FACULTY OF ENGINEERING
AND ARCHITECTURE

Garbage collection abstractions for high-level GPU languages

Jonathan Van der Cruysse
Student number: 01709392

Supervisor: Prof. dr. ir. Bjorn De Sutter
Counsellor: Tim Besard

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2018-2019

GHENT
UNIVERSITY

Assent to loan

June 23, 2019

Preface

Before you lies my master's thesis "Garbage collection abstractions for high-level GPU languages," a detailed description of what I believe is the first implementation of fully transparent garbage collection for Graphics Processing Units (GPUs).

I have had an interest in compilers and managed languages since high school. This thesis afforded me a wonderful opportunity to cultivate that interest. It also challenged me to learn more about and work with things that I had no prior experience with, including GPUs and garbage collection—the main subject matters here.

I would like to extend my sincere gratitude to my supervisors, Prof. Bjorn De Sutter and Tim Besard, for guiding me, for introducing me to new and interesting paradigms and tools, and for their insightful feedback. I also wish to thank the Julia community in general and Valentin Churavy and Keno Fischer in particular for their warm reception of my pull requests.

I enjoyed discussing ideas with friends and family. I feel particularly thankful to my parents, who supported me and offered me sage advice. Finally, I would like to thank you, dear reader, for taking the time to read my master's thesis. I hope you enjoy it.

Jonathan Van der Cruysse

June 23, 2019

Garbage collection abstractions for high-level GPU languages

Master's dissertation submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering

Jonathan Van der Cruysse
Supervisors: Prof. dr. ir. Bjorn De Sutter, Tim Besard
Faculty of Engineering and Architecture, Ghent University

Abstract

Scientific computing is evolving in two major directions: (1) increased GPU use to take advantage of evolving hardware and (2) high-level programming languages such as Julia that offer both excellent programmer productivity and performance. Recent research reconciles these two evolutions by compiling high-level languages to GPU instructions.

Traditionally, GPUs are programmed using dedicated low-level programming languages, e.g., CUDA and OpenCL. Garbage Collector (GC)-managed object allocation, a language feature that is crucial to high-level languages, does not have an equivalent in CUDA and OpenCL's programming models. Hence, state-of-the-art high-level language to GPU compilers either refuse to compile object allocations or implement object allocation by allocating memory and then never freeing it, perpetually leaking memory until no more objects can be allocated.

We extend the CUDAnative Julia-to-GPU compiler with the first fully transparent GC for GPU kernels. To build our GC, we introduce lightweight, low-level, platform-agnostic GC abstractions in the Julia compiler. The abstractions that underpin our GC are designed to support alternative GC implementations equally well. Our GC improves on the state of the art by implementing object allocation in a fully transparent way and without memory leaks. Liberally-allocating GPU kernels equipped with our GC obtain a mean speedup of 2× over trivial memory management based on CUDA `malloc`, the preferred approach in the state of the art.

Keywords: GPU, Julia, Compilers, Garbage Collection

Garbage collection abstractions for high-level GPU languages

Jonathan Van der Cruysse

Supervisors: Bjorn De Sutter, Tim Besard

Abstract—**Recent research reconciles high-level languages with Graphics Processing Unit (GPU) programming by compiling the former to instructions for the latter.**

Traditionally, GPUs are programmed using dedicated low-level programming languages, e.g., CUDA and OpenCL. Garbage Collector (GC)-managed object allocation, a language feature crucial to high-level languages, does not have an equivalent in CUDA and OpenCL's programming models. Hence, state-of-the-art high-level language to GPU compilers either refuse to compile object allocations or implement object allocation by allocating memory and then never freeing it, perpetually leaking memory until no more objects can be allocated.

We extend the CUDAnative Julia-to-GPU compiler with the first fully transparent GC for GPU kernels. To build our GC, we introduce lightweight, low-level, platform-agnostic GC abstractions in the Julia compiler. These abstractions are designed to support alternative GC implementations equally well. Our GC improves on the state of the art by implementing object allocation in a fully transparent way and without memory leaks. Liberally-allocating GPU kernels equipped with our GC obtain a mean speedup of 2× over trivial memory management based on CUDA `malloc`, the preferred approach in the state of the art.

Keywords—**GPU, Julia, Compilers, Garbage Collection**

I. Introduction

As the field of high-performance computing evolves, two salient trends are emerging.

First and foremost: Scientific computing applications nowadays typically rely on GPUs rather than Central Processing Units (CPUs) as carefully controlled use of the former permits order-of-magnitude speedups compared to typical use of the latter. For this reason, GPU computing has become mainstream in a wide range of scientific applications that includes weather forecasting [12], geospatial indexing [32], chemistry [31], medicine [23] and machine learning [1].

A second trend is the appearance of productive, high-level scientific computing languages that match the performance of low-level programming languages such as Fortran, C and C++. The Julia programming language is a prominent example: The performance of Julia programs can match that of equivalent C/C++ programs in spite of the fact that Julia offers many core language features—dynamic typing, garbage collection, etc.—that are typically associated with "slower" programming languages such as Python, Ruby and Perl [6].

These two trends—increased hardware performance and increased programmer productivity—are wedded by projects that compile high-level programming languages to instructions suitable for execution on the GPU. CUDAnative is an example of such a project: It compiles Julia code for NVIDIA GPUs [4].

Because of the unique characteristics of GPUs and a lack of mature infrastructure in GPU programming environments, projects that compile high-level languages for GPUs often restrict the set of supported language features. In CUDAnative's case, Julia language constructs that map well to the low-level CUDA programming model for NVIDIA GPUs are supported, but others are not. Consequently, CUDAnative's GPU *kernels*, that is, GPU programs, are written in a low-level dialect of Julia that precludes idiomatic features such as mutable `struct` types, arrays, exception handling, and others.

In addition to reducing the feature set available to manually written GPU kernels, language features not supported by CUDAnative also break compositional schemes that attempt to transparently expose the compute capabilities of GPUs. To name one example, the `CuArray` type mimics the functionality of Julia's normal array type, but transparently runs array computations on the GPU to obtain superior performance [4]. `CuArray` hides the complexity of GPU programming from its users by automatically synthesizing GPU kernels for common operations. These automatically synthesized kernels are also affected by CUDAnative's restricted set of permissible language features. This forces programmers to give special consideration to how `CuArray` instances are used, rather than treating them as fully-featured implementations of an abstract array type.

Many of the features that are not supported by CUDAnative have one thing in common: They rely on the presence of some automatic memory management scheme—specifically, they expect a GC to be in place.

GPU computing has historically relied on low-level languages to construct programs suitable for execution on a GPU. These low-level languages never necessitated the creation of a GC for GPUs. Consequently, there exist—to the best of our knowledge—no drop-in GCs for GPU kernels. GC algorithms have been developed for GPUs, but these experiments required programmer intervention for interrupting kernels when memory runs out, triggering an actual collection and restoring the kernel's state [28].

In this work, we make the following main contributions:

1. We design GPU-friendly compiler abstractions for

garbage collection and implement these abstractions in the Julia compiler.

2. Based on these abstractions, we design a transparent GC for GPUs kernels. We implement our GC in CUDAnative, targeting Julia code that is compiled for NVIDIA GPUs.

II. Background

This section briefly describes the state of the art in Julia-to-GPU compilation, emphasizing topics that are material to the contributions presented in the next sections.

A. GPU programming

GPU programming can deliver large performance increases over traditional CPU programming. This performance boost can be attributed to the execution model that underpins GPU programming: CPUs are optimized for executing sequential instruction streams whereas GPUs are optimized for executing instructions in a massively parallel fashion.

CUDAnative, which we will shortly discuss in more detail, targets NVIDIA GPUs only. Since our contributions are implemented as changes to Julia and CUDAnative, we will restrict ourselves to the CUDA programming model for NVIDIA GPUs.

GPU programming diverges from CPU programming in a number of ways. The most prominent differences between the two programming paradigms are related to parallelism and memory management.

1) Parallelism

Parallelism is a fundamental feature of GPUs that deeply affects both functional and non-functional aspects of GPU programs. We introduce the following CUDA terminology:

- A *kernel* is a GPU program. Every launched kernel consists of a set of threads.
- A *thread* is a semi-independent unit of execution. All threads in a launched kernel execute the same program. The control flow of threads can diverge based on the data they process.
- Threads are grouped by the hardware into *warps:* fixed-size collections of threads.
- Threads are grouped by programmers into *blocks*. The threads in a block can synchronize and communicate efficiently through shared memory [26].

NVIDIA GPUs implement Single Instruction, Multiple Threads (SIMT), an idiosyncratic version of the popular Single Program, Multiple Data (SPMD) programming model [11, 18]. Threads in different warps run in parallel and independently of each other, like CPU threads. Inside of a warp, threads exhibit a peculiar form of parallelism that characterizes SIMT: Instructions from threads in the same warp are executed in lockstep and in parallel. That is, at every step, threads in a warp execute the same instruction on different data in parallel. When control flow in these threads diverges, the divergent control-flow paths are executed sequentially until control flow reconverges. This phenomenon, where different threads in the same warp execute different control-flow paths, is known as *thread divergence* [19, 27]. Thread divergence impacts functional and non-functional aspects of GPU kernels. On the non-functional side, thread divergence reduces performance. On the functional side, thread divergence can cause thread starvation: Until the currently running threads complete, the other threads will not make any progress at all [9].

2) GPU memory

NVIDIA GPUs feature a complex partitioning of on-chip memory that includes global, constant, texture, shared, and local memory spaces. Each of these memory spaces are optimized for different use cases and differ in both their functional and non-functional properties [19, 27]:

- **Global memory** is read-write global memory.
- **Constant memory** is read-only global memory.
- **Texture memory** is read-only global memory that is optimized for 2D access patterns.
- **Shared memory** is read-write per-block memory.
- **Local memory** is read-write per-thread memory.

In addition to these various kinds of on-chip memory, GPU kernels may also access host memory directly, provided that it is *page-locked.* Page-locked memory, also known as *pinned* memory, is guaranteed by the operating system to always be present in random-access memory—it will never be paged out to disk.

Finally, *unified memory* is similar to pinned memory in that it is accessible to both CPUs and GPUs. Unlike pinned memory, which is essentially CPU memory that is shared with the GPU, unified memory pages automatically migrate to the device that is currently using them, which can be either a CPU or a GPU [13].

B. The Julia programming language

The Julia programming language [5, 10] is a dynamic programming language in a similar vein to Python, Matlab and R. Julia sets itself apart from these other dynamic languages in being designed with speed in mind: Julia is Just In Time (JIT) compiled to native code by the industrial-strength LLVM compiler framework [15].

C. CUDAnative

CUDAnative is a Julia package that compiles Julia code to Parallel Thread Execution (PTX) instructions suitable for execution on NVIDIA GPUs [4, 20]. Kernels are encoded as regular Julia functions, which are transparently compiled to GPU instructions by CUDAnative.

CUDAnative is designed to take advantage of existing Julia compiler infrastructure to compile Julia code for NVIDIA GPUs. Code that is compiled by CUDAnative passes through approximately the same machinery that is used for compiling regular Julia code. CUDAnative intercepts LLVM Intermediate Representation (IR) generated by the Julia compiler just before that IR becomes target-specific and sends it to LLVM's PTX back-end instead of sending it to the code generator for the CPU, as the unmodified, CPU-targeting Julia compiler would have done [25]. This mechanism allows CUDAnative to compile low-level Julia code for GPUs.

The Julia compiler relies on a runtime library to implement sophisticated language features, such as garbage collection, exception handling, and dynamic multiple dispatch. Julia's runtime library is CPU-specific and hence CUDAnative cannot reuse it. Whenever the Julia compiler generates a call to the runtime library, CUDAnative must cope with that call somehow. Broadly speaking, CUDAnative has two mechanisms for handling calls to the Julia runtime library.

1. If the runtime library function can reasonably be re-implemented for GPUs, then CUDAnative can supply its own version of the runtime library function and link it with kernels.

2. If the runtime library function's semantics or signature rule out a GPU implementation, then CUDAnative throws a compile-time error. In effect, this disables the use of a language feature for all CUDAnative kernels.

GC-managed object allocation straddles these two categories. Prior to this work, object allocation in CUDAnative was implemented as a GPU runtime library function that used a trivial memory management scheme, calling the CUDA `malloc` function to allocate memory but never returning said memory when the object became unnecessary, effectively creating a memory leak. Furthermore, memory leaked by a kernel was not reclaimed when the kernel terminated, meaning that memory eventually ran out if a program kept on launching kernels, even if those kernels used memory sparingly.

Hence, the previous proof-of-concept implementation of object allocation only partially satisfied the semantics of the Julia runtime function it implemented: It dutifully allocated new memory, but failed to recycle that memory when no longer in use.

III. Garbage collection abstractions for Julia

The first main contribution of this work is the addition of a new layer of abstraction to the Julia compiler. Our abstraction allows for easy implementation of generational, non-moving, precise GCs [29] without overly constraining that interface between the application and the GC.

A. *Requirements*

As a novel contribution, we introduce two metrics for evaluating the appropriateness of abstractions in general and apply them to the problem at hand:

1. **Robustness**: The degree to which an abstraction can accommodate divergent implementations of some feature.

2. **Thinness**: The amount of engineering required to lower the abstraction to something concrete.

One can easily come up with an abstraction that is maximally robust and minimally thin: The source code being compiled, for example. Similarly, a maximally thin and minimally robust abstraction would be a compiler's output. In practice, what we want is an abstraction that is sufficiently robust to accommodate the implementations we have in mind while being as thin as possible given this constraint.

B. *Preexisting abstractions*

The Julia compiler has two relevant preexisting abstractions for garbage collection. The primary difference between these abstractions is how they represent the *root set,* that is, the set of all object references in global variables and active function frames. Any GC needs to be able to inspect or approximate said root set to know which objects are alive and which ones are dead.

The first abstraction we will discuss is *fully lowered LLVM IR:* The final form of the LLVM IR produced by Julia, just before that IR is sent to the CPU-specific code generator. At this level of abstraction, the root set is represented by a linked list of stack-allocated arrays of root pointers. Each such array of root pointers is called a *GC frame.* The fact that GC frames are allocated on the stack makes this abstraction insufficiently robust for our GC, which allocates objects on the GPU and collects them on the CPU. The GPU can access its own stack memory, but the CPU cannot access GPU stack memory. Because the CPU is responsible for collections, both the CPU and GPU must be able to access the root set, which is not the case for fully lowered LLVM IR.

By taking a step backward in the Julia compiler's pipeline, we arrive at an *address space–based abstraction* that uses LLVM pointer address spaces to identify pointers that belong in the root set [17]. The address space–based abstraction is sufficiently robust for our GC but it is not thin enough: Lowering the abstraction is a complex process. Specifically, it is lowered in one big step to fully lowered LLVM IR by a transform called *GC frame lowering*. If we were to use this abstraction to implement our GPU GC, then we would have to re-implement GC frame lowering, plus some minor modifications.

[1]In a compiler, an intrinsic (function) is a function whose meaning is intimately understood by the compiler. Additionally, intrinsic functions are declared but never defined: They are lowered to something concrete before the end of the compiler's pipeline. Consequently, the native code generated by the compiler never calls intrinsics, unlike normal functions.

C. Low-level GC intrinsics

Because the address space–based abstraction is not thin enough and fully lowered LLVM IR is not robust enough, we implement an intermediate abstraction based on low-level intrinsics.[1] We split up the GC frame lowering transform into two passes: A large, GC-agnostic pass that takes the address space–based abstraction and translates it to our new abstraction and a small, GC-specific pass that lowers our new abstraction to fully lowered LLVM IR. To implement a custom GC, one only needs to implement a different GC-specific pass.

At the heart of our abstraction are four intrinsics that capture the essence of root set management:

1. `julia.new_gc_frame(n)` allocates a new GC frame that can accommodate at least `n` GC roots. It returns a pointer to that frame.

2. `julia.push_gc_frame(gcframe, n)` takes a GC frame and registers it with the GC. The size of that GC frame is also provided for the GC-specific lowering's convenience; the lowering can effortlessly discard it if it is not needed.

3. `julia.get_gc_frame_slot(gcframe, i)` accepts a pointer to a GC frame and produces a pointer to the `i` th root pointer in that GC frame.

4. `julia.pop_gc_frame(gcframe)` unregisters a GC frame, removing its contribution to the root set.

These four intrinsics have concrete semantics. Hence, they can be lowered formulaically to a root set management implementation. At the same time, they give us enough wiggle-room to support a number of different root set management data structures.

IV. A garbage collector for CUDAnative

We now discuss the implementation of a semi-precise, mark-and-sweep GC for CUDAnative kernels based on the abstractions from the previous section.

A. High-level design

CUDAnative targets systems that include both GPUs and GPUs. We can use either device to allocate memory and collect garbage, provided that we have a reliable means of communication between these two devices during kernel executions.

Our GC allocates memory on the GPU and collects garbage on the CPU. We believe that a CPU is the most suitable device for this task: CPUs are known to excel at workloads that feature non-streaming memory accesses and synchronized access to resources, typical properties of mark and sweep GCs [16, 21].

The GC uses free lists to allocate memory [30]. To reduce free list contention, the GC groups threads and assigns a local free list to every group. These local free lists are intended for quickly allocating small objects. Our GC also has a global free list, shared by all threads, for allocating large objects. Users can control the number of local free lists as well as the initial amount of memory managed by the local and global free lists.

When a thread's local and global free lists run out of memory, the GPU triggers an interrupt that makes the CPU run a simple mark-and-sweep algorithm. Said algorithm finds garbage objects and returns them to the free lists from which they originate. If the collector finds a free list to be memory-starved even after a collection, then it will allocate additional memory and append it to the free list in question.

B. Interrupts

To give GPU kernels a reliable channel for requesting actions from the CPU, we implement GPU *interrupts.* The semantics of an interrupt are as follows:

- A GPU thread requests an interrupt and waits for the current interrupt to complete. The current interrupt may be the interrupt requested by the GPU thread or may be an already-pending interrupt requested by some other thread. When the interrupt completes, the GPU thread is made aware of whether it started a new interrupt or merely waited for an existing interrupt.

- When the GPU requests a new interrupt, the CPU takes note of this fact and runs an *interrupt handler,* that is, a function that is assigned to the kernel at kernel launch time.

Additionally, GPU threads can also wait for the current interrupt—if any—to complete without requesting a new interrupt themselves.

Our GC uses this flexible interrupt mechanism to implement collections: When a GPU thread runs out of memory, it requests a new interrupt or waits for the current interrupt to complete. The interrupt handler corresponds to the GC's collection algorithm.

C. Safepoints

Many garbage collection algorithms require that object references, including the root set, do not change during collections. The mark-and-sweep algorithm we use to implement collections is no exception. GPU kernels are massively parallel, meaning that object references may well change during collections unless we take action, breaking the assumption made by the collection algorithm.

Safepoints are a common approach to work around this issue [2]. A safepoint is a point in the instruction stream of a program at which a collection may safely occur. By pausing threads at safepoints during a collection, we can ensure that object references do not change during said collection, vindicating the collection algorithm's assumption.

The Julia compiler already has a notion of safepoints. We inject an additional pass into the Julia compilation pipeline that places a *safepoint polling function* at every safepoint. Such a polling function simply tests if a collection interrupt is pending and stops the current thread if it is. It also sets a flag, telling the collector that the current warp is ready for collection.

The collector waits for all warps to enter a safepoint before performing a collection, ensuring that the entire kernel is paused during the collection. After the collection, all paused warps resume execution.

D. *Root set management*

Since the GC uses the GPU for allocation and the CPU for collection, we need GC-managed memory and auxiliary data structures to be accessible to both the GPU and that CPU. For this reason, we use pinned host memory for all memory managed by the GC.

As previously hinted, this implies that the root set management scheme for Julia's CPU GC does not work for our GPU GC: The CPU GC stores roots in a linked list of stack-allocated GC frames. GPU stack memory is accessible to the GPU only; we instead want to use pinned memory for the root set, which is accessible to both the CPU and the GPU.

The GPU GC allocates a fixed-size buffer of pinned memory to every thread at kernel launch time. We implement the GC abstractions we introduced in the previous section to use that buffer as a stack of GC roots. When a collection is triggered, the GC scans the root buffers, allowing it to precisely identify the GC roots. Object references stored in other objects are approximated conservatively by inspecting all pointer-sized chunks of data in every object.

V. Evaluation

In this section, we evaluate the GC's performance both in a functional and a non-functional sense.

A. *Functional aspects*

First and foremost, we evaluate the GC's functional aspects, which translates into the following question: "Which high-level language features does the GC enable?" We distinguish between two different classes of features:

- **Directly enabled features:** these features require nothing more than a working implementations of the GC-related functions of the Julia runtime library.

  Most prominently, this category includes object allocation, specifically mutable and/or recursive `struct` allocation. Mutable and/or recursive `struct` types are a central part of Julia's type system. In Julia, they represent the only "right way" to implement a number of standard data structures. Examples include linked lists, binary trees, and compiler IRs.

- **Indirectly enabled features:** these features depend on a GC implementation plus some additional, dedicated runtime library functions.

  We show that our GC is also suitable for this category of features by implementing thirteen crucial low-level array-related functions in the GPU version of the Julia runtime library. These functions form the backbone of Julia's low-level array interface. They enable a large amount of array-related features, including (1) array creation, (2) reading and writing array elements, (3) querying an array's dimensions, (4) inserting elements into arrays, (5) deleting elements from arrays, (6) increasing the capacity of arrays, (7) wrapping unmanaged buffers in arrays, (8) array comprehensions [14], and (9) high-level array functions such as `fill`, `fill!` and `similar`.

We verify the veracity of all of these claims by constructing a set of benchmark programs that implement various algorithm in a idiomatic ways, that is, using the features described above.

B. *Non-functional aspects*

For our GC to be useful in practice, we also need its non-functional aspects to be up to scratch. We will verify that this is the case based on three performance-related questions:

1. What is the GC's *variable overhead*, that is, how well does the GC perform in terms of run time compared to trivial memory management schemes when run on a set of benchmarks that rely strongly on dynamic memory allocation?

2. How does the initial GC heap size affect the GC's performance?

3. What is the GC's *constant overhead*, that is, how well does the GC perform in terms of run time compared to trivial memory management when run on a set of benchmarks that do not rely on dynamic memory allocation at all?

1) *Variable overhead*

To determine the GC's variable overhead, we re-use the functional benchmarks, but this time measure their wall clock time. The functional benchmarks are designed to model idiomatic Julia code applied to GPUs. They allocate objects quite liberally and are hence suitable for ascertaining the variable overhead of the GC.

Figure 1 shows the slowdown of benchmarks per configuration, normalized to CUDA `malloc` and averaged out over the benchmark suite. Specifically, every benchmark, configuration pair was run for 90 s by the BenchmarkTools package [8]. Then, the median of these measurements was taken, as per the BenchmarkTools manual's recommendations [24]. Each median was normalized with regard to the median for the CUDA `malloc` configuration of the applicable benchmark. The bars in Figure 1 represent the means of those medians, grouped by configuration.

The "CUDA `malloc`" configuration uses its eponymous function to allocate memory. The "Unoptimized GC" configuration uses the GC described in this work but assigns it a single global free list and no local free lists. The "Optimized GC" configuration uses eight local free lists. We also implemented a bump allocator [30]—a lower bound on run times. Every configuration has 60 MiB heap, except for CUDA `malloc`, which has a 64 MiB heap for practical reasons that do not affect our results.

