Characterizing Massively Parallel Polymorphism

Mengchi Zhang
Purdue University
West Lafayette, Indiana, USA
zhan2308@purdue.edu

Ahmad Alawneh
Purdue University
West Lafayette, Indiana, USA
aalawneh@purdue.edu

Timothy G. Rogers
Purdue University
West Lafayette, Indiana, USA
timrogers@purdue.edu

***Abstract*—GPU computing has matured to include advanced C++ programming features. As a result, complex applications can potentially benefit from the continued performance improvements made to contemporary GPUs with each new generation. Tighter integration between the CPU and GPU, including a shared virtual memory space, increases the usability of productive programming paradigms traditionally reserved for CPUs, like object-oriented programming. Programmers are no longer forced to restructure both their code and data for GPU acceleration. However, the implementation and performance implications of advanced C++ on massively multithreaded accelerators have not been well studied.**

In this paper, we study the effects of runtime polymorphism on GPUs. We first detail the implementation of virtual function calls in contemporary GPUs using microbenchmarking. We then propose *Parapoly*, the first open-source polymorphic GPU benchmark suite. Using *Parapoly*, we further characterize the overhead caused by executing dynamic dispatch on GPUs using massively scaled CPU workloads. Our characterization demonstrates that the optimization space for runtime polymorphism on GPUs is fundamentally different than for CPUs. Where indirect branch prediction and ILP extraction strategies have dominated the work on CPU polymorphism, GPUs are fundamentally limited by excessive memory system contention caused by virtual function lookup and register spilling. Using the results of our study, we enumerate several pitfalls when writing polymorphic code for GPUs and suggest several new areas of system and architecture research that can help alleviate overhead.

I. Introduction

General-Purpose Graphics Processing Unit (GPGPU) interfaces like CUDA [1] and OpenCL [2] have grown to support modern C++. While GPUs have the potential to improve parallel code's performance and energy-efficiency, a barrier to their adoption as general-purpose accelerators is programmability. CUDA and OpenCL have, to varying degrees, supported object-oriented code on GPUs for several generations. However, the compiler and runtime technology for CUDA is more mature. Table I chronicles the state of CUDA programming features and NVIDIA GPU capabilities over the last decade. The programming features accessible on GPUs have been steadily increasing with each release of CUDA and each new hardware generation. Despite this increased support for C++ and object-oriented code, the implementation and runtime effects of polymorphism on GPUs have not been well studied.

Decades of work on runtime systems, compilers, and architecture in CPUs have improved the execution of these productive programming techniques enough to make general, reusable code commonplace. However, the implications of productive programming techniques on GPUs must be studied to grow the subset of applications that benefit from GPU acceleration and increase programmer productivity. This paper takes the first steps to understand the overheads and tradeoffs of polymorphic, object-oriented code in the GPU space, such that future systems can run productive code more efficiently on massively parallel accelerators.

The adoption of GPGPU programming in non-traditional spaces such as sparse data structures [3] and graph analytics [4] has demonstrated that massively parallel accelerators still achieve significant performance and energy-efficiency gains over CPUs when running complex applications with enough parallelism. High-performance closed source packages, like the OptiX raytracing library from NVIDIA [5], use dynamic dispatch and a quick GitHub search reveals more than 35k device-side virtual functions in the wild. These signs indicate an interest in executing polymorphic code on GPUs, warranting a well-constructed exploration into its behavior.

This paper performs the first detailed characterization of polymorphic programs on GPUs and provides a foundation for future software systems and architectural techniques to evaluate the object-oriented paradigm. To study this problem from the ground up, we construct a set of microbenchmarks to understand the implementation of virtual function calls on GPUs and their performance effects in isolation. We then propose the first polymorphic GPU benchmark suite: *Parapoly*. The *Parapoly* suite is constructed by porting scalable, multithreaded CPU applications to CUDA without changing the core algorithm or underlying data structures. These applications come from the domains of model simulation, graph analysis, and computer graphics. The applications in *Parapoly* represent a future where parallel code written for the CPU can be seamlessly offloaded to the GPU without forcing the programmer to restructure their code.

Using the *Parapoly* benchmark suite, we analyze the runtime overheads associated with polymorphic GPU code, which is implemented by virtual functions in C++. Work in CPUs shows that the overhead of using virtual functions can be divided into direct and indirect costs [6]. The direct cost of a virtual function refers to the instructions added to retrieve and branch to a function pointer dynamically. The indirect cost quantifies the compile-time optimization opportunities lost because the target function is unknown until runtime. We quantify the effect of both direct and indirect overhead in *Parapoly* by exploring the effect different compilation

TABLE I: Progression of NVIDIA GPU programmability and performance

Year	2006	2010	2012	2014	2018	2021
CUDA toolkit	1.x	3.x	4.x	6.x	9.x	11.x
Programming features	Basic C support	C++ class inheritance & template inheritance	C++ new/delete & virtual functions	Unified memory	Enhanced Unified memory. GPU page fault	CUDA C++ standard library
GPU Architecture	Tesla G80	Fermi	Kepler	Maxwell	Volta	Ampere
Peak FLOPS	346 GFLOPS	1 TFLOPS	4.6 TFLOPS	7.6 TFLOPS	15 TFLOPS	19.5 TFLOPS

techniques have on runtime overheads. This paper details the reasons for this overhead and identifies areas where software and hardware support for polymorphism can mitigate this performance penalty.

The runtime performance cost of polymorphism is a long-studied problem in the CPU world [6]–[13]. A significant amount of CPU hardware research has focused on improving the predictability of indirect branches that implement calls to virtual functions. However, a fundamental difference between CPUs and GPUs is that GPUs do not use any speculative execution. The high area, complexity, and energy overheads that speculative execution would have on GPUs make techniques like branch prediction and out-of-order execution unviable. Instead, GPUs use thread-level parallelism to hide latency, making the extraction of instruction-level parallelism less critical. Our experimental results show that the limiting factor when executing polymorphic code on GPUs is the memory system. The memory accesses required to perform the virtual function lookup, and the register spilling that occurs at virtual function boundaries increase the load/store unit pressure by an average of $2\times$. These extra per-thread memory accesses still exist in CPUs but are effectively hidden by the cache hierarchy. In GPUs, the sheer number of threads accessing discrete objects overwhelms both cache capacity and throughput such that memory bandwidth becomes the primary bottleneck, resulting in average performance degradation of 77% versus inlining all the virtual function calls.

This paper makes the following contributions:

- It performs the first detailed analysis of polymorphic virtual function calls on GPUs. By reverse-engineering CUDA binaries and constructing a set of microbenchmarks, we detail the implementation of dynamic dispatch in CUDA programs.
- It introduces the first open-source polymorphic benchmark suite for GPUs: *Parapoly*. Constructed from scalable, polymorphic CPU frameworks and applications, *Parapoly* is representative of reusable code that does not need to be re-written for GPU acceleration, allowing system researchers and architects to explore the implications of productive programming practices on GPUs.
- It demonstrates that there are different performance bottlenecks when executing polymorphic code on GPUs than on CPUs. The direct overhead on CPUs primarily stems from mispredicting indirect branches, while the indirect overhead comes from missed ILP extraction strategies. In GPUs, both the direct and indirect overhead is dominated by additional memory traffic from accessing virtual tables and excessive register spills, respectively.
- Based on our observed overheads, we identify pitfalls that should be avoided when creating polymorphic GPU code and suggest areas in the architecture and system software that can be improved to increase the performance.

