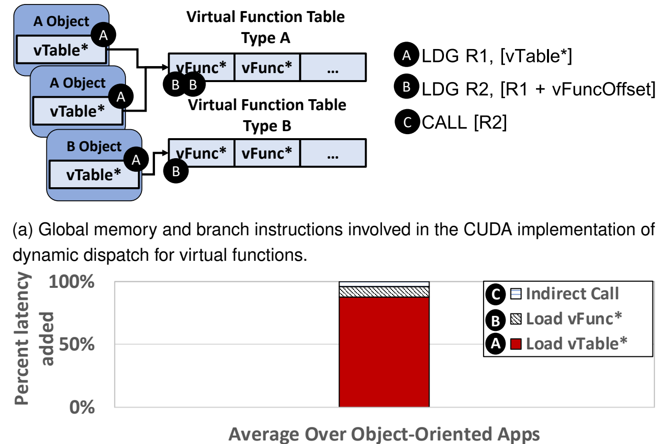
Judging a Type by its Pointer: Optimizing Virtual Function Calls on GPUs

Mengchi Zhang, Ahmad Alawneh, Timothy G. Rogers
School of Electrical and Computer Engineering, Purdue University

1. Motivation

The programmability of parallel accelerators is a major barrier to their general adoption. Modern, complex software relies heavily on reusable, object-oriented frameworks that use inheritance and virtual functions. Although programming extensions like CUDA [2], OpenCL [17] and OpenACC [1] have expanded the subset of C++ supported on GPUs, efficiently executing object-oriented code still requires significant porting effort for both functionality and performance. To alleviate the functionality problem, we propose the first CPU/GPU allocator that enables objects with virtual functions to be allocated on the CPU then used on the GPU without programmer intervention. Using both our new allocator and legacy techniques, we perform the first study of virtual function calls and dynamic dispatch on GPUs, identifying a different set of bottlenecks than observed on CPUs. Decades of work on runtime systems, compilers and architectures for CPUs have improved the execution of object-oriented applications enough to make them commonplace [4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 18, 23, 24]. We seek to do the same for GPUs.

Figure 1 demonstrates the motivation for a novel solution on GPUs. Figure 1 illustrates the implementation and added latency of calling a virtual function, known as the direct cost [11], using CUDA. Similar to C++ implementations on CPUs, CUDA implements virtual functions by storing a virtual table (vTable) for each type that contains virtual function (vFunc) pointers. Each concrete object instance contains a pointer to its vTable. When calling a virtual function, a pointer to the vTable is loaded Ⓐ, then the table is accessed to obtain the virtual function pointer Ⓑ. Finally, an indirect branch is called using the address loaded from the table Ⓒ. Figure 1b plots a breakdown of the latency added by each of these instructions using PC sampling on GPU-enabled implementations of object-oriented applications [19, 20, 21, 22] on an NVIDIA V100. 87% of the direct cost comes from the load to the vTable pointer Ⓐ. Since each object has a private copy of its vTable pointer, if each thread is accessing a different object, the load at Ⓐ will be diverged, generating a request to a different memory location from each thread. However, since the many objects being accessed come from a much smaller number of types, many threads will ultimately access the same vTable, resulting in coalesced memory accesses and more cache hits for Ⓑ. If the vTable pointer load can be avoided, most of the direct cost from calling virtual functions can be eliminated as well. To address this issue, we propose two novel, complementary techniques, one implemented completely in software: Coordinated Object Allocation and function Lookup (*COAL*) and one that requires a small hardware change: *TypePointer*. Both techniques remove the need to dereference an object's pointer to call its virtual functions.



(a) Global memory and branch instructions involved in the CUDA implementation of dynamic dispatch for virtual functions.

(b) Breakdown of the average virtual function call overhead across object-oriented GPU apps, described in Section 7 of the full paper, executing on an NVIDIA V100.

Figure 1: Direct cost of virtual function calls in GPUs.

