GPUs as an Opportunity for Offloading Garbage Collection [*]

Martin Maas, Philip Reames, Jeffrey Morlan, Krste Asanović, Anthony D. Joseph, John Kubiatowicz
University of California, Berkeley
{maas,reames,jmorlan,krste,adj,kubitron}@cs.berkeley.edu

Abstract

GPUs have become part of most commodity systems. Nonetheless, they are often underutilized when not executing graphics-intensive or special-purpose numerical computations, which are rare in consumer workloads. Emerging architectures, such as integrated CPU/GPU combinations, may create an opportunity to utilize these otherwise unused cycles for offloading traditional systems tasks. Garbage collection appears to be a particularly promising candidate for offloading, due to the popularity of managed languages on consumer devices.

We investigate the challenges for offloading garbage collection to a GPU, by examining the performance trade-offs for the mark phase of a mark & sweep garbage collector. We present a theoretical analysis and an algorithm that demonstrates the feasibility of this approach. We also discuss a number of algorithmic design trade-offs required to leverage the strengths and capabilities of the GPU hardware. Our algorithm has been integrated into the Jikes RVM and we present promising performance results.

Categories and Subject Descriptors D.3.4 [*Processors*]: Memory management (garbage collection); I.3.1 [*Hardware Architecture*]: Parallel processing

General Terms Design, Experimentation, Languages, Performance, Algorithms

Keywords parallel garbage collection, mark and sweep, SIMT, GPU, APU

1. Introduction

Graphics Processing Units (GPUs) have been part of commodity systems for more than a decade. While frameworks such as CUDA and OpenCL have enabled GPUs to run general-purpose workloads, their additional compute power is rarely utilized other than by graphics-intensive applications (e.g. games), special-purpose computations (e.g. image processing) or scientific simulations (e.g. fluid dynamics). In the absence of such workloads, the GPU is underutilized on most systems.

Today, we are seeing the first chips that integrate CPUs and GPUs into a single device. These changes open up a whole new set of application scenarios since they eliminate the copying overhead that is traditionally associated with moving data between the CPU and a dedicated GPU. We expect future hardware to move even further in this direction by providing a shared address space and possibly cache-coherence between CPU and GPU.

This development entails an opportunity to move traditional systems workloads to the GPU. Garbage collection appears to be a particularly good candidate for this, since garbage-collected languages such as C# and Java account for a significant portion of code running on consumer devices. Offloading their GC workloads to the GPU allows us to harvest the GPU's unused compute power, leaving the CPU free to perform other tasks such as JIT compilation, garbage collection for other memory spaces, or running mutator threads (if the GPU is used by a concurrent garbage collector).

Garbage collection is a workload that is arguably well-suited for running on the GPU, especially once the copy overhead between CPU and GPU disappears. Graph traversals (a key component of many garbage collectors) have already been efficiently demonstrated on GPUs [12].

Recent work by Veldema and Philippsen [21] has shown that garbage collection for GPU programs can be efficiently performed on the GPU itself. We take the next step and ask whether it is feasible to offload garbage collection workloads from conventional programs running on the CPU, and what it takes to achieve this goal. In this, we explore a different direction in choice of algorithm, as well as design trade-offs that are specific to our scenario. Garbage collection on the GPU is a challenging problem due to the GPU's SIMD-style programming model, and the need to design algorithms that make explicit use of available parallelism and memory bandwidth, while avoiding serialization of execution. The contributions of our work are as follows:

- We present an analysis of the heap graphs of several Java benchmarks, to evaluate the theoretic feasibility of using the GPU for garbage collection (Section 3).
- We prototype a GPU-based garbage collector integrated into the *Jikes Research Virtual Machine* [1] (Section 4), a Java VM maintained by the research community.
- We show a new algorithm, and variations thereof, for performing the mark phase of a mark & sweep garbage collector on the GPU. Our algorithm differs from previous work by using a frontier queue approach instead of a data-parallel algorithm. We also discuss trade-offs and optimizations to make it efficient on a GPU.

The objective of this work is not to present a single tuned implementation. The implementation presented in Section 4 is mainly for the purpose of illustrating a particular point in the design space. Our

[*] Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. Philip Reames was also supported by the National Science Foundation (Award #CCF-1017810).