II. Object-Oriented Code on GPUs

Object-oriented programming is defined by four major characteristics: (1) *Data is represented as discrete objects*: data and the operations on said data are coupled together. Concretely, we use C++ classes that contain member variables and methods that operate on those variables. (2) *Polymorphism*: A hierarchy of data types is constructed, such that derived classes share some data and methods with base classes. Derived classes also override virtual functions in the base class, necessitating a runtime function lookup to determine which code should be called. C++ class inheritance is used to define the polymorphism in our applications. (3) *Abstraction*: The precise implementation of a method, or concrete type of a class does not need to be known by code that uses that class. (4) *Encapsulation*: Data internal to a class or class hierarchy cannot be directly accessed or modified by code outside the implementation of that class. Polymorphism allows code that utilizes objects to be written with a higher level of abstraction and encapsulation, resulting in virtual functions, which incur runtime overhead.

The implementation details of object-oriented features on NVIDIA GPUs are not described in any public documentation. We obtain the information in this section by reverse-engineering binaries compiled with object-oriented programming. We perform all our analysis using an NVIDIA Volta GPU. However, we examined code from several different GPU generations and observe that the implementation of object-oriented code on NVIDIA GPUs has not significantly changed since it was first supported in the Fermi architecture. 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.

When using object-oriented programming, there are two general forms of overhead: one-time overheads when objects are created/destroyed, and recurring overheads that occur when member functions are called. In workloads with frequent object creation and destruction, initialization overheads can be significant. However, many scalable applications pre-allocate data structures to avoid parallel dynamic memory allocation. While dynamic memory management on GPUs is an interesting problem, it is not the focus of this paper. However, we do quantify the cost of pre-allocating data in these polymorphic applications. We envision the most common initial use-case for object-oriented programs on GPUs to be when the bulk of the objects in the program are created and initialized either by the CPU or by the GPU in an initialization phase.

A. Runtime Object-Oriented Features

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 space. The compiler enforces encapsulation with public, protected, and private variables and method scope. Virtual function calls are used to implement runtime polymorphism.

At compilation time, virtual function tables are created in CUDA's constant memory address space. In CUDA programs, constant memory is a small, cached memory space generally used to store variables constant across all threads in a kernel. The constant space is private to each CUDA kernel and is initialized when the program is compiled. CUDA does not support code sharing across kernels or dynamic code loading (like Linux does with .so files). As a result, the code for every virtual function call is embedded in each individual kernel's instruction address space. Therefore, code for the same virtual function implementation has a different address in different kernels. To support object creation in one kernel and use in another, a layer of indirection is added.

When a new type is allocated, a second virtual function table for the same type is created in global memory. This global table is initialized with references to the constant memory table, which is different for each kernel. When new objects are constructed, they contain a pointer to the global virtual function table for their type, which persists across kernels. When a virtual function is called, the global table is read and a constant pointer is returned. The constant table contains the actual address for the function's instructions in this particular kernel. When an object allocated in one kernel has a virtual function called in another kernel, the constant memory for the calling kernel is read to find the function's implementation. This is important because it adds an additional level of overhead not found in polymorphic CPU implementations. However, we observe that the constant memory accesses do not add significant overhead in practice. Table II, discussed in Section III, presents the implementation and overhead of this indirection in more detail.

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, commonly referred to as control-flow divergence, resulting in decreased execution efficiency. Note that this control-flow divergence is no worse in polymorphic code than in the same program without virtual function call overheads.

```
// Base class
class BaseObj {};

// 32 classes with Func implementation
class Obj_0 : public BaseObj {
    __device__ Func_1(input, output, numCompute)
          {
        while(numCompute--)
            output += input;
        }
};
...
class Obj_31 : public BaseObj {
    __device__ Func_31(input, output, numCompute
          ) {
        while(numCompute--)
            output += input;
        }
};

// Initialization kernel
__global__ init(BaseObj** objArray, int
     divergence_level) {
  ...
  switch (tid % divergence_level) {
    case 0:
        objArray[tid] = new Obj_1();
    ...
    case 31:
        objArray[tid] = new Obj_31();
  }
}

// Computation kernel
__global__ compute(BaseObj** objArray, float*
     inputs, float* outputs, int numCompute, int
      divergence_level) {
  ...
  switch (tid % divergence_level) {
    case 0:
        objArray[tid]->Func_0(inputs[tid],
             outputs[tid], numCompute);
    ...
    case 31:
        objArray[tid]->Func_31(inputs[tid],
             outputs[tid], numCompute);
  }
}
```

Fig. 1: Psuedo-code for switch-based microbenchmark.

III. Microbenchmarking Virtual Function Calls on GPUs

Just like in C++, CUDA implements polymorphism via virtual function calls. Virtual functions are implemented by an indirect call instruction that jumps to the appropriate function on a per-thread basis. Unlike indirect function calls on CPUs, an indirect call on an NVIDIA GPU can branch up to 32 different ways. To evaluate the performance overhead introduced by virtual function calls, we create two microbenchmarks with identical control-flow.

```
// Base class with vFunc
class BaseObj {
    virtual __device__ vFunc(...) {}
};

// 32 derived classes with vFunc implementation
class Obj_0 : public BaseObj {
    __device__ vFunc(input, output, numCompute)
         {
        while(numCompute--)
            output += input;
    }
};
...
class Obj_31 : public BaseObj {
    __device__ vFunc(input, output, numCompute)
         {
        while(numCompute--)
            output += input;
    }
};

// Initialization kernel
__global__ init(BaseObj** objArray, int
     divergence_level) {
  ...
  switch (tid % divergence_level) {
    case 0:
        objArray[tid] = new Obj_1();
    ...
    case 31:
        objArray[tid] = new Obj_31();
  }
}

// Computation kernel
__global__ compute(BaseObj** objArray, float*
     inputs, float* outputs, int numCompute) {
  ...
  objArray[tid]->vFunc(inputs[tid], outputs[tid
       ], numCompute);
  ...
}
```

Fig. 2: Psuedo-code for virtual function microbenchmark.

One microbenchmark (Figure 1) uses a switch to arbitrate the control flow, and the other (Figure 2) uses polymorphism, where control flow is determined by virtual function calls. We also implemented the switch-based microbenchmark using if-then-else conditionals and observe that the CUDA compiler generates the same code in both cases. The body of each virtual function (and switch case) contains a loop that performs a configurable number of floating-point additions on fixed input data. We verified that in both cases, the compiler generates different function bodies for the 32 different function implementations. To study the overhead of virtual functions as the work in each function scales, both the switch-based code and the virtual function code can scale the number of floating-point additions performed in each control flow path from 1 to 32k. We call this the compute density of the microbenchmark. Both microbenchmarks can scale the number of control flow paths taken by each warp from 1 to 32 to depict the effects of control-flow divergence overhead. We call this the virtual function divergence (*dvg*). At 1 virtual function call per warp, all threads in the warp make the same virtual function call. At 32 virtual function calls per warp, each thread in a warp calls a different virtual function.

