# POSTER: Quantifying the Direct Overhead of Virtual Function Calls on Massively Parallel Architectures

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

***Abstract*—Programmable accelerators aim to provide the flexibility of traditional CPUs, with greatly improved performance and energy-efficiency. Arguably, the greatest impediment to the widespread adoption of programmable accelerators, like GPUs, is the software engineering overhead involved in getting the code to execute correctly. To help combat this issue, GPGPU computing has matured from its origins as a collection of graphics API hacks to include advanced programming features, including object-oriented programming. This level of support, in combination with a shared virtual memory space between the CPU and GPU, make it possible for rich object-oriented frameworks to execute on GPUs with little porting effort. However, executing this type of flexible code on a massively parallel accelerator introduces overhead that has not been well studied.**

**In this poster, we analyze the direct overhead of virtual function calls on contemporary GPUs. Using the latest GPU architectures and compilers, this poster performs the analysis of how virtual function calls are implemented on GPUs. We quantify the direct overhead incurred from contemporary implementations and show that the massively multithreaded nature of GPUs creates deficiencies and opportunities not found in CPU implementations of virtual function calls.**

***Index Terms*—GPU, Object-Oriented Programming**

## I. Introduction

*General-purpose Graphics Processing Unit* (GPGPU) programming extensions like CUDA [1], OpenCL [2] and OpenACC [3] enable the execution of C/C++ code on GPUs. While GPUs offer the potential for high performance and energy efficiency, a major barrier to their adoption as general-purpose accelerators is programmability. To help alleviate this problem, the subset of C++ supported on GPUs has grown to include much of the C++ standard as well as a shared virtual address space with the CPU. However, the overhead of virtual function calls in a massively mutlithreaded environment has not been studied. In this poster, we study the execution of an object-oriented microbenchmark that measures virtual function overhead on current GPUs. Just like in CPUs, objects allocated on the GPU have virtual member functions, where the implementation of the function is not known until runtime. This dynamic dispatch is known to cause both direct and indirect overhead in CPUs [4]. Direct costs are attributed to the extra instructions needed to call the virtual function (virtual table lookups etc.), while indirect costs result from the lack of compiler optimizations across virtual function boundaries. This poster focuses on the direct cost of calling virtual functions on GPUs.

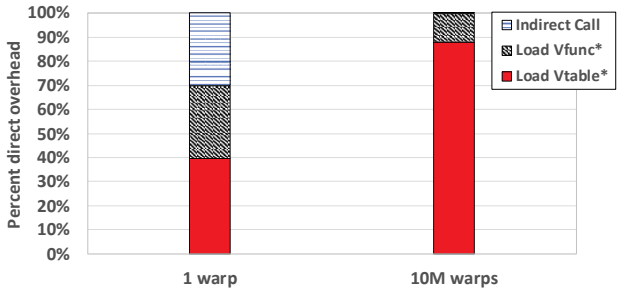


Fig. 1: Direct overhead of virtual function calls in GPUs for 1 warp vs. 10M warps on an NVIDIA Titan V. In this example, each thread accesses a different object in the same type.

## II. Virtual Function Call and its overhead on GPUs

Table I chronicles the state of CUDA programming features and NVIDIA GPU capabilities over the last decade. The programming features accessible on GPUs has been steadily increasing with each new release of CUDA and each new hardware generation. Despite this increased support for programming features, little attention has been given to the effect virtual function calls have on performance.

Since the implementation details of virtual function calls on NVIDIA GPUs are not described in any public documentation, we obtain the information in this poster by reverse-engineering binaries compiled with object-oriented programming. We perform all our analysis using the recently released NVIDIA Titan V GPU, however, we examine PTX and SASS code from several different GPU generations and observe that the implementation of object-oriented code has not changed significantly. We also note that although other GPU vendors do not support object-oriented features like virtual function calls, we anticipate that the observations made in this study would hold for other massively parallel accelerators.