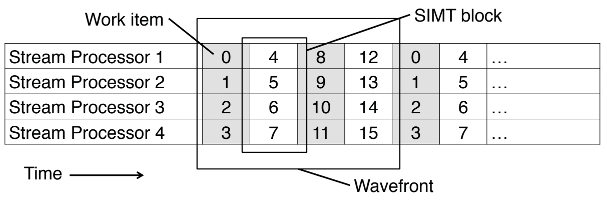


Figure 1. Scheduling of work-items for a fictional compute unit with 4 stream processors and a wavefront size of 16.

main goal is to assess whether using the GPU to offload garbage collection is feasible and to identify obstacles that need to be overcome. The GPU space is evolving rapidly and a compelling application workload such as garbage collection might influence the direction of that evolution.

After giving a brief introduction to the GPU hardware and programming model, the first part of our work (Section 3) consists of analyzing a selection of program heap graphs from the DaCapo benchmark suite [6] to determine whether heaps of real-world applications exhibit sufficient regular parallelism to take advantage of the GPU. We then present the implementation of our GPU-based collector (Section 4). This is followed by a discussion of design choices for our mark algorithm (Section 5), and experimental results (Section 6). We close with a discussion of the implications of our work, on current and future hardware (Section 7).

2. GPU Programming Model

This section provides a general introduction to the hardware and programming model of a GPU. Note that throughout this paper, we use the terminology from *OpenCL*; the terminology used by *CUDA* (the other major framework) is synonymous.[1]

GPUs provide a SIMT (*Single Instruction Multiple Thread*) programming model. SIMT is an extension of SIMD (Single Instruction Multiple Data) with support for hardware execution masking to handle divergent control paths within an instruction block. Computation is described in terms of a program *kernel* which is executed by a number of *work-items* (a.k.a. threads).

The basic building block of a GPU is a *streaming multiprocessor* (SM), or *compute unit*, which contains a single instruction decoder and a number – typically between 8 and 64 – of *stream processors* (SP). The stream processors execute the same instruction in lockstep, but with different register contexts (each of them stores the registers and a small amount of memory for each of its work-items). Within the compute unit, stream processors share access to a *Local Data Store*, a small fast block of dedicated memory.

Work-items are grouped into *wavefronts* (Figure 1); each wavefront typically contains four times the number of stream processors. The work-items of the wavefront are interwoven such that each stream processor executes the same instruction four times – once for each quarter of the wavefront. To handle divergence of control flow within a wavefront (e.g. one work-item takes a branch while another work-item does not), the hardware will perform masked execution. Both sides of the branch will be executed, but only some of the work-items will be enabled. For good performance, it is critically important to minimize the amount of divergence.

Wavefronts are in turn grouped into *workgroups*; 256 to 1,024 work-items per workgroup are common. When a given wavefront stalls because of memory access, another ready wavefront begins executing. Context switches between wavefronts are extremely fast

[1] For further reading, the AMD OpenCL Programming Guide [3] and the OpenCL v1.2 Specification [15] provide all the detail one might require.

(usually a single cycle) since each work-item in the entire workgroup retains its dedicated registers at all times. To maximize memory bandwidth, a kernel should maximize the number of wavefronts that are able to perform independent memory accesses.

GPUs have a number of compute units which share access to a memory region known as *global memory*. On today's devices, the number of compute units varies from 2-4 on a low-end device to as many as 12-40 in high-end devices. For discrete GPUs (such as graphics cards), global memory is dedicated hardware on the device; for integrated GPUs (where CPU and GPU share the same package), it will often be a reserved area of the main system memory. The discrete approach has the advantage of much faster access times, but requires slow (on the order of 8GB/s) explicit copies between CPU and GPU, using DMA over the PCIe bus. In current-generation parts, neither approach participates in the cache-coherency protocol of the CPU; this means that communication between CPU and GPU must be done explicitly through the OpenCL interface.

3. Preliminary Analysis

Garbage collection in general – and the mark phase of a mark & sweep garbage collector specifically – is a memory-bound problem. As such, the main challenge of any implementation is to process and issue memory requests at a sufficiently high rate to fully utilize the available memory bandwidth. As we will discuss more in Section 5, being able to process a large number (i.e. hundreds) of objects in parallel is essential for meeting this goal on a GPU.