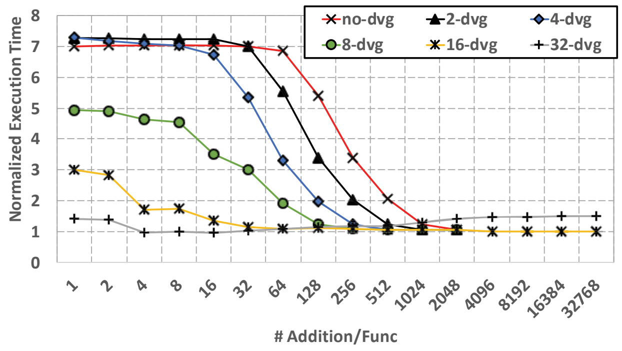

Fig. 3: The execution time of the object-oriented virtual function microbenchmark, normalized to the switch-based control flow management microbenchmark at the same compute density and level of divergence. We use the number of additions per function (*# Addition/Func*) to represent the compute density and it varies on the x-axis. Each data series represents a different level of control-flow divergence (*dvg*).

Figure 3 plots the execution time of the virtual function microbenchmark, normalized to the switch-based microbenchmark with the same compute density and control flow. We quantify the compute density with the number of floating-point additions per function (*# Addition/Func*). We also scale the number of threads in each microbenchmark to occupy the whole GPU and run each experiment 3 times, taking the median execution time. The y-axis represents the overhead added by calling virtual functions versus using less exotic control flow management. At low-levels of compute-density, there is a large variance in the overhead added at different levels of control flow divergence (*dvg*). At low compute and low divergence, the overhead caps out at approximately 7.2×. As the level of divergence increases, the overhead of calling virtual functions drops off and settles at only 1.3× when there is a 32-way divergence. As the divergence increases, the overhead of calling the virtual function becomes a smaller portion of the overall execution time. For example, at 32-way divergence, the program spends 32× more time in the serialized execution of the diverged functions than in the non-divergent (no-*dvg*) case. As the compute density increase, the overhead of the virtual function calls is mitigated since the computation in each virtual function dominates execution time, dwarfing the direct overhead of virtual function calls. By a compute density 4, the overhead of virtual functions in the fully-diverged case reaches nearly 0. However, in the non-diverged case (no-dvg), a compute density of 1024 floating-point additions per function is necessary to hide the virtual function overhead.

Figure 3 demonstrates that the direct overheads caused by virtual functions can be significant, depending on the level of runtime divergence and the computation to perform in each virtual function, or traverses a different block of the switch statement. To evaluate which aspects of the overhead contribute most to the slowdown, we use the NVIDIA Visual

TABLE II: Key direct overhead pseudo-assembly instructions for the virtual function call. The table lists the overhead at different levels of concurrency and the number of memory accesses generated by each load instruction. The overhead and memory divergence numbers are taken from the workload with no control-flow divergence (no-*dvg*) at a compute density of 1. R2 contains objArray pointer in Figure 2. Ovhd = Overhead, tid = thread index, cmem = constant memory, fid = function index, AccPI = Accesses per Instruction.

Instruction	Description	%Ovhd 1 warp	%Ovhd 10M warps	AccPI
1: LDG R2, [R2+tid*8]	Ld object ptr	18%	41%	8
2: LD R4, [R2]	Ld vTable ptr	34%	52%	32
3: LD R4, [R4+fid*8]	Ld cmem offset	26%	$< 0.1\%$	1
4: LDC R6, cmem[R4]	Ld vfunc addr	0%	7%	1
5: CALL R6	Call vfunc	26%	$< 0.1\%$	-

Profiler [14] to profile the memory accesses and latency added by each instruction of the virtual function overhead. Since the low divergence microbenchmarks have the most overhead, we study the no-*dvg* case in detail. Table II details the assembly instructions primarily responsible for the virtual function overhead, which consist of 4 additional load instructions to look up the function pointers and an indirect function call. The first load reads the pointer for this thread's object from global memory. The second load gets the global memory virtual function table (vTable) pointer (stored in the object's first 8 bytes) for this object type from memory. This load is generic (no **'G'** global specifier in the assembly instruction) because the compiler cannot statically determine which memory space the object was allocated in. The third load reads the constant memory offset for this virtual function from the global vTable. The final load accesses the kernel's constant memory space to find the actual code location, and finally, the call instruction jumps to the location.

To quantify the overhead added by each operation both with and without massive multithreading, we run the microbenchmark with one warp then again, with 10 million warps. Table II details the overhead (obtained using the GPU's PC sampling profiler) added by each instruction in both the 1 warp and 10M warp cases. In the single warp case, the first three load instructions and the function call contribute roughly the same level of overhead. However, with 10 million warps, almost all the overhead comes from the first two loads. Interestingly, multithreading is able to cover the long latency of the CALL instruction but the memory system cannot provide enough bandwidth to cover the memory latency. On each memory instruction, 32 threads execute in lock-step. Therefore, a single memory instruction can generate up to 32 different memory accesses. To cut down on the number of memory accesses generated, GPUs use memory coalescing hardware to group accesses from threads in a warp into 32-byte chunks. For example, if all 32 threads in one warp access the same 32-byte segment, only one memory access is generated by the instruction. We quantify the number of accesses generated per instruction in Table II. Since the threads in a warp are accessing the same object type, loads 3 and 4 (which access the vTable) only generate one access. However, loads 1 and 2 access different object instances and generate more accesses, resulting in more overhead. This study demonstrates that in GPUs, the direct overhead added to the memory system dominates the execution time. In single-threaded CPUs, accurate branch prediction can cover the latency of the call instruction. However, GPUs need multithreading to hide the branch latency, Multithreading places increased pressure on the memory system, resulting in large overhead.

IV. Parapoly: A Massively Parallel Polymorphic Benchmark Suite

To study the effects of object-oriented programming in a realistic setting, we propose *Parapoly*, the first open-source benchmark suites of polymorphic GPU applications. We first describe the workloads selected and inputs in Section IV-A. To isolate virtual function overheads, we create a series of different representations for Parapoly's workloads using different function calling methods, detailed in Section IV-B. We outline the features of workloads in Section IV-C, and finally, we present the setup for our experiments in Section IV-D.

A. Workloads and Inputs

Parapoly is constructed using scalable CPU applications from the fields of graph processing, model simulation and graphics rendering. Specifically, the applications that constitute *Parapoly* come from GraphChi-C++/Java framework [17], [18], [20], DynaSOAr [15], [16], and an open-source GPU ray-tracer [19], [21]. Table III presents the names, abbreviations, and descriptions of all workloads. We use the virtual functions defined in the GraphChi framework and ray tracer. For the DynaSOAr workloads, we mirror an object hierarchy present in CPU implementations of its object-oriented applications. In the latest CUDA implementation, it is impossible to create objects with virtual functions on the CPU and use them on the GPU. This limitation is likely due to needing two separate virtual function tables (one for the CPU and one for the GPU). Although this is not a fundamental limitation, the purpose of this paper is to study the performance effects of polymorphism using contemporary GPUs and systems. Therefore, each *Parapoly* application includes an initialization phase, where all the objects are constructed, and an execution phase, where the computation for each workload is performed.

