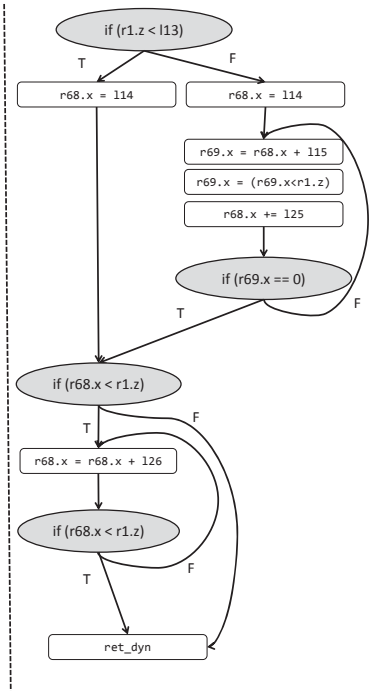

Figure 3: Computation Tree for the NBody Kernel

4. EVALUATION

The goals of the experimental evaluations presented in this section are the following:

- to demonstrate the benefits of combining symbolic execution with dynamic instrumentation in reducing kernel runtime overheads; and
- to show that symbolic execution for non-divergent code is much more efficient than its runtime instrumentation.

4.1 Experimental Setup

All experiments are performed on a system with an Intel Core i7 running Ubuntu 12.04 LTS x86-64, equipped with an AMD Radeon HD 7770 GPU. Benchmark applications for experiments are chosen from the AMD OpenCL SDK[1] and the Rodinia Benchmark Suite[7]. Seven applications are used in this study, described in Table 1. They are selected to ensure good coverage in terms of control-flow irregularity, since the approach using symbolic execution is dependent on the presence of control-flow regularity. Consequently, the selected applications (1) have no control-flow statements (*MatrixTranspose*, *FastWalshTransform*), (2) have uniform control-flow (*NBody*), (3) have mostly divergent code (*Reduction*, *BitonicSort*), and (4) have both divergent and uniform code sections (*LUD*).

Benchmark	Domain	Source
Bitonic Sort	Sorting	AMD SDK
FastWalsh Transform	Signal Processing	AMD SDK
LUD (diagonal kernel)	Linear Algebra	Rodinia
Matrix Transpose	Linear Algebra	AMD SDK
Nearest Neighbor	Data Mining	Rodinia
NBody Simulation	Physics	AMD SDK
Reduction	Sorting	AMD SDK

Table 1: Benchmark Applications

4.2 Metrics

Results for the activity factor and memory intensity metrics are shown in Figure 5, where four of the seven applications have 100% activity factor, while the other three