The core of our mark algorithm is a highly parallel queue-based breadth-first search. Objects to be processed are added to a *frontier queue*. For each item in the queue, we take it off, mark the object, and then add each outbound reference to the queue. This processes objects in order of increasing distance from the root set (i.e. increasing *depth*). At each depth, there is a fixed *width* (or *beam*) of nodes available for processing. If this available width is greater than the number of work-items, we can keep the entire device busy and make efficient progress through the traversal.

Any practical collector can do no better than an ideal collector which examines every object at a given depth in a single iteration. To understand this theoretical best case garbage collector on real programs, we examined the heap structure of benchmarks from the DaCapo 9.12 [6] benchmark suite. We examine two attributes of heap graphs: their general shape (i.e. depth, width per iteration, etc.) and the distribution of outbound references across objects. The latter has a significant performance impact on GPUs due to the divergence problem described previously (Section 2).

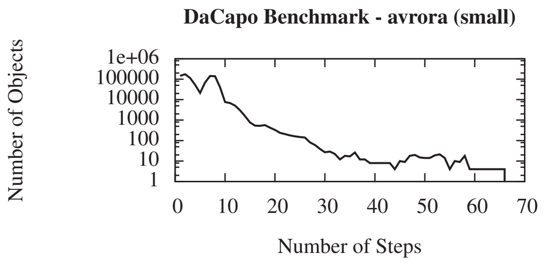
The data collection for this section was performed using an instrumented Jikes garbage collection plan. (We also repeated the analysis using a plugin for the Oracle HotSpot VM, but do not report the results since they were similar). All collection was done using the optimizing compiler; optimization affects the frequency of collection and thus the heap graphs' shapes. We ran the small and default configurations of a subset of the benchmarks.[2]

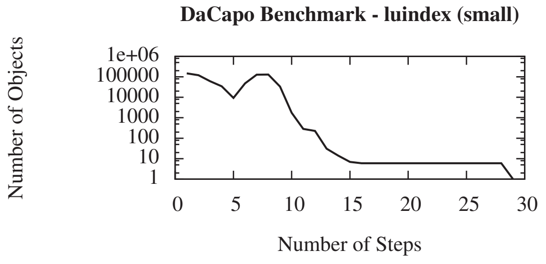
For further details about the structure of common Java heaps, we recommend Barabash and Petrank's paper [5]. They analyze heap depth and approximate shape for a previous release of the DaCapo benchmarks, as well as several Java SPEC benchmarks.

3.1 Structural Limits on Parallelism

We first examined the general shape of the heap graphs as traversed by the ideal collector. We were interested in determining whether

[2] We are only reporting a subset of the benchmark suite since several benchmarks did not work on a vanilla Jikes RVM running on our evaluation system. This is a known problem and unrelated to our garbage collector.





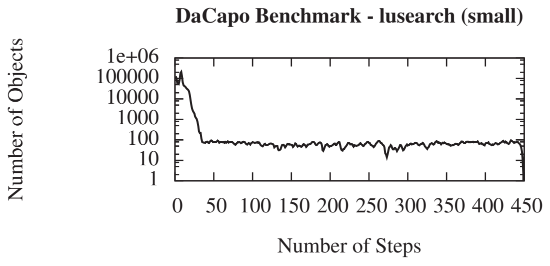


Figure 2. Number of objects at each depth during an idealized breadth-first traversal starting from the root set. The figures were chosen to exemplify classification by degree of parallelism.

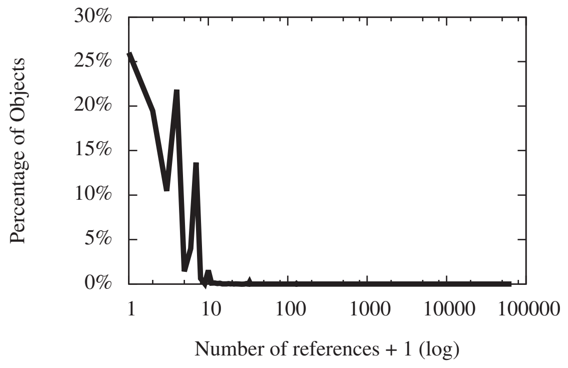


Figure 3. Distribution of the number of references within objects on the heap. The majority of objects have few outbound references, but some rare objects have hundreds or thousands.

there were structural limits that would prevent the degree of parallel processing that the GPU requires for efficiency.