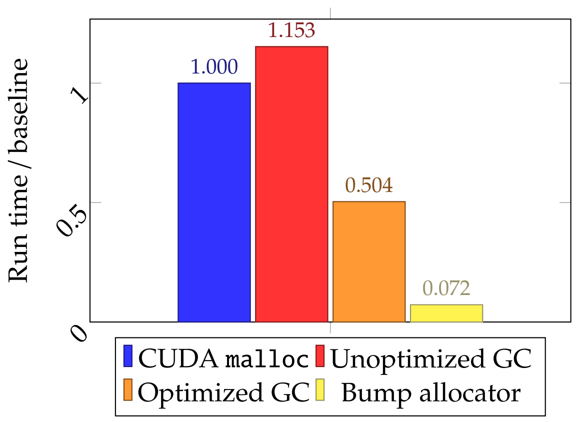

Figure 1: Variable overhead of unoptimized, optimized GCs relative to trivial memory management schemes. Lower scores are better.

2) *Initial heap size*

To determine the impact of the initial GC heap size on performance, we re-run the functional benchmarks, but this time with GC heap sizes that vary from 10 MiB to 60 MiB. The latter is more than enough memory for all benchmarks in our benchmark suite, meaning that no collections should take place. The former is well below that mark, allowing us to quantify the overhead of collections.

Figure 2 plots how the mean normalized run times of benchmarks evolve with initial GC heap sizes. The figure shows that the initial heap size barely affects performance, with larger initial heaps performing slightly worse than smaller ones. We believe that the initial heap size's limited impact on performance is due to the fact that allocation times dominate, as can be inferred from Figure 1. Interrupt and collection overheads appear to be mild in comparison, as evidenced by Figure 2. Smaller initial heap sizes reduce the amount of up-front effort required for allocating and initializing potentially unused sections of the heap, explaining why smaller heaps appear to perform slightly better than their larger counterparts.

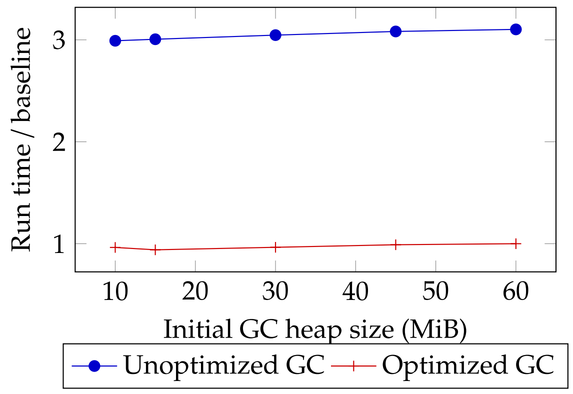

Figure 2: Normalized run times of benchmarks with unoptimized, optimized GCs at various initial heap sizes. Lower scores are better.

3) *Constant overhead*

We determine the constant overhead of the GC by performing the same experiment as for variable overhead, but with benchmarks from a Julia port of the Rodinia benchmark suite [3,7]. Rodinia is designed to represent typical scientific computing loads for GPUs. None of the Rodinia benchmarks allocate memory dynamically, which makes them suitable candidates for ascertaining the constant overhead of using a GPU GC.

Figure 3 depicts the result of this experiment: kernel run times relative to CUDA `malloc` are plotted for each Rodinia benchmark. The x axis identifies CUDA `malloc`–configured kernel run times. Short-lived kernels have the highest relative constant overhead; relative constant overhead for long-lived kernels is negligible.

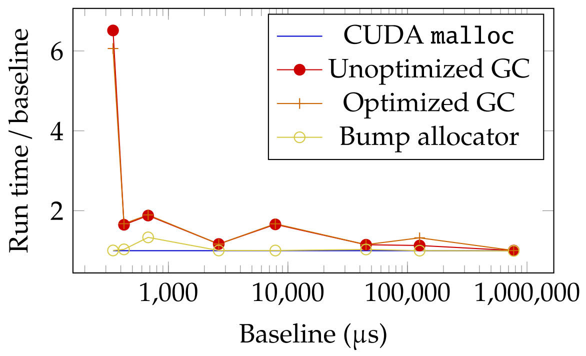

Figure 3: Constant overhead of unoptimized, optimized GCs relative to a trivial memory management scheme scheme built on CUDA `malloc`, by benchmark duration. Lower scores are better.

VI. Conclusion

We implemented the first fully transparent GC for GPU kernels. Our GC is implemented as a modified version of the CUDAnative package, which compiles Julia code to NVIDIA GPU kernels.

To implement our GC, we introduced a new low-level intrinsics-based GC abstraction in the Julia compiler. This abstraction is sufficiently robust to implement both Julia's CPU GC and the GPU GC presented in this work. Despite its robustness, the abstraction is quite thin: It can be implemented by a formulaic lowering, even for divergent implementation strategies.

Experiments produce hopeful results regarding the GPU GC's functional and non-functional aspects. On the functional side, the GC directly enables object allocation and indirectly enables arrays, a crucial data structure in idiomatic Julia.

In terms of variable overhead, benchmarks are on average 2× faster when they are configured to use our GC instead of CUDA `malloc`, the allocation function that is used to implement object allocation in unmodified CUDAnative and the Rootbeer Java-to-CUDA compiler [22]. To the best of our knowledge, these two projects are the only high-level language to GPU compilers that implement fully transparent object allocation.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[2] Attanasio, C. R., Bacon, D. F., Cocchi, A., and Smith, S. A comparative evaluation of parallel garbage collector implementations. In *International Workshop on Languages and Compilers for Parallel Computing* (2001), Springer, pp. 177–192.

[3] Besard, T., and Foket, C. Benchmark suite for heterogeneous computing infrastructures. `https://github.com/JuliaParallel/rodinia`. Accessed on May 8, 2019.

[4] Besard, T., Foket, C., and De Sutter, B. Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems 30*, 4 (2019), 827–841.

[5] Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM review 59*, 1 (2017), 65–98.

[6] Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145* (2012).

[7] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)* (2009), Ieee, pp. 44–54.

[8] Chen, J., and Revels, J. Robust benchmarking in noisy environments. *arXiv preprint arXiv:1608.04295* (2016).

[9] Durant, L., Giroux, O., Harris, M., and Stam, N. Inside Volta: The world's most advanced data center GPU. *NVIDIA Developer Blog,* ***`https://devblogs.nvidia.com/inside-volta`*** (May 2017). Accessed on May 24, 2019.

[10] Edelman, A. Julia: A fresh approach to parallel programming. In *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), IEEE, pp. 517–517.

[11] Fung, W. W., and Aamodt, T. M. Thread block compaction for efficient SIMT control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture* (2011), IEEE, pp. 25–36.

[12] Govett, M. W., Middlecoff, J., and Henderson, T. Running the NIM next-generation weather model on GPUs. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010), IEEE Computer Society, pp. 792–796.

[13] Harris, M. Unified Memory for CUDA Beginners. *NVIDIA Developer Blog,* ***`https://devblogs.nvidia.com/unified-memory-cuda-beginners/`*** (June 2017). Accessed on June 4, 2019.

[14] Introducing Julia contributors. Introducing Julia: Arrays and tuples. `https://en.wikibooks.org/wiki/Introducing_Julia/Arrays_and_tuples`, February 2019. Accessed on June 11, 2019.

[15] Lattner, C., and Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (2004), IEEE Computer Society, p. 75.

[16] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., et al. Debunking the 100× GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH computer architecture news 38*, 3 (2010), 451–460.

[17] LLVM project. LLVM language reference manual. `http://llvm.org/docs/LangRef.html`, April 2019. Accessed on May 1, 2019.

[18] Nickolls, J., Buck, I., and Garland, M. Scalable parallel programming. In *2008 IEEE Hot Chips 20 Symposium (HCS)* (2008), IEEE, pp. 40–53.

[19] Nvidia. *CUDA C programming guide v10.1*. Nvidia Corporation, March 2019.

[20] Nvidia. *Parallel Thread Execution ISA v6.4*. Nvidia Corporation, March 2019.

[21] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. GPU computing. *Proceedings of the IEEE 96*, 5 (2008).

[22] Pratt-Szeliga, P. C., Fawcett, J. W., and Welch, R. D. Rootbeer: Seamlessly using GPUs from Java. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on* (2012), IEEE, pp. 375–380.

[23] Pratx, G., and Xing, L. GPU computing in medical physics: a review. *Medical physics 38*, 5 (2011), 2685–2697.

[24] Revels, J., Arslan, A., Ahrens, P., Adams, L., Herriman, J., Goerz, M., Johnson, S. G., and Mauro. BenchmarkTools manual. `https://github.com/JuliaCI/BenchmarkTools.jl/blob/master/doc/manual.md`, November 2018. Accessed on June 10, 2019.

[25] Rhodin, H. A PTX code generator for LLVM. Bachelor's thesis, Saarland University, Saarbrücken, Germany, October 2010.

[26] Romero, M., and Urra, R. CUDA Overview. `http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm`. Accessed on May 27, 2019.

[27] Sanders, J., and Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[28] Veldema, R., and Philippsen, M. Iterative data-parallel mark&sweep on a GPU. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 1–10.

[29] Wilson, P. R. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management* (1992), Springer, pp. 1–42.

[30] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management* (1995), Springer, pp. 1–116.

[31] Wu, X., Koslowski, A., and Thiel, W. Semiempirical quantum chemical calculations accelerated on a hybrid multicore CPU–GPU computing platform. *Journal of chemical theory and computation 8*, 7 (2012), 2272–2281.

[32] Zhang, J., You, S., and Gruenwald, L. Indexing large-scale raster geospatial data using massively parallel GPGPU computing. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2010), ACM, pp. 450–453.

Contents

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
CLR	Common Language Runtime
CPU	Central Processing Unit
GC	Garbage Collector
GHC	Glasgow Haskell Compiler
GPU	Graphics Processing Unit
IR	Intermediate Representation
JIT	Just In Time
JVM	Java Virtual Machine
LOC	Lines of Code
PTX	Parallel Thread Execution
SIMT	Single Instruction, Multiple Threads
SPMD	Single Program, Multiple Data
SSA	Static Single Assignment

1 Introduction

As the field of high-performance computing evolves, two salient trends are emerging.

First and foremost: Scientific computing applications nowadays typically rely on GPUs rather than Central Processing Units (CPUs) as carefully controlled use of the former permits order-of-magnitude speedups compared to typical use of the latter. For this reason, GPU computing has become mainstream in a wide range of scientific applications that includes weather forecasting [38], geospatial indexing [85], chemistry [84], medicine [63] and machine learning [1].

A second trend is the appearance of productive, high-level scientific computing languages that match the performance of low-level programming languages such as Fortran, C and C++. The Julia programming language is a prominent example: The performance of Julia programs can match that of equivalent C/C++ programs in spite of the fact that Julia offers many core language features—dynamic typing, garbage collection, etc.—that are typically associated with "slower" programming languages such as Python, Ruby and Perl [17].

These two trends—increased hardware performance and increased programmer productivity—are wedded by projects that compile high-level programming languages to instructions suitable for execution on the GPU. CUDAnative is an example of such a project: It compiles Julia code for NVIDIA GPUs [15].

Because of the unique characteristics of GPUs and a lack of mature infrastructure in GPU programming environments, projects that compile high-level languages for GPUs often restrict the set of supported language features. In CUDAnative's case, Julia language constructs that map well to the low-level CUDA programming model for NVIDIA GPUs are supported, but others are not. Consequently, CUDAnative's GPU *kernels*, that is, GPU programs, are written in a low-level dialect of Julia that precludes idiomatic features such as mutable `struct` types, arrays, exception handling, and others.

In addition to reducing the feature set available to manually written GPU kernels, language features not supported by CUDAnative also break compositional schemes that attempt to transparently expose the compute capabilities of GPUs. To name one example, the `CuArray` type mimics the functionality of Julia's normal array type, but transparently runs array computations on the GPU to obtain superior performance [15]. `CuArray` hides the complexity of GPU programming from its users by automatically synthesizing GPU kernels for common operations. These automatically synthesized kernels are also affected by CUDAnative's restricted set of permissible language features. This forces programmers to give special consideration to how `CuArray` instances are used, rather than treating them as fully-featured implementations of an abstract array type.

Many of the features that are not supported by CUDAnative have one thing in common: They rely on the presence of some automatic memory management scheme—specifically, they expect a GC to be in place.

GPU computing has historically relied on low-level languages to construct programs suitable for execution on a GPU. These low-level languages never necessitated the creation of a GC for GPUs. Consequently, there exist—to the best of our knowledge—no drop-in GCs for GPU kernels. GC algorithms have been developed for GPUs, but these experiments required programmer intervention for interrupting kernels when memory runs out, triggering an actual collection and restoring the kernel's state [76].

This thesis has two main goals:

1. To design GPU-friendly compiler abstractions for garbage collection and implement these abstractions in the Julia compiler.

2. To design memory management schemes for GPUs that build on the abstractions from the previous item. We implement the memory management schemes in CUDAnative, targeting Julia code that is compiled for NVIDIA GPUs.

We will start off with an overview of the state of the art in compiling Julia for GPUs as well as a discussion of a number of topics that are relevant to the design of a GC that works on GPUs.

Next, we outline our core contributions. Our first contribution is a set of changes to the Julia compiler that allow for the development of custom GC implementations. Our second and main contribution is a fully-featured mark-and-sweep GC for GPUs.

We then evaluate functional and non-functional aspects of the GPU GC and its supporting infrastructure. Finally, we discuss select related work and conclude the thesis.

2 Background

This chapter discusses the state of the art in Julia-to-GPU compilation and GPU garbage collection, with a particular emphasis on topics that are material to the contributions presented in the remainder of this thesis.

2.1 General-purpose GPU programming

GPUs can deliver large performance increases over traditional CPU programming. This performance boost can be attributed to the execution model that underpins GPU programming: CPUs are optimized for executing sequential instruction streams whereas GPUs are optimized for executing instructions in a massively parallel fashion.

This section discusses some differences between CPU and GPU programming. As per our stated goal of targeting Julia code that is compiled for NVIDIA GPUs, we will restrict ourselves to NVIDIA's CUDA programming model [57].

2.1.1 Parallel execution semantics

Parallelism is a fundamental feature of GPUs that deeply affects both functional and non-functional aspects of GPU programs. To manage this parallelism adequately, the CUDA GPU programming language introduces a complex hierarchy of hardware and software parallelism. We introduce the following CUDA terminology:

- A *kernel* is a GPU program. Every launched kernel consists of a set of threads.
- A *thread* is a semi-independent unit of execution. All threads in a launched kernel execute the same program. The control flow of threads can diverge based on the data they process.
- Threads are grouped by the hardware into *warps:* fixed-size collections of threads.

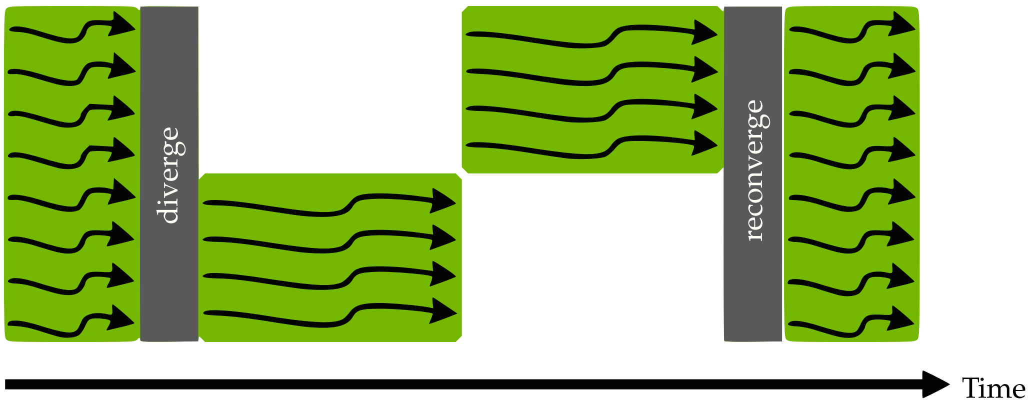

Figure 2.1: Thread divergence within a warp. Adapted from Durant et al. [28]

Warps and SIMT

NVIDIA GPUs implement Single Instruction, Multiple Threads (SIMT), an idiosyncratic version of the popular Single Program, Multiple Data (SPMD) programming model [35, 55]. Threads in different warps run in parallel and independently of each other, like CPU threads. Inside of a warp, threads exhibit a peculiar form of parallelism that characterizes SIMT: Instructions from threads in the same warp are executed in lockstep and in parallel. That is, at every step, threads in a warp execute the same instruction on different data in parallel. When control flow in these threads diverges, a scenario isomorphic to Figure 2.1 occurs: The divergent control-flow paths are executed in a sequential fashion until control flow reconverges. This phenomenon, where different threads in the same warp execute different control-flow paths, is known as *thread divergence* [57, 68].

Thread divergence impacts both functional and non-functional aspects of GPU kernels. On the non-functional side, subdividing a warp into groups of threads that must be processed in sequence reduces parallelism, impacting run time performance. On the functional side, sequentially processing divergent threads implies that thread starvation may occur: Until the currently running threads complete, the other threads will not make any progress at all [28].

Blocks and grids

In addition to the hardware-side partitioning of threads into warps, CUDA also exposes mechanisms for programmatically partitioning threads into *blocks* and *warps.*

A block is a collection of concurrently-executing threads, possibly spread out across multiple warps. Threads in a block are each assigned a block-locally unique

```
1 __global__ void kernel_sum(float* a, float* b, float* c)
2 {
3   int id = blockDim.x * blockIdx.x + threadIdx.x;
4   c[id] = a[id] + b[id];
5 }
6
7 // ...
8
9 kernel_sum<<<blocksPerGrid,threadsPerBlock>>>(a, b, c);
```

Listing 1: A CUDA vector addition kernel.

identifier. Additionally, the threads in a block can synchronize and communicate efficiently through shared memory [67].

A grid is a collection of blocks [42]. Blocks in a grid cannot communicate through shared memory and cannot synchronize [67]. Every block in a grid has a grid-locally unique identifier and every grid has a kernel launch–locally unique identifier.

When a kernel launch is programmed, the programmer is free to decide on the number of threads per block, the number of blocks per grid, and the number of grids to launch, subject to some constraints; the hardware imposes upper bounds on the sizes of blocks and grids. These dimensions may be specified as scalars, 2D vectors, or 3D vectors. In any case, the total number of threads per block, blocks per grid, and grids to launch corresponds to the product of the elements of the respective dimension vectors: A block with dimensions $(2, 3, 2)$ contains $2 \times 3 \times 2 = 12$ threads. Identifiers for threads, blocks, and grids are encoded as 3D vectors in the space defined by their respective dimensions.

To illustrate this, Listing 1 introduces a CUDA kernel that instructs every thread to add two values together. The `kernel_sum<<<blocksPerGrid,threadsPerBlock>>>` syntax launches kernel `kernel_sum` with `threadsPerBlock` threads per block and `blocksPerGrid` blocks per grid. We assume that both `threadsPerBlock` and `blocksPerGrid` are scalars. Because the number of grids is not explicitly specified, a value of one is implied. Line 3 of Listing 1 computes a globally unique thread index, which is then used to the `a`, `b`, and `c` arrays.

2.1.2 GPU memory

NVIDIA GPUs feature a complex partitioning of on-chip memory that includes global, constant, texture, shared, and local memory spaces. Each of these memory spaces are optimized for different use cases and differ in both their functional and non-functional properties [57, 68]:

- **Global memory** is read-write per-grid memory.
- **Constant memory** is read-only per-grid memory.
- **Texture memory** is read-only per-grid memory that is optimized for 2D memory access patterns.
- **Shared memory** is read-write per-block memory.
- **Local memory** is read-write per-thread memory.

In addition to these various kinds of on-chip memory, GPU kernels may also access host memory directly, provided that it is *page-locked*. Page-locked memory, also known as *pinned* memory, is guaranteed by the operating system to always be present in random-access memory—it will never be paged out to disk.

Finally, *unified memory* is similar to pinned memory in that it is accessible to both CPUs and GPUs. Unlike pinned memory, which is essentially CPU memory that is shared with the GPU, unified memory pages automatically migrate to the device that is currently using them, which can be either a CPU or a GPU [40].

2.2 The Julia programming language

The Julia programming language styles itself as a "fresh approach to numerical computing" [16, 30]. In programming language terminology, it is a dynamic programming language in a similar vein to Python, Matlab and R.

Julia sets itself apart from these other dynamic languages because Julia was designed with speed in mind: Julia is Just In Time (JIT) compiled to native code by the industrial-strength LLVM compiler framework [50].

To compile dynamic code to efficient machine instructions, the Julia compiler aggressively specializes functions definitions for their argument types. For instance, consider the `add` function in Listing 2. Because `add`'s argument types are not constrained, it can be applied to any two arguments for which a binary addition

```julia
1 function add(x, y)
2     return x + y
3 end
4
5 add(2, 42)
6 add(1.0, 2.0)
```

Listing 2: Function specialization in Julia.

function is defined. Furthermore, because Julia is a dynamic language, we cannot know how that binary addition function is implemented.

Typical dynamic languages implementations such as the CPython interpreter resolve such uncertainty by dynamically looking up the definition of the binary addition function for its argument types `x` and `y` prior to every invocation of the addition function.

Julia takes a radically different approach: Whenever the Julia compiler encounters a call to a function, it compiles the callee specifically for the argument types at the call site. When compiling such a specialized callee, the Julia compiler uses type inference to statically resolve dynamism at compile time. Compiled callees are cached, so specialized compilation is a one-time cost for every combination of a function and a tuple of argument types.

Returning to our example, this means that the `add` function will be compiled differently based on how it is called. For instance, the call on line 6 will compile `add` to native code that performs an integer addition and returns. Similarly, the call on line 7 will compile `add` to native code that performs a floating-point addition and returns. In both cases, all dynamism is resolved by the time control enters the `add` function. This translates into an opportunity to generate highly efficient code for `add`, similar to what one might expect from a C compiler.

This process of aggressive specialization allows code compiled by Julia's JIT compiler to match the performance of equivalent C/C++ code, as evidenced by a number of micro-benchmarks [45].

2.2.1 Compilation pipeline

Like many other LLVM-based programming language implementations, the Julia compiler accepts Julia source code as input, turns it into a Julia-specific Intermediate Representation (IR), optimizes that IR and finally uses it to generate LLVM IR that is

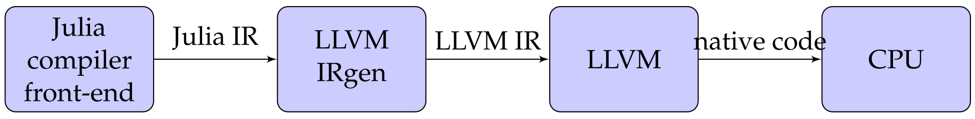

Figure 2.2: The Julia compiler's pipeline.

compiled to native code. This process is summarized graphically by Figure 2.2, a figure that we will expand in Section 2.3 as we discuss how CUDAnative extends the Julia compiler's pipeline.