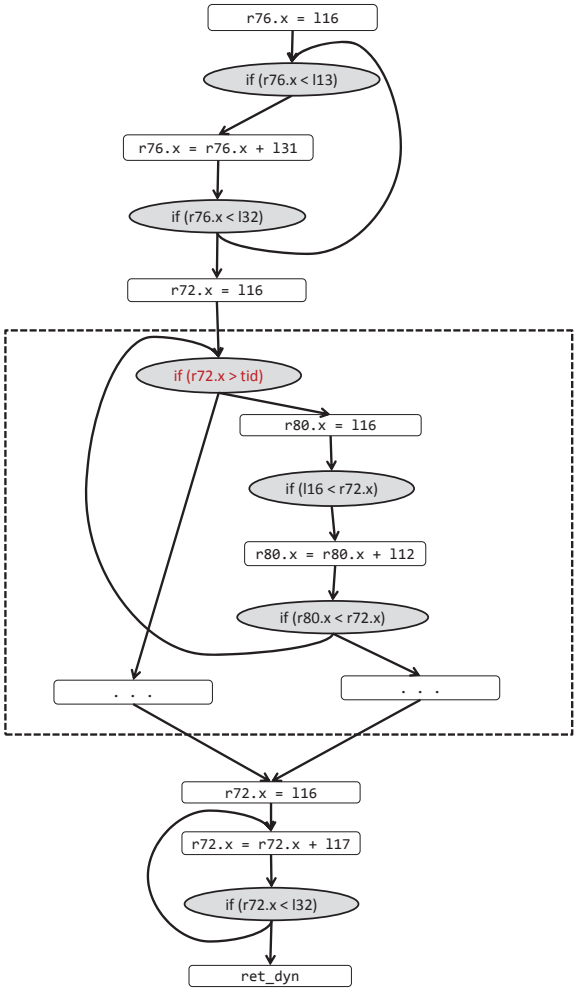

Figure 4: Computation Tree for the LUD Diagonal Kernel

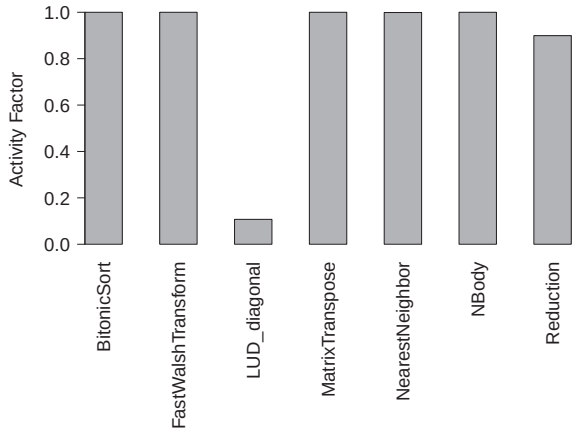

Figure 5: Activity Factor

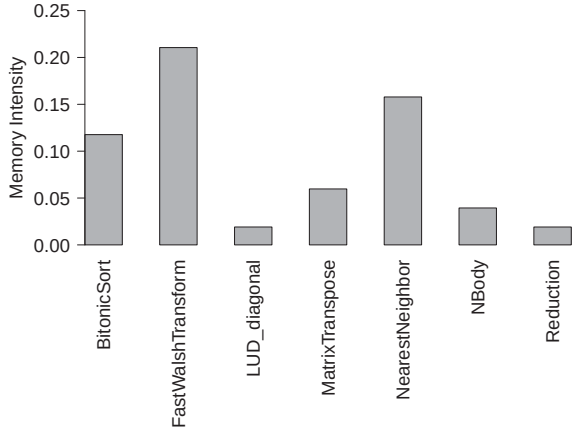

Figure 6: Memory Intensity

Reduction, LUD, and *NearestNeighbor*, have varying levels of control-flow irregularity. *NearestNeighbor* has close to a 100% activity factor, although most of its code is embedded within an *if* statement that can cause thread divergence. This is because *NearestNeighbor* proceeds with its computation only if the current thread index is less than the total number of records that need to be processed. As a result, if the number of records that need to be processed is slightly less than the total number of threads, as happens to be the case, most threads will perform the computation, resulting in a close to a 99% activity factor.

Interestingly, *BitonicSort* also has divergent code sections, but exhibits a 100% activity factor. This is because the IL code for the *BitonicSort* kernel takes advantage of the `component-wise conditional move` statement. If the divergent code section simply needs to assign a value to a variable based on a condition, this can be accomplished by replacing that code section with a conditional move statement. This, then, eliminates the divergent code sections, resulting in *BitonicSort* also exhibiting uniform control flow.

The memory intensity for most applications is low. This is because most of the application kernels make use of local memory to reduce the number of uncoalesced global memory accesses, and memory intensity computes the ratio of *global* memory operations to total dynamic instructions. Note that global memory can be up to 150x slower than local memory so this is usually a significant optimization. Memory intensity is high for application kernels that are generally smaller in total size and use global memory directly, such as *FastWalshTransform*, *NearestNeighbor*, and *BitonicSort*.

4.3 Dynamic Instrumentation Overheads

Figure 7 presents the kernel runtime overheads that result from applying dynamic instrumentation to all kernels, vs. selectively instrumenting kernel code based on information flow analysis and symbolic execution. We took an average of ten runs for each kernel, with and without each instrumentation, and the final overhead results are averaged for both of the two GPU metrics we implemented, activity factor and memory intensity. Given that both activity factor and memory intensity are warp-level and counting-based instrumentations, the overheads for each of them followed the exact same trend for all of our application kernels.

Note that the kernel runtime slowdown for three of the seven kernels, namely *FastWalshTransform*, *LUD*, and *NBody*, are quite high, ranging from 8-24x. This high overhead is attributed to the presence of many, small dynamic basic blocks in these kernels. Since both activity factor and mem-

ory intensity metrics contribute per-basic block overhead in terms of memory bandwidth and computation, kernels with a large number of dynamic basic blocks exhibit a larger slowdown. In addition, we believe that our implementation of these metrics also contributes to the high slowdown associated with dynamic instrumentation of all kernels in general. Due to the lack of special instructions that can provide us with wavefront-level information in the AMD IL, we chose to use atomics and memory barriers on local memory to obtain this data. We went with a non-optimal implementation choice because the HSA IL, which will replace the AMD IL for future AMD systems, has support for such intrinsics. Therefore, with HSA IL, we no longer will need to rely on atomic and barrier operations to implement these metrics. Note that irrespective of the implementation details of the instrumentation, the program analysis techniques we present in this paper to perform more fine-grained instrumentation of GPU kernels are orthogonal and complementary in reducing the overheads. The take-away is that dynamic instrumentation always has some runtime overhead associated with it, so any technique that can help reduce or eliminate the need for instrumentation altogether, while precisely characterizing the desired runtime behavior of the application, is beneficial.

