Judging a Type by Its Pointer: Optimizing GPU Virtual Functions

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

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

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

ABSTRACT

Programmable accelerators aim to provide the flexibility of traditional CPUs with significantly improved performance. A well-known impediment to the widespread adoption of programmable accelerators, like GPUs, is the software engineering overhead involved in porting the code. Existing support for C++ on GPUs allows programmers to port polymorphic code with little effort. However, the overhead from the virtual functions introduced by polymorphic code has not been well studied or mitigated on GPUs.

To alleviate the performance cost of virtual functions, we propose two novel techniques that determine an object's type based only on the object's address, without accessing the object's embedded virtual table pointer. The first technique, Coordinated Object Allocation and function Lookup (*COAL*), is a software-only solution that allocates objects by type and uses the compiler and runtime to find the object's vTable without accessing an embedded pointer. *COAL* improves performance by 80%, 47%, and 6% over contemporary CUDA, prior research, and our newly-proposed type-based allocator, respectively. The second solution, *TypePointer*, introduces a hardware modification that allows unused bits in the object pointer to encode the object's type, improving performance by 90%, 56%, and 12% over CUDA, prior work, and our new allocator. *TypePointer* can also be used with the default CUDA allocator to achieve an 18% performance improvement without modifying object allocation.

CCS CONCEPTS

• **Software and its engineering** → **Polymorphism**; • **Computer systems organization** → **Single instruction, multiple data**.

KEYWORDS

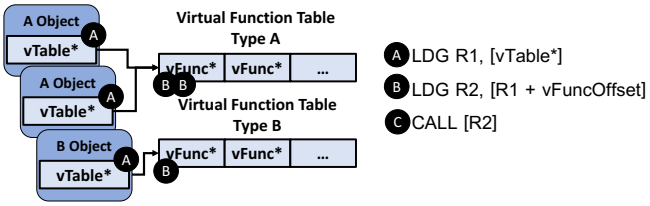
GPU, Virtual functions, Object-oriented programming

ACM Reference Format:
Mengchi Zhang, Ahmad Alawneh, and Timothy G. Rogers. 2021. Judging a Type by Its Pointer: Optimizing GPU Virtual Functions. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446734

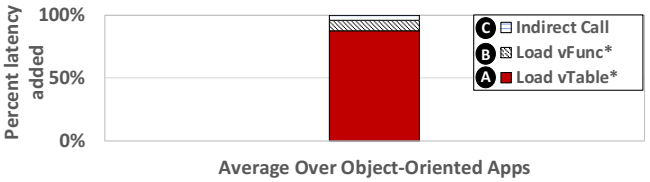
*These authors equally contributed to this work.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446734



(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 (Section 7) executing on an NVIDIA V100.

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

1 INTRODUCTION

General-Purpose Graphics Processing Unit (GPGPU) programming extensions like CUDA [2], OpenCL [32] and OpenACC [1] 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. These features can make it possible for reusable, object-oriented frameworks, written for multithreaded CPUs to execute on GPUs with little to no porting effort. However, no contemporary GPU programming language allows objects with virtual functions to be shared between the CPU and a GPU with a discrete memory space. As a result, little work has been done to evaluate the overhead of GPU virtual functions.

Despite these impediments, a number of high-performance, closed source packages, like the OptiX raytracing library from NVIDIA [39] make use of virtual functions. Intel has also developed an iHRC compiler [6] that has basic support for virtual function calls on integrated Intel GPUs. A Github survey for instances of CUDA virtual function calls reveals more than 35k GPU-side virtual functions in the wild. These numbers indicate that there is clear interest in executing virtual functions on the GPU, but their performance and usability in contemporary systems hinders their widespread use.

Figure 1 illustrates the implementation and added latency of the instructions involved in calling a virtual function, known as

the direct cost [21], using contemporary CUDA[1]. Similar to C++ implementations on CPUs, CUDA implements virtual functions by storing a virtual table (vTable) that contains virtual function (vFunc) pointers for each type. Each object instance contains a pointer to its vTable. To call 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 Program Counter (PC) sampling across 11 GPU-enabled implementations of object-oriented applications [35–37, 40] executing on an NVIDIA V100. 87% of the direct cost comes from the load to the vTable pointer Ⓐ. Since each object has it's own private copy of the vTable pointer, 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, so can most of the direct cost of calling virtual functions.

This observation is fundamentally different from what is observed on CPUs, where fewer threads and more cache-per-thread make the vTable pointer load an effective prefetch for the object members that reside on the same cache block, which will likely be accessed inside the virtual function itself. In GPUs, there are 2 reasons this prefetching is less effective: (1) the many threads executing at once are likely to thrash the caches, decreasing the likelihood of hitting on object members, and (2) even if the subsequent member accesses hit in the cache, cache bandwidth is wasted on loading vTable pointers for many concurrent threads.

In CPUs, which rely on extracting instruction level parallelism from a single thread, the predictability of the indirect branch Ⓒ is a major concern [29, 30, 33]. On GPUs, which do not use branch prediction, multithreading provides enough independent work that speculation and out-of-order execution is not necessary. On GPUs, the problem with virtual function calls is the memory system.

We propose two techniques that completely avoid accessing the per-object vTable pointer by identifying an object's type based only on the object's address in memory. The first technique, Coordinated Object Allocation and function Lookup (*COAL*), is implemented completely in user-level CUDA. *COAL* coordinates the compiler and GPU object allocator to place objects of the same type within a set of address ranges. Without any programmer intervention, compiler-generated code maps the object's address to its type. The second solution, *TypePointer*, requires a small hardware change to the Memory Management Unit (MMU) to ignore the 15 unused bits in the GPU virtual address space, where 64-bit values represent a 49-bit virtual address. *TypePointer* uses these bits to encode the object's vTable location within the pointer to the object. When an object is allocated, the runtime embeds an offset in these bits, which is recovered using a simple sequence of shift and mask instructions before a vFunc is called.

[1]In CUDA there is a level of indirection between Ⓑ and Ⓒ which loads from constant memory to account for different function locations in different kernels. We omit it here for clarity (see Section 2).

Prior work on supporting high-level languages on GPUs has either removed the ability to access non-primate types on the GPU [28, 41], or added support for virtual function calls through embedded class tags and switch statements [3, 6, 45]. All prior solutions that support virtual functions must still access the object to determine its type, which we have identified as the key bottleneck. Switch-statement based approaches, such as Intel's Concord [6] effectively eliminate the converged vTable access Ⓑ, while still accessing a type field embedded in each object instance, which is similar to the access to the virtual function pointer Ⓐ. To the best of our knowledge, no prior implementation of virtual functions on either CPUs or GPUs has attempted to determine an object's type based only on the object's address.

To study real object-oriented workloads on GPUs, we introduce a new object allocation mechanism, that allows the CPU and GPU to share objects with virtual functions on NVIDIA GPUs with discrete physical memory. Using this framework, we evaluate eleven multithreaded, object-oriented workloads [35–37, 40, 46], in both simulation and on a silicon GPU. The goal of this paper is to remove the need to completely restructure the design of multithreaded CPU code, while still achieving significant gains on the GPU. Decades of work on runtime systems, compilers and architectures for CPUs have improved the execution of object-oriented applications enough to make them commonplace. We seek to do the same for GPUs.

We make the following contributions:

- We demonstrate that there are different performance bottlenecks when executing virtual functions on GPUs. CPUs suffer significant performance loss from mispredicting indirect branches, however, GPUs primarily suffer from additional memory traffic caused by translating thousands of virtual function calls in parallel.
- We propose *COAL*, a software-only solution that allocates objects of each type consecutively, such that an automated lookup function can determine an object's type based only on the object's address.
- We propose *TypePointer*, a hardware mechansim that makes use of unused bits in the virtual address space to encode each object's type, removing the lookup overhead and allocator complexity of *COAL*.
- We introduce a Shared Object Allocation (*SharedOA*) framework that allows the CPU and GPU to share objects with virtual functions through unified virtual memory. *SharedOA* eases the GPU porting process, allowing the CPU and GPU to seamlessly share data types.

We demonstrate that, combined with our improvements to the memory allocator, *COAL* and *TypePointer* improve the performance of object-oriented code on GPUs by 80% and 90% respectively over CUDA and add an additional 6% and 12% on top of the performance improvements made by our proposed allocator. Since *TypePointer* is allocator-independent, we also evaluate it in simulation when applied on top of the default CUDA allocator, demonstrating an 18% performance improvement.