2.2.2 Garbage collection

A number of idiomatic Julia language features implicitly rely on having a GC in place that allocates and collects memory. Listing 3 is an anthology of such language features: (1) mutable and recursive data structures, (2) arrays, (3) exception handling, and (4) dynamic multiple dispatch. Of these features, mutable and recursive data structures depend solely on a GC and arrays depend on a GC plus some additional bookkeeping logic. Exception handling and dynamic multiple dispatch depend on a GC as well as complex, dedicated support mechanisms. The following list is a detailed explanation of why each aforementioned feature depends on GC support:

1. **Mutable data structures**, that is, `mutable struct` types are managed by the GC. Whenever an instance of a `mutable struct` type is created, it is stored in a chunk of GC memory. When the instance becomes unreachable, also known as *dead*, the GC will eventually *collect* the instance, recycling its memory for use by other objects.

   `Vector2` as defined on line 2 of Listing 3 is an example of a mutable data structure. Consequently, any `Vector2` instance will be managed by the GC and no `Vector2` can be allocated without a GC or other automatic memory management scheme.

   The same holds for recursive data structures such as `BinaryTreeNode` on line 7 of Listing 3. Non-recursive, immutable `struct` instances are a notable exception: They are not managed by the GC unless they are *boxed*, that is, copied to a GC-managed region of memory. Boxing is used to treat objects of different types uniformly and occurs when, e.g., storing a non-recursive, immutable `struct` instance in a dynamically-typed field.

2. **Arrays** are also managed by the GC. Line 15 of Listing 3 creates an array containing three elements, line 16 performs an array *comprehension:* Syntactic sugar for creating a new array that is the result of applying a mapping function to another array.

   In either case, the GC needs to allocate storage whenever an array is created.

3. **Exception handling** uses exception objects to describe errors. The exception handling mechanism that processes thrown exceptions does not know which type of exceptions will be thrown in advance, so it only manipulates GC-managed object references, turning raw objects such as non-recursive, immutable `struct` values into object references by boxing them.

   So, even if the `BoundsError` object created and thrown on line 20 of Listing 3 is a non-recursive, immutable `struct` instance, then the `throw` function will still box the `BoundsError` object, requiring GC intervention.

4. **Dynamic multiple dispatch** occurs when the Julia compiler cannot statically infer the types of a call's arguments.

   Julia includes a feature called *multiple dispatch,* which allows for a function to have multiple definitions, each with a different argument list. Many modern programming languages have a similar feature called *function overloading,* which is the same as *static* multiple dispatch, that is, multiple dispatch that can be resolved at compile time.

   Multiple dispatch is exemplified by lines 26 and 27 of Listing 3: `typeName` has two definitions: One for `Int` values and another for `Float64` values. The precise definition that is invoked when `typeName` is called depends on the arguments with which `typeName` is called.

   Now consider line 29 of Listing 3 and take `val` to be some unknown quantity. When the Julia compiler cannot tell what a call's argument types are as it is compiling the call, the Julia compiler will insert code that runs Julia's call resolution logic and compiles a specialized version of the callee whenever the call is encountered. This is called *dynamic* multiple dispatch and it differentiates Julia's dynamic dispatch mechanism from other languages' overloading implementations.

   Like exception handling, dynamic multiple dispatch depends on the ability to create GC-managed objects such that arguments of unknown types can be inspected and processed in a uniform manner by the call resolution logic.

The features from Listing 3 are pervasive in typical Julia code. To support them, Julia ships with a GC that is part of Julia's runtime library. Julia's runtime library is coded in C++. Whenever the Julia compiler encounters a feature that requires intervention from the GC, it generates a call to an appropriate GC-related runtime library function.

Julia's runtime library is specific to the architecture for which it is compiled. Because of this, existing runtime library functionality cannot be reused when the Julia compiler is coaxed into generating code for any architecture other than the one for which it was compiled. This is not a real limitation for typical Julia usage, but it is an issue for CUDAnative, which we will discuss in the next section.

2.3 CUDAnative: A Julia-to-GPU compiler

CUDAnative is a Julia package that compiles Julia code to Parallel Thread Execution (PTX) instructions suitable for execution on NVIDIA GPUs [15,58].

Listing 4 is a taste of what that looks like in practice: Kernels are encoded as regular Julia functions, which are transparently compiled to GPU instructions by CUDAnative. In this case, the `vadd` function is used as a GPU kernel by starting it with the `@cuda` macro on line 9. The `vadd` kernel from Listing 4 is both functionally and structurally equivalent to the CUDA kernel from Listing 1. Indeed, line 2 of Listing 4 computes a unique thread index and line 3 loads two values, adds them and stores the result.

2.3.1 Compilation strategy

CUDAnative is designed to take advantage of existing Julia compiler infrastructure to compile Julia code for NVIDIA GPUs. Figure 2.3 depicts the interactions between the Julia compiler and CUDAnative: Code that is compiled by CUDAnative passes through approximately the same machinery that is used for compiling regular Julia code. CUDAnative intercepts LLVM IR generated by the Julia compiler just before that IR is transformed in architecture-specific ways and sends it to LLVM's PTX back-end instead of sending it to the code generator for the CPU, as the unmodified, CPU-targeting Julia compiler would have done [66].

At this point, one might expect that CUDAnative can compile arbitrary Julia code for GPUs. This is true to some extent: CUDAnative's apt use of the Julia compiler means that it can handle all Julia *syntax.* However, the Julia compiler relies on a

```julia
# Mutable data structures.
mutable struct Vector2
    x::Float64
    y::Float64
end

# Recursive data structures.
struct BinaryTreeNode
    value::Int
    left::Union{BinaryTreeNode,Nothing}
    right::Union{BinaryTreeNode,Nothing}
end

# Arrays.
xs = [1, 2, 42]
ys = [x * x for x in xs]

# Exception handling.
try
    throw(BoundsError())
catch ex
    # ...
end

# Dynamic multiple dispatch.
function typeName(x::Int) = "Int"
function typeName(x::Float64) = "Float64"

name = typeName(val)
```

Listing 3: Julia language features that rely on having a GC.

```julia
function vadd(a, b, c)
    i = (blockIdx().x-1) * blockDim().x + threadIdx().x
    c[i] = a[i] + b[i]
    return
end

# ...

@cuda threads=len vadd(d_a, d_b, d_c)
```

Listing 4: CUDAnative's `vadd.jl` example, which computes the sum of two arrays.

runtime library to implement more sophisticated language features, such as garbage collection, exception handling, and dynamic multiple dispatch. Julia's runtime library is CPU-specific and hence CUDAnative cannot reuse it. Consequently, whenever the Julia compiler generates a call to the runtime library, CUDAnative must cope with that call somehow.

Broadly speaking, CUDAnative has two mechanisms for handling calls to the Julia runtime library.

1. If the runtime library function can reasonably be re-implemented for GPUs, then CUDAnative may supply its own version of the runtime library function and link it with kernels.

   For instance, prior to this work, CUDAnative did not support arbitrary uses of GC-related functionality but did implement boxing and unboxing functions for primitive types by implementing homonymous functions in the CUDAnative GPU runtime library.

2. If the runtime library function's semantics or signature rule out a GPU implementation, then CUDAnative throws a compile-time error. In effect, this disables the use of a language feature for all CUDAnative kernels.

Instances of the latter are painfully abundant. Core features such as exception handling, garbage collection, and dynamic multiple dispatch are implemented by the runtime library in such a way that CUDAnative cannot reasonably be expected to re-implement the runtime library functionality.

Sometimes, the fundamental issue blocking CUDAnative implementations for advanced Julia language features is that these *features* are truly incompatible with

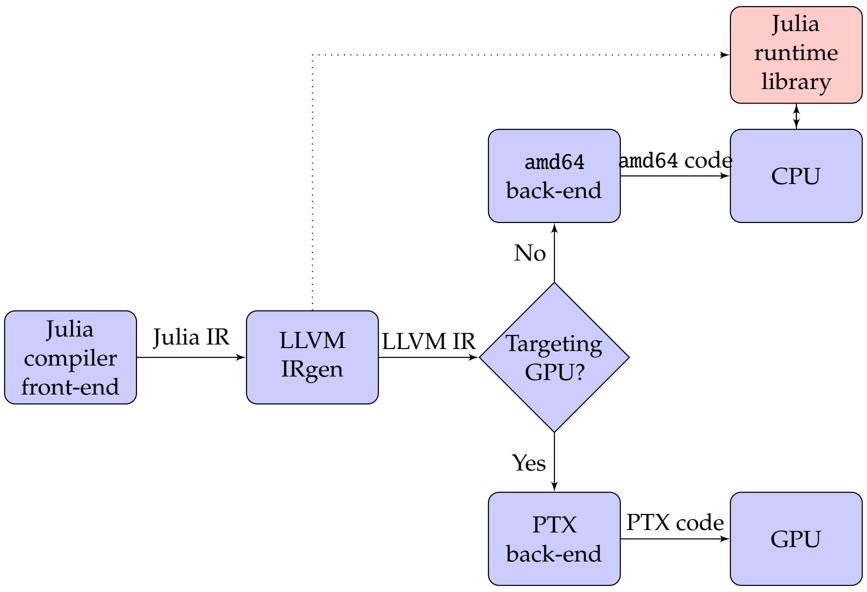

Figure 2.3: CUDAnative's original compilation pipeline.

GPU environments. For instance, Julia's interpretation of dynamic multiple dispatch may require compiling a specialized version of a function using the Julia and LLVM compilers, neither of which runs on GPUs.

But more often, the issue at hand is that the Julia runtime library's *implementations* of advanced language features is what blocks a successful GPU implementation. For example, this thesis shows that garbage collection can be implemented for GPU kernels, albeit not using the same interface as the Julia runtime library's GC. Similarly, exception handling has been implemented for Java code that compiles to GPU kernels, but that implementation does not adhere to the `setjump-longjump` exception handling framework preferred by the Julia compiler [62].

Notice the dotted arrow in Figure 2.3. This arrow represents a dependency on the interface exposed by the runtime library and complicates matters for CUDAnative. As can be inferred from the positioning of the arrow, the Julia compiler generates calls to runtime library functions before CUDAnative has an opportunity to intervene. So the proverbial damage is already done by the time CUDAnative receives LLVM IR from the Julia compiler—CUDAnative is tied to the Julia runtime library's interface.

2.3.2 Workarounds for problematic runtime library functions

For some language features, the restrictions imposed by the Julia runtime library's interface can be short-circuited by a language-level re-implementation.

To illustrate this, we will consider the challenge of implementing arrays for GPUs. The Julia compiler lowers one-dimensional array creation to calls to `jl_alloc_array_1d`, a function that takes two arguments: the array's element type and the number of elements in the array.

At the time of writing, there is little hope of re-implementing `jl_alloc_array_1d` as a GPU runtime library function because `jl_alloc_array_1d`'s interface forces it to rely on reflection to discover essential information such as the size of an element in the array. Specifically, the issue is that `jl_alloc_array_1d` is given a pointer to the element type of the array to allocate. The Julia compiler places type information in CPU memory, not in GPU memory. This is no burden on `jl_alloc_array_1d`'s CPU implementation, which can freely dereference pointers to find the size of the element type's instances. However, the GPU cannot successfully dereference pointers into arbitrary CPU memory, making basic facts about the element type impossible to discover and a direct re-implementation of `jl_alloc_array_1d` infeasible. We hence conclude that supporting arrays by naively re-implementing the Julia runtime library's interface is out of reach.

To work around this issue for kernels that do require an array type, CUDA-native offers `CuDeviceArray`: A user-defined type that uses multiple dispatch to implement the high-level interface exposed by built-in arrays. This allows for `CuDeviceArray` to be manipulated in the same way as built-in arrays, without constraining `CuDeviceArray` to the problematic low-level interfaces defined by Julia's runtime library.

In general, we will call this strategy *user-defined alternatives:* The practice of replacing data types and functions not supported by CUDAnative with CUDA-native-compatible language-level implementations.

Contextual dispatch

When calls to problematic runtime library functions are generated by functions from Julia's standard library, we can bypass the runtime library by re-implementing those exact functions specifically for the GPU using a process called *contextual dispatch* [77].

For example, contextual dispatch allows us to replace the CUDAnative-unsupported `sin` function from the Julia standard library with a GPU-friendly version in such a way that the original version is called on the CPU and the GPU-specific version is called on the GPU.

Limitations

User-defined alternatives and contextual dispatch are potent mechanisms, but they do have their limitations. Most prominently, there are some Julia language features that are implemented by the compiler directly, without calling functions from Julia's standard library. Neither workaround applies to such features.

Mutable and recursive data structures are a prime example: the Julia compiler directly lowers allocations of such structures to runtime library calls; no functions from the standard library are involved. Due to the standard library not being involved, there is no interface to mimic or override, making both contextual dispatch and user-defined alternative types unsuitable.

Array literals, array comprehensions and exception handling suffer the same fate: they all bypass the standard library in the same way, making language-level workarounds inadequate, especially if some degree of source code compatibility is to be maintained.

2.3.3 The case for native language features

We have thus far discussed why implementing certain Julia language features for GPUs is difficult. This section presents arguments for why implementing missing language features for GPUs is useful.

First off, there is *ease of programming:* Implementing the very language features that make Julia suitable for high-performance computing at a high level of abstraction for CPUs would allow for the same idioms to be applied to GPUs.

This also applies to the workarounds presented in the previous section, limitations notwithstanding. User-defined alternative types and functions can bring programming patterns to the GPU, albeit in a slightly different form. Contextual dispatch can bridge the gap entirely in some cases.

A second argument for GPU support for Julia language features is *code reuse:* The Julia standard library defines many data types and functions that might be useful in CUDAnative kernels. Hash maps and sets, for example, are ubiquitous in many algorithms.

The standard library assumes that all Julia language features are supported, so it does not restrict itself to a subset of Julia for its implementation. CUDAnative kernels, however, do need to restrict themselves to a subset of the Julia language. Consequently, many types and functions from the standard library cannot be used in CUDAnative kernels because these types and functions rely on CUDAnative-unsupported language features. This is also the case for hash maps and sets. If a CUDAnative kernel needs to make use of such a data structure, then it needs to create its own.

No workaround can remedy this: GPU-friendly versions of many standard library types and functions can surely be created, but doing so does not address the issue of code reuse. Indeed, it is in fact a form of code duplication.

Finally and perhaps most importantly, we also need to consider *compositional schemes* that build on CUDAnative. Such compositional schemes attempt to transparently expose the power of GPUs in a programmer-friendly manner, that is, without forcing the idiosyncrasies of GPU programming onto programmers. We discuss two such compositional schemes: CuArrays and GPUifyLoops.

CuArrays

CuArrays is a Julia package that defines a type—aptly called `CuArray`—that implements the same interface as regular Julia arrays [13]. `CuArray` instances differ from Julia's built-in arrays in that they store their contents in GPU memory and manipulate those contents using GPU kernels.

The `CuArray` type is designed to *transparently* expose the computational power of GPUs: a `CuArray` instance can be manipulated in exactly the same way as a normal array, but `CuArray` objects automatically perform computations in a massively parallel fashion on the GPU rather than relying on the CPU, as the built-in array type does. In short, `CuArray`'s promise is to offer the convenience of array programming and the power of GPUs.

Moreover, there is quite a bit of synergy between the `CuArray` type and Julia's type system. For instance, consider Listing 5. On line 1, a function called `dot` is defined. `dot` computes the dot product of two vectors by first performing a pointwise multiplication and then summing up the results. `dot`'s implementation is idiomatic Julia code, as one might write for built-in Julia arrays. Indeed, line 7 computes the dot product of two built-in Julia arrays.

Line 10 demonstrates the extraordinary usefulness of the `CuArray` type. `CuArray` uses multiple dispatch to implement the array interface in such a way that computations are performed by the GPU instead of the CPU. By passing two `CuArray` instances to `dot`, we completely change the underlying implementation of the dot product: The pointwise multiplication and subsequent reduction are performed by the GPU this time around. This is quite a feat, considering that `dot` does nothing to explicitly support GPU acceleration.

```julia
dot(A, B) = sum(A .* B)

xs = [1, 2, 3]
ys = [4, 5, 6]

# Compute the dot product on the CPU.
z_1 = dot(xs, ys)

# Compute the dot product on the GPU.
z_2 = dot(CuArray(xs), CuArray(ys))
```

Listing 5: CuArrays in action.

The abstraction afforded by `CuArray` allows for functions and libraries originally designed with built-in arrays in mind to be GPU-accelerated without any modifications at all to those functions and libraries. It does so by automatically generating GPU kernels for common operations such as *map*, *broadcast* and *reduce*.

However, `CuArray`'s abstraction breaks down when a function fed to such an operation does not adhere to the restricted language feature set offered by CUDAnative.

Suppose, for example, that we would like to map an array of integers to the number of prime factors of each integer in the array. This is exactly what Listing 6 does: It defines a naive factorization function, `factorize`, as well as a function that computes the number of prime factors for an integer, `factor_count`. Then, on line 22, `factor_count` is used to map an array of integers to the arity of their factorizations. The `factor_count.` syntax represents a *broadcast*, which is in this case equivalent to applying `factor_count` to every element of the array.

Line 27 of Listing 6 attempts to run the same computation on the GPU using CuArrays, but fails. The use of an array in `factorize` injects a `jl_alloc_array_1d` call into the GPU kernel generated for the broadcast, triggering an error message that is bound to be cryptic at best for programmers who are not intimately acquainted with CuArrays' implementation.

This example demonstrates that a single use of a CUDAnative-unsupported language feature can destroy the abstraction offered by CuArrays, effectively making CuArrays' abstraction somewhat leaky and brittle. Furthermore, there is nothing special about arrays as a CuArrays-breaking feature. Any CUDAnative-unsupported

```julia
# Naively factorizes an integer.
function factorize(n)
  results = []
  i = 2
  while n > 1
    if n % i == 0
      n = div(n, i)
      push!(results, i)
    else
      i += 1
    end
  end
  return results
end

# Computes the number of factors of an integer.
factor_count(n) = length(factorize(n))

xs = [15, 44, 60]

# Computes the number of factors for each element in `xs` (CPU).
factor_count.(xs)

# Computes the number of factors for each element in `xs` (GPU).
# Throws an error: "unsupported call through a literal pointer
# (call to jl_alloc_array_1d)"
factor_count.(CuArray(xs))
```

Listing 6: CuArrays' abstraction is leaky for features not supported by CUDAnative.

language feature will do, from mutable and recursive data types to exception handling and dynamic multiple dispatch.

Any feature that breaks CuArrays' abstraction is problematic. One could plausibly create a copy of such a function or library and modify it to become GPU-specific, but that runs contrary to one of CuArrays' design goals: To transparently accelerate GPU-unaware functions and libraries.

GPUifyLoops

GPUifyLoops is a Julia package that aims to reduce code duplication for high-performance code that targets both CPUs and GPUs [23]. GPUifyLoops introduces a variety of Julia macros that enable programmers to construct Julia functions that can execute efficiently both on CPUs and GPUs.

Listing 7 shows how GPUifyLoops can be used to define a kernel, called `kernel` in this case, that can run both on CPUs and GPUs.

The `@loop` macro invocation on line 2 expands to `for i in 1:size(A, 1)`—a typical for loop—when compiled for the CPU and `for i in threadIdx().x` when compiled for the GPU. In the latter case, Julia understands the for loop to be semantically equivalent to a block of code wherein `i` equals `threadIdx().x`, the thread index in a GPU kernel.

The `@synchronize` macro invocation on line 5 synchronizes the threads in a block when run by the GPU and does nothing when run by the CPU.

Lines 10 and 15 show how the kernel can be applied with relative ease to both CPU and GPU data. In practice, this allows for a straightforward partitioning of workloads between CPUs and GPUs without duplicating code.

Like CuArrays, GPUifyLoops' viability is affected by features unsupported by CUDAnative. Indeed, because GPUifyLoops kernels target both CPUs and GPUs, they need to restrict themselves to a subset of the Julia language and functions that is supported by both CPU and GPU targets. Consequently, GPUifyLoops kernels cannot use any CUDAnative-unsupported features nor can they use user-defined alternatives, because the latter are designed to be GPU-specific. Contextual dispatch can resolve this issue in some cases, but its applicability is limited.

2.3.4 Proof-of-concept object allocation for CUDAnative kernels

We have considered three arguments for implementing currently-unsupported language features in CUDAnative: ease of programming, code reuse and compositional

```julia
function kernel(A)
  @loop for i in (1:size(A,1); threadIdx().x)
    A[i] = 2 * A[i]
  end
  @synchronize
end

# Run the kernel on the CPU.
data = Array{Float32}(undef, 1024)
kernel(data)

# Run the kernel on the GPU.
kernel(A::CuArray) = @launch CUDA() kernel(A, threads=length(A))
data = CuArray{Float32}(undef, 1024)
kernel(data)
```

Listing 7: GPUifyLoops in action. Adapted from `https://github.com/vchuravy/GPUifyLoops.jl/blob/master/examples/simple.jl`.

schemes. Workarounds exist that apply mostly to ease of programming, but have limited pertinence to the other two use cases.

The absence of language features underpinned by the GC is particularly painful in CUDAnative because there is not always a cut-and-dried workaround for these features and their use is idiomatic in Julia. The latter point is evidenced by Listing 6, the integer factorization example.

As identified in Section 2.2.2, mutable and recursive data structures depend exclusively on an automatic memory management scheme such as a GC to work correctly. Other unsupported language features also depend on a GC, but they require varying degrees of additional, dedicated logic.

As a stopgap measure, there exists a proof-of-concept implementation of the Julia GC's allocation function for CUDAnative kernels, implemented as a homonymous function in CUDAnative's GPU runtime library. Said implementation of memory allocation calls CUDA `malloc` to allocate memory for objects and then never frees that memory. In the remainder of this work, we will call memory management schemes that allocate but never free *trivial*.

The proof-of-concept CUDA `malloc` based re-implementation of the Julia GC's allocation function allows for mutable and recursive data structures to be created by CUDAnative kernels. The proof-of-concept allocation implementation does not support other language features such as arrays that depend on dedicated Julia runtime functions, although support for these features can be added by re-implementing those functions in the GPU runtime library.

However, the proof-of-concept allocation implementation has two fundamental limitations that diminish its usefulness in practice:

1. It leaks memory as kernels execute. Long-running and liberally-allocating kernels can hence run out of memory—the CUDA `malloc` heap is only 8 MiB by default.

2. Memory allocated by a kernel using CUDA `malloc` is not and cannot be automatically freed after the kernel completes. That is, memory leaks persist across kernel invocations. Because of this, even kernels that allocate sparingly will eventually run out of memory if enough kernels are executed in sequence. Having a long-running program invoke a sequence of kernels is common in practice and the CUDA `malloc` heap can no longer be expanded once a kernel has run [57], making permanent GPU memory leaks a serious issue.

2.3.5 Goals of this thesis

The goal of this thesis is to implement a GC for CUDAnative-compiled Julia code, in the first place to support mutable and recursive data structures without leaking memory. To accomplish this, we discern two sub-goals:

1. Untether CUDAnative from the Julia runtime library's interface by abstracting over GC-related constructs in the LLVM IR generated by the Julia compiler. Push the CPU-specific lowering of that abstraction to Julia runtime library calls down into the parts of the Julia compiler that explicitly target CPUs.

2. Implement a GC for CUDAnative-compiled Julia code. Have CUDAnative lower our new GC abstraction down to calls to that GC's interface.