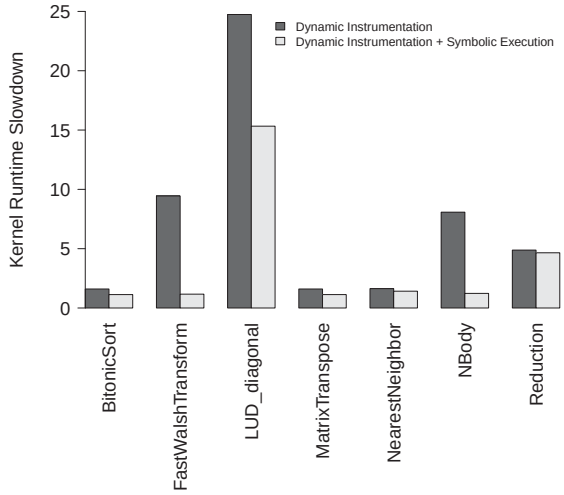

Figure 7: Kernel Runtime Slowdown due to Instrumentation

As expected, kernels that have either uniform or no control-flow (i.e., *BitonicSort, MatrixTranspose, NBody,* and *FastWalshTransform*), exhibit almost no kernel runtime slowdown. This is because these kernels are evaluated entirely via symbolic execution. *NBody* and *FastWalshTransform* benefit most from symbolic execution, because not only do these kernels have uniform control-flow, but they also have a large number of dynamic basic blocks. Since the activity factor instrumentation operates at basic-block level, kernels with many short basic blocks pay a higher runtime price than those with a few large basic blocks. *MatrixTranspose, BitonicSort,* and *NearestNeighbor* fall into the latter category.

Both *Reduction* and *LUD* have a combination of divergent and uniform code segments. However, the benefit of symbolic execution is much greater for the *LUD* kernel than the *Reduction* kernel. This correlates directly with the extent of divergent code present in these kernels. *LUD* has a good mix of divergent and uniform code segments while *Reduction* has a much larger divergent code section, with only a few instructions that fall in the non-divergent code section. As a result, we are able to symbolically execute more of the *LUD* kernel versus the *Reduction* kernel. Note that despite the *LUD* kernel having a more substantial uniform code segment than the *Reduction* kernel, it is still smaller than its divergent code section. Nevertheless, we are still able to get almost a 38% improvement by using symbolic execution in combination with dynamic instrumentation for this kernel.

4.4 Breakdown of Runtime Overheads

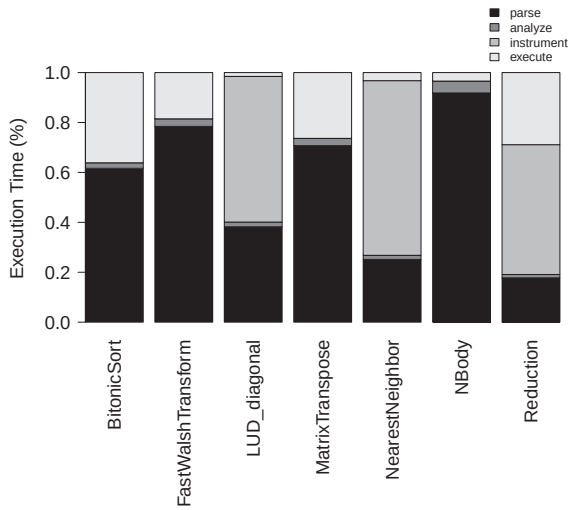

Figure 8: Dynamic Instrumentation Breakdown of Execution Time

Figure 8 presents the breakdown of execution times due to the various stages described in our instrumentation pipeline: (1) parsing the binary to extract the IL kernel and constructing the control-flow graph (shown as `parse`), (2) performing information flow analysis and symbolically executing the kernel (shown as `analyze`), (3) inserting instrumentation code into the IL kernel and updating the binary (shown as `instrument`), and (4) executing the kernel on the GPU backend (shown as `execute`).