2. Limitations of the State of the Art

State of the Art solutions for object-oriented programming on GPUs have several limitations. From a performance standpoint, no prior work has identified the bottlenecks of dynamic dispatch on GPUs. The closest work to our own is Intel's Concord [3], which adds virtual function call functionality to heterogeneous CPU/GPU objects in a machine with shared physical memory and a GPU that cannot perform indirect jumps. Instead of using vTables and function pointers to implement virtual functions, Concord embeds a type field within the object and uses a statically compiled switch statement to select the appropriate function implementation. Neither Concord nor contemporary CUDA implementations of virtual functions avoid the overhead of accessing the object to determine its type (i.e., the Ⓐ access in Figure 1). To the best of our knowledge, we are the first work for either CPUs or GPUs that performs virtual functions calls without dereferencing the object. Finally, no existing infrastructure can share objects with virtual functions between the CPU and a discrete GPU.

Operation	State-of-the-art: CUDA	Software Only: COAL	Hardware Support: TypePointer
Ⓐ Get vTable*	$Acc \propto NumObjects$	$Acc \propto NumTypes$	0 Acc
Ⓑ Get vFunc*	$Acc \propto NumTypes$	$Acc \propto NumTypes$	$Acc \propto NumTypes$
Ⓒ Call vFunc*	Indirect Branch	Indirect Branch	Indirect Branch

Table 1: Overhead of calling virtual functions in prior work and our proposed techniques. Acc=Number global accesses.

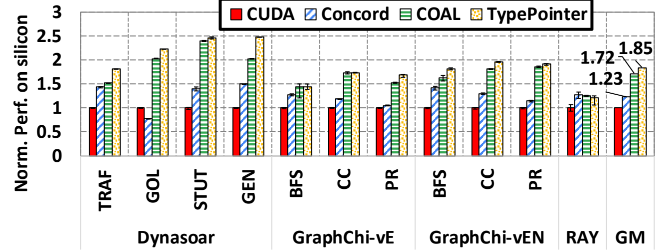


Figure 2: Performance, normalized to CUDA on a silicon V100 GPU, averaged over 10 runs (error-bars=max and min).

3. Key Insights

- We should rethink virtual function implementations for GPUs, which we demonstrate have fundamentally different bottlenecks than CPUs. GPUs primarily suffer from the additional memory traffic caused by performing thousands of virtual function calls in parallel.
- Using two novel techniques, the direct overhead of virtual function calls on GPUs is greatly reduced by determining an object's vTable location based only on the object's address.

4. Main Artifacts

This paper presents two novel techniques, one implemented in software only, *COAL*, and one that requires minimal hardware support, *TypePointer*. Both techniques reduce the number of memory accesses required to call virtual functions on GPUs. Table 1 details the three abstract actions that happen when a virtual function is called and enumerates the number of global memory accesses required for the baseline and our proposed solutions. CUDA accesses each object instance to obtain the object's vTable*, meaning that memory accesses are proportional to the number of accessed objects. In both our solutions, the vTable* is obtained without dereferencing the object pointer. *COAL* modifies the memory allocator to allocate objects of the same type in contiguous address ranges. Next, a software lookup function obtains the object's vTable* without accessing individual objects by testing the object pointer against all the allocated ranges. The lookup operation still generates memory accesses; however, memory accesses are now proportional to the number of types in the program, not the number of objects. Generally, $NumObjectInstances >> NumObjectTypes$, which results in less memory pressure using *COAL*. More importantly, there is significant reuse in the lookup function, where each thread walks a small, centralized data structure, regardless of which object it is accessing. In contrast, CUDA accesses thousands of discrete objects spread throughout memory to obtain their type. *TypePointer* is a more efficient, alternative solution to *COAL* that requires a small change to the compiler, allocator and hardware. Using a much smaller allocator change than *COAL*, *TypePointer* makes use of extra bits in the 64-bit object pointer (GPU unified memory uses a 49-bit virtual address space) to embed object type information inside the pointer to the object when it is allocated. *TypePointer* then uses a simple sequence of shift and mask instructions to obtain the object's vTable* without accessing main memory. *TypePointer* requires a small change to the GPU's Memory Management Unit (MMU) to ignore the unused bits in the virtual address.