The implementations of these sub-goals are discussed in detail in Chapter 3 and Chapter 4, respectively.

Visually, the architecture this thesis envisages and implements corresponds to Figure 2.4: The CUDAnative-specific parts of the compilation pipeline are no longer tied to the Julia runtime library's GC interface and CUDAnative has its own GC library.

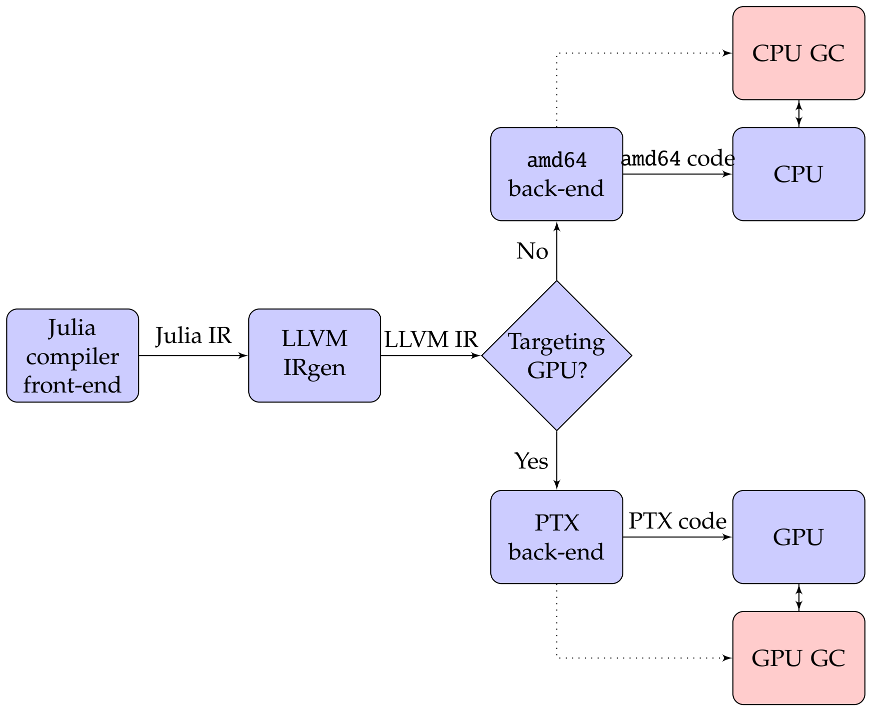

Figure 2.4: CUDAnative's modified compilation pipeline.

2.4 Garbage collection

We have discussed Julia's dependency on a GC and how that dependency affects CUDAnative, but we have thus far sidestepped the question of how a GC works. This section aims to remedy that.

Automatic memory management schemes, including garbage collection, allow programmers to allocate in-memory objects without having to explicitly free that memory when it is no longer in use; the GC can automatically determine which objects are still in use and which ones are not. Objects in the former category are called *live,* those in the latter are called *dead* or *garbage.* When pressured for memory, a GC will look for dead objects and free them, allowing for the memory they occupy to be reused [82].

Two approaches have evolved to address the problem of automatic memory management: *reference counting* and *tracing garbage collection.* Reference counting equips every object with a reference count, which is equal to the number of references to the object. When an object's reference count drops to zero, the application knows that there are no more references to the object and deallocates it.

While conceptually simple, reference counting cannot cope with cyclic object references: when one object refers to another object and vice-versa, then these two objects' reference counts will always be greater than zero, even if there are no other references to the objects other than the cycle. This is known as a *dead cycle* [18].

Julia uses tracing garbage collection, usually shortened to *garbage collection.* A tracing GC does not suffer from dead cycles. Tracing GCs discern live objects from dead objects by reducing their task to a graph problem: "Given a directed graph of objects in memory and a set of root objects, find and recycle objects not reachable from the root set."

In practice, the root set is the union of all global variables and local variables in active functions; all live objects can be found by following paths of object references that start at these variables. The edges in an object graph correspond to object references or pointers that are stored by objects.

In our discussion of various approaches to the problem of garbage collection, we will emphasize in particular those approaches that are used by Julia's GC and the GPU GC we will introduce in Chapter 3 of this thesis.

2.4.1 A taxonomy of garbage collectors

Over time, many different approaches to the problem of garbage collection have been proposed. We will categorize GCs based on five aspects of garbage collectors: precise/conservative, moving/non-moving, collection strategy, allocation strategy, and threading. There are other salient aspects of GCs, but they are not relevant in the context of this thesis.

Precise/conservative

A GC is supposed to find and recycle objects unreachable from the root set. Reachability is a known concept in graph theory, but how can we tell what an in-memory object's outgoing edges are? Similarly, how do we know the contents of the root set? These questions are precisely what gives rise to the precise/conservative distinction.

We say that a GC is *precise* if it can precisely determine the root set and every object's set of outgoing edges. Precise garbage collectors do not divine this information on their own. Rather, the compiler generates additional code or data that a precise garbage collector reads at run time to determine the root set and outgoing edges [10,41].

Julia's GC is precise: it detects references from one object to another by inspecting the type information of objects as well as the data in those objects' fields. Julia's GC represents the root set using a per-thread linked list of GC frames, that is, arrays of root object references. Section 3.2.2 will describe Julia's root set management discipline in more detail.

Conservative GCs are the very opposite of precise GCs: They cannot precisely tell what the root set and sets of outgoing edges are. To cope with this, they conservatively approximate these quantities [19].

Typical conservative GCs approximate references from one object to another by iterating over all aligned, pointer-sized slices of object data. If such a slice of data can be interpreted as a reference to another object, then the GC conservatively assumes that it is one. In a similar vein, conservative GCs can conservatively approximate the root set by iterating over all aligned, pointer-sized slices of data on the stack and in global memory.

For an example of a conservative GC, see the Boehm–Demers–Weiser GC, a conservative GC for the C and C++ languages, which do not offer any compiler support for GCs [19]. Despite its origins as a GC for C and C++, the Boehm–Demers–Weiser GC has become a popular choice for managed language implementations, including Mono and Crystal [49, 80].

Moving/non-moving

It is sometimes advantageous for a GC to be able to move objects around in memory. GCs that do so are called *moving* GCs. Two prominent motivations for moving objects are (1) reducing fragmentation and (2) collection strategies that depend on the GC's ability to move objects [82].

To move an object in memory, a GC needs to copy the object to its new address and update all pointers to the moved object. Updating pointers can be challenging because it implies that the root set must also be rewritten. This usually requires some form of compiler support: Code generated for moving GCs must still function in the face of changing object pointers.

Language implementations that offer no such compiler support all but rule out moving GCs: GCs for those implementations are *non-moving.* That is, any moving GC must also be a precise GC because moving GCs must not accidentally update non-pointer fields. Conservative GCs cannot be sure that a pointer-sized slice of data is an object reference—they merely conservatively assume that it is one.

The converse is not true. Not all precise GCs are moving, as evidenced by Julia's GC, which is precise and non-moving.

Collection strategy

Over the course of many years, a variety of strategies have been developed for finding and collecting garbage objects.

Mark and sweep is arguably the simplest collection strategy: It performs a traversal of the object graph, starting at the roots. Reachable objects are left untouched; the memory used for unreachable objects is reused. One of the key advantages of mark-and-sweep collectors is that they do not require any special compiler support: Mark-and-sweep collectors are non-moving by default and can be conservative [82].

More advanced strategies are typically superior to mark and sweep in terms of non-functional aspects such as time overhead, pause times, and concurrency. However, more advanced collection strategies typically require increasingly deep compiler support.

For instance, a *generational* GC subdivides allocated objects into two or more "generations," usually a "new" generation also known as a *nursery* for objects that have thus far not participated in a collection yet and an "old" generation for objects that have [8]. Based on the heuristic that "objects die young," generational GCs collect younger generations separately and more frequently than older generations.

Generational GCs are used in a variety of managed language implementations: Mono, the .NET framework, and Julia all implement generational GCs [49,81]. Julia's GC uses a conventional two-generation scheme.

A generational GC's core assumption when collecting a generation is that older generations do not point to younger generations. This allows the GC to compute liveness in younger generations without taking the objects in older generations into account.

However, this assumption does not always hold in imperative languages, which allow for objects in older generation to be modified to point to objects in younger generations. For instance, consider Listing 8. A new object is created and a reference to that object is stored in an object of which we will assume that it is in the "old" generation.

To avoid having to mark the entire old generation every time the nursery is collected, compilers that target generational GCs—including the Julia compiler—emit so-called *write barriers* on every write of an object reference to a field of some other object, as in Listing 9. Such a write barrier essentially informs the GC that a

```
new_object = create_new_object()
old_object.field = new_object
```

Listing 8: Code that violates a generational GC's core assumption.

```
new_object = create_new_object()
old_object.field = new_object
write_barrier(old_object, new_object)
```

Listing 9: A write barrier–augmented version of Listing 8. The write barrier is highlighted.

write is taking place, allowing it to compose a set of all younger-generation objects that may be live because they are pointed to from older-generation objects.

Allocation strategy

Memory allocation is a crucial component of a GC's functionality, as well as an area of research in its own right. We will consider two types of allocators in this section: free list allocators and bump allocators.

Free list allocators A *free list allocator* is a simple type of memory allocator that can be used either as a standalone allocator or as a component in a more complex system, such as a GC. A free list allocator functions by storing unallocated or *free* chunks of memory in a linked list—the free list [83].

To allocate a new object, a free list allocator traverses the free list until it finds a free chunk of memory that is sufficiently large to accommodate the object's size. That free chunk is then removed from the free list and allocated to the object. If the free chunk is larger than the size of the object to allocate, then the free chunk is split into two chunks first. The first chunk is allocated to the object and is just large enough for the object. Any remaining memory is moved into the second chunk and added to the free list.

To free a garbage object, a free list allocator adds that object's chunk of memory to the free list.

When multiple threads want to access the free list, a mutex is typically used to ensure that free list modifications are atomic.

Julia's GC uses free list allocators both for large objects and smaller objects. To minimize allocation latency for the latter category of objects, Julia's GC introduces

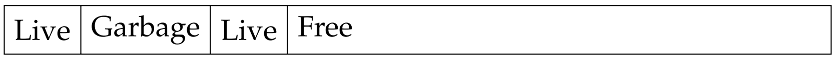

Figure 2.5: An illustration of a situation where a bump allocator cannot recycle garbage objects.

multiple "pools" of free lists that can be used for allocating objects. Large objects share a single free lists.

Bump allocators A *bump allocator* is an even simpler type of memory allocator. A bump allocator manages a pre-allocated, contiguous buffer of storage. The only state retained by a minimal bump allocator is a pointer to the first unallocated byte in that buffer. We will call this pointer the *free pointer.*

To allocate a chunk of memory, a bump allocator increments its free pointer by the number of bytes to allocate and returns the free pointer's old value.

This type of allocator has vanishingly little run time overhead because incrementing a pointer is extraordinarily cheap on modern hardware. Furthermore, bump allocators can be shared by multiple threads in a lock-free manner because pointer increments can be performed atomically using a dedicated atomic addition instruction.

The Achilles' heel of the bump allocator is that it leaks memory when used as a standalone allocator because it cannot effectively recycle memory independently. To see why, consider Figure 2.5. In that listing, we would like to recycle a garbage object, but we cannot: The free pointer is our only means of distinguishing between free and allocated memory. We cannot move the free pointer backward because that would inappropriately free the second live object in the figure.

Bump allocators can overcome this weakness by forming a symbiotic relationship with a moving GC. For example, consider a *semispace collector* [83]. Such a collector maintains two equally-sized, contiguous regions of free memory. Each region is managed by a bump allocator. Only one of these regions is in use at any given time.

When a collection is triggered, the collector moves all live objects from the active region to the inactive region. As the collector moves live objects, it ensures that these objects form a contiguous region of storage. The inactive region's free pointer is then set to the end of the last copied object in the inactive region. Finally, the collector marks the inactive region as active and vice-versa.

Semispace collectors can be highly efficient, even outpacing stack allocation for sufficiently large memory sizes [7]. Semispace collectors appear to be particularly popular in the Lisp family of programming languages [11, 27, 69].

Threading

Most prominent garbage collection algorithms were historically designed for single-threaded programs. In such programs, one can take for granted that the root set and object graph do not change during a collection.

However, this is not the case in multi-threaded programs: If one thread is collecting garbage and another is freely performing computations, then the latter might change the object graph and/or the root set. This is a problematic proposition for all of the collection strategies described in this section.

A common approach to work around this issue is based on the notion of *safepoints* [9]. A safepoint is a point in the instruction stream of a program at which a collection may safely occur. When a thread is in a safepoint, the state of its root set must be stable and correct. For moving GCs, a safepoint is also a point in the program at which object addresses may safely change due to a moving collection.

Safepoints do not arise by accident. Instead, the compiler deliberately ensures that certain points in the program become safepoints. Typically, function calls and loop back-edges are promoted to safepoints.

Interestingly, safepoints are not necessarily specific to multi-threaded programs. Compilers for single-threaded programs may also introduce safepoints as an optimization. By analyzing the instruction stream being compiled, the compiler may detect that ranges of instructions cannot possibly trigger a collection and promote to safepoints only those instructions that can trigger a collection. The Julia compiler is such a compiler. Only a handful of instruction types give rise to safepoints, allowing the compiler to materialize the root set at safepoints only instead of keeping the root set well-defined at all times.

In multi-threaded programs, these safepoints are augmented by the compiler with calls to a *safepoint polling function* [6]. Such a polling function checks if a collection is pending. If no collection is pending, then the polling function does nothing. If a collection is pending, then the polling function notifies the GC that the current thread is in a safepoint and waits for the collection to complete. The collection itself starts when all threads are in a safepoint.

By pausing non-collection threads, safepoint polling functions ensure that the root set and object graph do not change during a collection. Thanks to this property, collection algorithms originally intended for single-threaded programs become applicable to multi-threaded programs.

The Julia compiler does not insert safepoint polling functions; Julia programs remain, at the time of writing, mostly single-threaded.

3 Garbage collection abstractions for Julia

The first main contribution of this thesis is the addition of a new layer of abstraction to the Julia compiler, pursuant to the first goal set out in Section 2.3.5. This chapter describes that contribution, its requirements, and related preexisting Julia compiler machinery.

To create the conditions in which a GPU GC for Julia might thrive, we need to find or introduce an appropriate abstraction layer. On the functional side, CUDAnative needs be able to intercept the LLVM IR generated by the Julia compiler before the IR is made dependent on the low-level interface to Julia's CPU GC.

3.1 Requirements

We want an *appropriate* abstraction layer for GC information. But what, exactly, does "appropriate" mean for an abstraction, especially a compiler abstraction? It appears that this question has yet to be addressed in the literature. The nature of abstractions in computer science has been discussed extensively [24,47,48]. So has the practice of requirements engineering in general, in which abstraction is widely regarded as an important tool [56,61,75]. However, *requirements for abstractions* have eluded further investigation thus far.

As a novel contribution, we introduce two metrics for evaluating the appropriateness of abstractions in general and apply them to the problem at hand:

1. **Robustness**: The degree to which an abstraction can accommodate divergent implementations of some feature.

2. **Thinness**: The amount of engineering required to lower the abstraction to something concrete.

One can easily come up with an abstraction that is maximally robust and minimally thin: The source code being compiled, for example. Indeed, starting from the source code gives a compiler unparalleled latitude in how to compile said source code but implies that every source code–level abstraction needs to be lowered.

Similarly, a maximally thin and minimally robust abstraction would be a compiler's output. There are no abstractions left to speak of in native code—the code is the only implementation of itself.

In practice, what we want is an abstraction that is sufficiently robust to accommodate the implementations we have in mind while being as thin as possible given this constraint.

Concretely, we want to abstract over GC-related information in the Julia compiler in a such a way that supports both the existing CPU GC as well as a range of other GCs, including a GPU GC. This is the robustness constraint. Additionally, we want this abstraction to be wafer-thin: It should be possible to lower the abstraction in an entirely formulaic manner. Lowering the abstraction should not require convoluted analyses.

Our reasoning for introducing this fairly strict thinness requirement is that CUDAnative is developed separately—"downstream"—from Julia itself. Hence, duplicated logic in complicated abstraction lowerings would not just represent a one-time engineering effort; it would be a continuous maintainability headache. Bug fixes and improvements from Julia's lowering would have to be manually translated and applied to CUDAnative's lowering, which would be an undue burden on CUDAnative's limited development resources.

3.2 Preexisting abstractions

Figure 3.1 gives us an overview of all GC abstractions in the Julia compiler. A first, trivial abstraction we can consider is the Julia source code that is sent to the Julia compiler. Information that is pertinent to the GC is entirely implicit in this form.

The Julia compiler then parses the source code into an untyped Abstract Syntax Tree (AST) in which GC-related information is still implicit. The Julia compiler's semantic analysis components then generate a typed AST from the untyped AST. In such a typed AST, the distinction between GC-managed objects and other data can be made based on the types of variables and expressions.

An IR generation pass, abbreviated as *IRgen,* then generates LLVM IR from the typed AST. The IRgen pass annotates every pointer to a GC-managed object with a particular LLVM IR *address space* to distinguish between pointers to GC-managed objects and other pointers [31]. The former are part of the root set, the latter are not.

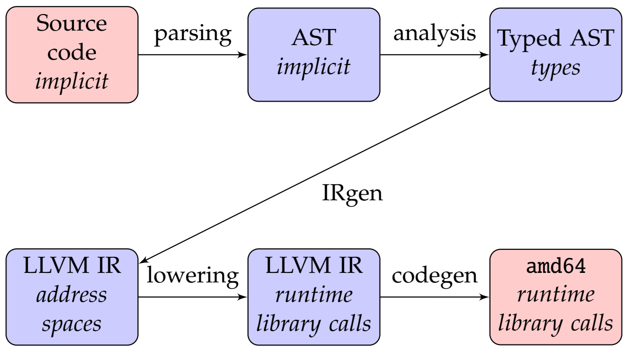

Figure 3.1: Overview of GC abstractions in the Julia compiler.

An LLVM address space is an integer identifier that is included in the type of an LLVM IR pointer [31]. LLVM's optimizer must preserve address spaces as it optimizes code, but the meaning of an address space is loosely-defined; back-ends and front-ends are free to introduce their own interpretations of address spaces.

Because of these characteristics, the Julia compiler can safely tag GC-relevant pointers in LLVM IR with appropriate address spaces and send the IR through LLVM's optimizer before running the GC frame lowering pass. The GC frame lowering pass replaces the implicit root set defined by address space–tagged pointer values with explicit calls to Julia's runtime library.

Finally, the fully lowered LLVM IR is sent to an appropriate LLVM back-end, which produces native code for the target architecture.

We observe that every abstraction-lowering transformation in Julia's compiler pipeline produces a more specialized and more target-specific representation than its input. That is, every lowering trades robustness for thinness. This is intentional: Compilation is a long and complex process. Breaking it down into multiple, smaller steps makes it easier to reason about.

3.2.1 Methodology

We have determined what the requirements are for an appropriate abstraction. With these requirements in mind, we can inspect preexisting abstractions over GC-related functionality in the Julia compiler. We will do so by evaluating the Julia compiler's abstraction layers for GC-related information in reverse order, ignoring the trivial abstractions of source code and native code.

That is, we will start at the Julia compiler's final LLVM IR output, which we will discover to be unsuitable for alternative GC implementations. We will then move backward in the Julia compiler's sequence of abstraction layers until we reach a point where the GC abstraction is sufficiently robust to support other GCs. Since every step of the Julia compiler pipeline trades robustness for thinness, the first sufficiently robust abstraction we find will also be the thinnest one.

We finally evaluate the thinness of that abstraction level, determine that it is insufficient, and conclude that we need a new abstraction layer.

3.2.2 Last abstraction layer: Runtime library calls

The final abstraction layer for GC-related functionality in the Julia compiler consists of LLVM IR that makes direct calls to GC functions in the Julia runtime library. This abstraction—the interface of the Julia runtime library's GC—is designed specifically to support Julia's precise, non-moving, generational GC. The abstraction is insufficiently robust to be of much use for developing a GPU GC. However, the patterns of code that appear at this abstraction layer are useful for pointing out the "gap" between this abstraction and the abstraction that precedes it; we will discuss the latter in the next section.

We identify 6 distinct patterns of GC-related LLVM IR at this abstraction level. Four of these patterns are related to root set management, one allocates memory, and one is related to write barriers in Julia's GC. We will now discuss the three concepts with their related patterns in more detail.

Root set management

Julia's CPU GC is precise. Hence, the LLVM IR generated by the Julia compiler must somehow include instructions that manage the root set: The set of all local and global variables that point to garbage-collected objects.

Julia's GC expects that all root pointers be stored in a linked list of GC *frames:* data structures that are essentially variable-length arrays of root pointers. Figure 3.2 visualizes such a linked list of GC frames. Every blank cell represents a root pointer and every GC frame contains a pointer to the next GC frame. A null pointer indicates the end of the linked list of GC frames.

Every Julia function that defines at least one variable pointing to a GC-managed object (1) allocates memory for a GC frame on entry, (2) appends it to the linked list of GC frames, (3) stores GC roots in the frame whenever necessary and (4) unregisters the frame when the function finishes execution.

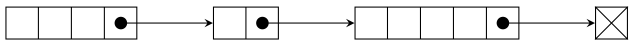

Figure 3.2: A linked list of GC frames. Blank cells contain pointers to GC-managed objects.

```julia
mutable struct Vector2
    x::Float64
    y::Float64
end

@noinline create_vec(x) = Vector2(x, x)

function gc_root_management(x)
    vec = create_vec(x)
    yield()
    return vec
end
```

Listing 10: An example Julia function to illustrate GC root management.

When the GC starts a collection, it walks the linked list of GC frames and marks every root pointer in that list as live. Linked lists of GC frames are a standard technique in managed language implementations, especially those that target platforms without support for scanning the stack for root pointers [10, 41].

We will now consider the LLVM IR that implements this root management scheme, as generated by the Julia compiler. Listing 10 is a small Julia script that is designed to trigger root set management logic. It defines three top-level entities:

1. A mutable data type called `Vector2`. Because it is a mutable data type, any instance of `Vector2` will become a GC-managed object.

2. A helper function `create_vec` that creates an instance of a `Vector2`. `create_vec` is marked `@noinline` to simplify the LLVM IR generated for `gc_root_management`.

3. A function `gc_root_management` that first calls `create_vec`, then calls the `yield` function, and finally returns `create_vec`'s result.

   By inserting the rather opaque `yield` function in between the `create_vec` call and the return statement, we force `gc_root_management` to keep `create_vec`'s result alive, that is, to store that result in a GC frame.

Listing 11 is a simplified version of the LLVM IR generated by the Julia compiler for the `gc_root_management` function. Of particular interest are the four root set management patterns. All four show up in the listing:

1. A GC frame pointed to by `%gcframe` is allocated and zero-filled. The GC frame is three pointer-sized elements in size to accommodate an integer value that indicates the size of the GC frame, a pointer to the next GC frame in the linked list of GC frames, and a single root that will be stored in the GC frame, in that order.