A selection of the graphs generated from the DaCapo benchmarks is shown in Figure 2; these graphs were picked to be representative of the three categories of graph shape we identified. The figures show the number of objects reachable – marked or unmarked – from a given step of the ideal breadth-first traversal starting at the root set. For presentation purposes, we picked the deepest traversal found within a couple of runs of each benchmark, as these are likely to be the least advantageous for the GPU.

All of the benchmarks begin with a short section of extreme parallelism. The first step is limited to the size of the root set (typically 600-1,000 objects), but the next few steps expand rapidly. Once this startup section is completed, our benchmarks fell into three categories. Some of the benchmarks – such as `luindex` – then complete within a small number of additional steps. A few – such as `avrora` – had moderate length sections of structurally limited parallelism. Unfortunately, there were also a few benchmarks – such as `lusearch` – which had long narrow sections ("tails") following the parallel beginning. A width of 30 to 80 represents at most 1/3 of the available parallelism on the GPU. As the number of available work-items per workgroup increases with time, this fraction may drop precipitously.

Since no hardware can execute an infinite number of threads, we repeated the analysis above with maximum step widths of 128, 256, 1,024, and 32,768. As expected, decreasing the number of items processed in each iteration increased the effective depth of the graph, but did not change the overall shape or categorization of any of the benchmarks.

Despite the limited parallelism towards the end of some collections, we conclude that heap graphs are sufficiently parallel for the purposes of garbage collection on a GPU. However, if one wants to minimize collection latency, having a mechanism to deal efficiently with long narrow tails in the heap graph is critical; we discuss our solution in Section 5.4. An alternative approach would be to insert artificial shortcut edges into the heap graph. Barabash and Petrank describe this strategy in detail [5].

3.2 Distribution of Outbound References

Prompted by Veldema and Philippsen's [21] findings, the second issue we examined was the distribution of the number of outbound references within each object. When processing one object per thread in a SIMT environment where each thread loops over the outbound references within its object, this distribution is critical to understanding and controlling divergence.

Our findings show that while the vast majority of objects have a small out-degree – 26% of objects have no outbound references (other than their class pointer), 76% have four or fewer, and 98% have 12 or fewer – the distribution has a very long and noisy tail. A small fraction of objects (less than 0.01%) have hundreds to thousands of outbound references. It is worth noting that our analysis does not distinguish between objects and arrays of references; we have manually confirmed that some of the high double-digit out-degree nodes are, in fact, objects.

The distribution of the number of references within objects can be seen in Figure 3. Given that the results across benchmarks are fairly uniform, we chose to present the distribution across all the collections of all the benchmarks for which we collected results.

Even leaving aside the extreme tail of the distribution, the distribution of references between objects means that blindly looping over the number of references will result in unacceptable divergence of threads. We discuss one solution for distributing references between work-items in Section 5.2.

4. Offloading Garbage Collection

In this section, we present challenges for offloading garbage collection to the GPU and discuss as well as measure different performance trade-offs. To substantiate our claims, we implemented a proof-of-concept GPU-based garbage collector for the *Jikes Research Virtual Machine* [1] (a Java Virtual Machine maintained by

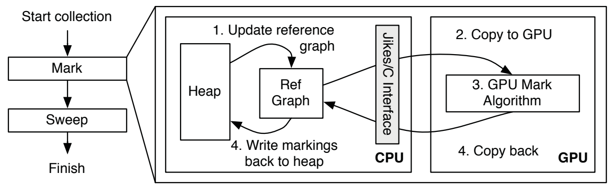


Figure 4. Overview of the collector integration.

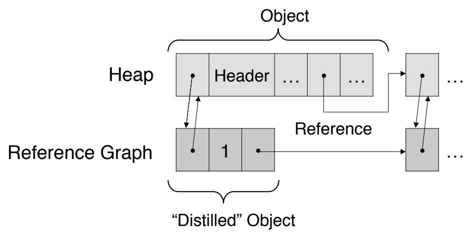


Figure 5. The reference graph structure.

the research community). This allows us to investigate performance trade-offs for full executions of real Java programs, by performing a series of macro and micro benchmarks.