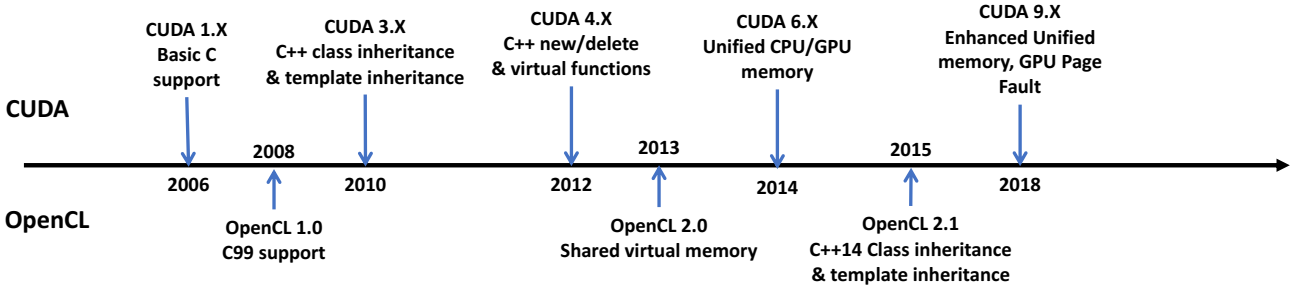


Figure 2: Evolution of programming features in CUDA and OpenCL

2 OBJECT-ORIENTED CODE ON GPUS

Figure 2 details the state of GPU programming features in CUDA and OpenCL over the last decade. Both platforms began by supporting basic C and have gradually added features such as support for objects, virtual functions and unified virtual memory. In C++, runtime polymorphism is achieved through object inheritance and implemented via virtual function calls. We focus this paper on virtual functions in CUDA, as other programming languages, including OpenCL do not support GPU-side virtual function calls. Despite supporting virtual function calls, contemporary CUDA requires that objects using them are manually allocated in GPU memory via device-side calls to *new*. We extend CUDA such that the GPU can accesses CPU-allocated objects with virtual functions.

Broadly speaking, overheads incurred by object-oriented programming fall into two categories: one-time overheads that take place when an object is created/destroyed, and recurring overheads incurred every time code interacts with an object. One-time costs can be substantial in workloads where objects are created and destroyed dynamically at a high frequency. However scalable, parallel applications, like the ones we study, often allocate data structures once then operate on them repeatedly. Although studying the implications of object initialization is an interesting problem [27], this paper focuses on the runtime phase of the algorithms.

To support virtual function calls, where the location of the code implementing the function is not known until runtime, the CUDA compiler and runtime supports dynamic binding. The runtime creates one vTable per-type that gets initialized once for the whole program. All objects contain a pointer to their type's vTable, which is initialized when the object is constructed. GPUs do not support dynamic code loading or code sharing across kernels (like Linux does with .so files). Therefore; the code for every virtual function potentially used in a kernel must be embedded inside each kernel's code. That means that the same virtual function implementation has different function addresses in different kernels. To support object creation in one kernel and use in another (where the virtual function's location in memory may be different), a layer of indirection is added to traditional CPU virtual function implementations. A second virtual function table is created in kernel-specific constant memory. The per-kernel constant memory tables contain the location of the virtual functions in each kernel's instruction memory. The global memory load Ⓑ in Figure 1 retrieves an offset into constant memory for the virtual function being called. Then a constant memory load between Ⓑ and Ⓒ loads the virtual function's address in the running kernel's instruction memory. Since this table is small, it fits in the dedicated constant memory cache and we did not observe it to be a bottleneck, hence we omit it when discussing Figure 1.

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

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

The implementation details of object-oriented features on NVIDIA GPUs are not public. We obtain the information in this section by reverse-engineering binaries with the NVIDIA profiler. We perform all our analysis using CUDA 10.1 on an NVIDIA V100 Volta, however, we examined code from several different GPU generations and observe similar behavior.

3 HIGH-LEVEL SOLUTION GOALS

Given the results in Figure 1, we design two independent solutions that reduce the cost of finding the vTable*: *COAL* in pure-software and *TypePointer* with hardware support. The goal of both solutions is to reduce the memory accesses required to obtain an object's type (hence its vTable location, operation Ⓐ in Figure 1). 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, using only the object pointer value itself. *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

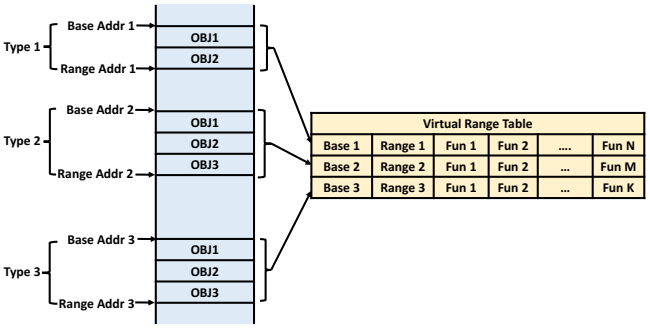


Figure 3: Type-based object allocator.

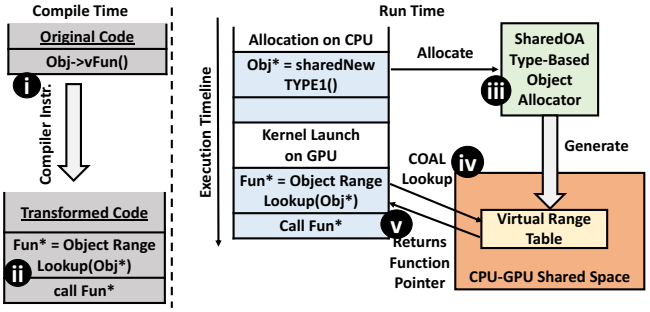


Figure 4: Overview of *COAL*.

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.

4 A TYPE-BASED SHARED OBJECT ALLOCATOR

To implement *COAL*, we design a type-based memory allocator that allows objects which make use of inheritance and virtual functions to be shared between the CPU and GPU, greatly easing the porting process. No industrial computing framework (CUDA, OpenCL, OpenACC, etc) supports the use of objects with virtual functions in unified virtual memory. To overcome this limitation, we design a Shared Object Allocator (*SharedOA*). *SharedOA* is written in user-level CUDA and allows users to allocate objects with inheritance and virtual functions in managed memory using a *sharedNew()* function. Objects allocated with *sharedNew()* store one CPU vTable pointer and one GPU vTable pointer.

The type-based allocator has two main functions: (1) dedicate contiguous chunks of memory to each object type, and (2) create a tracking structure with the address ranges of each type, which we call the virtual range table. There is an interesting challenge in predicting how large the region dedicated to each object type should be. If its too large, you risk wasting precious GPU memory space, too small and you will have to allocate many discrete regions for the same type, increasing the number of entries in the virtual range table. To alleviate this problem the allocator starts by allocating a small region size (i.e. 4K objects). If the region gets full, the allocator creates a new region with double the size. This doubling continues as more object are allocated and allows the regions sizes to adapt with the demands of the workload. Further, when regions of the same type happen to be allocated contiguously, the allocator attempts to merge the contiguous regions into one larger region. This system reduces the potential for memory fragmentation while limiting the total number of allocated regions, which can have a detrimental performance impact on *COAL*. We discuss this trade-off in Section 5 and evaluate its effect on performance in Section 8.2.

Figure 3 illustrates an example of *SharedOA*'s operation. In the example, there are three types that each have their own memory region values, e.g., (*Base 1*, *Range 1*). *OBJ1* can be easily identified as *TYPE1* if the address of *OBJ1* is between *Base Addr 1* and *Range Addr 1*. The allocator can create new regions for the same type by simply allocating a new chunk, and adding a new entry for another region of the same type of object. The allocator stores this information in global memory by augmenting the traditional virtual function tables with base and range values, which we call the virtual range table (shown on the right hand side of Figure 3). This table is accessible from the compiler generated code described in Section 5. Allocating objects of the same type contiguously is similar to how existing small-object allocators in modern operating systems work [7, 9]. Small-object allocators in CPUs are primarily used to improve allocation time, prevent fragmentation and reduce space overhead. However, in GPUs, we observe that our type-based *SharedOA* results in better object packing than the default CUDA allocator, which can have a positive impact on runtime performance, independent of allocation time. We evaluate the effect of *SharedOA* without *COAL* in Section 8.2. Using the *SharedOA* framework, we believe there is an interesting space in studying shared object allocation in GPUs that is orthogonal to the virtual function problem.