We use the inputs from DynaSOAr [15], [16] for the DynaSOAr workloads and the DBLP network with approximately 300k vertices and 1M edges for GraphChi-vE/vEN workloads. To provide enough objects to render, we create 1000 objects and randomize the position and the size of the objects in the scene for the ray tracer. We run *Parapoly* on an NVIDIA Volta V100 GPU. We also verified that *Parapoly* workloads run on Accel-Sim in trace-driven SASS simulation [22], where microarchitecture studies can be performed [1].

[1]The code for the workloads can be found at https://github.com/purdue-aalp/Paraploy

TABLE III: Workloads, abbreviations and their descriptions for *Parapoly*. GraphChi-vE workloads apply virtual functions to only edges for graphs, while GraphChi-vEN workloads use virtual functions on both edges and nodes.

Workload	Abbreviation	Description
DynaSOAr Workloads [15], [16]		
Traffic	TRAF	A Nagel-Schreckenberg model traffic simulation to model streets, cars and traffic lights for traffic flows.
Game of Life	GOL	A cellular automaton formulated by John Horton Conway.
Structure	STUT	Structure uses the finite element method to simulate the fracture in a material. The benchmark models the material with springs and nodes.
Generation	GEN	Generation is an extension of GOL benchmark. The cells in Generation have more intermediate states which lead to more complicated scenarios.
Collision	COLI	Simulates the movement of particals according to gravitational forces with collision.
NBody	NBD	Simulates the movement of particals according to gravitational forces.
GraphChi-vE workloads from GrapChi-C++ Framework [17]		
Breadth First Search	BFS	Traverses graph nodes and updates a level field in a breadth-first manner. The GraphChi-vE BFS implementation defines an abstract class for edges, ChiEdge, and a concrete classEdge, which implements all the virtual functions of ChiEdge.
Connected Components	CC	Connected Component is commonly used for image segmentation and cluster analysis, it employs an iterativenode updates according to the labels of adjacent nodes.
Page Rank	PR	Page rank is a classic algorithm to rank the pages of search engine results using iterative updates for each node.
GraphChi-vEN Workloads from GraphChi-Java Framework [18]		
Breadth First Search	BFS	The GraphChi-vEN BFS implementation also defines an abstract base class for vertex, ChiVertex, and a concrete class vertex, which implements ChiVertex's virtual functions.
Connected Components	CC	GraphChi-vEN CC is similar to GraphChi-vE above. However, GraphChi-vEN CC has both virtual edges and vertices.
Page Rank	PR	GraphChi-vEN PR is similar to GraphChi-vE above. However, GraphChi-vEN PR has both virtual edges and vertices.
Open Source Ray Tracer [19]		
Raytracing	RAY	RAY performs global rendering of of spheres and planes. The algorithm traces light rays through a scene, bouncing them off of objects, and back to the screen.

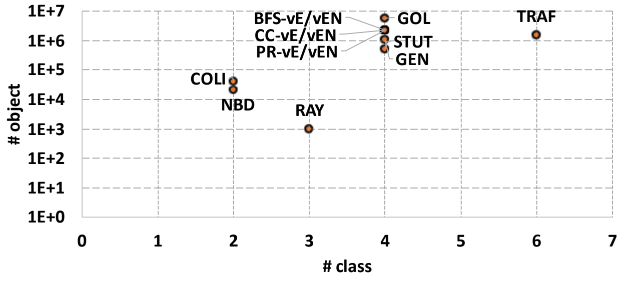

Fig. 4: The number of classes (*#class*) and objects (*#object*) in object-oriented workloads.

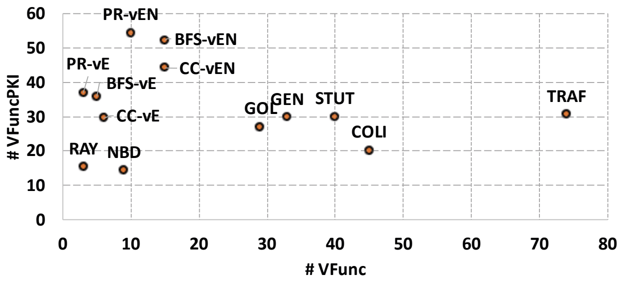

Fig. 5: The number of static virtual functions (*#VFunc*) and dynamic virtual functions called per thousand instructions (*#VFuncPKI*) in *Parapoly* workloads. *#VFuncPKI* = number of virtual functions per kilo instructions.

B. Optimized Workload Representations

To isolate the overheads associated with the virtual functions in object-oriented applications, we create two additional implementations for each workload, one where regular direct function calls are used instead of virtual function calls and another that removes the function call entirely by inlining the formerly virtual functions at compile time. The first representation removes the direct overhead from performing the virtual function lookup, while still incurring all the normal function calling costs. The second representation avoids calling a function entirely and enables aggressive inter-procedural optimizations at the expense of code size.

The representations of all three implementations are as follows:

- **Virtual function implementation (*VF*)**: The *Parapoly* applications with all the associated virtual function calling overhead.
- **No virtual function implementation (*NO-VF*)**: Virtual functions prevent optimizations because the targets are unknown. In *NO-VF*, we restructure the function calls such that the function targets are known at compilation time, and there is no direct overhead. Note that inlining is disabled, so the applications still need to call the functions.
- **Inline implementation (*INLINE*)**: Function inlining enables the compiler to further reschedule the code and avoid calling the function entirely. Here, all direct and indirect overhead is eliminated.

We discuss the impact of using the three different program representations (*VF*, *NO-VF*, and *INLINE*) in Section V.

C. Parapoly Workload Features

Object-oriented workloads usually define a small number of classes and construct many concrete instances of those classes. We describe the number of the classes (*#class*) and objects (*#object*) of *Parapoly*'s workloads in Figure 4. There are less than 10 class types in all the workloads, while the numbers of objects created range from 10^3 to 10^7. Those numerous objects are used across a massive number of threads on GPUs. The allocation and initialization of millions of objects can incur significant setup overhead, which we explore in Section V.

TABLE IV: Breakdown of how different representations save on aspects of direct and indirect overhead.

	Direct Cost	Indirect Cost
NO-VF	Save cost on virtual function lookup	Save cost with interprocedural optimization and eliminating register spills
INLINE	Save cost on function call	Save cost with code re-scheduling

We detail the number of static virtual functions (*#VFunc*) and the number of dynamic virtual function called per thousand instructions (*#VFuncPKI*) in Figure 5. A higher *#VFunc* demonstrates that the application implements a variety of virtual function calls, while a higher *#VFuncPKI* indicates frequent use of those functions at runtime. Figure 5 demonstrates that *Parapoly* exhibits significant diversity in both the frequency and number of virtual functions called and implemented. Notice that the GraphChi-vEN workloads have higher *#VFuncPKI* than GraphChi-vE workloads although they have the same number of objects and classes in Figure 4. This is because GraphChi-vEN workloads utilize virtual functions for vertices and edges, whereas in GraphChi-vE, only edges have virtual functions, as shown in Table III. Generally, a workload with a higher *#VFuncPKI* could introduce more overhead if virtual functions are called frequently, implying that there may be little work in each virtual function.