The `parse` step is required, irrespective of whether we symbolically execute and/or instrument a GPU kernel. Note, however, that the kernels that can be analyzed entirely via symbolic execution do not have an `instrument` component. But for applications that have both `analyze` and `instrument` components, such as *LUD, NearestNeighbor,* and *Reduction*, it is evident that the runtime cost associated with analyzing the kernels is much lower than actually instrumenting them. This can be attributed to the efficiency of the analysis phase. The analysis involves iterating over the basic blocks of the control-flow graph and symbolically executing only the control-flow instructions and the instructions that the control-flow operations depend upon, for a single thread. The final number of executed instructions is multiplied by all threads in the grid. In this way, only a subset of the kernel is symbolically executed, whereas the `instrument` step involves inserting instrumentation code for all basic blocks that may exhibit thread divergence, as well

as generating an updated binary with the instrumented IL code.

Finally, note that the cost of analyzing a kernel varies with its static code size, and kernel execution happens on the GPU versus the CPU. Therefore, depending on the complexity and the size of the kernel, the kernel execution component may be larger or smaller than `analyze` component. For example, if the kernel size is large, as in the case of *NBody*, the analysis cost is more than the actual kernel execution. But for less complex, smaller kernels, such as *MatrixTranspose* and *BitonicSort*, kernel execution is more expensive.

5. FUTURE WORK

As a dynamic instrumentation framework, GPU Lynx has a unique capability to support online, profile-driven optimizations, code transformations, and resource management policies. By extending GPU Lynx to support execution on AMD GPU backends, we are in a position to investigate automatic run-time resource allocation and decision-making for the integrated CPU-GPU environments, supported by AMD's Accelerated Processing Units (APU). Additionally, we plan to support the HSA IL as the backend IR to explore the possibilities of profiliing on a variety of future GPU backends. We also believe that with the HSA IL support, we will be able to significantly lower instrumentation overheads that were implementation-specific, resulting from the lack of wavefront-level instructions in the AMD IL. We believe that the combination of our program analysis techniques and HSA IL support will make our framework much more viable and attractive to enable profile-guided optimizations and resource management with minimal runtime overheads.

6. RELATED WORK

GPU Lynx[12] is a dynamic instrumentation engine that enables customized instrumentation routines to be inserted into GPU kernels dynamically and transparently. Our framework extends GPU Lynx to enable the execution of OpenCL applications on the AMD GPU backend target. Caracal[10] is a framework that leverages the GPU Ocelot dynamic compiler to execute CUDA applications on AMD GPUs. Similar to our work, Caracal also exposes an IR abstraction for the AMD IL but with the intent to translate PTX into IL to enable execution of CUDA programs on AMD GPUs. Our work, on the other hand, focuses on dynamically instrumenting IL code for direct execution on AMD backend targets.

In this work, we focus on using program analysis techniques, such as concolic execution, to alleviate instrumentation overheads. GKLEE [19] is a concolic verification and test generation framework aimed at analyzing GPU programs for correctness and performance bugs. Unlike our work, which focuses on a runtime-based approach to profiling GPU applications, GKLEE's main contribution is a symbolic virtual machine that models the execution of GPU programs on open inputs.

Preexisting work on workload characterization of GPU applications includes [17] and [13]. In [17], the authors propose an empirical model for predicting the performance of GPU kernels, using emulation to capture metrics such as memory efficiency, activity factor, and branch divergence. Goswami et al. [13] characterize several workloads from the CUDA SDK, Parboil, and Rodinia suites as well, proposing a set of microarchitecture agnostic metrics, obtained via a PTX simulator. They then use clustering analysis to understand relative workload similarities. The nomenclature and the definition of the GPU-specific metrics studied in this paper are similar to the ones discussed in [17].

GPU simulators and emulators [6, 8, 17], visualization tools built on top of such simulators [4], and performance modeling tools [23, 5] are generally intended for offline program analyses to identify bottlenecks and predict performance of GPU applications. Our work focuses on performing online analysis of applications kernels *efficiently* via dynamic instrumentation, particularly for those metrics that capture wavefront-level details to characterize GPU utilization and efficiency.

NVIDIA's Visual Profiler[22] and AMD's CodeXL[2] were released to address the profiling needs of developers of GPU compute applications. Each provides a selection of metrics to choose from by reading performance counters after applications have run. Although in some cases these utilities provide similar information that can be obtained via GPU Lynx, neither support the ability to insert customized, user-defined instrumentation procedures, nor support selective online profiling. Note that although in this paper, we have studied two warp-level metrics, the GPU Lynx engine can support a wide array of metrics, customized to the end-users' needs. To the best of our knowledge, GPU Lynx is the only GPU performance analysis tool that provides this kind of flexibility. For example, with our framework, users can profile applications at different granularities, such as on per-thread, per-wavefront, per-thread-block, or per-core basis. Further, as discussed in this work, our framework allows for fine-grained instrumentation of selective code sections. Finally, since our framework supports the ability to toggle profiling on and off at kernel execution boundaries, while the application is running, we can support online, profile-driven optimizations, code transformations, and resource management policies.