2. The GC frame is then initialized. This consists of two distinct steps:

   a) First, the first element of the GC frame is set to the number of root pointers in the GC frame plus one. This allows the GC to tell how many root pointers are stored in the GC frame when it walks the linked list of GC frames.

   b) Next, the GC frame is inserted into the linked list of GC frames. This is done by storing the head pointer of the linked list of GC frames in the second field of the GC frame. Said head pointer is then updated to point to the GC frame. To acquire this head pointer, we consult a thread-local data structure called the "PTLS states."

3. The code subsequently calls the `create_vec` function and stores its result in the GC frame. This result value will be considered live by the GC as long as it remains in the GC frame and the GC frame remains registered in the linked list of GC frames.

   After the call to `create_vec`, we call `yield`. `yield` may trigger garbage collections, but this will not make the GC collect `create_vec`'s result because that result will be considered live thanks to its presence in the GC frame.

4. Finally, the GC frame is removed from the linked list of GC frames by undoing the action performed in item 2.

We observe a number of properties of this final abstraction layer.

1. The code is hardwired to use a linked list of GC frames as opposed to some other data structure for managing the root set.

2. At this level of abstraction, GC frames are always allocated using an `alloca` instruction, that is, they will be placed in the GPU's stack memory. GPU

```
define %jl_value_t* @gc_root_management(i64) {
top:
; (1) Allocate a GC frame that can store GC-managed objects.
%gcframe = alloca %jl_value_t*, i32 3
call void @llvm.memset.p0i8.i32(
  %jl_value_t* %gcframe, i8 0, i32 24, i32 0, i1 false)

; (2) Initialize the GC frame.
; (2.a) Set the GC frame's size field.
%2 = getelementptr %jl_value_t*, %jl_value_t** %gcframe, i32 0
%3 = bitcast %jl_value_t** %2 to i64*
store i64 2, i64* %3
; (2.b) Append the GC frame to the linked list of GC frames.
%ptls = call %jl_value_t*** @julia.ptls_states()
%4 = getelementptr %jl_value_t**, %jl_value_t*** %ptls, i32 0
%5 = getelementptr %jl_value_t*, %jl_value_t** %gcframe, i32 1
%6 = bitcast %jl_value_t** %5 to %jl_value_t***
%7 = load %jl_value_t**, %jl_value_t*** %4
store %jl_value_t** %7, %jl_value_t*** %6
%8 = bitcast %jl_value_t*** %4 to %jl_value_t***
store %jl_value_t** %gcframe, %jl_value_t*** %8

; (3) Create a GC-managed object, put its address in the GC frame.
%9 = call %jl_value_t* @create_vec(i64 %0)
%10 = getelementptr %jl_value_t*, %jl_value_t** %gcframe, i32 2
store %jl_value_t* %9, %jl_value_t** %10

; Call 'yield'.
call void @yield()

; (4) Remove the GC frame from the linked list.
%12 = getelementptr %jl_value_t*, %jl_value_t** %gcframe, i32 1
%13 = load %jl_value_t*, %jl_value_t** %12
%14 = getelementptr %jl_value_t**, %jl_value_t*** %ptls, i32 0
%15 = bitcast %jl_value_t*** %14 to %jl_value_t**
store %jl_value_t* %13, %jl_value_t** %15

; Return.
ret %jl_value_t* %9
}
```

Listing 11: GC root management at the lowest level of abstraction.

stack memory is only accessible to the GPU in the CUDA programming model [57,66].

Due to technical restrictions that will be explained in more detail in Section 4.2.2, some degree of CPU participation in collections is a hard requirement if one wants to obtain a GPU GC that can expand its heap—a desirable feature.

Furthermore, traditional mark-and-sweep collection algorithms correspond to mostly-sequential pointer-chasing logic with highly irregular memory access patterns. Such workloads do not play to the strengths of GPUs, but CPUs excel at them [52,79].

In summary, some CPU participation in collections is required and CPUs are more suitable for collection algorithms. It then stands to reason that we use the CPU for collecting the GPU GC's heap. However, this is precisely what is prohibited by storing the root set in GPU stack memory.

3. A pointer into the linked list of GC frames is stored in the "PTLS states" thread-local data structure, queried by a call to the `julia.ptls_states` intrinsic:[1]
`%ptls = call %jl_value_t*** @julia.ptls_states()`.

   The PTLS states are an amalgamation of various pieces of thread-local information from many different components of the Julia runtime including garbage collection but also signal handling, IO handling, pseudo-random number generation and others. This makes PTLS states a highly CPU-specific data structure and hence no GPU GC should be built on top of it.

At this point, it is fair to conclude that this abstraction is insufficiently robust to support a GPU GC. However, the abstraction's underlying *concepts* are by and large sufficiently robust. Indeed, the parts of the abstraction that rule out a GPU GC implementation are—quite frankly—trivial details. If we could somehow tailor the abstraction to use a different type of memory for GC frames and remove the reliance on PTLS states, then we would have something that is workable for a GPU GC.

Memory allocation

The fifth pattern we observe is memory allocation. To allocate GC-managed memory, LLVM IR generated by the Julia compiler will call either the `jl_gc_pool_alloc`

[1]In a compiler, an intrinsic (function) is a function whose meaning is intimately understood by the compiler. Additionally, intrinsic functions are declared but never defined: They are lowered to something concrete before the end of the compiler's pipeline. Consequently, the native code generated by the compiler never calls intrinsics, unlike normal functions.

function or the `jl_gc_big_alloc` function, depending on the size of the allocation. These functions could plausibly be re-implemented by a GPU GC. The only wrinkle is that both functions take a pointer to PTLS states as an argument, but this dependency could reasonably be removed by a cleanup pass.

Write barriers

The Julia compiler places write barriers right after writes that store an object reference in a field of some other object. This gives rise to the sixth pattern: A write barrier implemented by a call to the `jl_gc_queue_root` function from the Julia runtime library. This is not a real blocker for implementing a GPU GC: Write barriers can be elided for non-generational GCs and generational GCs can simply provide their own implementation of `jl_gc_queue_root`.

Conclusion

We conclude that the final abstraction layer offered by the Julia compiler is *almost* sufficiently robust to support a GPU GC. Two aspects that block a GPU GC implementation are (1) insufficient control over how GC frames are allocated and managed, and (2) pointers to PTLS states being passed around.

3.2.3 Penultimate abstraction layer: Address spaces and intrinsics

Since the last abstraction layer is not quite robust enough for our purposes, it is only logical for us to move on to the next-to-last one. Neither the penultimate nor the ultimate abstraction layer in the Julia compiler has a standard name, but the transformation that turns the former into the latter is called "late GC frame lowering." We will therefore henceforth refer to the next-to-last abstraction layer as "before GC frame lowering" and the final abstraction layer as "after GC frame lowering."

Whereas LLVM IR after GC frame lowering is characterized by its hardwired GC frame management scheme and its reliance on runtime library calls, LLVM IR before GC frame lowering operates at a high level of abstraction. Specifically, there is no explicit GC frame management to speak of. Instead, the pointers that are to be included in a function's GC frame are given a special *pointer address space.* A pointer address space in LLVM is an integer value that is included in the type of a pointer. The semantics of a pointer address spaces are entirely target-specific [53].

Listing 12 corresponds to Listing 11 prior to GC frame lowering. It succinctly captures the gist of Listing 10 without the low-level implementation details of

```
define %jl_value_t addrspace(10)* @gc_root_management(i64) {
top:
%1 = call %jl_value_t addrspace(10)* @create_vec(i64 %0)
call void @yield()
ret %jl_value_t addrspace(10)* %1
}
```

Listing 12: GC root management before GC frame lowering.

Listing 11. First, Listing 12 calls `create_vec`, then `yield` is called and finally `create_vec`'s result is returned.

To distinguish between pointers to GC-managed objects and pointers to other data, the Julia compiler tags all instances of the former with address space 10. These address space annotations are encoded as `addrspace(10)` in the LLVM IR from Listing 12.

This abstraction allows us to distinguish between different pointer types and treat them accordingly, but does not commit itself to any particular root set management technique one way or the other; it is quite robust in that sense.

Object allocation and write barriers are handled by dedicated intrinsic functions at this layer of abstraction: `julia.gc_alloc_obj` and `julia.write_barrier`, respectively.

Appropriateness

The abstraction presented by LLVM IR prior to GC frame lowering is more than robust enough to allow for a GPU GC to be implemented on top of it.

However, there is quite a large gap in terms of abstraction that is monolithically closed by the GC frame lowering pass. To close this gap efficiently, the GC frame lowering pass performs a complicated analysis that determines when, if ever, to store GC roots in a GC frame. To reduce stack memory consumption, it also tries to reuse GC frame slots by using a graph coloring algorithm [32].

If we were to try and lower this abstraction to GC frame managing code for a GPU GC directly, then we would essentially have to replicate the GC frame lowering algorithm or settle for a simpler but less effective ersatz algorithm. Neither option is very appealing.

In conclusion: The abstraction prior to GC frame lowering is sufficiently robust for our purposes. But it is not thin enough.

3.3 Low-level GC intrinsics

As it turns out, GC frame lowering seems to be a major tipping point in terms of GC-related abstractions: The abstraction that precedes it is quite robust but not very thin; the abstraction that succeeds it is not robust enough.

This observation gives rise to this thesis' first contribution: An additional layer of abstraction inside the Julia compiler that is wafer-thin and just robust enough to support the existing precise, non-moving, generational GC as well as GCs that require less compiler support. The abstraction arises naturally by refactoring the GC frame lowering pass into two passes:

1. A mostly GC-agnostic GC frame lowering pass that produces optimized yet abstract instructions for GC frame management in the form of intrinsics.

2. A GC-specific lowering pass that replaces these intrinsics with concrete implementations that are appropriate for Julia's CPU GC.

Crucially, virtually all of the complexity of the GC frame lowering pass is moved into the GC-agnostic pass—the GC-specific lowering is entirely formulaic. This makes the GC-specific lowering's implementation practically trivial. Hence, implementing a different lowering for another GC is easy in this scheme.

Figure 3.3 illustrates the compilation pipeline that corresponds to the lowering scheme we are introducing. It is satisfyingly similar to the modified compilation pipeline we envisioned as the first major goal of this thesis in Section 2.3.5.

3.3.1 Abstraction

We hinted in the previous section that the abstraction produced by the GC-agnostic lowering is intrinsics-based. We will now discuss the details of the abstraction: The intrinsics and their meanings.

The abstraction introduces 6 new intrinsics, each of which corresponds exactly to one of the patterns of code as discussed in Section 3.2.2. Like those patterns, we bin the intrinsics into three distinct categories based on the functionality they provide: Root set management, memory allocation, and write barriers.

Root set management

We introduce four intrinsics that capture the essence of GC frame management. The GC-specific lowering expands each of these intrinsics to its corresponding pattern.

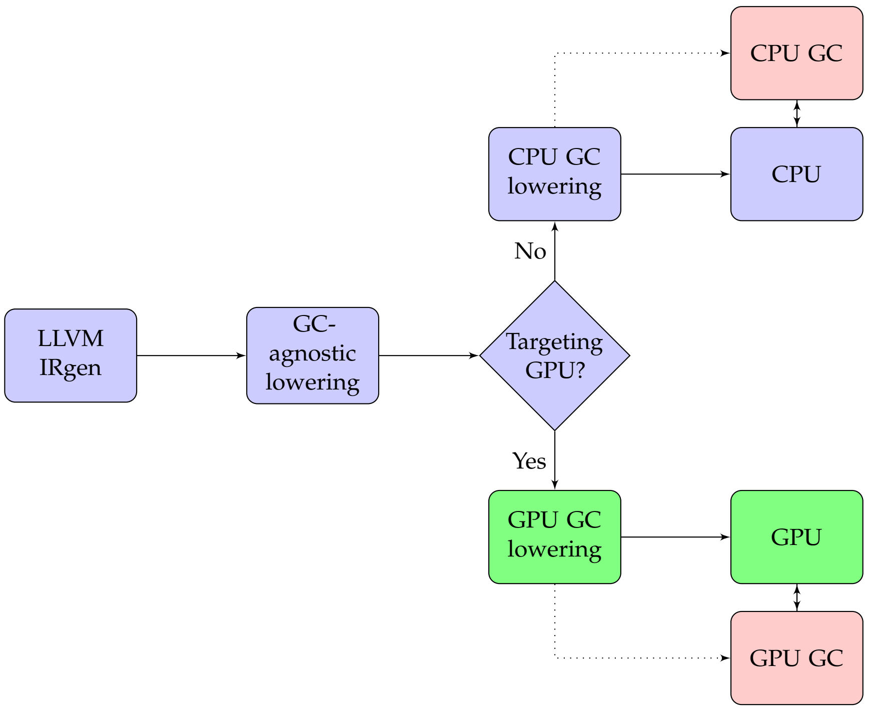

Figure 3.3: Modified GC lowering scheme. The "GPU GC lowering" and "GPU" blocks refer to CUDAnative compiler infrastructure and its targeted hardware, respectively.

1. `julia.new_gc_frame(n)` allocates a new GC frame that can accommodate at least `n` GC roots. It returns a pointer to that frame.

2. `julia.push_gc_frame(gcframe, n)` takes a GC frame and registers it with the GC. The size of that GC frame is also provided for the GC-specific lowering's convenience; the lowering can effortlessly discard it if it is not needed.

3. `julia.get_gc_frame_slot(gcframe, i)` accepts a pointer to a GC frame and produces a pointer to the `i`th root pointer in that GC frame.

4. `julia.pop_gc_frame(gcframe)` unregisters a GC frame, removing its contribution to the root set.

Listing 13 demonstrates what these intrinsics look like in practice when applied to our running example. The robustness of the abstraction becomes apparent by comparing Listing 13 to Listing 11: The low-level patterns we identified in section Section 3.2.2 are all abstracted over and generalized by replacing them with intrinsics.

The thinness of the new abstraction can be observed by taking Listing 12 into consideration as well. Indeed, while sufficiently general to support alternative GC

```
define %jl_value_t* @gc_root_management(i64) {
top:
; (1) Allocate a new GC frame big enough for one root.
%gcframe = call %jl_value_t** @julia.new_gc_frame(i32 1)

; (2) Register it with the GC.
call void @julia.push_gc_frame(%jl_value_t** %gcframe, i32 1)

; (3) Create a Vector2 instance and add it to the GC frame.
%1 = call %jl_value_t* @create_vec(i64 %0)
%2 = call %jl_value_t** @julia.get_gc_frame_slot(
    %jl_value_t** %gcframe, i32 1)
store %jl_value_t* %1, %jl_value_t** %2

; Call 'yield'.
call void @yield()

; (4) Unregister the GC frame.
call void @julia.pop_gc_frame(%jl_value_t** %gcframe)
ret void
}
```

Listing 13: GC root management after GC-agnostic GC frame lowering.

implementations, the intrinsics in our new abstraction can be expanded formulaically. The address space–based abstraction from Section 3.2.3, on the other hand, depends on complex algorithms for an efficient lowering.

Memory allocation

When targeting Julia's CPU GC, the Julia compiler emits a call to an allocation function. The exact allocation function depends on the size of the object to allocate: `jl_gc_pool_alloc` is used for small objects and `jl_gc_big_alloc` for large ones. That is not necessarily something every GC wants to do.

Accordingly, we abstract over the exact allocation function by introducing a fifth intrinsic: `julia.gc_alloc_bytes(ptls, n)`. It allocates `n` bytes of storage, plus sufficient headroom for an object type tag.[2] As a historical concession, it also takes a pointer to PTLS states.

[2]The Julia compiler marks all GC-managed objects with a tag that identifies the object's type.

This approach differs from the one taken by the root set management intrinsics, which do not require a pointer to the PTLS states as a parameter. We believe the root set management intrinsics' design to be superior to `julia.gc_alloc_bytes`, as PTLS states are a highly CPU-specific and implementation-specific construct whereas the intrinsics introduced for this abstraction layer are designed to be platform-agnostic. However, the preexisting `julia.gc_alloc_obj` intrinsic from Section 3.2.3 already takes a pointer to the PTLS states. It would be silly to discard the PTLS states pointer argument when lowering `julia.gc_alloc_obj` to `julia.gc_alloc_bytes` only to recover it when `julia.gc_alloc_bytes` is in turn lowered to a call to an allocation function.

That being said, a pointer to the PTLS states can be trivially discarded when lowering the intrinsic, so it does not impose a real burden on lowerings that do not or cannot rely on PTLS states.

Write barriers

Similarly, we define a sixth intrinsic, `julia.queue_gc_root(root)`, that marks an object as being pointed to from another object that may not be in the same generation. The CPU GC targeting lowering expands this intrinsic to a call to `jl_gc_queue_root`, which essentially has the same semantics.

3.3.2 Implementation

We implemented the contributions described thus far in a fork of the Julia compiler.[3] Concretely, we first refactored the old GC frame lowering pass, `LateLowerGCFrame`, by introducing a helper class `JuliaPassContext` that deduplicates the repetitive pattern of finding and/or defining intrinsics and well-known functions.[4]

After that, we incrementally extracted all CPU GC-specific logic from pass `LateLowerGCFrame` and placed them in a separate pass: `FinalLowerGC`. Both passes rely on `JuliaPassContext` to juggle the intrinsics and well-known functions the passes accept and emit. We finally expanded the Julia compiler's test suite to test `LateLowerGCFrame` and `FinalLowerGC` passes in isolation.

Our contributions are designed to be suitable for inclusion in the upstream Julia project. To that end, we sent a pull request to the upstream Julia repository, which

[3]See `https://github.com/jonathanvdc/julia/tree/staged-gc-lowering` for the exact changes.

[4]A well-known function is a real function that the compiler knows intimately. Unlike intrinsics, well-known functions are backed by definitions.

has since been reviewed, approved and, merged in.[5] In total, our pull request consisted of 988 additional lines of code and 273 removed lines.

[5]See `https://github.com/JuliaLang/julia/pull/31135` for the pull request.

4 A garbage collector for CUDAnative

The abstractions introduced in the previous chapter are a means to an end: To implement alternative garbage collectors. This chapter discusses two GPU memory management schemes built on those abstractions: A trivial memory management scheme and a general-purpose GC. The trivial memory management scheme is compatible with any memory allocator, including CUDA `malloc` and a bump allocator that we implement ourselves. Like most GCs, our GC needs fine-grained control over allocation and hence uses a custom-designed allocator.

4.1 Trivial memory management

To demonstrate the robustness of the abstraction layer introduced at the end of the previous chapter and enable limited dynamic memory allocation for GPU kernels, we first implement a trivial memory management scheme: One that allocates objects but never bothers to free them.

Trivial memory management does not impose any requirements on the allocator other than the basic requirement of reserving a slice of memory. Because of this, the trivial memory management scheme is completely independent of the allocator in use: It is entirely compatible with existing, non-GC memory allocators. This is not the case for more advanced memory management schemes, which continue to manage memory after it is allocated, imposing additional requirements on the allocator.

In the context of GPU programming, trivial memory management is actually not such a radical idea. Indeed, to the best of our knowledge, there are no managed language implementations that target GPUs and use a full-fledged GC. At the time of writing, the only known GPU-targeting managed language implementation that supports transparent dynamic allocation other than CUDAnative is Rootbeer, a Java-to-CUDA compiler [62]. Rootbeer uses trivial memory management.

Furthermore, typical GPU kernels are short-lived and rarely rely on dynamic memory allocation. In using a trivial memory management scheme, we obviate the need for setting up a GC heap while providing support for crucial language features such as exception handling.

Exception handling is used to handle exceptional conditions, but such conditions rarely occur in production code. Disallowing language features outright means that many kernels fail to compile, even if these features' implementations would otherwise never be executed at run time. Trivial memory management offers an alternative in this case.

The trivial memory management scheme is also a useful yardstick for measuring the overhead of an actual GPU GC, allowing us to quantify whether using a GC on a GPU is sensible or not from a performance perspective by comparing it to non-GC allocators.

4.1.1 GC intrinsic lowering

We demonstrate that trivial memory management can readily be implemented based on the GC intrinsics introduced at the end of the previous chapter by expanding those intrinsics using a set of straightforward rules.

1. GC frames are lowered as stack-allocated arrays of pointers. Intrinsics that register and unregister GC frames are simply deleted.

2. The `julia.gc_alloc_bytes` intrinsic is replaced by a call to an allocation function. By default, this is the CUDA memory allocation function: `malloc`. Kernel invocations are free to specify an alternative allocation function if they so choose.

3. All calls to the `julia.queue_gc_root` intrinsic are deleted.

This lowering replaces the proof-of-concept implementation of trivial memory management in CUDAnative mentioned in Section 2.3.4. Said previous implementation was built on the address space–based abstraction from Section 3.2.3. The new lowering instead departs from the abstraction level defined in Section 3.3.

Trivial memory management nets us a fully functional implementation, ushering in a host of Julia features that would ordinarily require intervention from the GC for CPU platforms. The only downside is that memory runs out after a while if dynamic memory allocation is used liberally.

4.1.2 A bump allocator

Kernels that complete quickly are unlikely to exhaust the pool of available GPU memory, even if they allocate liberally. However, memory allocated by CUDA `malloc` outlives the kernel that allocated it. Since trivial memory management never frees memory, this implies that repeatedly launching kernels from the same process will cause GPU memory to run out for that process if those kernels use trivial memory management and CUDA `malloc`.

Additionally, CUDA `malloc` appears to be quite slow in practice, as we will observe in Chapter 5.

To address these two issues, we implement a bump allocator for GPU kernels that serves as an alternative to CUDA `malloc` within the framework of trivial memory management. The allocator gives rise to the following workflow:

1. Before launching a kernel, the CPU allocates a pool of GPU memory, unified memory, or pinned memory.
2. When the kernel is launched, a pointer to the start and end of said pool is passed to the kernel.
3. The kernel runs. When a GPU thread needs to allocate memory, it does so by atomically incrementing a pointer.
4. The kernel finishes and the CPU deallocates the entire memory pool.

The bump allocator integrates well with our trivial memory management scheme, which is designed to accommodate any allocation function. Hence, calls to the bump allocator are injected into functions using the same mechanism as for CUDA `malloc`. That is, allocations are automatically lowered to call the bump allocation function rather than CUDA `malloc`. Kernels do not have be modified to use the bump allocator; the code that launches the kernel merely has to specify that it wishes to use a bump allocator.

The bump allocator has two major benefits over CUDA `malloc`. First and foremost, it ensures that memory leaks do not outlive kernel launches. Second, we expect a bump allocator to represent a lower bound on allocation latency. Allocating an object using a bump allocator consists of atomically incrementing a pointer and little else. It is difficult to imagine a functioning allocator that does even less work. This sentiment is confirmed in Chapter 5, which shows the bump allocator to be very fast indeed.

4.2 A garbage collector for GPU memory

The trivial memory management scheme discussed in the previous section is acceptable for low-level GPU kernels that rarely allocate GC-managed objects. However, as outlined in Section 2.3.3, code reuse and compositional schemes can give rise to kernels that do not restrict themselves to the low-level programming paradigm for which a trivial memory management scheme is useful.

This motivates the emergence of a memory management scheme that does not leak memory, a GC for objects allocated on the GPU.

4.2.1 Related work

Our case for a GPU GC is a natural consequence of the fact that we mean to support high-level programming on GPUs. That notwithstanding, the concept of a GC for a programming language that compiles to GPUs is actually quite novel. Indeed, as previously stated, we know of no GPU programming language implementations that use a GC in a transparent way, as one might expect for a managed language that runs on a CPU.