5 COORDINATED OBJECT ALLOCATION AND FUNCTION LOOKUP (COAL)

COAL leverages the runtime object allocator described in Section 4 to determine an object's type based on which address range the object's pointer falls into. Figure 4 presents an overview of *COAL*'s components. The left side of Figure 4 illustrates the compile-time instrumentation of GPU code and the right side presents an example

of how *COAL* works in co-ordination with the object-allocator at runtime. To implement the lookup portion of *COAL*, we use the compiler to statically instrument the GPU code, replacing the traditional virtual table pointer access (❶ in Figure 4) with a lookup mechanism based on object ranges, followed by a call to the function pointer resulting from the lookup (❷).

On the right side of Figure 4, the runtime *SharedOA* allocator is invoked by the CPU and type-based allocation is performed (❸). When the GPU kernel is launched, the transformed code calls the *COAL* lookup implementation (❹), which returns the correct function pointer (❺).

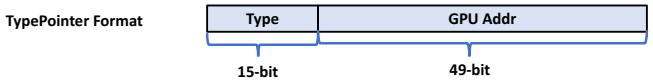
Algorithm 1: Scan algorithm for the virtual range table

```
Function ObjectRangeLookup(objectAddr, funcIndex)
    node = 0;
    nextNode = 0;
    while True do
        if objectAddr in node.left.range then
            nextNode = 2 * node + 1;
        else if objectAddr in node.right.range then
            nextNode = 2 * node + 2;
        else
            return NULL;
        end
        if nextNode >= treeSize then
            return node.vfuncTable[funcIndex];
        end
    end
end
```

The role of code generation in *COAL* is two fold: (1) to instrument virtual functions with a pre-defined range checking algorithm that will find the appropriate virtual function to call based on the object's address, and (2) to determine which virtual function calls to instrument with range checking. In some cases, the cost to perform the range search will outweigh the benefit of accessing the object. This is a heuristic-based decision that can be decided by multiple factors. For this work, choose not to instrument a virtual function with COAL if we can statically determine that every thread in a warp will be accessing the same object instance when they call the virtual function. There are several call points in the apps we study where this is true. We have observed that removing coalesced loads to the same object does not outweigh *COAL*'s overhead.

The compiler inserted code must first access the virtual range table to determine the object's type. Once the type is known, the compiler looks up the correct virtual function to call by indexing into the object's virtual range table, in the same way traditional vTable lookups operate.

To implement the scan in *COAL*, we organize the types into a segment tree [16]. Each leaf node contains the base and the range address of one type, while each internal node includes the minimum and maximum address boundaries of two child nodes. Our balanced segment tree algorithm is shown in Algorithm 1 and has $O\left(\log_2\left(K\right)\right)$ complexity, where K is the number of object ranges. The compiler implements these operations, as it has knowledge



(a) Format of the *TypePointer* returned by the *new* operation.

R_{obj}: The register containing the object address (known by the compiler)
$R_{vTablesStartAddr}$: Beginning address of the contiguous page table storage (a fixed register similar to the virtual page table register)
$\#vFuncOffset$: The byte offset for the vfunc within the vTable (known by the compiler)

```
1: SHR      Ra,   R_obj,   #49
2: ADD      Ra,   Ra,      R_vTablesStartAddr
3: LDG      Ra,   #vFuncOffset (Ra)
4: CALL     Ra
```

(b) Instructions implementing virtual function calls with *TypePointer*

Figure 5: TypePointer format and operations.

of which register contains the object address, and which virtual function is being called.

The chunk size used by the memory allocator introduces a trade-off between using large and small chunks for the object ranges. For *COAL*, the optimal chunk-size varies per-application both because of variance in the object size and the number of objects created. To help adjust for this variance, the *COAL* allocator creates chunks based on the number of objects in the range, not raw byte values. Therefore larger objects are given larger chunk sizes. A sensitivity study of *SharedOA*'s initial chunk-size is presented in Section 8.2.

6 TYPEPOINTER

In this section, we present *TypePointer*, a more efficient alternative to *COAL* that locates an object's vTable without generating additional memory requests. *TypePointer* exploits unused bits in the GPU's 64-bit virtual address space to encode the type's vTable location. Although more efficient than *COAL*, *TypePointer* requires a small hardware change and has a finite limit on the number of types that can use it. *TypePointer* requires a small change in each of the memory allocator (Section 6.1), compiler (Section 6.2) and hardware (Section 6.3). Section 6.4 describes *TypePointer*'s limitations.

6.1 *TypePointer* Memory Allocator Modifications

TypePointer's memory allocator must encode each object's type in the pointer returned from the new operation. Both the memory allocator and the GPU's virtual to physical memory translation hardware must agree on which bits of the virtual memory space are used to encode the object type. The memory allocator then creates the virtual function tables for all allocated objects in a contiguous memory region such that the unused bits in the pointer can be used as an offset into the space allocated for vTables. Figure 5a describes the format of the *TypePointer* returned by the allocator. Allocating vTables contiguously allows *TypePointer* to load the vTableStartAddr into a register at the beginning of the program,

similar to how the page table register is initialized with the page table's address. Note that it appears vTables are already allocated contiguously in CUDA.

The unused 15-bits in the virtual address space allows us to encode 32kB worth of vTable space, which is enough for 4k virtual function pointers, shared among all the types used in the program. This is more than sufficient for the programs we study in this paper, however if more pointers are required, we can pad each type's vTable such that they are all the same size and use the 15-bits as an index, which gets multiplied by the vTable size to determine the offset. This would allow us to map 32k different types, which is many more types than most heavily object-oriented CPU programs [14]. In the extremely unlikely case that a program has more than 32k types, the compiler can make this determination at link time and choose to use *TypePointer* on a subset of types, and use *COAL* or the traditional CUDA vFunc lookup as a backup mechanism. Note that *TypePointer* can be implemented in either our type-based allocator (which we evaluate in Section 8.1), or in the default CUDA allocator, evaluated in Section 8.2.

6.2 *TypePointer* Compiler Modifications

To call a virtual function using *TypePointer*, the compiler inserts a sequence of instructions to extract the vTable offset from the unused 15-bits of the object's pointer. Figure 5b lists the assembly instructions that shift the object's address (line 1), add the offset to the beginning of the contiguous vTable space (line 2), loads the appropriate vFunc* based on the vFuncOffset (line 3), then calls the function (line 4). To implement a version of *TypePointer* that can map more types, an index (instead of a byte-offset) can be stored in the object pointer. The ADD instruction (line 2) is then replaced with a fused multiply-add that multiplies a vTable size register (which is initialized upon program launch) by the index. In this implementation the system must ensure that the vTables for all object types are padded to the maximum vTable size, potentially wasting space. In the benchmarks we study, *TypePointer* works with either solution. In the worst case, the padded implementation consuming only 360 additional bytes with the total space devoted to vTables being $< 1k$.

6.3 *TypePointer* Hardware Modifications

To implement *TypePointer* in a production system, a small change must be made to the MMU, in order to avoid triggering exceptions when unused bits of the virtual address space are modified. This would involve a small change to the logic in the MMU that would not introduce any practical overhead. Such a modification could also be implemented with an enable flag such that the feature can be disabled on applications that do not require *TypePointer* support. To study *TypePointer* on a real machine, we develop a prototype that avoids MMU exceptions when attempting to access objects allocated with *TypePointer*. In particular, we discovered that the CUDA unified memory allocator returns a consistent pattern in the upper 15-bits of allocated addresses. Leveraging this fact, we modified our memory allocator to encode type information in the upper-most 15 bits, then replace these bits with the known consistent pattern before we access the fields of the object. This implementation adds some additional overhead to mask out our type information at runtime, but gives us the ability to evaluate *TypePointer* on a real machine. We also evaluate *TypePointer* in simulation using Accel-Sim [31]. In simulation we evaluate *TypePointer* using the default CUDA memory allocator (Section 8.2) and confirm that our prototype has similar performance to a machine with a modified MMU.

6.4 *TypePointer* Limitations

TypePointer embeds type information in an object's pointer when the object is constructed. Generally, valid C++ code that constructs objects with the *new* operator can use these special pointers without any effect on functionality. However, there are some corner-cases where, without additional runtime checking, using *TypePointer* may produce a different result than the same program without *TypePointer*. In particular, if the program: (1) Manipulates the pointer bits in C, clobbering the upper 15 bits of the pointer value. Generally, this will produce undefined behavior in C/C++ and will cause *TypePointer* to break. (2) Uses abusive C-style pointer casting that converts a pointer of one type into another. Again, this will generally cause undefined behavior without *TypePointer*, but there are more instances where incorrect execution will occur when *TypePointer* is used. (3) Mixes the *TypePointer* allocator with the other allocators that are unaware of the type embeddings.