D. Experimental Setup

All our experiments are performed on an NVIDIA Volta V100 GPU. The CUDA 10.1 toolkit is used, including the NVCC compiler toolchain, runtime library, and SDK utilities. We also use the CUDA Nsight Compute Command Line Interface (CLI) 10.1 [14] and NVIDIA Binary Instrumentation tools (NVBit) [23], [24] to profile and instrument the workloads.

V. Characterizing Parapoly

This section performs a comprehensive evaluation of object-oriented workloads running on an NVIDIA Volta V100 GPU. We first summarize the performance and break down execution time into the initialization and computation phases in Section V-A. We then study the performance differences between the three different application representations defined in Section IV-B. Then, we study dynamic instructions, memory accesses, and cache behaviors to explain the overheads in Section V-B. Finally, we detail the compiler optimizations enabled by the different representations of the *Parapoly* applications in Section V-C.

A. Performance Breakdown

Contemporary NVIDIA GPUs utilize dynamic allocation to construct objects [25]. To understand the fraction of application time spent in initialization versus computation we plot the breakdown in Figure 6. Although initialization consumes more than half of the total execution time on average, the breakdown of the two phases varies significantly depending on the workload. For example, COLI, NBD, and RAY spend more than 95% on computation, while BFS, CC, and PR spend

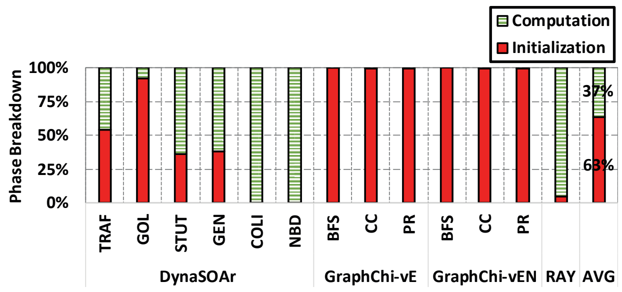

Fig. 6: Initialization and computation phases breakdown on *Parapoly*.

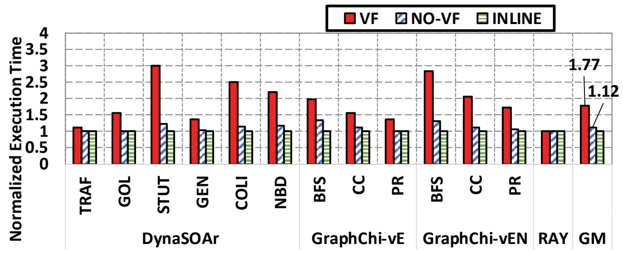

Fig. 7: The execution time for *VF*, *NO-VF* and *INLINE* implementations normalized to *INLINE* implementation for *Parapoly*. This limit study indicates the overhead when unknowing the target and when disable inlining.

99% on their time in initialization. We found that most of the initialization time is due to dynamically allocating the many thousands to millions of objects on the GPU [26]. Generally, the applications with the most objects in Figure 4 have the greatest relative overhead. The situation is most pronounced in the graph applications, where there is relatively little work per object. RAY, COLI, and NBD do significantly more work per object. This data indicates that there is significant room for improvement in GPU-side dynamic memory allocators when allocating small objects.

Driesen et al. [6] break the runtime overhead of virtual function calls into direct and indirect cost. The additional instructions required to dynamically load a function pointer beyond what is required for traditional function calling is the direct overhead, and the overhead incurred from missed

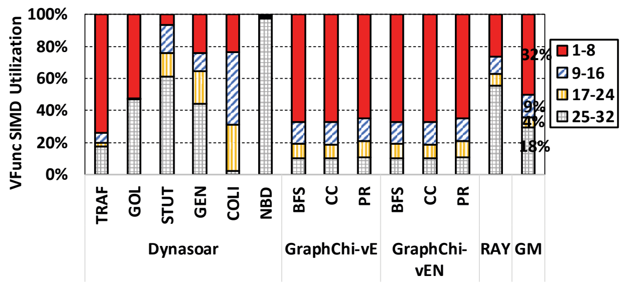

Fig. 8: The SIMD utilization for virtual function on *Parapoly* workloads.

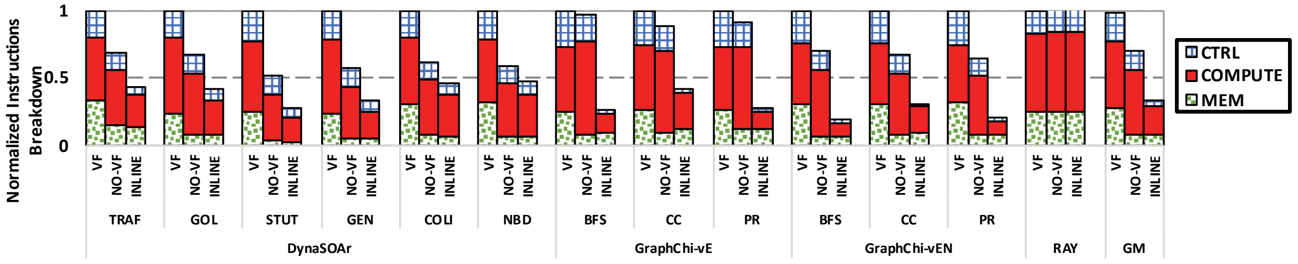

Fig. 9: The dynamic warp instruction breakdown for *NO-VF* and *INLINE* normalized to *VF*. Instructions are categorized into memory (*MEM*), compute (*COMPUTE*) and control (*CTRL*) instructions.

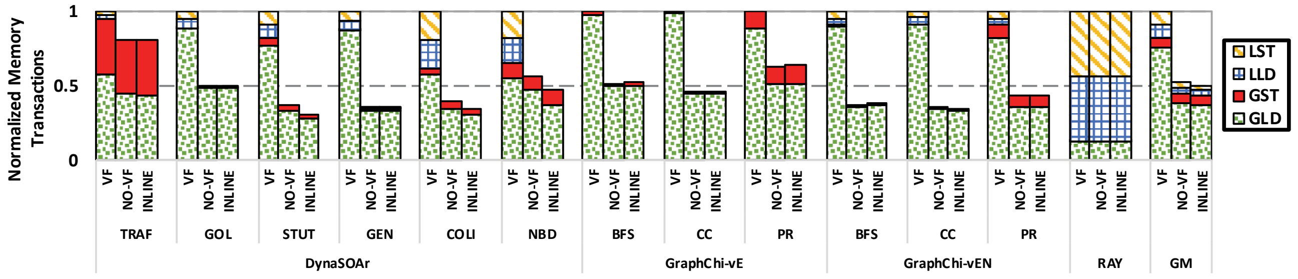

Fig. 10: The number of transactions for global loads (*GLD*), global stores (*GST*), local loads (*LLD*) and local stores (*LST*) for *NO-VF* and *INLINE* normalized to *VF*.

compile-time optimizations is the indirect overhead. For example, indirect cost can take the form of worse code scheduling or missed inter-procedural optimizations. Our two alternative implementations of *Parapoly* (*NO-VF* and *INLINE*) help us quantify what fraction of the overhead comes direct and indirect costs. Table IV breaks down the cost savings enabled by both optimizations. *NO-VF* eliminates the cost for function lookup and performs inter-procedural optimizations. *INLINE* further lessens function call overhead since the compiler can reschedule the code and no function calling penalty is paid.