The idea of a GC for GPU kernels is not entirely unprecedented: Veldema and Philippsen implemented a mark and sweep garbage collector for GPUs [76]. Their GC both allocates and collects memory on the GPU, but garbage collection is not *transparent* in their implementation: When their GC runs out of memory, it returns a null pointer. The GPU kernel programmer has to check for null pointers after every allocation and decide how to handle them. Veldema and Philippsen recommend that programmers handle these null pointers by introducing logic that terminates the kernel, triggers a collection and then restarts the kernel.

We argue that Veldema and Philippsen's approach, while novel, is invasive even for relatively simple kernels. Indeed, it requires that every allocation is rewritten from the ground up, precluding reuse of code that was originally written for CPU execution such as Julia's standard library. Furthermore, aborting kernels is easy, but writing code that restores a kernel's state exactly is delicate and labor-intensive.

Maas et al. [54] as well as Abhinav and Nasre [2] have proposed modified versions of Veldema and Philippsen's collector. Their collectors are designed to offload garbage collection of CPU memory to the GPU in order to reduce garbage collection times. Their work is innovative, but it does not address the issue of transparent garbage collection for GPU kernels.

4.2.2 High-level design

The lack of a cut-and-dried solution to the problem of transparent garbage collection for GPU kernels prompted us to design a solution from the ground up. Since this brings us into uncharted territory and the novelty of our solution lies in the problem it solves rather than *how* it solves that problem, we decided on a design carefully constructed to maximize the odds of success in a functional sense. The non-functional aspects of the implementation of course cannot be neglected—a common theme in system software—but we envisage our GC as more of proof of concept that might well serve as a starting point for future innovation in this area of research than the *ne plus ultra* of GPU garbage collection.

For this reason, we implemented a free list based allocator and a straightforward conservative, non-moving mark and sweep collection algorithm.

More prominently, we decided on a design wherein allocations are performed directly by the GPU and collections are performed by the CPU. This may be slightly controversial: As evidenced by Veldema and Philippsen's work, it is not strictly necessary to refer collections back to the CPU. Indeed, it is not inconceivable that the GPU might be instructed to reuse its already-running threads for the purpose of collecting garbage.

We base our decision to move collections to the CPU in their entirety on the following points, which are in line with our earlier reasoning in Section 3.2.2:

- The CPU can allocate additional memory at will and share it with the GPU using the `cudaHostAlloc` function of the CUDA Application Programming Interface (API) [57]. To the best of our knowledge, GPUs cannot do the same: They are assigned a fixed-size, usually tiny heap for dynamic memory allocations. Once the heap runs out, CUDA `malloc` returns null pointers.

  We consider the ability to expand the GC heap to be a crucial feature of any GC. For this reason, we foresee that the GC will have to fall back to the CPU whenever the GC heap proves insufficiently large after a collection, regardless of which device performs the collection. Hence, the GC needs some mechanism for faulting back to the CPU either way; on-GPU collections will not rescue us from having to implement one.

- Veldema and Philippsen report that by using 256 GPU threads for collections they can achieve a speedup of up to 11× compared to using just one GPU thread for collections [76]. However, Veldema and Philippsen do not compare their GC to a CPU GC, even though CPUs are known to excel at workloads that feature non-streaming memory accesses and synchronized access to resources, typical properties of mark-and-sweep GCs that use free lists such as Veldema and Philippsen's [52, 60]. One might reasonably expect a CPU GC to offer a speedup equal to or even in excess of the 11× speedup compared to a single GPU thread. Indeed, Maas et al.'s work shows that their reference CPU GC outperforms their GPU GC on almost every benchmark [54]. Only Abhinav and Nasre report that their implementation outperforms the reference GC, with an average performance improvement of 10% [2].

- Veldema and Philippsen's collection algorithm is fairly complex and consists of many different kernel launches and synchronization steps, all of them driven by the CPU. We estimate that it would take significant effort to rework their algorithm as a GPU-only algorithm.

  Implementing a simple CPU mark-and-sweep algorithm, on the other hand, is certainly less risky and we expect it to at least be competitive with a GPU-only algorithm, as per the previous point.

  Moreover, a straightforward collection algorithm is fully in line with our design methodology for the GC: Our primary mission is to focus on functional aspects rather than on convoluted optimizations.

We anticipate one notable non-functional downside to having the CPU perform collections for the GPU GC: Memory has to be copied back and forth between the CPU and the GPU when collections occur. To this end, we will examine the additional overhead imposed by collections in Chapter 5.

While the CPU is responsible for performing our GC's collections, the GPU is responsible for triggering those collections: When the GPU runs out of memory, it notifies the CPU that a collection is due, waits for the CPU to finish collecting garbage and resumes. We call this general scheme, where the GPU requests that the CPU performs some action, a GPU *interrupt.*

4.2.3 GPU interrupts

In the literature, we discern two mechanisms that fit our definition of an interrupt: GPU-to-CPU *callbacks* [71] and GPU *system calls* [78]. Both essentially achieve the same result, with the former being more reliant on software and the latter on hardware.

- GPU-to-CPU callbacks are interrupts that are defined purely in software: A GPU thread sets an interrupt flag and waits until the CPU gives the GPU an all-clear signal. Similarly, the CPU waits for the GPU to set the interrupt flag and then runs its interrupt logic before giving the all-clear signal.
- GPU system calls instead rely on hardware support to send a signal from a GPU thread to the CPU and vice-versa, requiring no busy waiting from either device. This is an intuitively more attractive solution, but research into this approach has thus far focused on AMD GPUs rather than NVIDIA GPUs and required modifications to the hardware drivers [78]. This latter requirement is problematic considering that NVIDIA's drivers are closed source, unlike AMD's.

  The key to re-implementing GPU system calls for NVIDIA devices seems to have eluded us so far. We believe that the most promising approach to implementing system calls is by hijacking the `brkpt` breakpoint instruction defined by the PTX ISA using the CUDA Debugger API. This is also the approach taken by the researchers who implemented GPU system calls for AMD GPUs, as we learned from private correspondence with one of the authors.

  However, for unknown reasons, the CUDA Debugger API did not work when we tried to use it, freezing the main thread when the CUDA Debugger API was instructed to launch a kernel from the same process that runs the CUDA Debugger API. This appears to be a known deficiency with the CUDA Debugger API, but we are not aware of any workarounds [72].

A high-level interrupt API

To sidestep current issues in implementing GPU system calls for NVIDIA devices without ruling out the possibility of a future system calls–based interrupt implementation, we introduce an abstraction over the concept of an interrupt and implement it for now using GPU-to-CPU callbacks.

Specifically, we associate a single *interrupt handler* with every kernel launch. The interrupt handler is executed exactly once per interrupt. We also define the following high-level GPU-facing API for interrupts:

1. `interrupt_or_wait()::Bool` requests an interrupt and waits until the interrupt completes. At most one interrupt may be active at any time per

launched kernel. If an interrupt is already running, then this function waits for that interrupt to complete but does not request an interrupt of its own. `interrupt_or_wait` returns `true` if an interrupt was successfully requested by this function; otherwise, `false`.

2. `wait_for_interrupt()` waits for the current interrupt to finish, if an interrupt is currently pending or running.

3. `wait_for_interrupt(fun::Function)` waits for the current interrupt to finish, if an interrupt is currently pending or running. `fun` is repeatedly executed until the interrupt finishes.

4. `interrupt()` repeatedly requests an interrupt until one is requested successfully.

This API represents a versatile mechanism that can be used to implement both non-idempotent interrupts like incrementing a counter and idempotent interrupts such as triggering a collection. We will focus on the latter use case. The literature discusses alternative use cases, including disk I/O, network I/O, and debugging [71,78].

4.2.4 Collection

Collections are triggered by a GPU thread when it runs out of memory to allocate. To trigger a collection, the thread calls `interrupt_or_wait()`, which will either wait for the active collection to finish or start a brand new one.

Safepoints

It is of paramount importance that all GPU threads are paused during garbage collection. If they are not, then they might modify the object graph in unpredictable ways. Moreover, the GC frame lowering pass only guarantees that all live objects are reachable in the object graph when all threads are in a select set of positions in the program, the safepoints from Section 2.4.1. The GC frame lowering pass assumes that only function calls are safepoints, at the caller side. The state of the root set is undefined at all other points in the program.

To ensure that all GPU threads are in a safepoint for the entire duration of a collection, we assign a *safepoint state* to every warp. There are three legal values a safepoint state can have: (1) “not in safepoint”, (2) “in safepoint” and (3) “in perma-safepoint”.

```
function gc_safepoint()
    wait_for_interrupt() do
        gc_set_safepoint_flag(in_safepoint; overwrite = false)
    end
    return
end
```

Listing 14: Safepoint polling function implementation.

When the interrupt handler fires on the CPU in response to a collection request from the GPU, it will first set every warp's state to "not in safepoint" unless that warp is in a perma-safepoint.

Once the interrupt handler notices that all warps are in either the "in safepoint" or "in perma-safepoint" state, the interrupt handle will proceed by starting the actual garbage collection algorithm.

The "not in safepoint" and "in safepoint" states are commonplace in GCs for multi-threaded programs [9]. A *perma-safepoint* is a bit of an oddity. The perma-safepoint state differs from the normal safepoint state in that the collector will not wait for warps that are in the perma-safepoint state. The collector will wait for warps in the "not in safepoint" or "in safepoint" states to reach a safepoint polling function.

Perma-safepoints are useful for situations where we know that a warp cannot reach a safepoint. Currently, perma-safepoints are used for two purposes:

- Warps that have finished executing a kernel put themselves in the perma-safepoint state just before they finish. Such warps can no longer change the root set or the object graph, nor can they reach a normal safepoint. Hence, a perma-safepoint is the only "right" state for such warps.

- Warps that request a collection also place themselves in the perma-safepoint state prior to requesting the collection.

When a GPU thread enters a normal safepoint, it will execute a compiler-injected call to the safepoint polling function. Listing 14 contains the verbatim implementation of the safepoint polling function. Said function waits for the current collection, if any, to complete and repeatedly sets the warp's safepoint state to "in safepoint" until the collection completes.

Collection algorithm

The GC's collection algorithm consist of four distinct phases.

1. **Mark:** the GC traverses the entire object graph, starting at the roots. During this traversal, it marks all reachable objects as live.

   During the marking process, roots are identified precisely but object references in object fields are approximated conservatively by scanning live objects for aligned, pointer-sized slices of data that point into GC-managed objects.

   We had initially hoped to implement a fully precise collector, but quickly learned that doing so would be infeasible within the scope of this work. The Julia compiler emits type metadata that any GC should be able to inspect in order to precisely find object references, but an alternative precise GC implementation is hampered by the diversity and complexity of this type metadata. In the Julia compiler, the type metadata gives rise to a 565-line marking function that manages a custom stack and relies on computed `goto` statements [37] as the primary means for control flow.[1] The format of this metadata is both delicate to parse and an implementation detail subject to change without notice.

   Ideally, we would refactor the CPU GC's marking logic to depend on a helper function that accepts type metadata and produces a description of all object references in an instance of such a type. This helper function could then be exported for use by alternative GC implementations. However, this would be a large change to which the Julia community might not be receptive and it is not clear at this time what the impact of such an abstraction would be on GC pause times. We can only assume that the marking function's current shape is due to it being designed and optimized specifically to minimize pause times.

2. **Sweep:** the GC iterates through all allocated objects that are not live and deallocates them.

3. **Defragment:** the GC iteratively merges adjacent free list entries to combat memory fragmentation.

4. **Expand:** if the GC determines that the amount of free memory is insufficiently large, then it will allocate a new chunk of pinned memory and turn it into

[1]See the `gc_mark_loop` function defined in `https://github.com/JuliaLang/julia/blob/dea494038df310f8e19de740d9a16b764210f9c3/src/gc.c`.

a free list entry. This step allows the GC to accommodate memory-hungry applications with ease without introducing an exorbitantly large initial GC heap size.

After collecting garbage, the interrupt handler terminates and GPU threads are free to resume execution.

4.2.5 Choice of memory type

One aspect of the GC that we have thus far not discussed in detail is the type of memory it uses. This would be a moot point for a CPU GC, but GPUs offer a variety of different memory kinds.

We consider three types of GPU memory:

1. **Device memory.** Under normal circumstances, device memory would be the straightforward choice for our GPU GC's heap. But as it turns out, we cannot use device memory for this purpose.

   Device memory needs to be copied explicitly between the CPU and the GPU. If we were to use device memory for the heap, then our GC's mark phase would have to copy every live GC-managed object from the GPU and inspect it for references to other objects.

   This is not a problem in and of itself. Rather, the semantics of the copy functions are an issue for our GC: Memory copies can at best be coaxed into running *concurrently* with the GC-enabled kernel, but they cannot be forced to run *in parallel* with said kernel.

   In practice, our experiments indicate that the GPU's scheduler may schedule crucial memory copies for execution *after* the kernel completes. This causes a deadlock: The kernel cannot resume execution until the GC-related memory copy completes and the GC-related memory copy will not even be started until the kernel terminates.

2. **Unified memory.** Unified memory is next in line to become our preferred type of memory for the GPU GC's heap. Since unified memory pages migrate to the device that is currently using them, we would expect a GC heap based on unified memory to be approximately as efficient as one based on device memory.

   Misfortune strikes again. Accessing a unified memory page from a CPU generates a segmentation fault if that page has been touched by a still-active kernel running on a pre–Pascal microarchitecture GPU [40]. This is precisely what happened when we initially experimented with a unified memory GC heap: The GPU would allocate and populate an object in unified memory, after which the CPU would try to inspect it during a collection, triggering a segmentation fault.

Pascal is a fairly recent microarchitecture: the first Pascal hardware premiered in 2016. Consequently, we cannot reasonably expect all CUDAnative users to have a Pascal or post-Pascal GPU, ruling out unified memory as a substrate for our GPU GC.

3. **Pinned memory.** By exclusion, we arrive at pinned memory as a means for implementing the GPU GC. Pinned memory can be manipulated directly by both the CPU and the GPU, which is highly convenient for collections in general and the mark phase in particular. Most importantly, pinned memory does not suffer from the show-stopping limitations of device memory and unified memory.

Our GPU GC implementation uses pinned memory for all of the data it manages, that is, both the GC's auxiliary data structures and the GC heap. Additionally, we designed the GPU GC so that it will be trivial to migrate to unified memory in the future, when post-Pascal has become the norm.

4.2.6 Root set management

Our use of pinned memory has a profound impact on the root set management scheme. Since GPUs cannot allocate pinned memory, we have the CPU set aside a fixed-size pinned memory buffer for every GPU thread. This buffer is used to store root pointers and is used like a stack, managed using a top-of-stack pointer. The four GC frame management intrinsics we introduced in the previous chapter are lowered as follows:

1. `julia.new_gc_frame(n)` returns the root buffer's top-of-stack pointer.
2. `julia.push_gc_frame(gcframe, n)` increments the top-of-stack pointer by `n` pointers.
3. `julia.get_gc_frame_slot(gcframe, i)` indexes `gcframe` at index `i`.
4. `julia.pop_gc_frame(gcframe)` sets the top-of-stack pointer to `gcframe`, restoring it to the state it was in when `julia.new_gc_frame` was called.

In this scheme, root buffers can overflow. The same limitation affects the CPU GC: Allocating too many GC frames will make the stack overflow. Root buffer overflows are rare in practice both for the CPU GC and the GPU GC because stack sizes and root buffer sizes are chosen to be reasonably large.

When a root buffer does overflow, a check in the GPU GC implementation of `julia.push_gc_frame` detects this overflow and prints a message indicating that an overflow occurred, along with the index of the thread that caused the overflow.

4.2.7 Allocation

In its most basic mode of operation, the GC's allocator manages a single free list. To give the GC an overview of all allocated chunks of memory, the GC's allocator also manages an *allocation list:* an analogous list of all allocated chunks of memory. Whenever a new object is allocated, the chunk of memory allocated to that object is moved from the free list to the allocation list.

All free list modifications are performed atomically to fend off race conditions. To guarantee atomicity, we equip the free list with a global mutual exclusion lock and acquire said lock whenever we want to allocate memory.

Arenas

The basic allocation scheme described thus far is perfectly functional, even on a GPU. However, it serializes memory allocations, which would be undesirable even for CPUs and doubly so for massively parallel GPUs.

To combat this phenomenon, we will introduce multiple *arenas.* We define an arena as a possibly non-contiguous region of memory managed by an allocator that is fully independent from the other allocators. Hence, an allocator will only make threads contend for lock acquisitions or other resources within the confines of a single arena.

Our reason for introducing the notion of an arena is that arenas are composable: We can use multiple arenas to build a composite arena. We propose two ways to compose arenas.

1. **In parallel:** to reduce contention, we can take a sequence of arenas and assign each arena to a group of threads. Since contention is by definition an intra-arena phenomenon, two groups of threads can allocate memory in a fully independent fashion from their respective arenas, boosting parallelism.

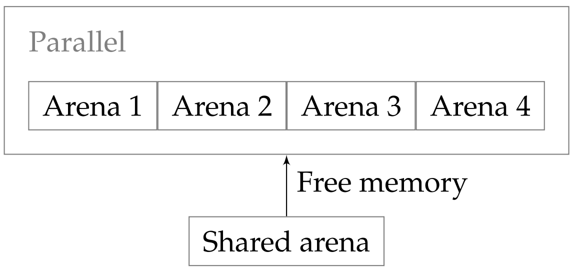

Figure 4.1: Composed arena allocator.

2. **Hierarchically:** a hierarchy of arenas is readily established by trying to allocate from one arena first and moving on to the next arena if the first arena cannot satisfy the request.

We also note that, in a hierarchical composition scheme, compatible arenas can "steal" resources from each other. For example, a free list entry can painlessly be removed from one free list and added to another.

Taking this into account, we now equip the GPU GC with an arena that includes two layers of arena composition, as in Figure 4.1. The outer composition is hierarchical, with the first layer in the hierarchy being a number of parallelly composed free lists and the second layer being a single, shared free list. The latter is granted generous amounts of memory, but incurs more contention. The parallel free lists are each assigned a modest amount of memory. When one of the parallel free lists runs out of memory, it tries to steal a large chunk of memory from the shared free list. This large chunk can then be subdivided into many small chunks by the parallel free list without the contention overhead of the shared list.

4.3 Limitations

Before concluding this section, we identify two main limitations in our GPU GC: concurrent kernel launches and dynamic parallelism.

4.3.1 Concurrent kernel launches

The CUDA programming model allows for kernels to be launched in such a way that the kernels can run concurrently. CUDAnative also exposes this functionality.

Our GC's core design is well-suited to accommodate multiple kernel launches as all GC state is local to a kernel launch. However, our polling-based interrupt implementation blocks the CPU thread that launches a kernel. This would not be an issue in most contemporary programming languages: Launching a separate, lightweight thread for polling the GPU and handling interrupts would allow the main thread to continue executing and launch concurrent kernels.

However, Julia does not support launching new threads. Adding this functionality to the Julia compiler itself would also be a rather large endeavor, as going from a single-threaded to a multi-threaded execution model requires non-obvious changes to core language implementation components. For example, the Julia compiler would have to be taught to inject calls to safepoint polling functions.

The easiest solutions to the problem of concurrent kernel launches would be to wait for Julia to gain full-fledged threading support.

Alternatively, we could re-implement interrupt handling and garbage collection logic in a language that does support multi-threading. Doing so would significantly complicate CUDAnative's architecture, so this option is not our preferred outcome.

Finally, a more elegant option would consist of implementing non-blocking interrupts for CUDAnative. However, it is unclear to us how that can be done; our efforts to implement GPU system calls for CUDAnative have failed so far.

4.3.2 Dynamic parallelism

Dynamic parallelism refers to the practice of launching new CUDA kernels from already-running CUDA kernels on the GPU [3, 4]. Dynamic parallelism has been part of the CUDA API since CUDA 5.0 and CUDAnative has recently gained support for dynamic parallelism as well [12].

The GC's design and implementation predates the addition of dynamic parallelism to CUDAnative. Hence, the GC was not designed with dynamic parallelism in mind. Two aspects of the GC's design are incompatible with dynamic parallelism:

1. A fixed-size array of safepoint states is allocated to every warp before a kernel starts. A dynamic kernel launch produces additional warps that need to be managed by the GC, but the safepoint array cannot be extended because its size is fixed when the original kernel is launched.

2. A fixed-size array of root pointer stacks is allocated to every GPU thread before a kernel starts. Again, a dynamic kernel launch would require additional root pointer stacks, which the current design cannot accommodate.

In the future, the GC can be updated to use different, extensible data structures for safepoint states and root sets. This would allow for GC-enabled dynamic kernel launches.

For the time being, GC-enabled dynamic kernel launches are prohibited. However, GC-enabled kernel launches can still dynamically launch non-GC kernels.

4.4 Conclusion

We designed and implemented two distinct lowering strategies: trivial memory management and a GC. The former is aimed at enabling sparingly used language features in typical GPU kernels and conforms to the state of the art in memory management for managed language implementations that target GPUs. The latter is designed to support idiomatic Julia code. It is, to the best of our knowledge, the first implementation of fully transparent garbage collection for GPU kernels.

Designing, implementing, and refining the GPU GC required significant effort that translates to 3696 additional lines of CUDAnative code. To verify the GPU GC's functional and non-functional properties, we also implemented 1315 lines worth of GC benchmarks.

The GPU GC is designed for direct integration into CUDAnative. To that end, we have sent a pull request containing the GPU GC implementation,[2] which is currently under review.

[2]See `https://github.com/JuliaGPU/CUDAnative.jl/pull/419` for the current state of the pull request.

5 Evaluation

The previous chapter presented the design of a GPU GC, as implemented for CUDAnative.[1] However, a mere design does leave some important questions unanswered. In this chapter, we will answer the following questions.

1. Qualitatively, which language features are enabled directly by the GC?
2. Furthermore, can the GC be leveraged—as we previously claimed—to implement language features that rely on having a GC but require additional logic for a working implementation?
3. Quantitatively, what is the GC's *variable overhead*, that is, how well does the GC perform in terms of run time compared to trivial memory management when run on a set of benchmarks that rely strongly on memory allocation?
4. Quantitatively, how does the initial GC heap size affect the GC's performance?
5. Quantitatively, what is the GC's *constant overhead*, that is, how well does the GC perform in terms of run time compared to trivial memory management when run on a set of benchmarks that do not rely on memory allocation at all?

We identify the first two questions as related to functional requirements and the last three points as pertaining to non-functional requirements. When it comes to non-functional aspects, we are interested mainly in telling if the GC described in this thesis has acceptable overhead, that is, if its overhead is sufficiently small for its use to be justifiable. We will answer all five questions in order.

5.1 Functional aspects

When it comes to the GC's functional aspects, we distinguish between *directly enabled* language features and *indirectly enabled* language features. The former category refers to Julia language features that can be wholly implemented solely by introducing a

[1]See `https://github.com/jonathanvdc/CUDAnative.jl/tree/gc-staging` for the source code.

memory management scheme. The latter refers to features that rely on a memory management scheme plus some dedicated runtime library support. These categories are related to questions one and two of our evaluation, respectively.

5.1.1 Directly enabled language features

The language feature that is directly enabled by GC is mutable and/or recursive `struct` instantiation.