7 EXPERIMENTAL METHODOLOGY

We evaluate the effectiveness of *COAL* and *TypePointer* on an NVIDIA Volta V100 GPU. For all our experiments we compile the workloads with full optimizations (-O3) using CUDA 10.1. We use the CUDA command-line profiler NVProf [38] and the CUDA Visual Profiler to collect the profiling data for all techniques. To collect data we run each program 10 times and report the average as well as the maximum and minimum performance of the computation kernels, as reported by NVProf. For the counter statistics, NVProf runs the applications several times and reports the average, which we found to have very low variance. To permit the workloads enough heap space, we use the CUDA functions *cudaDeviceSetLimit()* and *cudaLimitMallocHeapSize* and set the heap to 4GB.

To implement *SharedOA*, we override the default CUDA memory allocator. Since we implement *SharedOA* entirely in user-level CUDA code and do not have access to device driver code or the finalizing compiler, we run a tiny initialization kernel to leverage CUDA's vTable creation mechanism and update each object's GPU vTable*. This tiny kernel is run only once before the first kernel call and consumes, on average, 0.15% of the total initialization time. The init kernel can be completely avoided by implementing *SharedOA* inside the CUDA backend. We implement a PTX-level compiler transform to access vTables from the GPU-side vTable pointer.

To evaluate *TypePointer*, we implement a software-only prototype (described in Section 6.3) that bypasses the need to modify the MMU. We use this prototype to evaluate *TypePointer* in Section 8.1. In addition, we also evaluate *TypePointer* in simulation using the V100 model in v1.0.0 of the SASS-based Accel-Sim + GPGPU-Sim [31] simulator. We use the simulator to evaluate *TypePointer* both with and without the software overhead introduced to avoid MMU errors in our prototype, which we find to be insignificant. Therefore; we only include only real hardware results

Table 2: Workloads. # Objects=Number of object instances created. # Types=Number of types in the program. #vFunc=Total number of virtual functions in the compiled code. vFuncPKI=Dynamic virtual function calls per thousand instructions.

Workload	Description	# Objects	# Types	vFuncs	vFunc PKI
Dynasoar Workloads [22, 46]					
Traffic (TRAF)	A Nagel-Schreckenberg model traffic simulation to model streets, cars and traffic lights for traffic flows.	1573714	6	74	30.6
Game Of Life (GOL)	Game of life is a cellular automaton formulated by John Horton Conway. This benchmark has two abstract classes Cells and Agent.	5645916	4	29	26.9
Structure (STUT)	Structure uses the Finite element method to simulate the fracture in a material. The benchmark models the material with springs and nodes.	525000	4	40	30.0
Generation (GEN)	Generation is an extension of gol benchmark. The cells in Generation have more intermediate states which lead to more complicated scenarios.	1048576	4	33	29.8
GraphChi-vE Workloads [35]					
Breadth First Search (BFS)	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.	2254419	4	5	35.9
Connected Components (CC)	Connected Component is commonly used for image segmentation and cluster analysis, it employs an iterative node updates according to the labels of adjacent nodes.	2254419	4	6	29.5
Page Rank (PR)	Page rank is a classic algorithm to rank the pages of search engine results using iterative updates for each node.	2254419	4	3	36.9
GraphChi-vEN Workloads [36]					
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.	2254419	4	15	52.2
Connected Components (CC)	GraphChi-vEN CC is similar to GraphChi-vE described above. However, GraphChi-vEN CC has both virtual edges and nodes.	2254419	4	15	44.2
Page Rank (PR)	GraphChi-vEN PR is similar to GraphChi-vE described above. However, GraphChi-vEN PR has both virtual edges and nodes.	2254419	4	10	54.4
Open Source Ray Tracer [40]					
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.	1000	3	3	15.4

in Section 8.1. We also use the trace-based simulator to evaluate the effect of *TypePointer* when using the CUDA allocator (Section 8.2).

We evaluate eleven representative applications from different scalable, multi-threaded CPU frameworks [35–37, 40] and contemporary object-oriented GPU workloads [22, 46]. The workloads we study focus on graph analytics [35–37], model simulations [22, 46] and raytracing-based rendering [40], where parallel, object-oriented programming is a natural fit. The workloads and their characteristics are listed in Table 2.

8 EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness of both *COAL* and *TypePointer* compared to the contemporary CUDA implementation of virtual functions, the type-tag implementation in Intel's Concord [6] work and our type-based *SharedOA* allocator. We perform functional validation on all the implementations to guarantee they produce the same results. We study the following techniques:

CUDA: We implement the workloads using the default CUDA virtual function implementation mechanism. By default, CUDA does not support shared objects between the CPU and GPU. Therefore, an object initialization kernel is run prior to executing the compute kernels. The time required for this initialization is not included in the CUDA results.

Concord: We support virtual function calls as described in Concord [6], where a *type* field is embedded in each object instead of a virtual function pointer. When a virtual function is called, the compiler inserts a switch statement, which reads the embedded type tag, then jumps to the appropriate function body. Concord does not support true runtime polymorphism, since the call targets must be known at compile-time.

SharedOA: We implemented the type-based shared object allocator as described in Section 4. Objects of the same type are allocated together in the same contiguous region.

COAL: We implement COAL as described in Section 5. *COAL* is built on top of *SharedOA*, hence the performance improvements demonstrated by *COAL* over CUDA and Concord are a combination of the allocator effects and the effect of removing vTable* lookups.

TypePointer: We implement *TypePointer* as described in Section 6 inside the *SharedOA* allocator described in Section 4. To apply *TypePointer*, we perform instrumentation at each virtual function call to get the vFunc* and add instructions at member variable references

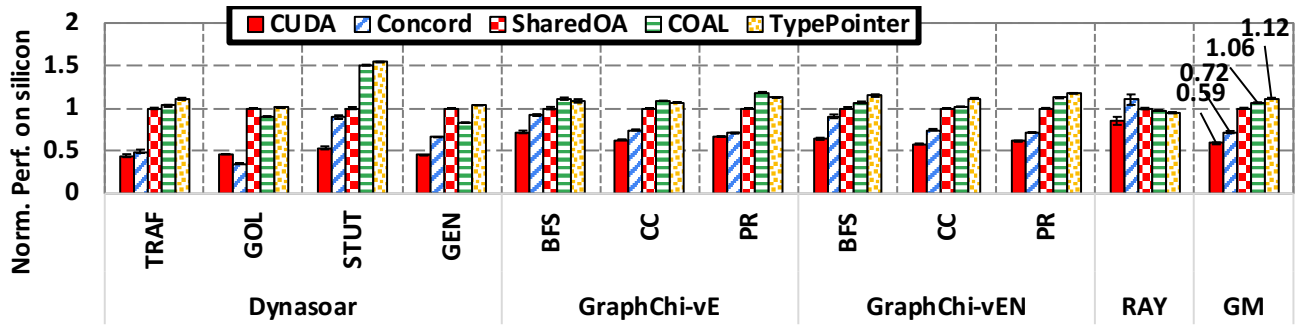


Figure 6: Performance, normalized to *SharedOA* on a silicon V100 GPU, averaged over 10 runs (error-bars=max and min).