4.1 High-level Overview

We modified Jikes' `MarkSweep` garbage collector to offload its mark phase to the GPU. The steps performed by the collector are shown in Figure 4. Our mark phase is performed on a *reference graph* data structure, a self-contained version of the heap that only contains references but no other fields (an implementation detail which we discuss in the next sections). This structure is kept up-to-date during program execution or filled in on each collection. We then invoke our collector in a native call which sets up our mark kernel and runs it on the GPU. The CPU is idle until the mark completes (a production-grade implementation would perform other tasks during this time). Upon completion, execution returns to Jikes and the markings are transferred back into the heap. The sweep phase is then performed by Jikes. Note that the intermediary copying steps are merely an implementation detail of our prototype, and not inherent in our approach.

While a real-world collector will have to offload the sweep phase as well, we think that Veldema et al. have sufficiently covered this aspect in their work, so we focus on the mark phase for brevity. A complete collector would perform the sweep phase on the GPU (immediately after the mark algorithm) and only copy the resulting free lists back to the Jikes RVM.

4.2 Object Layout

The difficulty of integrating our collector into Jikes was exacerbated by the VM's object layout. To identify the reference fields within each object, it is necessary to look up their offsets in an array that is stored with the type information of the current class. This adds up to three levels of indirection (type information block, type info, offset array), and incurs a significant performance penalty on the GPU, due to the lack of caching. We therefore require a different object layout, which lays out the references of an object consecutively and contains the object's number of references in the object header. The layout introduced by the Sable VM [8] achieves this requirement with minimal overhead [9] by laying out reference fields to the right of the object header, and non-reference fields to the left.

4.3 Reference Graph

While high-end GPUs may provide up to 3 GB of global memory, today's consumer GPUs often only have 256-512 MB. Even though our test platform (see Section 6.1) can access the system's main memory, it can only map 128 MB at a time.[3] This became problematic, as this size was too small to hold the heaps of several DaCapo benchmarks (when including Jikes' memory spaces).

For the purposes of evaluation, we solved this problem by building a condensed version of the heap which we call a *reference graph*. The reference graph is stored in a separate space and contains an entry for each object on the main heap, consisting of a pointer to the original object, the number of references, and a consecutive list of all outbound references as pointers into the reference graph (Figure 5). Arrays are represented in the same way. This emulates the object layout presented in Section 4.2, but reduces the size of the heap such that it fits on the GPU. Due to the lack of caching on the GPU, this approach does not give a performance advantage, while it allows us to evaluate our collector on real-world heaps that otherwise would have been too large to fit on the GPU.

We found that the reference graph approach gave us a sufficient reduction in size to evaluate the DaCapo benchmarks on our collector. The following table shows the cumulative sum of heap sizes across all collections within a run, as well as the equivalent numbers for the reference graph.[4] This allows us to estimate that the reference graph approach reduces the size of our graph by about 75% on average:

	# GCs	Cum. Heap	Cum. Graph	Ratio
avrora	9	256 MB	80 MB	31.2%
jython	114	10499 MB	3301 MB	31.4%
luindex	7	178 MB	35 MB	19.8%
lusearch	77	7078 MB	515 MB	7.3%
pmd	14	809 MB	233 MB	28.9%
sunflow	39	2935 MB	658 MB	22.4%
xalan	23	1686 MB	456 MB	27.1%

We experimented with two different approaches to building and maintaining the reference graph. Both of them allocate a node in the reference graph whenever a new object is allocated.

- The most basic approach fills in the reference graph immediately before performing a collection. It performs a linear scan through the distilled objects in the reference graph, follows the references of each corresponding original object, and copies the pointers for the corresponding distilled objects to the reference graph (Figure 5).
- The reference graph can also be built while running the mutator threads: this turns every reference write into a double-write to two locations, which can be implemented as either a write barrier or issuing a second write instruction in the compiler. We prototyped the simpler write barrier approach.

While these approaches obviously differ in performance, we refrain from performing a deeper analysis, as this problem is somewhat orthogonal to our approach and current hardware trends are indicating that the memory available to the GPU will soon be large enough to store the entire heap.

4.4 Launch Overhead

Equipped with the reference graph, our collector calls into a C function which initializes OpenCL, copies (or maps) the reference graph to the GPU and launches the mark kernel. Launching a kernel execution incurs both a fixed startup cost, and a variable cost

[3] This value was determined experimentally.

[4] We used a heap size of 100 MB for all of these runs.