We measure the overhead introduced by *VF* and *NO-VF*, normalized to *INLINE* in Figure 7. *INLINE* is used as the normalization factor since it is the representation with the least overhead. Disable inlining (*NO-VF*) introduces a 12% overhead over *INLINE*, while using virtual function introduces extra 65% overhead, which is a total 77% overhead, relative to *INLINE*. Figure 7 also demonstrates the diversity of *Parapoly*. Some of the workloads, like RAY and TRAF, suffer relatively little performance loss versus *INLINE*. Others, like STUT and BFS-vEN, suffer a much greater loss in performance due to both function calling and virtual functions. Generally, the bulk of the added overhead comes between *NO-VF* and *VF*. We detail the reasons for the overheads in Section V-B.

B. *Profiling Analysis*

Figure 8 plots the SIMD utilization of each *Parapoly* workload. SIMD utilization represents how many lanes of each warp instruction are active when the instruction executes and provides a measure of control-flow divergence in an application. A SIMD utilization of 32 indicates that the instructions executed with all lanes active, whereas 1 means that only one thread was active in each warp. *Parapoly* Workloads have a relatively diverse divergence distribution. NBD and STUT have less divergence, while GraphChi-vE and GrapChi-vEN show more divergence. Both the virtual function features and the level of divergence affect the overhead of virtual functions on GPUs. As we noted in Section III, SIMD poor utilization generally decreases the effect of virtual function overheads. However, SIMD utilization alone is not enough to predict the runtime overhead. The compute density and frequency of calls in the application is also important. As a concrete example, RAY has a relatively high SIMD utilization, compared to the graph applications. However, RAY's higher compute-density and lower frequency of virtual function calls results in its overhead being significantly lower than the graph applications.

We measure the dynamic instruction breakdown for *NO-VF* and *INLINE* normalized to *VF* in Figure 9. We classify instruction types as either memory (*MEM*), compute (*COMPUTE*) or control (*CTRL*). On average, *NO-VF* and *INLINE* execute 41% and 2.8× less instructions than the *VF* implementation of *Parapoly* respectively. Interestingly, the bulk of the memory instruction reduction comes from *NO-VF* because it avoids the virtual function lookup and allows for some inter-procedural optimizations. There is a significant reduction in the number of compute instructions executed by *INLINE* because it can avoid the large number of move instructions (which are counted as compute) required to setup the function calls. However, the reduction in compute instructions does not translate into as much of a performance gain as *NO-VF*. This indicates that the primary source of the overhead in *Parapoly* comes from the additional memory accesses added by virtual function table

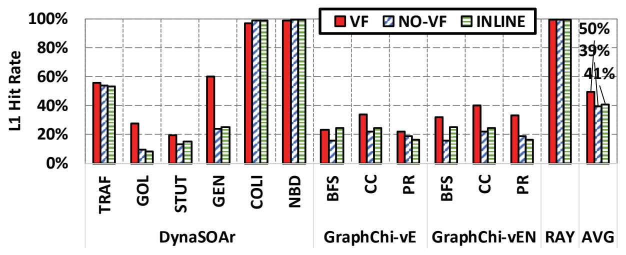

Fig. 11: L1 hit rate for virtual function (*VF*), no virtual function (*NO-VF*) and function inlining (*INLINE*) implementations.

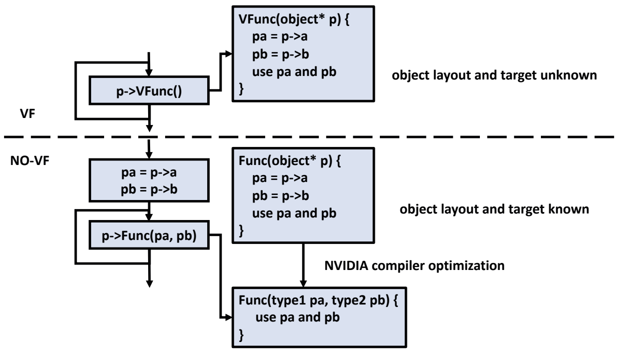

Fig. 12: Example demonstrating how NVCC will pre-load registers with function members if function targets are known at compilation time.

lookups and register spilling.

NVIDIA GPUs execute one warp instruction for 32 threads. Thus, each memory instruction may access up to 32 locations in a warp. To quantify the change in the number of memory transactions that result from *NO-VF* and *INLINE*, we measure the number of transactions for global loads (*GLD*), global stores (*GST*), local loads (*LLD*), and local stores (*LST*) in Figure 10. 76% of memory transactions are global loads, and *NO-VF* reduces global loads by 37% versus *VF* by removing virtual function lookup overhead. Moreover, knowing function targets avoid registers spilling to local memory, which is costly on GPUs. This reduces 66% of local loads and stores. *INLINE* has a minimal effect on memory transactions, reducing compute instructions in the form of moves (Figure 9). Register spills and fills affect some applications much more than others. COLI, NBD, and RAY all experience significant memory traffic in *VF*. When virtual function calls are eliminated, COLI and NBD are no longer forced to spill and fill registers, and locals are eliminated. In RAY, the local accesses come from local arrays, which are irrelevant to virtual function calls.

We also measure the L1 cache hit rate for *VF*, *NO-VF* and *INLINE* in Figure 11. The L1 cache hit rate drops from *VF* to *NO-VF* since *NO-VF* removes numerous global loads to shared virtual function tables that have locality. Interestingly, even though the cache hit rate worsens using *NO-VF* the overall performance improves because there are fewer accesses to the cache. This demonstrates that although these accesses have locality, L1 cache throughput on hits is a bottleneck when many objects access their virtual function tables at once.

C. Observed Compilation Time Optimizations

In this subsection, we delve into some of the compiler optimizations that NVCC cannot perform when virtual functions are called. Figure 12 depicts an example that *NO-VF* can optimize versus *VF* implementation. *VF* (top) calls a virtual function VFunc() that is unknown until run time, while *NO-VF* (bottom) knows the target at compilation time and calls a normal function Func(). Overhead associated with getting the virtual function pointer is not shown for clarity, as this example focuses on one aspect of indirect overhead. In the VFunc() implementation, the object's fields ($p \to a$ and $p \to b$) must be loaded into registers every time the VFunc() is called. If the function is called in a loop, this will involve successive loads to the variables. In the Func() version, the compiler can hoist the member loads outside the function call and assume that the values are loaded into registers (pa and pb) when the function is called. Note that we also discover that the NVIDIA assembler optimizes function parameter passing with registers instead of pushing them on the local memory stack due to local memory overhead on GPUs. This example demonstrates one concrete way the compiler is able to effectively reduce memory pressure with indirect optimizations, even if a function is not inlined. Knowing the target at compilation time also allows for smarter register allocation. If we cannot determine the target at compilation time, the virtual function has to spill the registers it uses to local memory. This introduces some of the extra local loads and stores for *VF* in Figure 10. On the contrary, the compiler can coordinate the register usage, and the local accesses can be eliminated in the *NO-VF* implementation.