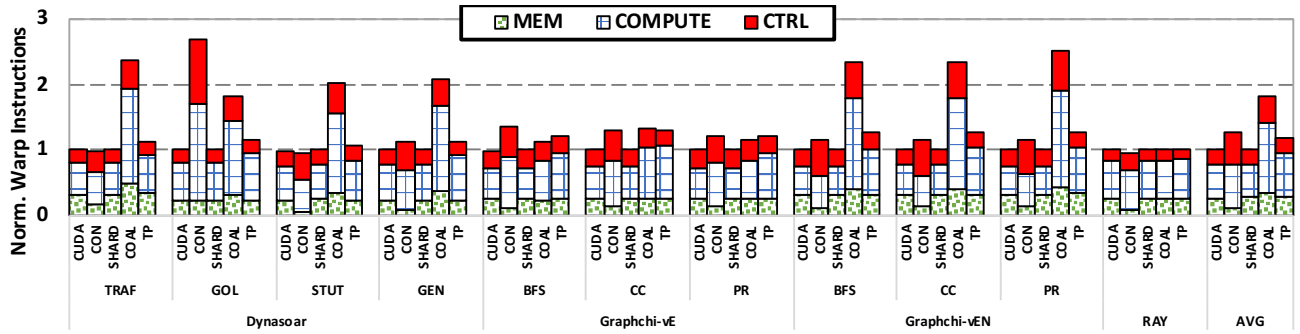


Figure 7: Dynamic warp instruction breakdown for CUDA, Concord (CON), *COAL* and *TypePointer* (TP) normalized to *SharedOA* (SHARD). We break instruction types into memory (MEM), compute (COMPUTE) and control (CTRL).

to remove the *TypePointer*'s type bits. In order to provide a clean comparison of *TypePointer* to *COAL*, we implement *TypePointer* on top of our custom *SharedOA* allocator. However, since *TypePointer* is allocator-independent, we also evaluate the effect it has when applied to the default CUDA allocator in Section 8.2.

8.1 Experiments on COAL and TypePointer

Figure 6 plots the speedup of CUDA, Concord, *COAL* and *TypePointer*, normalized to *SharedOA*. CUDA and Concord suffer a performance loss versus *SharedOA*. Packing objects of the same type into the same region of memory has a net positive impact on performance that the default CUDA allocator is not able to capture. Adding *COAL* and *TypePointer* on top of *SharedOA* further improves performance over *SharedOA* by 6% and 12% respectively. Although *COAL*'s geomean performance improvement over *SharedOA* is somewhat muted, there is wide variation in the magnitude of the improvement. In STUT, *COAL* achieves a 1.5× improvement over *SharedOA*, while in GEN and GOL, *COAL* suffers a slight performance regression versus *SharedOA*, but still significantly outperforms CUDA and Concord. Since Concord avoids loading a true vFunc* (Ⓑ in Figure 1a), it demonstrates some performance improvement over CUDA, at the expense of flexibility and an increase in code size comparing to CUDA. However, Concord must still access a field in each object to determine the object's type, which is an operation similar to Ⓐ in Figure 1a. If each thread is accessing a different object, this load will be highly diverged and overwhelm the memory system. *COAL* and *TypePointer* demonstrate a speedup because they eliminates this diverged load. Since *TypePointer* eliminates all the accesses for vTable* shown in Table 1, *TypePointer* achieves better performance than *COAL* on all the workloads. Interestingly, Concord performs slightly better for RAY. A closer inspection of RAY reveals that a number of the virtual function calls are not diverged. RAY has several loops where each thread accesses the same renderable object to determine if the ray collides with this object. In these instances, *COAL*'s static analysis will not instrument the virtual function since the overhead of the lookup is likely to be higher than the gain. For *TypePointer* on RAY, *TypePointer*'s performance is roughly equivalent to Concord (within the margin of error), since removing the vTable* accesses is a less significant bottleneck. Concord is able to take advantage of inter-procedural optimizations enabled by statically knowing all the potential call targets, which the other techniques (that support true dynamic dispatch) cannot.

Figure 7 shows the dynamic warp instruction breakdown for CUDA, Concord, *COAL* and *TypePointer* normalized to *SharedOA*. CUDA has the same instruction breakdown as *SharedOA* since the allocator does not affect the execution of instructions. To apply different techniques, Concord, *COAL* and *TypePointer* increase the instructions by 28%, 83% and 19%. Concord adds a number of compute and control instructions to the program and reduces memory instructions by half. This is due to the fact that Concord uses low overhead switch statements to replace high cost indirect function

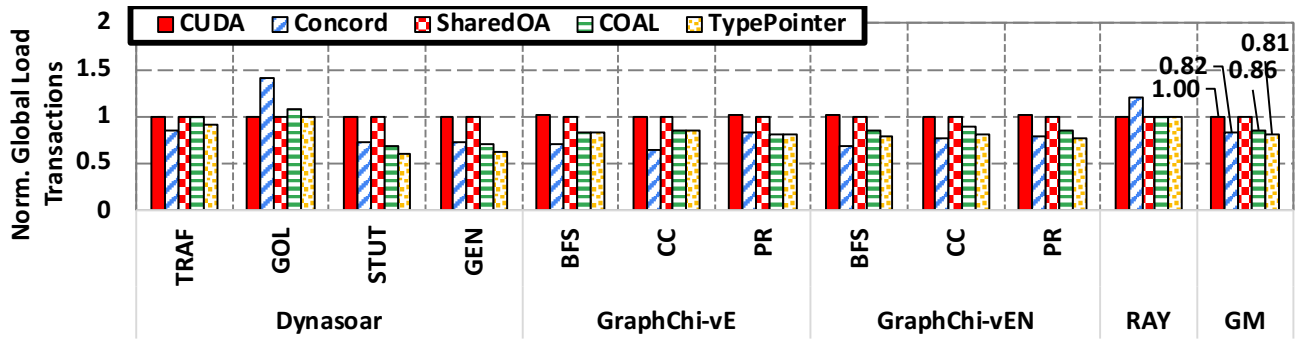


Figure 8: Global load transactions, normalized to *SharedOA* on a Silicon V100 GPU.

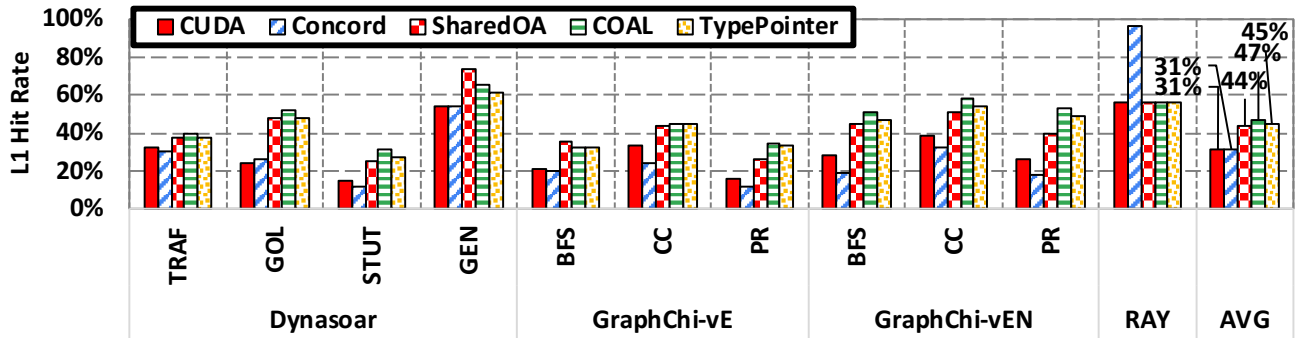


Figure 9: L1 cache hit rate on a silicon V100 GPU.

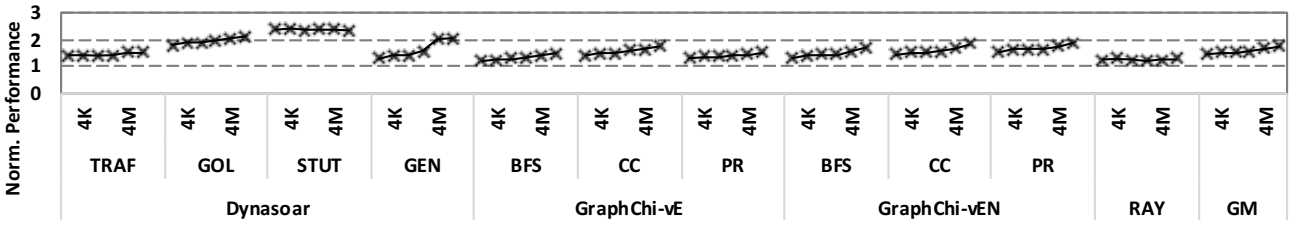
calls. *COAL* increases all categories of instructions, but the memory instructions have a higher cache hit rate and the compute instructions can be effectively hidden with multithreading. Versus *COAL*, *TypePointer* decreases both compute and memory instructions, requiring far fewer instructions to compute the object's vTable* and resulting in fewer memory transactions (Figure 8). Keep in mind that one dynamic warp instruction can generate up to 32 memory access, which is not reflected in examining the instruction breakdown in Figure 7 alone.