The layout of objects (and structures) in CUDA follows the C++ standard, where fields defined sequentially within the object are laid out sequentially in the virtual address

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00063

TABLE I: Progression of NVIDIA GPU programmability and performance

| **Year** | 2006 | 2010 | 2012 | 2014 | 2018 |
|---|---|---|---|---|---|
| **CUDA toolkit** | 1.x | 3.x | 4.x | 6.x | 9.x |
| **Programming features** | Basic C support | C++ class inheritance & template inheritance | C++ new/delete & virtual functions | Unified CPU/GPU memory | Enhanced Unified memory. GPU Page Faults. |
| **GPU Architecture** | Tesla G80 | Fermi | Kepler | Maxwell | Volta |
| **Peak FLOPS** | 346 GFLOPS | 1 TFLOPS | 4.6 TFLOPS | 7.6 TFLOPS | 15 TFLOPS |

space. Virtual function calls are used to implement runtime polymorphism. To support virtual function calls, where the location of the code implementing the function is not known until runtime, the CUDA compiler supports dynamic binding. At compile-time, a virtual function table is created in CUDA's constant memory address space. In CUDA programs, constant memory is a small, cached memory space that is generally used to store variables constant across all threads in a kernel. GPUs do not currently support dynamic code loading or code sharing across kernels (like Linux does with .so files). Therefore, the code for every virtual function call in a kernel must be embedded inside each kernel's instruction stream. That means that the same virtual function implementation has different addresses, depending on the kernel it is called from. To support object creation in one kernel and use in another, a layer of indirection is added to traditional CPU virtual function implementations. A second virtual function table is created in global memory when objects are allocated. Inside the global memory table, pointers into constant memory, which is unique per-kernel, are initialized. When a virtual function is called on an object that has been allocated in another kernel, the constant memory for the calling kernel is read to find the location of the function's implementation. This adds an extra level of overhead not found in CPU object-oriented implementations. When new objects are constructed, they contain a pointer to the global virtual function table for the object type being constructed. When a virtual function is called, the global virtual table pointer is used to reference constant memory, where the address of the function implementation is stored. There is no dynamic inlining or just-in-time compiler optimizations performed in contemporary GPUs to mitigate the cost of calling virtual functions. An indirect call instruction from the GPU's instruction set is used to jump to the virtual function. GPUs use a lock-step Single Instruction Multiple Thread (SIMT) execution model where sequential threads in the program are bound together into warps when scheduled on the GPU's Single Instruction Multiple Data (SIMD) datapath. In NVIDIA machines, 32 threads execute in lock-step across a warp. Consequentially, when a virtual function is called across a warp, each thread in the warp can potentially jump to a different virtual function implementation, depending on the objects being accessed in parallel threads. When threads across a warp traverse different control flow paths, those paths cannot be executed in the same instruction. This results in a serialization of the divergent control-flow paths.

Figure 1 plots the direct overhead caused by calling virtual functions on a GPU. The overhead is broken down into 3 basic components: the cost of the indirect function call itself (Indirect Call), the cost of loading the virtual function locations from the virtual table as well as loading function pointers from constant memory (Load Vfunc*) and the cost of loading the pointer to the virtual table (Load Vtable*). To demonstrate the effects of multithreading, overheads for both 1 warp and 10 million warps are plotted. In the single warp case, 30% of the overhead comes from the indirect function call itself. With only one warp, there is not enough parallelism to hide the deep branch pipeline. The loads split the remaining overhead roughly equally. The overhead looks much different when the workload is scaled to 10 million warps. The function call latency is completely hidden by multithreading, and the loads dominate the overhead. In particular, the load to the virtual table pointer accounts for 85% of the direct overhead. Using the latest CUDA compiler, each object contains a pointer to its virtual table in memory. This implementation results in significant cache-contention when thousands of threads call thousands of virtual functions in quick succession. However, the number of distinct object types accessed in quick succession is several orders of magnitude smaller. The cost of loading function pointers from constant memory is mitigated due to the caching of the function pointers belonging to less types. Based on these facts, we conclude that the overhead of the virtual table pointer loads should be mitigated on massively parallel architectures.

## III. Conclusion

In this poster, we describe how virtual function calls are implemented in contemporary GPUs. Through microbenchmarking we characterize the overhead of virtual function calls on GPUs. While CPUs access very few objects simultaneously, and have sophisticated mechanisms like out-of-order and speculative execution to hide the latency of difficult to predict branches from virtual function calls, GPUs have no such hardware and instead, rely on massive multithreading. We find that virtual function calls on GPUs are heavily bottlenecked by memory system throughput.

## References


[1] "NVIDIA CUDA C Programming Guide," https://docs.nvidia.com/cu-da/cuda-c-programming-guide/index.html, NVIDIA Corp., 2016, accessed August 6, 2016.

[2] "OpenCL," http://www.khronos.org/opencl/, Khronos Group, 2015, accessed July 6, 2018.

[3] "OpenAcc," https://www.openacc.org/, OpenACC.org, 2019, accessed April 15, 2019.

[4] U. Hölzle and D. Ungar, "Optimizing dynamically-dispatched calls with run-time type feedback," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.