VI. Discussion

In this section, we first summarize the primary sources of overhead that programmers writing object-oriented code on contemporary GPUs should avoid. We then document our discovered opportunities for mitigating this overhead, which the systems and architecture communities can use to enable higher performance object-oriented code on GPUs.

A. Pitfalls of GPU Object-oriented Programming

There are two implementation decisions when writing object-oriented code for GPUs that lead to the biggest pitfalls:

- **Using virtual functions in non-diverged control-flow**: As demonstrated in Section III, the relative overhead of calling virtual functions is increased in dense code without control-flow divergence. Having less active threads limits the number of objects being accessed at once and helps mitigate memory pressure. This suggests that the use of virtual functions in the most regular functions should be avoided in CUDA.
- **Large, register-heavy virtual function implementations**: To address the memory system as the primary

bottleneck for object-oriented Programs on GPUs, programmers should avoid excessive spills and fills from large function bodies that the compiler cannot optimize effectively.

B. *Optimization Opportunities in GPUs*

Two potential opportunities to decrease the cost of virtual functions on GPUs are:

- **Alternative virtual function implementations**: Based on our analysis, the implementation of virtual functions on GPUs is remarkably similar to CPU implementations. Given the vastly different memory and contention characteristics on GPUs, there appears to be an opportunity to rethink how virtual function calls are implemented in a massively multithreaded environment.
- **New compilation opportunities**: Our exploration has demonstrated that the indirect cost of virtual functions on GPUs can also place significant pressure on the memory system. GPUs already employ a CPU-side just-in-time (JIT) compiler to translate PTX into SASS. It may be possible to leverage this dynamic compilation phase to devirtualize functions for certain threads where the compiler knows which object types they touch.

VII. Related Work

In this section we detail work in the programmability, benchmark creation and memory allocation spaces this work touches on.

GPU Programmability: A body of work exists on enabling CPU-like programmability infrastructures on GPUs. The use of a file system abstraction [27], network stack [28], IO system [29], and more advanced memory management [30] are examples of this. Work on supporting productive languages on GPUs [31]–[33] focuses on primitive data structures but not polymorphism and virtual functions. OpenCL supports the creation of GPU objects but does not support runtime polymorphism [2], [34], [35]. CUDA [1] started to support polymorphism beginning in 2012, as shown in Table I. As the programmability evolves, object-oriented programming as well as polymorphism is expected to be better supported and improved on GPUs.

GPU Benchmarks: There are diverse GPU benchmark suites [21], [36]–[38], and object-oriented CPU suites [39]–[42] available. There are also workloads for object-oriented programming on GPUs [15], [16], where they focus on object allocation. However, no existing GPU benchmark suites focuses on polymorphism and no work has examined the effects of object-oriented code with virtual functions on massively parallel accelerators.

Indirect Branching: A body of CPU work improves indirect branch [8] prediction [9]–[11], addressing the performance loss from misspeculation on CPUs. Other work has looked at profile-guided techniques [8], [12], [13], which increase single-threaded performance and make the code better suited to conditional branch predictors. In GPUs, the primary method of handling an indirect branch has been patented [43]. Intel Concord [44] utilizes conditional branches to simulate the functionality of virtual functions in a customized compiler for integrated Intel CPU/GPU systems. Prior work in the CPU space has applied various JIT and static compilation techniques to eliminate the need for virtual functions calls [45]–[50] There is currently no GPU-specific work on this problem. CPU work also attempts to apply JIT optimizations [45], [46] that infer allowed types at call-sites such that recompilation can be performed.

Memory Allocations on GPUs: A set of work [25], [51]–[53] has been exploited to support better memory allocations on parallel architectures. Xmalloc [51] implements a lock-free allocator with pre-defined space management. ScatterAlloc [52] utilizes bitmap to prevent allocation collisions. Issac et al. [25] splits resource allocation tasks to two-stages to improve the allocation throughput on NVIDIA GPUs. Springer and Masuhara [15], [16] develop a parallel memory allocator for object-oriented programs on GPUs. Winter et al. [26] perform a survey benchmarking contemporary dynamic allocators on GPUs. As we indicated in Section V-A, allocations comprise the main part the total execution time for some of the workloads in *Parapoly*. Therefore, high throughput parallel memory allocation is still an active area to exploit for object-oriented applications on GPUs.

GPU Optimizations on Object-oriented Programs: GPU work has been done to mitigate memory [54]–[60] and control flow [61]–[67] irregularities in GPU-unfriendly applications. While some of these techniques are potentially effective when applied to *Parapoly*, their effect on object-oriented code has not been thoroughly studied. Generally, the type of applications that these techniques benefit most have a low compute-to-memory ratio. One way to interpret the conclusions from our study is that GPU virtual functions can lower the compute-to-memory ratio of applications.

VIII. Conclusion

We perform the first study of polymorphic code on GPUs. Using microbenchmarking, we reverse-engineer the implementation of virtual functions on GPUs. We then go on to introduce *Parapoly*, the first open-source polymorphic benchmark suite on GPUs. *Parapoly* includes a diverse set of workloads from model simulation, graph analytics, and computer graphics. We dissect the overhead of polymorphism on these workloads through careful code transformations, demonstrating an average 77% overhead versus inlining functions.

Using *Parapoly*, we isolate the source of polymorphic overhead, demonstrating that the bottlenecks are fundamentally different from those previously explored in the CPU space. The memory system, not ILP extraction mechanisms, dominates the overhead introduced by virtual function calls on GPUs. While the massively multithreaded nature of the GPU hides the latency of the function call itself, it introduces excessive memory system contention. Moreover, unknown virtual function targets block inter-procedural optimizations such that register spills add to the memory contention problem.

Finally, we provide guidance on how polymorphism can be optimized on GPUs, suggesting areas where the system and architecture can be improved. It is clear from our characterization that there are significant opportunities for improving the performance of productive programming practices on GPUs. This work takes steps towards making GPUs more general-purpose and improving programmer productivity.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their feedback which helped improve this paper. This work was supported, in part, by NSF CCF #1943379 (CAREER) and the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA.

REFERENCES

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

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

[3] "NVIDIA cuSPARSE," http://docs.nvidia.com/cuda/cusparse, NVIDIA Corp., 2016, accessed August 6, 2016.

[4] NVIDIA, "NVGraph Library," https://developer.nvidia.com/nvgraph, NVIDIA, 2018, accessed Aug 20, 2018.