Figure 8 plots the number of global load transactions generated by the respective techniques. *COAL* reduces loads by 14% because it eliminates the diverged, low-locality vTable* load. However, additional global loads are added to perform the range check. Concord reduces loads by a greater fraction than *COAL* because it does not have the lookup overhead associated with performing the range check. However, as we will see in Figure 9, the loads generated by *COAL* are all to the same structure and hit in cache, whereas the L1 hit rate in Concord stays relatively constant versus CUDA. Since *TypePointer* does not access memory to find the object's vTable, it efficiently decreases global load transactions by an average of 19%.

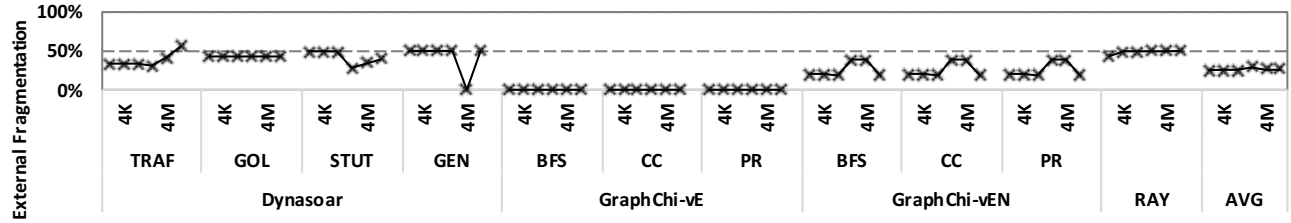
Finally, Figure 9 plots the L1 cache hit rate for CUDA, Concord, *SharedOA*, *COAL* and *TypePointer*. Although Concord removes a significant number of loads from the program, it generally decreases the hit rate of the L1 cache. Concord removes many of the L1 hits in the original CUDA program. *COAL*, on the other hand, sees a large increase in L1 hit rate on all the applications, with the exception of RAY, where *COAL* instruments relatively few instructions. The hit rate improves becasue *COAL* removes the diverged load (Ⓐ in Figure 1) that often missed in the L1 cache. All the extra loads instructions added to access the virtual range table in global memory turn into hits in the L1 cache, since the thousands of in-flight objects are composed of relatively few types (Table 2). RAY is an outlier for Concord, where the extra global loads added exhibit significant locality. The L1 hit rate for *TypePointer* remains relatively constant or even falls (versus *COAL*) in some applications, however, it generates fewer transactions than *COAL*.

8.2 Allocator Effects

As mentioned in Section 4, the memory allocator alone can have an impact on the performance of compute kernels. Although the details of the CUDA allocator are not public, we observe that it does not allocate objects of the same type consecutively and adds additional padding between allocated objects. Although not the primary focus of this paper, the more tightly-packed objects have a positive performance impact in our applications. Figure 6 shows the performance of *SharedOA* over CUDA, alongside the other techniques evaluated in the paper for context. *SharedOA* alone is able to out-perform CUDA by 41% because objects of the same type tend to be accessed together and packing them in the same region decreases divergence and increases cache performance. Applying *COAL* on top of *SharedOA* improves performance by a further 6%. In some apps, like STUT, *COAL* significantly improves performance over *SharedOA*, where in others, the overhead of *COAL* causes a small performance drop, although *COAL* is always significantly



(a) *COAL*'s performance with different initial chunk sizes noramlized to CUDA on a silicon V100 GPU. Each *X* increases the initial region size by 4× from 4k to 4M objects.



(b) *SharedOA* external fragmentation with different initial chunk sizes on a silicon V100 GPU. Each *X* increases the initial region size by 4× from 4k to 4M objects.

Figure 10: Effect of the allocator's initial region size on performance and fragmentation.

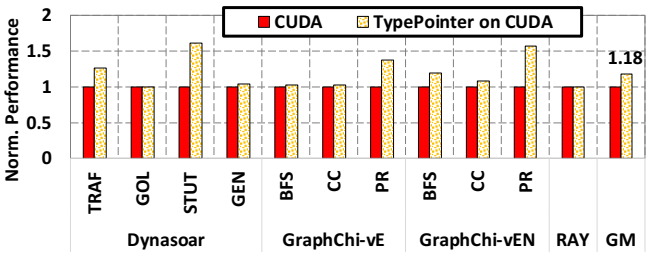


Figure 11: *TypePointer* performance on the CUDA allocator in a V100 GPU simulation.

better than CUDA. Using *TypePointer* instead of *COAL* removes the performance overhead from *COAL*.

COAL's performance can vary based on the number of objects initially allocated per region, before the region doubling starts. Figure 10a sweeps the number of objects in the initial region size from 4k to 4M. Generally, the performance of *COAL* is stable across initial region sizes, with only GEN demonstrating a significant performance increase at 2M. To better understand the fragmentation caused by our proposed allocation scheme, Figure 10b plots *SharedOA*'s external fragmentation when the number of the objects in the initial region size ranges from 4k to 4M. The fragmentation varies from 17% at 128k to 27% at 4M. Similar to other small-object allocators, *SharedOA* does not suffer from internal fragmentation.

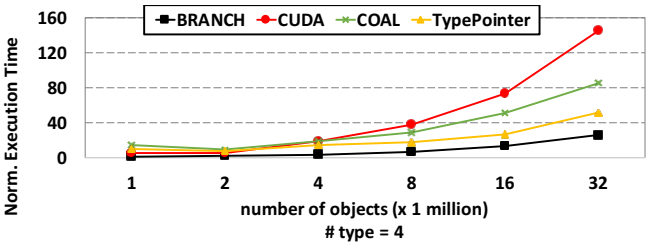
To show that *TypePointer* is not dependent on a particular memory allocator, we also evaluated it in simulation on the default CUDA memory allocator, which is plotted in Figure 11. Overall, *TypePointer* is able to improve performance by 18% without any major changes to the CUDA allocator.

Although not the focus of this paper, we also performed an evaluation of our *SharedOA* allocator's object initialization performance versus the CUDA allocator. For the object initialization phase, *SharedOA* outperforms the default CUDA allocator by a geometric mean 80× over all our applications. Since *SharedOA* is a host-side allocator, we eliminate the huge synchronization overhead imposed by performing device-side allocation of objects with virtual functions.

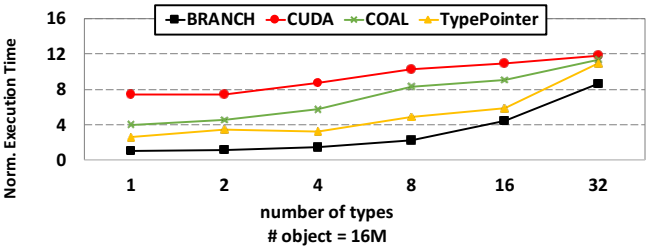
8.3 Scalability Study

To understand performance as types and objects scale, we perform a study using a set of microbenchmarks that have high vFuncPKI. One benchmark uses the standard CUDA implementation, where objects are allocated on the GPU and use virtual function calls (CUDA). Another use branches to arbitrate different "types" (BRANCH), without any objects or object-oriented program. In all benchmarks, threads scale with the number of objects and the compute inside the function call is a simple addition operation. BRANCH decides which function to call based on register values and does not access memory for the function call. It represents control flow on GPUs without memory overhead.

Using 4 types, we vary the number of objects from 1 million to 32 million in Figure 12a. As the number of objects increase, the slowdown for CUDA versus BRANCH reaches 5.6×, while *COAL* and *TypePointer* continue to linearly track BRANCH, demonstrating 3.3× and 2.0× slowdown over the idealized BRANCH microbenchmark at 32M objects. In Figure 12b, we fix the number of objects at 16M and scale the number of types being accessed by one warp. As the number of types accessed by one warp increases, the performance universally degrades across all the benchmarks, since branch divergence increases. COAL consistently tracks closely to BRANCH. At 32 types, the relative difference between the various methodologies becomes small. In highly diverged code, the overheads associated with virtual functions are less pronounced. Since fewer threads are active, memory has less contention and most of the performance loss comes from poor SIMD utilization.



(a) Execution time, normalized to BRANCH with 1M objects, on a silicon V100 GPU as the number objects scale.



(b) Execution time, normalized to BRANCH with 1 type, on a silicon V100 GPU as the number of types scale.

Figure 12: Scalability experiments on microbenchmarks.

9 RELATED WORK

GPU Programmability. Prior work on supporting productive languages in GPUs [28, 41, 45] has primarily focused on supporting primitives and simple data structures on the GPU, opting to avoid the use of virtual functions in GPU code. We explicitly focus on efficiently executing GPU virtual function calls, rethinking their implementation in massively multithreaded environments by employing a coordinated effort between the compiler and runtime system. A body of work also exists on providing CPU-like programmability for GPUs. Support for a file system abstraction [44], network stack [34] and more advanced memory management [42] are examples of this. Although these works share the same motivation as ours, they are focused on other aspects of programmability.