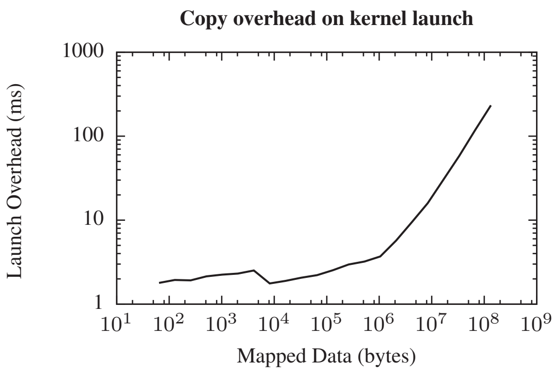


Figure 6. Kernel launch latency presented as a function of the amount of data being mapped onto the GPU (in ms).

related to the kernel itself and the size of memory being mapped to the GPU. We incur these costs once per garbage collection. To analyze their impact, we measured the launch latencies of two microbenchmarks: an empty kernel, and our mark algorithm (Section 5) when run on a single object. Each experiment was repeated 20 times and we present the average latencies.

- The results from the first microbenchmark show that the first invocation of a kernel incurs an overhead (around 1 ms) which is not repeated for successive runs. We believe this to be the time to copy the kernel itself. For the remaining runs, the overhead hovers between 0.33 ms and 0.62 ms. The execution time of the empty kernel was always below 0.01 ms (i.e. negligible).
- The results from the second benchmark show that overhead scales roughly linearly with mapped memory for sizes above 2 MB (Figure 6). Below that threshold, the overhead is dominated by the fixed cost. We want to note that there is substantial variation in results, particularly for larger sizes. The range of times for 128 MB was from 181 ms to 292 ms.

While our test platform supports zero-copy mapping between CPU and GPU, the device drivers available for Linux do not currently support this feature. The Windows drivers do, but we chose to run our experiments on Linux for consistency with the rest of our results. We therefore incur a copy-overhead, which can be seen in the linear scaling for large memory sizes; this resembles the behavior on traditional (discrete) GPU architectures. We argue that the majority of this overhead will go away once zero-copy mapping is supported (except for some cache write-back costs).

The remaining launch time can be largely discounted for purposes of assessing validity, as long as the number of kernel launches is small. Launch times have been trending downward at a steep rate and we expect them to decrease further, since this is clearly a general problem for many GPU workloads. For this reason, we exclude launch overheads from the execution times of our kernels.

4.5 Copy-back Overhead

After performing the mark phase on the GPU, our collector incurs an additional overhead from copying the marked reference graph back into main memory and transcribing the mark bits into the original heap structure. This is necessary for the integration with Jikes, but would not occur in a real-world collector that integrates both mark and sweep phase: after finishing the sweep phase on the GPU, the collector would simply move the resulting free-list to the CPU, ideally in a low-overhead, zero-copy operation. For this reason, we ignore this overhead for the purpose of our evaluation.

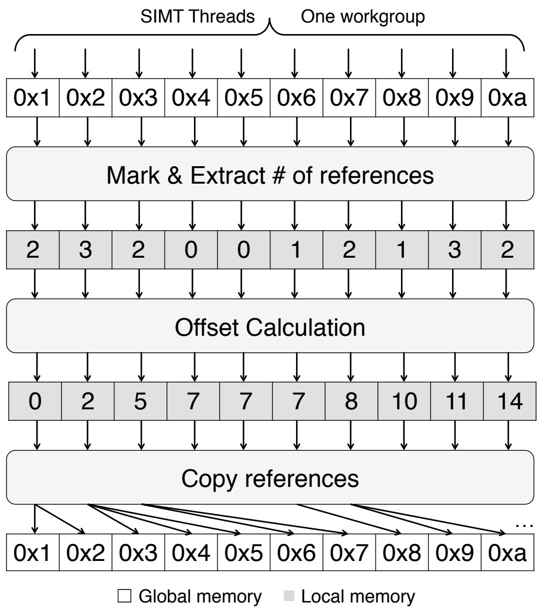


Figure 7. Structure of the baseline algorithm.

5. Algorithm and Optimizations

The core part of our collector consists of an algorithm that performs a parallel mark phase on the GPU, using n work-items (on our platform, $n = 256$). Our approach is based on maintaining a *frontier queue* that contains pointers to objects to be processed; we do not differentiate between arrays and objects. The kernel processes these elements in a loop: at each iteration, it removes up to n pointers from the queue, marks them, and adds the address of any referenced objects to the end.