The GC hence supplants the proof-of-concept implementation for mutable and/or recursive `struct` allocation mentioned in Section 4.1.1. Said proof-of-concept implementation consists of a trivial memory management scheme based on CUDA `malloc`, causing it to leak memory permanently. That is, the proof-of-concept implementation supports object allocation for some time, until a kernel exhausts the pool of available GPU memory. Once that happens, no kernel can allocate objects anymore.

The GC improves on the proof-of-concept implementation by ensuring that memory does not leak. When the GC is enabled, allocations in one kernel do not affect the amount of memory available to the next kernel and even in-memory memory leaks are combated by occasional garbage collection phases.

Mutable and/or recursive `struct` types are a central part of Julia's type system. In Julia, they represent the only "right" way to implement a number of data structures that are standard in all of computer science. For example, consider Listing 15, which implements a singly linked list. A singly linked list, represented by the abstract and generic `List` type, can either be empty (`Nil`) or nonempty (`Cons`). In this case, `struct Cons` is both mutable and recursive. This is the idiomatic Julia way to implement a linked list, as evidenced by the fact that the FunctionalCollections Julia package independently developed a near-identical definition of a linked list,[2] which is published in a production-ready package.

To further demonstrate the usefulness of mutable and/or recursive `struct` instances on GPUs, we created example programs that use such `struct` instances to implement a variety of nontrivial algorithms that includes linked list reductions, binary tree searches, heap-allocated fixed-sized array operations, a genetic algorithm, and a basic optimizer for a hypothetical Static Single Assignment (SSA) form compiler

[2]See `https://github.com/JuliaCollections/FunctionalCollections.jl/blob/master/src/PersistentList.jl` for the list definition.

```julia
abstract type List{T}
end

struct Nil{T} <: List{T}
end

mutable struct Cons{T} <: List{T}
  value::T
  next::List{T}
end
```

Listing 15: Types for a singly linked list.

IR [26].[3] The source code for these algorithms is idiomatic Julia code that does not function without a memory management scheme in place.

5.1.2 Indirectly enabled language features

The majority of Julia language features that depend on GC support also depend on a number of runtime library functions. Such features include boxing, array creation and manipulation, exception handling and dynamic multiple dispatch.

Arrays

To show that our GC abstractions and implementation are an adequate substrate for indirectly enabled language features, we implement crucial low-level array functionality. Specifically, we implement the following functions:

1. `jl_alloc_array_1d` allocates a one-dimensional array. We implement this function by injecting a pass into the CUDAnative compiler pipeline that lowers calls to this function to appropriate LLVM IR. `jl_alloc_array_1d` cannot be implemented as a function in the GPU runtime library because it relies on type reflection. Such reflection consists of accessing data structures the Julia compiler has placed in CPU memory. This works well for programs that run on the CPU, but the GPU cannot access arbitrary CPU memory. By making

[3]See `https://github.com/jonathanvdc/CUDAnative.jl/tree/gc-staging/gc-benchmarks` for the example programs' source code. The programs referred to here are `binary-tree.jl`, `genetic-algorithm.jl`, `linked-list.jl`, `matrix.jl`, `ssa-opt.jl`, `static-arrays.jl` and `stream-queries.jl`.

the compiler lower `jl_alloc_array_1d`, we can perform the reflection ahead of time.

2. `jl_alloc_array_2d` allocates a two-dimensional array. To implement this function, we use the same strategy as for `jl_alloc_array_1d`.

3. `jl_alloc_array_3d` allocates a three-dimensional array. We use the same strategy as for `jl_alloc_array_1d`.

4. `jl_new_array` allocates an array of arbitrary dimensions. We use the same strategy as for `jl_alloc_array_1d` plus some additional logic to accommodate the fact that callers of `jl_new_object` may specify dimensions either using a run-time–constructed tuple in GPU memory or a compile-time constant tuple in CPU memory. The former requires that code be generated to read out the tuple's fields, the latter requires that dimensions be extracted at compile time.

5. `jl_ptr_to_array_1d` creates a one-dimensional array that uses a user-specified memory buffer as backing storage. To implement `jl_ptr_to_array_1d`, we use the same strategy as for `jl_alloc_array_1d`.

6. `jl_ptr_to_array` creates an array of arbitrary dimensions that uses a user-specified memory buffer as backing storage. We use the same strategy as for `jl_new_object`.

7. `jl_array_grow_end` expands an array by appending a number of default-initialized elements to the end of the array. To do so, `jl_array_grow_end` may have to reallocate and copy the entire array. We implement this function as a regular Julia function in the GPU runtime library.

8. `jl_array_grow_beg` expands an array by appending a number of default-initialized elements at the beginning of the array, which may require array buffer reallocation. We implement this function as a regular Julia function in the GPU runtime library.

9. `jl_array_grow_at` expands an array by appending a number of default-initialized elements at a user-specified index into the array, which may require array buffer reallocation. We implement this function as a regular Julia function in the GPU runtime library.

10. `jl_array_sizehint` hints that an array is likely to require a particular capacity. Such a hint is processed by preemptively resizing the array to that capacity. We implement this function as a regular Julia function in the GPU runtime library.

11. `jl_array_del_end` deletes a number of elements at then end of an array. We implement this function as a regular Julia function in the GPU runtime library.

12. `jl_array_del_beg` deletes a number of elements at then beginning of an array. We implement this function as a regular Julia function in the GPU runtime library.

13. `jl_array_del_at` deletes a number of elements at a user-specified index into an array. We implement this function as a regular Julia function in the GPU runtime library.

These functions form the backbone of Julia's low-level array interface. They enable a large amount of array-related features. These features include (1) array creation, (2) reading and writing array elements, (3) querying an array's dimensions, (4) inserting elements into arrays, (5) deleting elements from arrays, (6) increasing the capacity of arrays, (7) wrapping unmanaged buffers in arrays, (8) array comprehensions [44], and (9) high-level array functions such as `fill`, `fill!` and `similar`. We again rely on a series of example programs to substantiate each of these claims.[4]

Standard library types

The array functionality enabled by `jl_alloc_array_1d` and `jl_array_grow_end` is also sufficient for an array-based data type in the standard library, `BitVector`, to become usable from GPU kernels. `BitVector` is a dense array of one-bit Booleans that uses an array of integers as its backing storage. We successfully used it to implement arbitrary-width integer arithmetic and—to our delight—found that it worked out of the box, that is, without any modifications whatsoever.[5] This is quite a triumph, considering that the logic implementing `BitVector` is lengthy and complicated. Moreover, it was to the best of our knowledge never intended for use on GPUs and yet works flawlessly.

Other convenient array-based data types including dictionaries and hash sets are now almost within reach: The essential infrastructure for supporting these types is

[4]See `https://github.com/jonathanvdc/CUDAnative.jl/tree/gc-staging/gc-benchmarks` for the example programs' source code. The programs referred to here are `array-expansion.jl`, `array-features.jl`, and `array-reduction.jl`.

[5]See the `bitvector.jl` example program for the source code.

in place, but their implementation in the standard library relies on recursion for their internal logic. At the time of writing, recursion support has not yet been included in CUDAnative as it first requires changes to the Julia compiler.

In early 2018, a pull request that makes the changes necessary for CUDAnative to support recursion was opened in the Julia repository, but it has not yet been merged in.[6] Once these changes are merged in, CUDAnative will support recursion, and—barring unforeseen obstacles—dictionary and hash set data types will also work out of the box for GPU kernels.

We initially tried to merge the recursion-enabling pull request ourselves in hopes of verifying this claim, but the pull request had already diverged too far from the main Julia repository. Since it was posted, significant parts of the recursion-enabling pull request have been re-designed and re-implemented in separate pull requests to the Julia repository, which have been merged in. These later pull requests are incompatible with the recursion-enabling pull request. Weaving these incompatible pull requests together into a functional whole would be tantamount to a re-implementation of the recursion-enabling pull request, a task better left to the Julia contributor who opened the pull request in the first place.

5.2 Non-functional aspects

We now quantify the GC's variable and constant overhead by comparing it to two versions of the trivial memory management scheme as described in Section 4.1.1: One equipped with the CUDA `malloc` allocator and another with the bump allocator from Section 4.1.2. We also measure the GC's sensitivity to initial heap sizes.

All measurements reported in this section were performed by an Ubuntu 18.10 machine sporting a GeForce GTX 970 GPU [59] and an Intel Core i7-6700K CPU [25]. The machine has 16 GiB of main memory and the GPU has 4 GiB of device memory. When measuring GC overheads, we will take two GC configurations into consideration.

- **Unoptimized GC** uses a single shared free list that initially contains all free memory in the GC heap.
- **Optimized GC** has eight local free list arenas, each of which is initially assigned one tenth of the initial GC heap, as well as a shared free list arena containing two tenths of the initial GC heap.

[6]See `https://github.com/JuliaLang/julia/pull/25984` for the pull request.

The optimized GC corresponds to the arena-based composition scheme from Section 4.2.7. To ensure that the trivial memory management baseline's allocator does not run out of memory, we set the CUDA `malloc` heap size to 64 MiB. We set the bump allocator's heap to 60 MiB, which is slightly smaller than the `malloc` heap size because the bump allocator's heap is allocated from the `malloc` heap; taking 64 MiB is not an option. 60 MiB is more than enough memory for every benchmark we will run and can hence provide a like-for-like comparison with a 64 MiB CUDA `malloc` heap. Similarly, we set the initial GC heap size to 60 MiB unless otherwise stated.

5.2.1 Variable overhead

To measure the variable overhead of the GPU GC compared to trivial memory management, that is, the difference in run times with and without the GC, we run all of the example programs from the previous section and measure their wall-clock run times. There are no benchmark suites that measure GC overhead on GPUs because there are currently no other GCs for GPUs. Hence our decision to reuse the functional benchmarks: They represent a reasonable set of benchmarks that corresponds neatly to the features we implemented. The functional benchmark programs are more than sufficiently comprehensive to determine if the GC scheme we are proposing holds water.

Figure 5.1 summarizes the performance of the unoptimized and optimized GC implementations. Specifically, it shows the run times of the functional benchmarks using unoptimized and optimized GC implementations, as well as two baseline trivial memory management schemes. For easy comparison, these run times are normalized.

We initially intended to include other memory allocators for GPUs in our comparison, as they have been shown to deliver large speedups over CUDAnative. Xmalloc promises speedups of 211× compared to CUDA `malloc` [43]. ScatterAlloc achieves speedups between 10 and 100× compared to CUDA `malloc` [70]. Halloc claims speedups of 3× to 1000× compared to ScatterAlloc [5].

However, using existing alternative memory allocators from CUDAnative appears to be impossible: Such allocators are packaged as CUDA libraries. CUDA libraries consist of both GPU kernels and CPU functions. These two types of code are closely intertwined: The CPU functions refer directly to global variables in GPU memory, which are laid out at link time. This implies that we need to use the same linking strategy for both GPU kernels and CPU functions—if we link them separately, then the addresses they both refer to will diverge. However, the CUDA programming model does not support dynamic linking [57] and Julia only supports dynamic linking.

Measurements represent the wall clock time of kernels plus the amount of time required to construct and deconstruct a kernel's GC or bump allocator heap. Any actions performed prior to and after kernel execution, including memory transfer and compilation are not included. Measurements were made by the robust BenchmarkTools benchmarking package [22]. Every benchmark, memory allocation scheme pair was run for 90 s in total. Measurements were aggregated by taking their median rather than their mean, as per the BenchmarkTools manual's recommendations [65].

Some interesting phenomena can be discerned by observing Figure 5.1:

- The unoptimized GC is usually faster than the CUDA memory allocator. This is a bit of a surprise as our initial goal was to create a GC with acceptable overhead compared to trivial memory management. We never purposefully tried to outperform `malloc`.

  In hindsight, this is likely due to CUDA's `malloc` implementation rather than the unoptimized GC's implementation. Alternative GPU memory allocators appear to outperform `malloc` by substantial margins, which suggests that CUDA's `malloc` is implemented overly naively [5,43,70].

  Regardless, this result implies that the GC's performance is up to scratch: it greatly improves on the status quo ante established by the proof-of-concept trivial memory allocation scheme that CUDAnative previously used for object allocations.

- The optimized GC is always faster than the CUDA allocator barring the "matrix" benchmark. Usually, the optimized GC is faster by a substantial margin.

  As for the "matrix" benchmark: This benchmark allocates a small number of very large, long-lived chunks of memory and then manipulates that memory. We suspect that the GC's poorer performance there is mostly due to the fact that the GC uses pinned host memory whereas `malloc` produces device memory. The "matrix" benchmark appears to be the only benchmark where this is a bottleneck.

  The other benchmarks instead predominantly allocate many small, short-lived objects, the type of workload our GPU GC is tuned for. The same can be said

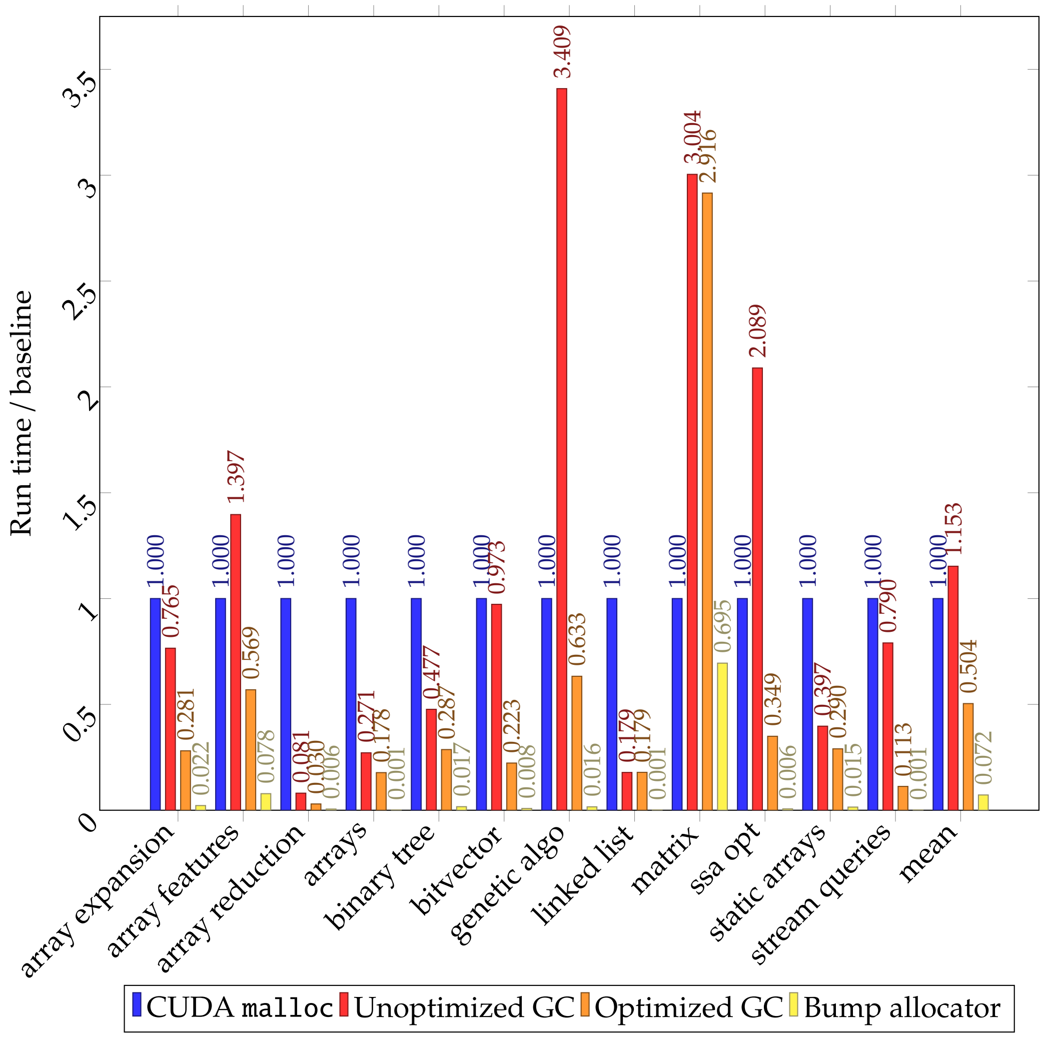

Figure 5.1: Variable overhead of unoptimized, optimized GCs relative to trivial memory management schemes. Lower scores are better.

for Julia's CPU GC: The CPU GC is a generational GC with specialized pools for allocating small objects, meaning that it is optimized specifically for many small objects with limited lifetimes.

Whether GC-enabled GPU kernels will lean toward a few large, long-lived objects or many small, short-lived objects—as CPU programs do—is an open question. Our GC is, to the best of our knowledge, the first fully transparent GC implementation for GPU kernels and hence we have no corpus of third-party GC-enabled GPU kernels to examine in search of the idiomatic allocation model for such kernels. The best we can do at this time is optimize for what is common in CPU programs, that is, many small, short-lived objects.

- The optimized GC is always faster than the unoptimized GC. The performance gap is relatively modest on some benchmarks but substantial on others. Additionally, the optimized GC appears to offer more predictable improvements over the CUDA allocator than the unoptimized GC.

- The bump allocation has excellent performance that represents a lower bound on allocation latencies, regularly achieving speedups of over 1000× relative to CUDA `malloc`. The bump allocator's performance remains out of reach of the GC for now, suggesting that there is room for improvement.

In summary: The optimized GC is almost always an improvement over CUDA `malloc` and the unoptimized GC is usually an improvement. We conclude that the GC's variable overhead compared to a CUDA `malloc`–based trivial memory management scheme—the status quo ante—is more than reasonable.

5.2.2 Initial heap sizes

The previous section evaluated the dynamic overhead of the GPU GC by comparing it to CUDA `malloc` and the bump allocator for identical heap sizes. In this section, we will ascertain what the effect is of the initial heap size on the GC's performance.

To that end, we run the same benchmarks we used to determine the GC's variable overhead and measure the run time of the unoptimized and optimized GC configurations for increasingly large initial heap sizes.

We do not restrict the GC's ability to increase the size of its heap beyond the initial size, as heap expansion is a normal and necessary part of the GC's collection algorithm. Hence, smaller heap sizes give rise to a "pay as you go" system where the GC heap is initially insufficiently large but gets expanded during collections to accommodate the working set size. Larger heap sizes imply a larger up-front investment and fewer collections during the kernel's execution.

The mechanism for expanding GC arenas is as follows: If the amount of free memory assigned to an arena remains below a certain threshold even after recycling garbage objects, then it is deemed to be *memory-starved.* For every memory-starved arena, a section of free memory is allocated and assigned to that arena. The size of the new section of memory equals the starvation threshold, which is 1 MiB for local arenas and 4 MiB for the global arena.

Figure 5.2 shows how the mean normalized run time across all benchmarks evolves as we increase the initial heap size for both the unoptimized and optimized GC configurations. Every data point in Figure 5.2 is the normalized median benchmark run time, averaged out over all benchmarks. Benchmark run times are normalized relative to the the optimized GC configuration with 60 MiB.

Figure 5.2 shows that the initial heap size barely affects performance, with larger initial heaps performing slightly worse than smaller ones. We believe that the initial heap size's limited impact on performance is due to the fact that allocation times dominate, as can be inferred from Figure 5.1. Interrupt and collection overheads appear to be mild in comparison, as evidenced by Figure 5.2. Smaller initial heap sizes reduce the amount of up-front effort required for allocating and initializing potentially unused sections of the heap, possibly explaining why smaller heaps appear to perform slightly better than their larger counterparts.

5.2.3 Constant overhead

To measure constant overhead, we repeat the same experiment as for variable overhead, but with a different set of benchmarks: A Julia port of the Rodinia benchmark suite [14,21]. Rodinia is designed to represent typical scientific computing loads for GPUs. None of the Rodinia benchmarks allocate memory dynamically, which makes them suitable candidates for ascertaining the constant overhead of using a GPU GC.

Figure 5.3 displays the GPU kernel run times for each Rodinia benchmark and memory management scheme, normalized against CUDA `malloc`. Every Rodinia benchmark consists of one or more kernels. Every measurement is the sum of the minimum kernel run times for every kernel. We compute the minimum of every kernel's run time as it is a robust measure of a kernel's execution time under ideal circumstances [22]. Only kernel execution times are measured here. Set-up and tear-down times are not included.

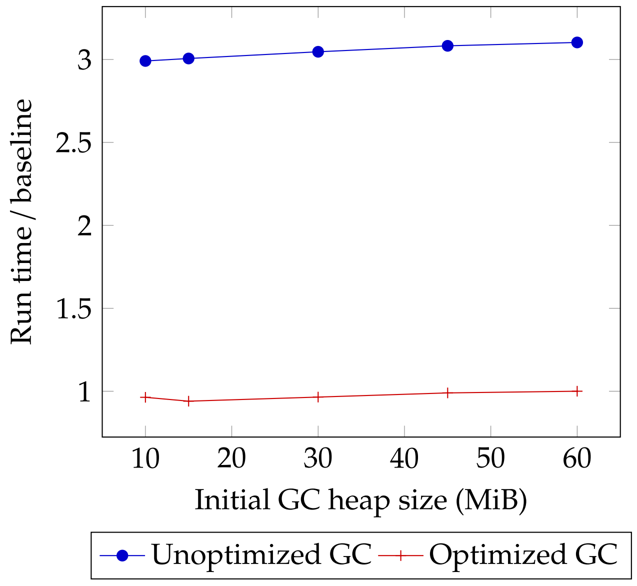

Figure 5.2: Normalized run times of benchmarks with unoptimized, optimized GCs. Lower scores are better.

There seems to be clear evidence that using a GC incurs some constant overhead: The relative additional cost of using a GC is never less than zero and quite large for a number of benchmarks, especially for "backprop".

Some additional overhead is only normal—every kernel is modified to support the GC but then never uses the GC, giving us all the drawbacks and none of the upsides. However, the amount of additional overhead is at first blush rather alarming: Even if we were to treat "backprop" as an outlier, then we'd still have large overheads for "pathfinder", "lud", and "bfs".

Figure 5.4 helps to explain this phenomenon. It plots CUDA `malloc`–configured kernel run times on the x axis and normalized run times for other configurations on the y axis. Clearly, benchmarks that spend less time in total on GPU kernels are more affected by the overhead. Indeed, an average run of the leftmost benchmark—"backprop"—spends less than a millisecond on its GPU kernels and is most affected. An average run of the rightmost benchmark—"streamcluster"—spends a little under a second on the GPU and is barely affected. These observations suggest that the GC's constant overhead is indeed a constant factor.