GPU benchmarks. There are numerous GPU benchmark suites [5, 10, 13, 48], and object-oriented CPU suites [8, 24, 51]. However, there are no publicly available GPU benchmark suites that contain object-oriented applications with virtual function calls. We believe this is, in part, because the performance implications we study have not been explored before.

Virtual Function Calls on CPUs. A vast amount of CPU work has improved indirect branch/indirect jump [20] prediction [29, 30, 33], addressing the performance loss from misspeculation on CPUs. Other works have looked at profile-guided techniques [11, 12, 20] for increasing single-threaded performance and making code better suited to conditional branch predictors. However, a fundamental difference between CPUs and GPUs is that GPUs do not use any speculative execution and the cost of profiling and recompiling thousands of concurrently executing threads is high. On the software side, Just-In-Time (JIT) compilation techniques in managed languages like Python and Java have removed much of the overhead via profiling techniques that recompile the code and inline virtual function calls at their call sites [17–19, 23, 26, 49, 52]. Dynamically recompiling GPU code on the fly is a challenge for several reasons. First, any inlining would have to be performed on a per-thread basis, as the same call-site will be used by thousands of threads, accessing thousands of objects. Second, there is no on-device JIT compiler for contemporary GPUs.

Object-Oriented Programming On GPUs. In GPUs, the SIMT stack mechanism to handle an indirect branch has been patented [15]. Barik et al. [6] develop Concord, which enables a subset of C++ to execute on integrated Intel CPU/GPU systems. Concord does not support virtual function calls through indirect branches, but instead relies on a set of if/else conditionals that test the object type and must still dereference the object pointer to determine its type. Other work [28, 41, 45] has supported advanced programming languages on GPUs, (e.g., Java and Ruby). However, no work has quantitatively analyzed the performance of virtual functions, or proposed a solution to the significant overhead they incur.

Memory Allocation On GPUs. A set of work [4, 25, 27, 47] attempts to implement efficient memory allocation on parallel architectures. Xmalloc [27] aims to implement a scalable GPU memory allocator with lock-free buffers to hold pre-defined size truncks and bins. Issac al. [25] formulate resource allocation to two-stages to improve the allocation throughput on Nvidia GPUs. ScatterAlloc [47] utilizes CUDA's dynamic allocator to perform coarse grained allocation, employing bitmaps to prevent collision. Halloc [4] uses bit arrays to represent free blocks and a hash function to search. DynaSOAr [22, 46] implements the first parallel memory allocator for object-oriented programs on GPUs. Like *SharedOA*, many of these allocators allocate data in large chunks, however they do it to improve allocation speed. We allocate large chunks of data such that we can determine an object's type by its address range.

Type-based management On CPUs. Prior work on supporting type-based management or addressing in object-oriented programs on CPUs [43, 50] has focused on improving garbage collection or reducing memory consumption. To the best of our knowledge, we are the first work on either CPUs or GPUs to perform virtual function calls without accessing objects to determine their type.

10 CONCLUSION

We examine the effects of executing object-oriented code on massively parallel architectures. Through a detailed analysis of the direct cost of virtual functions on a silicon V100 GPU, we demonstrate the memory system is the performance bottleneck, in particular the loads to the virtual table pointer in each object. To alleviate this pressure, we propose two techniques, one completely in software (*COAL*) and one with hardware support (*TypePointer*) that determine an object's type based only on the object's address. We study this problem on realistic object-oriented workloads, by implementing a shared object allocator that allows the CPU and GPU to share objects with virtual functions through unified virtual memory.

The software-only technique, (*COAL*) uses the runtime memory allocator to place objects of the same type in a set of ranges.

The compiler then inserts code to scan the address ranges and determines which function is called based only on the object's address. We evaluate *COAL* on a silicon GPU and demonstrate a 80%, 47% and 6% performance improvement over contemporary CUDA, prior academic work and our newly-proposed *SharedOA*, respectively. The second technique eliminates the lookup overhead of *COAL* with *TypePointer*. *TypePointer* encodes each object's type in unused bits in the GPU's 64-bit address space. Using a small modification to the GPU's MMU, *TypePointer* locates the vTable without accessing memory, improving performance by 90%, 56% and 12% over CUDA, previous work and *SharedOA*, respectively. We also implement *TypePointer* in simulation, applied on top of the default CUDA allocator and demonstrate an 18% performance improvement without changing how objects are allocated.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Alexandre Passos and the 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.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains the source code for the *SharedOA*, *COAL*, and *TypePointer* that applied to all workloads. We also include the instructions to configure, build, run, and acquire the workload's performance. Users can reproduce the results in Figure 6. We also contain a tutorial with examples to apply *SharedOA*, *COAL* and *TypePointer* to show that the three techniques are reusable on other CUDA applications.

A.2 Artifact Checklist

We list the artifact checklist below to formally describe our artifact:

- **Program:** Ported version of dynasoar, GraphChi and raytracing Workloads are included in the measurement repository.
- **Compilation:** GCC 7.5.0, CUDA 10.1, 10.2 or 11.1.
- **Transformations:** *COAL* and *TypePointer* requires PTX transformations which are implemented by python scripts. PyYAML is needed for the scripts.
- **Data set:** Included in the measurement repository.
- **Run-time environment:** Ubuntu 18.04.5
- **Hardware:** Intel x86 machine with NVIDIA Volta architecture GPU with at least 8GB GPU memory. We use V100 GPU with 32GB GPU memory in our experiments.
- **Execution:** NVProf commandline profiler from CUDA is used to measure the kernel execution time of each workloads.
- **Metrics:** Normalized performances are reported by scripts for each version of the workloads.
- **How much disk space required (approximately)?:** At least 1GB is needed to contain the measurement repository.
- **How much time is needed to complete experiments (approximately)?:** 20 minutes to compile and 40 minutes to run.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD 2-Clause "Simplified" License
- **Data licenses (if publicly available)?:** BSD 2-Clause "Simplified" License
- **Workflow framework used?:** Scripts are provided in the measurement repo, no other framework required.
- **Archived (provide DOI)?:** Provided in Github, and a snapshot in https://doi.org/10.5281/zenodo.4319923.

A.3 Description

A.3.1 How to Access. We provide the artifact evaluation with Github repositories:

- Evaluation scripts: https://github.com/brad-mengchi/asplos21_ae_script
- Measurement repository to compile and run the workloads: https://dgithub.com/brad-mengchi/asplos_2021_ae
- Tutorial for applying the *SharedOA*, *COAL* and *TypePointer*: https://dgithub.com/purdue-aalp/SharedOA

We also provide zipped version of the repositories on Zenodo: https://doi.org/10.5281/zenodo.4319923.

A.3.2 Hardware Dependencies. The *SharedOA*, *COAL* and *TypePointer* works on Intel x86 machine with NVIDIA Volta architecture GPU with at least 16GB GPU memory. The experiments in this paper use V100 GPU with 32GB GPU memory. 1GB CPU memory is required to contain the evaluation repositories.

A.3.3 Software Dependencies. Ubuntu 18.04 Linux is preferred to run the experiments. NVIDIA CUDA 10.1, 10.2 and 11.1 and GPU driver are required to compile and run the GPU workloads.

A.4 Installation from Github

We include the evaluation scripts in asplos21_ae_script repository and detail the instructions in README.md. Users need to clone the evaluation script repository to a directory:

```
# git clone https://github.com/brad-mengchi/asplos21_ae_script
```

We include setup.sh to configure environments for CUDA. CUDA_INSTALL_PATH need be set to the CUDA directory:

```
# export CUDA_INSTALL_PATH=<CUDA directory path>
```

Users can finish the configuration after setting up the environment:

```
# source setup.sh
```

A.5 Experiment Workflow

Users can use compile.sh to clone the workload repository and compile the workloads:

```
# source compile.sh
```

This compile.sh script clones the workload repository from https://github.com/brad-mengchi/asplos_2021_ae and builds the workloads. We generates 55 binaries for 11 workloads with different techniques (CUDA, Concord, *SharedOA*, *COAL* and *TypePointer*). All workloads binaries will be in the directory asplos_2021_ae/benchmarks/bin/10.1/release/.

