Characterizing the Runtime Effects of Object-Oriented Workloads on GPUs

Mengchi Zhang, Roland Green, Timothy G. Rogers
Department of Electrical and Computer Engineering, Purdue University
{*zhan2308, green349, timrogers*}*@purdue.edu*

***Abstract*—Modern GPGPU programming extensions like OpenCL and CUDA have supported object-oriented workloads on GPUs for several generations. However, no analysis of object-oriented workloads running on massively parallel accelerators has been investigated.**

This extended abstract presents a performance analysis of object-oriented workloads on a PASCAL Titan X GPU. Our characterization demonstrates that GPUs have different performance trade-offs when running object-oriented code than traditional CPUs. Where CPUs are sensitive to the misprediction of indirect branches that result from virtual function calls, GPUs are more sensitive to the additional memory system pressure that comes from loading pointers and virtual function table entries.

***Keywords*-GPGPU; Object-Oriented Programming**

I. Introduction

General-purpose programming extensions like CUDA and OpenCL have enabled the execution of C/C++ code on GPUs. GPUs offer the potential for high performance and energy efficiency, but a major barrier to their general adoption is still programmability.

As CPUs and GPUs become more tightly integrated and the complexity of applications benefiting from GPU acceleration increases, popular software engineering techniques like object-oriented (OO) design are likely to become more commonplace. There are several impediments to the widespread adoption of OO C++ in GPUs. The first is the complexity involved in creating and maintaining multiple sets of objects in the CPU and GPU memories. However, features such as unified virtual memory may help alleviate this issue by allowing user-level software to manage just one set of objects. Once objects are allocated and migrated to the GPU, the second greatest challenge is the runtime overhead of OO code in a massively multithreaded environment.

The performance cost of OO code is a long-studied problem in the CPU world [1]. Calling virtual functions, whose targets are not known until runtime has resulted in a significant amount of CPU hardware research focused on improving the predictability of indirect branches. A fundamental difference between CPUs and GPUs is that GPUs do not use any speculative execution. The high area, complexity and energy overheads of speculative execution on GPUs make techniques like branch prediction not viable. Additionally, GPUs use of thread level parallelism to hide latency makes the extraction of instruction level parallelism less important.

```
// Object Heirarchy
class RenderableObject {
   __device__ virtual float3 getNormal(...)=0;
   __device__ virtual float intersection(...)=0;
}

class Sphere : RederableObject {
   __device__ virtual float3 getNormal(...) {
       ...
   }
   __device__ virtual float intersection(...) {
       ...
   }
}

class Plane : RenderableObject {
   __device__ virtual float3 getNormal(...) {
       ...
   }
   __device__ virtual float intersection(...) {
       ...
   }
}
```

Figure 1: Pseudocode GPU ray tracer using OO.

II. Workloads

To begin studying this problem such that software systems and architectural techniques can be developed, we created a set of GPU OO workloads by transforming several non-OO GPU applications from GPGPU-Sim's ISPASS [2] and lonestarGPU [3] benchmark suites into OO equivalents:

Raytracing [2] (RAY): A high-level view of our OO implementation is shown in Figure 1. We created an abstract *RenderableObject* class and *Sphere/Plain* classes that inherit from the *Object* class. The simulation renders 2 spheres and 1 plane in a 512x512 image.

Breadth-First Search [3] (BFS): BFS with a worklist (BFSW) and BFS with a worklist and flags (BFSA) from the lonestarGPU benchmark suite. We create a *BFSGraph* class which derives from the *Graph* class with a virtual *processNode* function.

Single-Source Shortest Path [3] (SSSP): SSSP implements the Bellman-Ford algorithm. We create an *SSSPGraph* class which derives from the *Graph* class and overloads node processing functions.

Minimum Spanning Tree [3] (MST): Minimum Spanning Tree is imported from the lonestarGPU benchmark suite and uses Boruvka's algorithm. We create an *MSTGraph* class which derives from the *Graph* class.