The GC's constant overhead is due to a write to pinned memory that occurs at the end of every GC-enabled kernel to move the active warp from the "not in safepoint" state to the "in perma-safepoint" state. It stands to reason then that short-lived kernels will be most affected by this write's overhead. Relatively long-lived kernels,

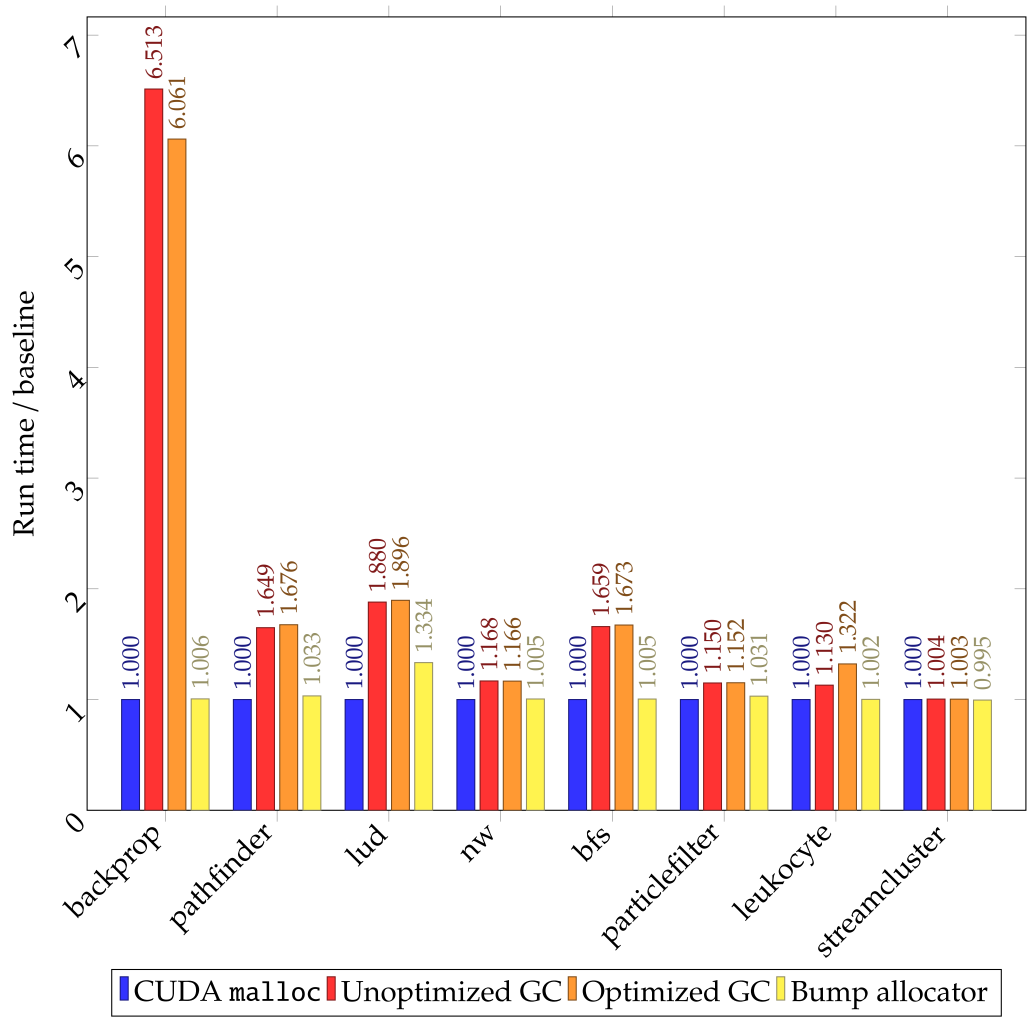

Figure 5.3: Constant overhead of unoptimized, optimized GCs relative to trivial memory management schemes. Lower scores are better.

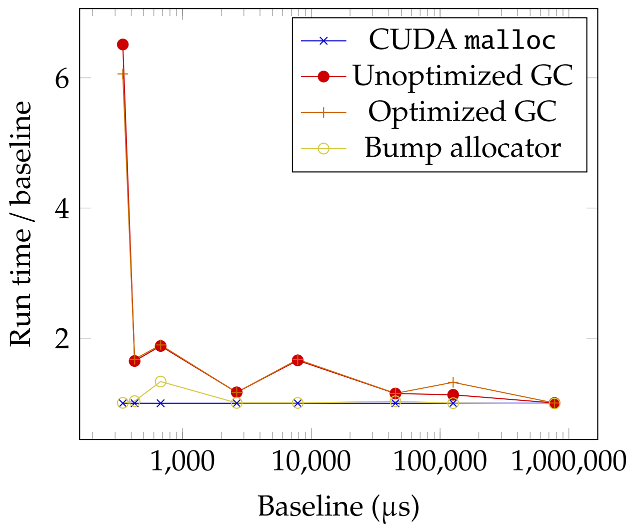

Figure 5.4: Constant overhead of unoptimized, optimized GCs relative to a trivial memory management scheme scheme built on CUDA `malloc`, by benchmark duration. Lower scores are better. Points are placed in the same order as the bars in Figure 5.3 for easy comparison.

on the other hand, are barely affected. The write to pinned memory occurs both for the "Unoptimized GC" and "Optimized GC" configurations, explaining why their results almost always coincide.

6 Related work

This chapter briefly describes work that is related to the contents of this thesis. It discusses the state of the art in three main fields of research: Abstractions for GC-related information in compilers, allocators for GPU memory and managed language implementations that target GPUs.

6.1 Garbage collection abstractions

All managed language implementations have to shepherd GC-related information through their compilation pipelines. Of particular interest is how compiler IRs represent the fact that a pointer may refer to a GC-managed object.

- High-level IRs tend to have a rich type system and use that type system to identify potential GC-managed object references.

  For instance, Java and C# code compiles to bytecode suitable for execution on the Java Virtual Machine (JVM) and Common Language Runtime (CLR), respectively. Both of these virtual machines' bytecodes use the type system to represent GC-managed object references [29,74].

- Low-level IRs are typically designed for low-level languages. Point in case: The two most prominent compiler frameworks—GCC and LLVM—were both designed with C-like languages in mind [39,50].

  Managed languages increasingly target these low-level compilers, particularly LLVM. Julia is a prime example of such a language. Other LLVM-targeting managed languages include Haskell [73], D [51] and Swift [36].

  Broadly, these languages can use some combination of the following three strategies to support precise garbage collection despite targeting a low-level compiler:

  1. **Eagerly lower GC abstractions to runtime library calls.** This is what, e.g., Glasgow Haskell Compiler (GHC) does. The major downside to this

approach is that it may rather liberally insert calls to unknown functions, potentially hampering the low-level compiler's optimizer.

2. **Extend the low-level compiler with GC abstractions.** This approach has spawned two distinct categories of GC abstractions for LLVM: The legacy `gcroot`, `gcread` and `gcwrite` intrinsics and the newer statepoint intrinsics.

   Neither abstraction seems to enjoy widespread adoption: As of March 2016, only two projects—Azul's Zing/C4 and Microsoft's `llilc`—are known users of statepoints [64]. We are not aware of any recent additions to this list.

3. **Take control of the optimization pipeline.** This is the Julia compiler's preferred approach. Julia uses LLVM's pointer address spaces feature to encode GC pointers, then runs a sequence of optimizations and eventually lowers the address spaces to lower-level GC-supporting instructions [46].

6.2 Allocators for GPU memory

Faced with CUDA `malloc`'s rather disappointing performance, a number of novel allocators have been developed for GPUs. These allocators include Xmalloc [43], ScatterAlloc [70] and Halloc [5]. All three try to reduce allocation times by relying on lock-free data structures based on carefully-orchestrated atomic operations.

In contrast, Veldema and Philippsen's GPU GC uses a simple free list allocator, just like the GC presented in this thesis [76].

6.3 Managed language implementations for GPUs

Compiling managed languages for GPUs seems to be a popular pursuit. Oftentimes, only a restricted subset of the language can be compiled for GPUs. For instance, Copperhead is a Python module that compiles a restricted subset of Python code down to efficient GPU kernels [20]. In a similar vein, Fumero et al.'s work on heterogeneous array programming allows programmers to express use a restricted subset of Java for expressing stream computations that run on the GPU [33, 34].

These language subsets typically include low-level language features such as arithmetic and function calls, but tend to elide the features that typify managed languages: object allocation and exception handling.

Rootbeer is a notable exception: It compiles Java code to GPU kernels and supports almost all features of the Java programming language, including object allocation and exception handling [62]. Rootbeer is quite similar to CUDAnative in that sense: Both projects aim to compile arbitrary code for execution on GPUs. Like the work presented in this thesis, Rootbeer offers a trivial memory allocation scheme based on CUDA's `malloc`. Rootbeer never frees and eventually runs out of memory.

7 Conclusion

Over the course of this thesis, we first discussed the unique properties of GPUs, Julia and CUDAnative. We then moved on to the design and implementation of GC-related abstractions in the Julia compiler, identifying the need for an abstraction level that is sufficiently robust to support a GPU GC and as thin as possible. We subsequently detailed the first main contribution of this thesis: A lightweight, intrinsics-based abstraction for precise, non-moving, generational GCs.

Next, we discussed how that abstraction can be used to implement both trivial memory management and a full-blown GC for GPUs. We expounded on the design of the latter as it represents both the second main contribution of this thesis and a major step forward compared to the state of the art. To the best of our knowledge, our fully transparent implementation of garbage collection for GPU kernels is the first of its kind.

To ensure that the GPU GC meets functional and non-functional requirements, we thoroughly evaluated the Julia language features that now work thanks to the GC as well as the variable and constant overhead of enabling the GC for GPU kernels. The results are encouraging: Despite its relatively straightforward design, our GC actually manages to outperform CUDA `malloc` in terms of variable overhead, on average offering speedups of 2×.

7.1 Future work

The work presented in this thesis is designed to be a starting point for future innovation in addition to its aforementioned role of pushing the envelope of managed languages that target GPUs. We hence see a number of directions for future research.

Firstly, an implementation of the remaining runtime functions that support Julia arrays and similar GC-dependent constructs is bound to be useful from a functional viewpoint: Such an implementation would allow for ever more Julia code to be compiled to GPU kernels.

With regard to non-functional requirements, it would be interesting to see if GPU system calls can be implemented efficiently for NVIDIA devices using existing mechanisms. System calls could then replace the CPU-to-GPU callbacks that now underpin the flexible interrupt mechanism introduced in this thesis for the purpose of triggering collections.

Finally, a lock-free allocator that truly plays to the strengths of GPUs would be a more than worthwhile addition to the GC. It would also be interesting to see how such an allocator integrates with the arena composition scheme introduced in this work.

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[2] Abhinav, and Nasre, R. `FastCollect`: offloading generational garbage collection to integrated GPUs. In *International Conference on Compilers, Architectures, and Sythesis of Embedded Systems (CASES)* (2016), IEEE, pp. 1–10.

[3] Adinets, A. Adaptive Parallel Computation with CUDA Dynamic Parallelism. *NVIDIA Developer Blog,* `https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/` (May 2014). Accessed on June 5, 2019.

[4] Adinets, A. CUDA Dynamic Parallelism API and Principles. *NVIDIA Developer Blog,* `https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/` (May 2014). Accessed on June 5, 2019.

[5] Adinetz, A. V., and Pleiter, D. Halloc: a high-throughput dynamic memory allocator for GPGPU architectures. In *GPU Technology Conference (GTC)* (2014), vol. 152.

[6] Agesen, O. GC points in a threaded environment.

[7] Appel, A. W. Garbage collection can be faster than stack allocation. *Information Processing Letters 25*, 4 (1987), 275–279.

[8] Appel, A. W. Simple generational garbage collection and fast allocation. *Software: Practice and Experience 19*, 2 (1989), 171–183.

[9] Attanasio, C. R., Bacon, D. F., Cocchi, A., and Smith, S. A comparative evaluation of parallel garbage collector implementations. In *International Workshop on Languages and Compilers for Parallel Computing* (2001), Springer, pp. 177–192.

[10] Baker, J., Cunei, A., Kalibera, T., Pizlo, F., and Vitek, J. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience 21*, 12 (2009), 1572–1606.

[11] Baker Jr, H. G. List processing in real time on a serial computer. *Communications of the ACM 21*, 4 (1978), 280–294.

[12] Besard, T. Dynamic parallelism. `https://github.com/JuliaGPU/CUDAnative.jl/pull/362`, May 2019. Accessed on June 5, 2019.

[13] Besard, T., Churavy, V., Edelman, A., and De Sutter, B. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Software Engineering* (2019).

[14] Besard, T., and Foket, C. Benchmark suite for heterogeneous computing infrastructures. `https://github.com/JuliaParallel/rodinia`. Accessed on May 8, 2019.

[15] Besard, T., Foket, C., and De Sutter, B. Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems 30*, 4 (2019), 827–841.

[16] Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM review 59*, 1 (2017), 65–98.

[17] Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145* (2012).

[18] Blackburn, S. M., and McKinley, K. S. Ulterior reference counting: Fast garbage collection without a long wait. In *ACM SIGPLAN Notices* (2003), vol. 38, ACM, pp. 344–358.

[19] Boehm, H.-J., and Weiser, M. Garbage collection in an uncooperative environment. *Software: Practice and Experience 18*, 9 (1988), 807–820.

[20] Catanzaro, B., Garland, M., and Keutzer, K. Copperhead: compiling an embedded data parallel language. In *Symposium on Principles and practice of parallel programming (PPoPP'11)* (2011), pp. 47–56.

[21] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In

2009 IEEE international symposium on workload characterization (IISWC) (2009), Ieee, pp. 44–54.

[22] Chen, J., and Revels, J. Robust benchmarking in noisy environments. *arXiv preprint arXiv:1608.04295* (2016).

[23] Churavy, V., Wilcox, L. C., Kozdon, J. E., and Ramadhan, A. GPUifyLoops.jl: Support for writing loop-based code that executes both on CPU and GPU. `https://juliagpu.gitlab.io/GPUifyLoops.jl/`, March 2019. Accessed on May 31, 2019.

[24] Colburn, T., and Shute, G. Abstraction in computer science. *Minds and Machines 17*, 2 (2007), 169–184.

[25] Corporation, I. Intel® Core™ i7-6700K Processor. `https://ark.intel.com/content/www/us/en/ark/products/88195/intel-core-i7-6700k-processor-8m-cache-up-to-4-20-ghz.html`, 2015. Accessed on June 12, 2019.

[26] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*, 4 (1991), 451–490.

[27] Dawson, J. L. Improved effectiveness from a real time lisp garbage collector. In *Proceedings of the 1982 ACM symposium on LISP and functional programming* (1982), ACM, pp. 159–167.

[28] Durant, L., Giroux, O., Harris, M., and Stam, N. Inside Volta: The world's most advanced data center GPU. *NVIDIA Developer Blog, `https://devblogs.nvidia.com/inside-volta`* (May 2017). Accessed on May 24, 2019.

[29] Ecma International. *Common Language Infrastructure (CLI).* June 2012.

[30] Edelman, A. Julia: A fresh approach to parallel programming. In *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), IEEE, pp. 517–517.

[31] Fischer, K., Churavy, V., Hatherly, M., Hildebrand, M., Zhou, M., Karpinski, S., Besard, T., Yu, Y., and Arslan, A. Working with LLVM. `https://github.com/JuliaLang/julia/blob/master/doc/src/devdocs/llvm.md`, August 2018. Accessed on June 2, 2019.

[32] Fischer, K., Yu, Y., Besard, T., Nash, J., Arslan, A., Vettorel, D., Churavy, V., and Zhou, M. LLVM late GC lowering. https://github.com/JuliaLang/julia/blob/master/src/llvm-late-gc-lowering.cpp, May 2019. Accessed on June 2, 2019.

[33] Fumero, J. J., Remmelg, T., Steuwer, M., and Dubach, C. Runtime Code Generation and Data Management for Heterogeneous Computing in Java. In *Proceedings of the Principles and Practices of Programming on The Java Platform - PPPJ '15* (New York, New York, USA, 2015), ACM Press, pp. 16–26.

[34] Fumero, J. J., Steuwer, M., and Dubach, C. A Composable Array Function Interface for Heterogeneous Computing in Java. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY'14* (2014), pp. 44–49.

[35] Fung, W. W., and Aamodt, T. M. Thread block compaction for efficient SIMT control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture* (2011), IEEE, pp. 25–36.

[36] García, C. G., Espada, J. P., Bustelo, B. C. P. G., and Lovelle, J. M. C. Swift vs. Objective-C: A new programming language. *IJIMAI 3*, 3 (2015), 74–81.

[37] GCC developers. Using the GNU Compiler Collection (GCC): Labels as Values. https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html. Accessed on June 4, 2019.

[38] Govett, M. W., Middlecoff, J., and Henderson, T. Running the NIM next-generation weather model on GPUs. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010), IEEE Computer Society, pp. 792–796.

[39] Griffith, A. *GCC: the complete reference*. McGraw-Hill, Inc., 2002.

[40] Harris, M. Unified Memory for CUDA Beginners. *NVIDIA Developer Blog, https://devblogs.nvidia.com/unified-memory-cuda-beginners/* (June 2017). Accessed on June 4, 2019.

[41] Henderson, F. Accurate garbage collection in an uncooperative environment. In *ACM SIGPLAN Notices* (2002), vol. 38, ACM, pp. 150–156.

[42] Homm, F., Kaempchen, N., Ota, J., and Burschka, D. Efficient occupancy grid computation on the gpu with lidar and radar for road boundary detection. In *2010 IEEE Intelligent Vehicles Symposium* (2010), IEEE, pp. 1006–1013.

[43] Huang, X., Rodrigues, C. I., Jones, S., Buck, I., and Hwu, W.-m. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology* (2010), IEEE, pp. 1134–1139.

[44] Introducing Julia contributors. Introducing Julia: Arrays and tuples. `https://en.wikibooks.org/wiki/Introducing_Julia/Arrays_and_tuples`, February 2019. Accessed on June 11, 2019.

[45] Julia contributors. Julia micro-benchmarks. `https://julialang.org/benchmarks/`. Accessed on April 23, 2019.

[46] Julia contributors. Working with LLVM. `https://docs.julialang.org/en/v1/devdocs/llvm/index.html#Representation-1`, August 2018. Accessed on November 2, 2018.

[47] Kiczales, G. Towards a new model of abstraction in software engineering. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems* (1991), IEEE, pp. 127–128.

[48] Kramer, J. Is abstraction the key to computing? *Communications of the ACM 50*, 4 (2007), 36–42.

[49] Köplinger, A., Turaev, M., and Hasitzka, A. Generational GC. `https://www.mono-project.com/docs/advanced/garbage-collector/sgen/`, February 2019. Accessed on June 1, 2019.

[50] Lattner, C., and Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (2004), IEEE Computer Society, p. 75.

[51] LDC Developers. LLVM-based D Compiler. `https://github.com/ldc-developers/ldc`. Accessed on May 8, 2019.

[52] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., et al. Debunking the

100× GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH computer architecture news 38*, 3 (2010), 451–460.

[53] LLVM PROJECT. LLVM language reference manual. `http://llvm.org/docs/LangRef.html`, April 2019. Accessed on May 1, 2019.

[54] MAAS, M., REAMES, P., MORLAN, J., ASANOVIĆ, K., JOSEPH, A. D., AND KUBIATOWICZ, J. GPUs as an opportunity for offloading garbage collection. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 25–36.

[55] NICKOLLS, J., BUCK, I., AND GARLAND, M. Scalable parallel programming. In *2008 IEEE Hot Chips 20 Symposium (HCS)* (2008), IEEE, pp. 40–53.

[56] NUSEIBEH, B., AND EASTERBROOK, S. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (2000), ACM, pp. 35–46.

[57] NVIDIA. *CUDA C programming guide v10.1.* Nvidia Corporation, March 2019.

[58] NVIDIA. *Parallel Thread Execution ISA v6.4.* Nvidia Corporation, March 2019.

[59] NVIDIA CORPORATION. GeForce GTX 970: Specifications. `https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-970/specifications`, 2014. Accessed on June 12, 2019.

[60] OWENS, J. D., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J. E., AND PHILLIPS, J. C. GPU computing. *Proceedings of the IEEE 96*, 5 (2008).

[61] POHL, K. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

[62] PRATT-SZELIGA, P. C., FAWCETT, J. W., AND WELCH, R. D. Rootbeer: Seamlessly using GPUs from Java. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on* (2012), IEEE, pp. 375–380.

[63] PRATX, G., AND XING, L. GPU computing in medical physics: a review. *Medical physics 38*, 5 (2011), 2685–2697.

[64] REAMES, P. Status of Garbage Collection with Statepoints in LLVM. `http://lists.llvm.org/pipermail/llvm-dev/2016-March/096360.html`, March 2016. Accessed on November 2, 2018.

[65] Revels, J., Arslan, A., Ahrens, P., Adams, L., Herriman, J., Goerz, M., Johnson, S. G., and Mauro. BenchmarkTools manual. `https://github.com/JuliaCI/BenchmarkTools.jl/blob/master/doc/manual.md`, November 2018. Accessed on June 10, 2019.

[66] Rhodin, H. A PTX code generator for LLVM. Bachelor's thesis, Saarland University, Saarbrücken, Germany, October 2010.

[67] Romero, M., and Urra, R. CUDA Overview. `http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm`. Accessed on May 27, 2019.

[68] Sanders, J., and Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[69] Sobalvarro, P. A Lifetime-Based Garbage Collector for LISP Systems on General-Purpose Computers. Tech. rep., Massachusetts Institute of Technology, Cambridge Artificial Intelligence Lab, 1988.

[70] Steinberger, M., Kenzel, M., Kainz, B., and Schmalstieg, D. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *2012 Innovative Parallel Computing (InPar)* (2012), IEEE, pp. 1–10.

[71] Stuart, J. A., Cox, M., and Owens, J. D. GPU-to-CPU callbacks. In *European Conference on Parallel Processing* (2010), Springer, pp. 365–372.

[72] TachyonicClock42, and Besard, T. CUDA debugger API. `https://devtalk.nvidia.com/default/topic/1030648/cuda-programming-and-performance/cuda-debugger-api/`, March 2019. Accessed on June 5, 2019.

[73] Terei, D. A., and Chakravarty, M. M. An LLVM backend for GHC. In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 109–120.

[74] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers* (2010), IBM Corp., pp. 214–224.

[75] Van Lamsweerde, A. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering* (2001), IEEE, pp. 249–262.

[76] Veldema, R., and Philippsen, M. Iterative data-parallel mark&sweep on a GPU. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 1–10.

[77] Verroken, H. Contextual language abstractions for low-level GPGPU programming in Julia. Master's thesis, Ghent University, June 2018.

[78] Veselỳ, J., Basu, A., Bhattacharjee, A., Loh, G., Oskin, M., and Reinhardt, S. K. GPU System Calls. *arXiv preprint arXiv:1705.06965* (2017).

[79] Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M., and Shringarpure, A. On the limits of GPU acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism* (2010), vol. 13.

[80] Wajnerman, J. Garbage collector. `https://crystal-lang.org/2013/12/05/garbage-collector.html`, December 2013. Accessed on June 1, 2019.

[81] Wenzel, M., Turn, N., Schonning, N., Mabee, D., Petrusha, R., B, M., Kotas, J., A, A., xaviex, Jones, M., Ciechan, M., Alan, Latham, L., and tompratt AQ. Fundamentals of garbage collection. `https://github.com/dotnet/docs/blob/master/docs/standard/garbage-collection/fundamentals.md`, May 2019. Accessed on June 1, 2019.

[82] Wilson, P. R. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management* (1992), Springer, pp. 1–42.

[83] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management* (1995), Springer, pp. 1–116.

[84] Wu, X., Koslowski, A., and Thiel, W. Semiempirical quantum chemical calculations accelerated on a hybrid multicore CPU–GPU computing platform. *Journal of chemical theory and computation 8*, 7 (2012), 2272–2281.

[85] Zhang, J., You, S., and Gruenwald, L. Indexing large-scale raster geospatial data using massively parallel GPGPU computing. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2010), ACM, pp. 450–453.