A.6 Evaluation and Expected Result

Users can use run.sh and get.sh to run the experiments and get the normalized performance on specific GPU. To run the experiments on GPU 0 with command below:

```
# source run.sh 0
```

To get the statistics after running on GPU 0 with command below:

```
# source get.sh 0
```

The script print out the normalized performance for workloads with techniques like below:

```
trafficV 0.443328212062
trafficV_CONCORD 0.469073964166
trafficV_MEM 1.0
...
RAY_COAL 0.945034105267
RAY_TP 0.935876022056
```

Above results can verify the experiments in Figure 6.

A.7 Tutorial for *SharedOA* with Examples

SharedOA, *COAL* and *TypePointer* techniques can be easily reusable on other CUDA applications. Therefore, we create a tutorial to show examples to apply three techniques on a simple program. We provide the tutorial on Github repository: https://github.com/purdue-aalp/SharedOA. The detailed instructions are in README.md. Users can clone this tutorial repository with:

```
# git clone https://github.com/purdue-aalp/SharedOA
```

The example for *SharedOA* are in `example/SharedOA/` directory, so users can build and run *SharedOA* example with the following command:

```
# cd example/SharedOA
# make
# ./main
```

A.8 Tutorial for *COAL* and *TypePointer* with Examples

The examples for *COAL* and *TypePointer* are in example/COAL/ and example/TP/ directories. Users can build and run *COAL* example with the following commands:

```
# cd example/COAL
# make
# ./main_COAL
```

Users can also build and run *TypePointer* example with the following commands:

```
# cd example/TP
# make
# ./main_TP
```

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

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

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

[3] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. 1989. Dynamic Typing in a Statically-Typed Language. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Austin, Texas, USA). https://doi.org/10.1145/75277.75296

[4] Andrew V Adinetz and Dirk Pleiter. [n.d.]. Halloc. https://github.com/canonizer/halloc.

[5] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[6] Rajkishore Barik, Rashid Kaleem, Deepak Majeti, Brian T. Lewis, Tatiana Shpeisman, Chunling Hu, Yang Ni, and Ali-Reza Adl-Tabatabai. 2014. Efficient Mapping of Irregular C++ Applications to Intergrated GPUs. In *International Symposium on Code Generation and Optimization (CGO)*. 33–43. https://doi.org/10.1145/2581122.2544165

[7] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. ACM, 117–128. https://doi.org/10.1145/378995.379232

[8] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Lee Han, Eliot Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Vol. 41. ACM, 169–190. https://doi.org/10.1145/1167515.1167488

[9] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Boston, Massachusetts) *(USTC'94)*. USENIX Association, USA, 6. https://dl.acm.org/doi/10.5555/1267257.1267263

[10] M. Burtscher, R. Nasre, and K. Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[11] Brad Calder and Dirk Grunwald. 1994. Reducing Indirect Function Call Overhead in C++ Programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 397–408. https://doi.org/10.1145/174675.177973

[12] C. Chambers, D. Ungar, and E. Lee. 1989. 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*. 49–70. https://doi.org/10.1145/74878.74884

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[14] Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas. 2017. ShortCut: Architectural Support for Fast Object Access in Scripting Languages. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3079856.3080237

[15] B.W. Coon, J.E. Lindholm, P.C. Mills, and J.R. Nickolls. 2010. Processing an indirect branch instruction in a SIMD architecture. https://www.google.com/patents/US7761697 US Patent 7,761,697.

[16] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1997. *Computational geometry*. Springer. 231–237 pages. https://doi.org/10.1007/978-3-540-77974-2

[17] Jeffrey Dean, Craig Chambers, and David Grove. 1995. Selective Specialization for Object-oriented Languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. 93–102. https://doi.org/10.1145/223428.207119

[18] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*. 77–101. https://dl.acm.org/doi/10.5555/646153.679523

[19] David Detlefs and Ole Agesen. 1999. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming*. https://dl.acm.org/doi/10.5555/646156.679839

[20] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 297–302. https://doi.org/10.1145/800017.800542

[21] Karel Driesen and Urs Hölzle. 1996. The Direct Cost of Virtual Function Calls in C++. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/236337.236369

[22] Dynasoar 2019. *CUDA Dynamic Memory Allocator for SOA Data Layout*. Retrieved June 18, 2020 from https://github.com/prg-titech/dynasoar

[23] Izzat El Hajj, Thomas B. Jablin, Dejan Milojicic, and Wen-mei Hwu. 2017. SAVI Objects: Sharing and Virtuality Incorporated. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Vol. 1. ACM, New York, NY, USA, Article 45, 24 pages. https://doi.org/10.1145/3133869

[24] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. 2011. 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)*. ACM, 319–332. https://doi.org/10.1145/1961296.1950402

[25] Isaac Gelado and Michael Garland. 2019. Throughput-oriented GPU Memory Allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 27–37. https://doi.org/10.1145/3293883.3295727

[26] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1899661.1869634

[27] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. 2013. Scalable SIMD-parallel memory allocation for many-core machines. *The Journal of Supercomputing* 64, 3 (01 Jun 2013), 1008–1020. https://doi.org/10.1007/s11227-011-0680-7

[28] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. 2015. Compiling and Optimizing Java 8 Programs for GPU Execution. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1109/PACT.2015.46

[29] Jose A. Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, and Yale N. Patt. 2008. 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* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, New York, NY, USA, 80–90. https://doi.org/10.1145/1353535.1346293

[30] John Kalamatianos and David R. Kaeli. 1998. Predicting Indirect Branches via Data Compression. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society Press, 272–281. https://dl.acm.org/doi/10.5555/290940.290997

[31] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM. https://doi.org/10.1109/ISCA45697.2020.00047

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

[33] Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. 2007. VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-based Dynamic Devirtualization. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM, 424–435. https://doi.org/10.1145/1250662.1250715

[34] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI)*. https://doi.org/10.1145/2963098

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

[36] Aapo Kyrola. [n.d.]. GraphChi-Java. https://github.com/GraphChi/graphchi-java.

[37] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI)*. https://dl.acm.org/doi/10.5555/2387880.2387884

[38] NVIDIA. 2018. NVIDIA profiling tools. https://docs.nvidia.com/cu-da/profiler-users-guide/index.html. Accessed Aug 20, 2018.

[39] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers* (Los Angeles, California) *(SIGGRAPH '10)*. ACM, New York, NY, USA, Article 66, 13 pages. https://doi.org/10.1145/1778765.1778803

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

[41] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. 2012. Rootbeer: Seemlessly Using GPUs for Java. In *High Performance Computing and Communications, 2012. HPCC '12. 14th IEEE International Conference on*. 375–380. https://doi.org/10.1109/HPCC.2012.57

[42] Sagi Shahar, Shai Bergman, and Mark Silberstein. 2016. ActivePointers: A Case for Software Address Translation on GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3007787.3001200

[43] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. 2002. Exploiting Prolific Types for Memory Management and Optimizations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, NY, USA, 295–306. https://doi.org/10.1145/503272.503300

[44] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. https://doi.org/10.1145/2499368.2451169

[45] Matthias Springer and Hidehiko Masuhara. 2016. 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* (Santa Barbara, CA, USA) *(ARRAY 2016)*. ACM, New York, NY, USA, 25–31. https://doi.org/10.1145/2935323.2935327

[46] Matthias Springer and Hidehiko Masuhara. 2019. DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access. In *ECOOP*. http://drops.dagstuhl.de/opus/volltexte/2019/10809

[47] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. 2012. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *2012 Innovative Parallel Computing (InPar)*. 1–10. https://doi.org/10.1109/InPar.2012.6339604

[48] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nadi Obeid, Li-Wen Chang, Nasser Anssari, Greg Daniel Liu, and Wen-mei W. Hwu. 2012. *IMPACT Technical Report, IMPACT-12-01*. University of Illinois, at Urbana-Champaign.

[49] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/354222.353189

[50] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. 2007. Java Object Header Elimination for Reduced Memory Consumption in 64-Bit Virtual Machines. *ACM Trans. Archit. Code Optim.* 4, 3 (2007), 17–30. https://doi.org/10.1145/1275937.1275941

[51] D. Zaparanuks and M. Hauswirth. 2010. Characterizing the design and performance of interactive java applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 23 –32. https://doi.org/10.1109/ISPASS.2010.5452075

[52] Olivier Zendra, Dominique Colnet, and Suzanne Collin. 1997. Efficient Dynamic Dispatch Without Virtual Function Tables: The SmallEiffel Compiler. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/263698.263728