[5] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: A general purpose ray tracing engine," in *Special Interest Group on Computer Graphics and Interactive Techniques Conference (SIGGRAPH)*, 2010, pp. 66:1–66:13.

[6] 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, pp. 306–323.

[7] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[8] L. P. Deutsch and A. M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1984, pp. 297–302.

[9] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, "VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-based Dynamic Devirtualization," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007, pp. 424–435.

[10] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal, and Y. N. Patt, "Improving the performance of object-oriented languages with dynamic predication of indirect jumps," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2008, pp. 80–90.

[11] J. Kalamatianos and D. R. Kaeli, "Predicting Indirect Branches via Data Compression," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1998, pp. 272–281.

[12] C. Chambers, D. Ungar, and E. Lee, "An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes," in *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1989, pp. 49–70.

[13] B. Calder and D. Grunwald, "Reducing Indirect Function Call Overhead in C++ Programs," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1994, pp. 397–408.

[14] NVIDIA, "NVIDIA profiling tools," https://docs.nvidia.com/cuda/profiler-users-guide/index.html, NVIDIA, 2018, accessed Aug 20, 2018.

[15] M. Springer and H. Masuhara, "DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access," in *European Conference on Object-Oriented Programming (ECOOP)*, 2019, pp. 17:1–17:37.

[16] Aapo Kyrola, "CUDA Dynamic Memory Allocator for SOA Data Layout," https://github.com/prg-titech/dynasoar, 2018, accessed Aug 20, 2018.

[17] Aapo Kyrola, "GraphChi-C++," https://github.com/GraphChi/graphchi-cpp, 2019, accessed Aug 20, 2019.

[18] Aapo Kyrola, "GraphChi-Java," https://github.com/GraphChi/graphchi-java, 2019, accessed Aug 20, 2019.

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

[20] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[21] 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.

[22] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[23] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019, p. 372–383.

[24] NVIDIA Corporation, "NVidia Binary Instrumentation Tool," https://github.com/NVlabs/NVBit, 2020, accessed Aug 20, 2020.

[25] I. Gelado and M. Garland, "Throughput-oriented GPU Memory Allocation," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2019, pp. 27–37.

[26] M. Winter, M. Parger, M. Parger, and M. Steinberger, "Are dynamic memory managers on GPUs slow?: a survey and benchmarks," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2021, pp. 219–233.

[27] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013, pp. 485–498.

[28] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 201–216.

[29] "NVIDIA Magnum IO," https://www.nvidia.com/en-us/data-center/magnum-io/, NVIDIA Corp., 2020, accessed August 6, 2020.

[30] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: A Case for Software Address Translation on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016, pp. 596–608.

[31] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar, "Compiling and optimizing java 8 programs for gpu execution," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015, pp. 419–431.

[32] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch, "Rootbeer: Seemlessly Using GPUs for Java," in *IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2012, pp. 375–380.

[33] M. Springer and H. Masuhara, "Object Support in an Array-based GPGPU Extension for Ruby," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY)*, 2016, pp. 25–31.

[34] "The OpenCL C++ Specification," https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_Cxx.pdf, Khronos Group, 2018, accessed May 12, 2018.

[35] "An Introduction to OpenCL C++," https://www.khronos.org/assets/uploads/developers/resources/Intro-to-OpenCL-C++-Whitepaper-May15.pdf, Khronos Group, 2015, accessed May 15, 2015.

[36] 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 *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[37] 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.

[38] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, *IMPACT Technical Report, IMPACT-12-01*, University of Illinois, at Urbana-Champaign, 2012.

[39] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley, “Looking Back on the Language and Hardware Revolutions: Measured Power, Performance, and Scaling,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2011, pp. 319–332.

[40] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, L. Han, E. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, “The DaCapo Benchmarks: Java Benchmarking Development and Analysis,” in *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006, pp. 169–190.

[41] D. Zaparanuks and M. Hauswirth, “Characterizing the design and performance of interactive java applications,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010, pp. 23 –32.

[42] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, “Renaissance: Benchmarking Suite for Parallel Applications on the JVM,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019, p. 31–47.

[43] B. Coon, J. Lindholm, P. Mills, and J. Nickolls, “Processing an indirect branch instruction in a SIMD architecture,” 2006, US Patent US7761697B1.

[44] R. Barik, R. Kaleem, D. Majeti, B. T. Lewis, T. Shpeisman, C. Hu, Y. Ni, and A.-R. Adl-Tabatabai, “Efficient Mapping of Irregular C++ Applications to Intergrated GPUs,” in *International Symposium on Code Generation and Optimization (CGO)*, 2014, pp. 33–43.

[45] D. Detlefs and O. Agesen, “Inlining of virtual methods,” in *European Conference on Object-Oriented Programming (ECOOP)*, 1999.

[46] 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, pp. 326–336.

[47] O. Zendra, D. Colnet, and S. Collin, “Efficient Dynamic Dispatch Without Virtual Function Tables: The SmallEiffel Compiler,” in *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1997, pp. 125–141.

[48] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, “Practical Virtual Method Call Resolution for Java,” in *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2000, pp. 264–280.

[49] J. Dean, D. Grove, and C. Chambers, “Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis,” in *European Conference on Object-Oriented Programming (ECOOP)*, 1995, pp. 77–101.

[50] J. Dean, C. Chambers, and D. Grove, “Selective Specialization for Object-oriented Languages,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995, pp. 93–102.

[51] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu, “Scalable simd-parallel memory allocation for many-core machines,” *The Journal of Supercomputing*, vol. 64, no. 3, pp. 1008–1020, Jun 2013.

[57] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, “Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs,” in *Pro-*

[52] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, “Scatteralloc: Massively parallel dynamic memory allocation for the gpu,” in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[53] A. V. Adinetz and D. Pleiter, “Halloc,” https://github.com/canonizer/halloc.

[54] M. Rhu, M. Sullivan, J. Leng, and M. Erez, “A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013, pp. 86–98.

[55] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Cache-Conscious Wavefront Scheduling,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012, pp. 72–83.

[56] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Divergence-Aware Warp Scheduling,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013, pp. 99–110.
ceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2013, pp. 157–166.

[58] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2013, pp. 359–406.

[59] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “Orchestrated Scheduling and Prefetching for GPGPUs,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013, pp. 332–343.

[60] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, “A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 41–53.

[61] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007, pp. 407–420.

[62] W. Fung and T. Aamodt, “Thread Block Compaction for Efficient SIMT Control Flow,” in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2011, pp. 25–36.

[63] T. G. Rogers, D. R. Johnson, M. O’Connor, and S. W. Keckler, “A Variable Warp Size Architecture,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 489–501.

[64] J. Meng, D. Tarjan, and K. Skadron, “Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010, pp. 235–246.

[65] M. Rhu and M. Erez, “CAPRI: Prediction of Compaction-adequacy for Handling Control-divergence in GPGPU Architectures,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 61–71.

[66] M. Rhu and M. Erez, “Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 356–367.

[67] M. Rhu and M. Erez, “The Dual-Path Execution Model for Efficient GPU Control Flow,” in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2013, pp. 235–246.