Sophisticated program analysis techniques for GPU computing have been studied extensively in dynamic compilation frameworks, such as [9], and code generation frameworks, such as [14]. GPU Ocelot[9] is a dynamic compilation framework designed to map the explicit data-parallel, GPU execution model onto multi-threaded, many-core x86 processors, leveraging the Low Level Virtual Machine (LLVM) [18] code generator. Grewe et al. [14], on the other hand, designed a code generation framework to automatically generate OpenCL code from data-parallel OpenMP GPU programs. Both works make use of intricate program analysis techniques. To the best of our knowledge, our work is the first to apply information flow analysis and symbolic execution to develop a more efficient instrumentation framework.

7. CONCLUSION

Dynamic binary instrumentation is a useful program analysis technique for GPU computing. However, dynamic instrumentation imposes overheads in the execution path of an application, and these costs may obviate the utility of such instrumentation. This paper shows how a combination of symbolic execution and dynamic instrumentation can be used to reduce runtime overheads for a specific class of GPU metrics, namely wavefront-level metrics. Using our program analysis approach for GPU computing, we are either able to remove kernel runtime overheads altogether, or are able to show 5-38% improvement in kernel runtime overheads. We envision that such techniques will be in-

tegrated into any robust dynamic instrumentation engine for GPU applications to further encourage the adoption of real-time introspection tools based on dynamic instrumentation, particularly for profile-driven optimizations and runtime resource management. We are currently integrating our framework with a profile-driven runtime to explore runtime resource management policies for integrated CPU-GPU execution environments.

Acknowledgements

This research was supported in part by a National Science Foundation SI2 award. We also gratefully acknowledge Dr. Richard Vuduc for his valuable feedback.

8. REFERENCES

[1] AMD. *AMD APP SDK*. AMD, 2.9 edition.

[2] AMD. *CodeXL*. AMD, 3.1 edition.

[3] AMD. *AMD Intermediate Language (IL)*. AMD, 2.4 edition, October 2011.

[4] A. Ariel, W. W. L. Fung, A. E. Turner, and T. M. Aamodt. Visualizing complex dynamics in many-core accelerator architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 164–174, White Plains, NY, USA, March 2010.

[5] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 105–114, New York, NY, USA, 2010. ACM.

[6] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, Boston, MA, USA, April 2009.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct. 2009.

[8] S. Collange, D. Defour, and D. Parello. Barra, a modular functional gpu simulator for gpgpu. Technical Report hal-00359342, 2009.

[9] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[10] R. Dominguez, D. Schaa, and D. Kaeli. Caracal: Dynamic translation of runtime environments for gpus. In *Proceedings of the 4th Workshop on General-Purpose Computation on Graphics Processing Units*, Newport Beach, CA, USA, March 2011. ACM.

[11] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan. A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In *Proceedings of the 4th Workshop on General-Purpose Computation on Graphics Processing Units*, Newport Beach, CA, USA, March 2011. ACM.

[12] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on gpgpu architectures. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 58 –67, april 2012. `http://code.google.com/p/gpulynx/`.

[13] N. Goswami, R. Shankar, M. Joshi, and T. Li. Exploring gpgpu workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10, 2010.

[14] D. Grewe, Z. Wang, and M. F. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *CGO '13: Proceedings of the 11th International Symposium on Code Generation and Optimization*. ACM, 2013.

[15] K. O. W. Group. *The OpenCL Specification*, December 2008.

[16] K. O. W. Group. *HSA ProgrammerÃćâĆňâĎćs Reference Manual: Virtual ISA and Programming Model, Compiler WriterÃćâĆňâĎćs Guide, and Object Format (BRIG)*, 0.95 edition, May 2013.

[17] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of ptx kernels. *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. Gklee: Concolic verification and test generation for gpus. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 215–224, New York, NY, USA, 2012. ACM.

[20] J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[21] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*. NVIDIA Corporation, Santa Clara, California, 2.1 edition, October 2008.

[22] NVIDIA. *NVIDIA Compute Visual Profiler*. NVIDIA Corporation, Santa Clara, California, 4.0 edition, May 2011.

[23] Y. Zhang and J. D. Owens. A quantitative performance analysis model for gpu architectures. In *17th International Conference on High-Performance Computer Architecture (HPCA-17)*, pages 382–393, San Antonio, TX, USA, February 2011. IEEE Computer Society.