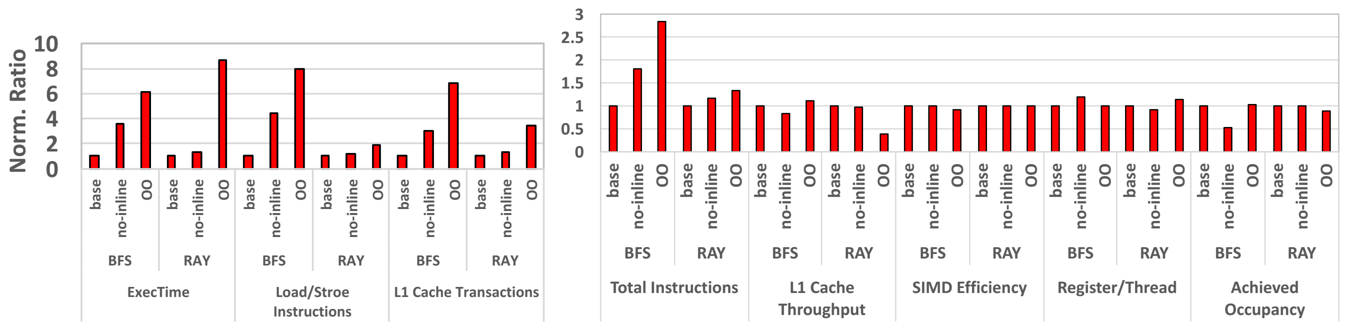

Figure 2: Ratio of statistics measured on Pascal TITAN X for inlined functions (baseline), non-inlined functions and full OO. Values normalized to the non-OO baseline.

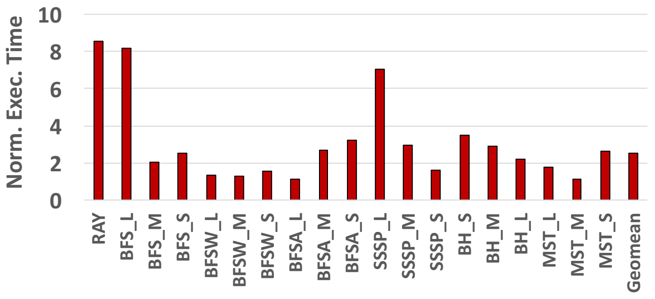

Figure 3: Execution time for OO workloads on a Pascal TITAN X. Normalized to a non-OO GPU implementation.

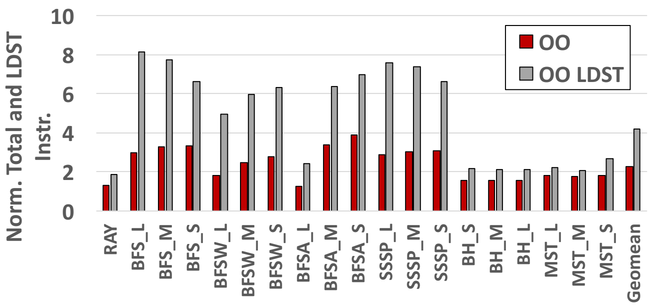

Figure 4: Instructions executed and load/store executed for OO implementations normalized to non-OO baseline

Barnes-Hut N-Body [3] (BH): BH is a star cluster based gravitational force algorithm to solve the N-Body problem. We create a *bhClass* class to hold position data, etc.

III. Experimental Results

Our experiments compare a non-OO version of the applications with our OO implementations.

Figure 3 shows the execution time for the object-oriented implementation on a Pascal Titan X GPU normalized to the baseline, non-OO version. Supporting object-oriented schemes introduces slowdowns which vary from 1.12× to 8.56× with an average of 2.50×.

Figure 4 presents the normalized instructions executed, subdividing them into all instructions (OO) and just the increase in loads and stores (OO LDST). Both values are normalized to their non-OO equivalent. The number of instructions rise (due to virtual table lookups and function calling instead of inlining). However, our graph applications see a much larger increase in memory instructions.

Figure 2 presents statistics collected from the RAY and BFS workloads. We compiled each baseline benchmark both with and without function inlining to quantify the overheads that come from simply calling a function, and what OO adds on top of that. The graph shows normalized values for execution time, warp SIMD efficiency, total number of instructions executed, L1 statistics available from the profiler, registers/thread usage and occupancy. BFS and RAY both experience a significant slowdown with OO code, but the reasons for each is slightly different. In BFS, a significant portion of the overhead comes from calling functions (as the no-inline implementation shows a 1.8× increase in execution time). However, in RAY, removing inlining has little effect. RAY's performance decrease comes primarily from a large drop in the L1 cache throughput, specific to OO code.

IV. Conclusion

This extended abstract provides an early look at the performance of OO programs on GPUs. Our initial results demonstrate that OO workloads on GPUs have different characteristics and bottlenecks than OO code on CPUs. Our study suggests that GPU OO programming places excessive stress on the memory system.

References

[1] K. Driesen and U. Hölzle, "The direct cost of virtual function calls in c++," in *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1996.

[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 163–174.

[3] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2012, pp. 141–151.