Veldema and Philippsen [21] identified synchronization as a core problem of such an approach: in an implementation where each work-item accesses the queue in an atomic operation, execution would be serialized and therefore very inefficient. Based on this observation, they discard the queue-based approach and instead show a data-parallel implementation that flips to the CPU after every iteration, to spawn a new set of work-items.

We avoid the problem of serialization by calculating in on-chip memory the total number of elements to remove and add to the queue, as well as their offsets. This avoids the need for per work-item atomic operations on the critical regions of the queue. At the same time, it avoids flipping between CPU and GPU, since we found the associated launch overhead to be too significant for this approach (Section 4.4).

Our algorithm is implemented as an OpenCL kernel which executes the code in Algorithm 1 (discussed below) in a loop until the frontier queue is empty. For the purposes of this explanation, assume that in_queue and out_queue are pointers to the parts of the queue where we are extracting elements from and where we store new elements, respectively. On each iteration, we remove up to n pointers from in_queue and examine the corresponding objects in parallel. We then mark all objects that have not been marked before and copy their references to the end of out_queue. This is done in three steps (Figure 7):

1. For all objects, read the number of references and whether the object has been marked. Objects that have already been marked are treated like an object with zero references.

2. Compute the offsets that the references of each object will have in *out_queue*, using either a prefix sum or histogram (discussed in Section 5.1). For the ease of exposition, assume the prefix sum approach for now, which lays out the references of an object consecutively, one object after another.
3. Copy all references within the objects to their new location in the frontier, using the previously calculated offsets to determine where to store the first reference of each object.

Only between iterations do we update the queue's start- and end-offsets. This can be done by a single work-item per workgroup, since all work-items know the number of elements that are removed from the queue (l below) and the number of elements that are added to the queue (which is given by the offset calculation – e.g. the right-most entry of the prefix-sum).

The following paragraphs give a more detailed description of the algorithm that is executed by each work-item. Note that id is the offset of each individual work-item within the workgroup.

Algorithm 1 One step of the GPU mark phase.

```
function MARK_PHASE ($id$, $in\_queue$, $out\_queue$)
 1: $x\,[work\_group\_size] \leftarrow (0,\ldots,0)$
 2: $l \leftarrow \min(length(in\_queue), work\_group\_size)$
 3: if $id \geq l$ then return
 4:
 5: $header \leftarrow mark(in\_queue\,[id]\,, mark\_bit)$
 6: if $\neg marked(header)$ then
 7:     $refcount \leftarrow ref\_count(header)$
 8: else
 9:     $refcount \leftarrow 0$
10: end if
11: $x\,[id] \leftarrow refcount$
12:
13: $offset \leftarrow compute\_offsets(id, x, l)$
14:
15: for $i = 0$ to $refcount - 1$ do
16:     $refptr \leftarrow in\_queue\,[id] + i + HEADER\_SIZE$
17:     $out\_queue\,[offset + i] \leftarrow *refptr$
18: end for
```

Lines 1-3 set up the necessary data structures and drop out of the function if there is no work to do for the work-item. Notably, x is allocated in the local scratchpad memory and used to efficiently calculate the offsets into the output queue.

Lines 5-11 implement the first part of the algorithm. It retrieves the object's header, in order to extract the marking and the reference count. It then stores the reference count in x.

Line 13 calculates the offsets for writing into the output queue. We implemented several options, which are discussed in Section 5.1. The presented algorithm uses a simple prefix-sum to determine the offset for each object in the output queue, and stores the object's references in consecutive slots after this offset. The next section discusses this aspect in detail.

Lines 15-18 describe the last part of the algorithm. The references are copied one-by-one into their dedicated locations in the output queue. The output calculation in the previous step ensures that no two references are written into the same slot, avoiding the requirement for synchronization or locking. In the given code, the fixed constant *HEADER_SIZE* represents the offset of the first reference from the beginning of the object header.

It is important to note that the *mark* operation does not need to use an atomic operation. Setting the mark bit is an idempotent operation and there is no correctness concern if a single object is processed multiple times. The slight performance loss due to redundant work – if an unmarked object gets added to the frontier multiple times and processed within the same iteration – is vastly outweighed by the cost of atomic operations.[5].