We implement both *COAL* and *TypePointer* using CUDA 10.1 and PTX compiler transformations. We evaluate both techniques on real hardware (an NVIDIA Volta V100), as well as in simulation (using NVIDIA SASS-based Accel-Sim [16]) over a collection of highly parallel object-oriented applications. To get the most reliable performance numbers, we evaluate *TypePointer* (which requires hardware changes) in real silicon by developing a prototype based on observed patterns in the unified memory allocator. This implementation is described in detail in the full paper's Section 6.2. A novel memory allocation infrastructure that enables objects with virtual functions to be shared between the CPU and GPU is also introduced.

5. Key Results

Figure 2 plots the performance improvement of *COAL* (72%) and *TypePointer* (85%) over State of the Art virtual function call implementations and memory allocators on a silicon V100 GPU. By reducing the number of expensive global memory accesses, both *COAL* and *TypePointer* can outperform a contemporary CUDA implementation of virtual functions as well as proposed work from Intel Concord [3], both of which access individual objects to obtain their type information.

6. Why ASPLOS

This paper touches on elements of memory allocation (operating systems), programming languages (compilers and object-oriented language implementation), and architecture (MMU modifications to enable *TypePointer*). Optimizing the performance of productive programming language constructs on accelerators is a relatively new topic, encouraged by ASPLOS.

7. Citation for Most Influential Paper Award

For pioneering work on decreasing the effort to make use of massively parallel accelerators with contemporary, productive programming techniques. By carefully identifying the key bottleneck in massively parallel virtual function calling, *COAL* and *TypePointer* are the first mechanisms to perform runtime virtual functions calls without dereferencing an object's pointer. *Judging a Type by its Pointer* has enabled new classes of workloads to take advantage of parallel acceleration.

References

[1] OpenAcc. https://www.openacc.org/, 2019. Accessed April 15, 2019.

[2] NVIDIA CUDA C Programming Guide. https://docs.nvidia.com/cu-da/cuda-c-programming-guide/index.html, 2020. Accessed August 6, 2020.

[3] Rajkishore Barik, Rashid Kaleem, Deepak Majeti, Brian T. Lewis, Tatiana Shpeisman, Chunling Hu, Yang Ni, and Ali-Reza Adl-Tabatabai. Efficient Mapping of Irregular C++ Applications to Intergrated GPUs. In *International Symposium on Code Generation and Optimization (CGO)*, pages 33–43, Feb. 2014.

[4] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 397–408, 1994.

[5] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in c++ programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1994.

[6] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 49–70, 1989.

[7] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization for Object-oriented Languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 93–102, 1995.

[8] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[9] David Detlefs and Ole Agesen. Inlining of virtual methods. In *ECOOP*, 1999.

[10] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302, 1984.

[11] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1996.

[12] Izzat El Hajj, Thomas B. Jablin, Dejan Milojicic, and Wen-mei Hwu. Savi objects: Sharing and virtuality incorporated. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 1, pages 45:1–45:24, New York, NY, USA, October 2017. ACM.

[13] Urs Hölzle and David 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.

[14] Jose A. Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, and Yale N. Patt. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 80–90, New York, NY, USA, 2008. ACM.

[15] John Kalamatianos and David R. Kaeli. Predicting Indirect Branches via Data Compression. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 272–281. IEEE Computer Society Press, 1998.

[16] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM, 2020.

[17] Khronos Group. OpenCL. http://www.khronos.org/opencl/, 2013.

[18] Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-based Dynamic Devirtualization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 424–435. ACM, 2007.

[19] Aapo Kyrola. GraphChi-C++. https://github.com/GraphChi/graphchi-cpp.

[20] Aapo Kyrola. GraphChi-Java. https://github.com/GraphChi/graphchi-java.

[21] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[22] Peter Shirley. Ray Tracing in One Weekend. https://github.com/petershirley/raytracinginoneweekend, 2018. Accessed Aug 20, 2018.

[23] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2000.

[24] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables: The smalleiffel compiler. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1997.