The described version of the algorithm performs no coordination between workgroups and can thus only exploit one compute unit per device. In Section 7.4 we expand on it and discuss load balancing and synchronization concerns in detail. We present a naïve proof-of-concept solution in Section 5.5.

5.1 Offset Calculation

Our first strategy for calculating the offsets for the output queue used Blelloch's prefix-sum algorithm from [11]. With this approach, all references of an object are stored in consecutive slots in the queue, and the offset of an object's first reference is the total number of references from work-items with a lower *id* than the one processing that object (Figure 8). When performed in local memory, the complexity of this approach is $O(\log n)$ parallel addition and local memory operations.

However, we discovered that this approach often takes up 40-50% of the kernel's total execution time, arguably due to the large number of accesses to local memory. We therefore implemented a different approach, based on a histogram. In this layout, the first references from all work-items with at least one reference appear first, followed by the second references, third references, etc.

Offsets for this layout are calculated by generating a histogram that counts the number of work-items that have at least one reference, at least two references, etc. The histogram is generated using atomic operations in local memory, by atomically counting the number of work-items with an object having $i = 1, 2, \ldots$ references or more. The sum of the first $i - 1$ entries of the histogram gives the global offset of the part of the output array where the i'th references begin (Figure 8). The atomic counting operation also gives each work-item a unique local offset into the i'th part, where it will write its reference. The following code replaces the last part of Algorithm 1, starting from line 13.

Algorithm 2 Histogram approach for the offset calculation.

```
 1: $hist \leftarrow (0,\ldots,0)$
 2: $global\_offset \leftarrow 0$
 3: $atomic\_max(\&max\_refcount, refcount)$
 4: for $i = 0$ to $max\_refcount - 1$ do
 5:     if $i < refcount$ then
 6:         $local\_offset \leftarrow atomic\_increment(\&hist\,[i]\,, 1)$
 7:         $refptr \leftarrow in\_queue\,[id] + i + HEADER\_SIZE$
 8:         $out\_queue\,[global\_offset + local\_offset] \leftarrow *refptr$
 9:     end if
10:     (memory barrier)
11:     $global\_offset {+}{=} hist\,[i]$
12: end for
```

This uses $O(max\_refcount)$ parallel atomic operations, where *max_refcount* is the maximum number of references among the objects currently processed by any of the work-items. Since the atomics are executed in local memory, they do not slow down global memory access and are comparatively fast. An additional advantage of this approach is that we are writing to consecutive items in the queue, which is efficient on our hardware.

[5] At least on AMD hardware, atomic operations are up to 5x slower than normal accesses, because they use the *complete path* vs the *fast path* for memory access [3] In the compiler available with the current version of the SDK (AMD APP SDK v2.6), using any atomic operation on global memory causes *all* global memory accesses to use the complete path. In practice, this has led us to avoid atomic operations wherever possible.

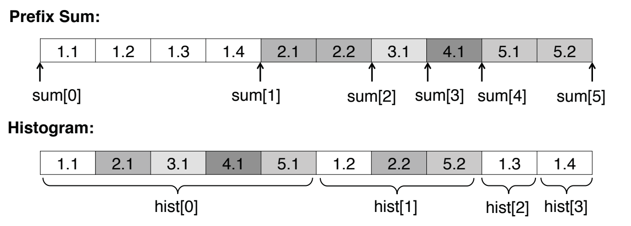


Figure 8. Different approaches for calculating offsets. $m.n$ describes the n'th reference of the m'th object (i.e. work-item).

5.2 Reducing Divergence

As discussed in Section 3, the majority of objects have a small out-degree (i.e. few references), while a few objects have a large numbers of references. In the algorithm above, each work-item loops over all references within the object it handles. This behavior is problematic for SIMT execution: when one work-item encounters a high-degree node, the remainder of the workgroup will stall until that work-item has completed its task. This results in low utilization of available parallelism and wastes available memory bandwidth.

To avoid this case, we extended our algorithm to let each work-item process at most a fixed number of references for each object (currently 16). This minimizes the worst case divergence in the loop. Objects that are longer than this are then stored on a non-blocking stack and (in the same iteration of the algorithm) processed in parallel. This is done by letting all work-items process one reference each in parallel, a very efficient way to perform a large copy operation. Like Veldema and Philippsen, we consider this an essential optimization. Our approach bears resemblance to theirs, but processes large arrays (and objects) immediately and does not require a new kernel launch.

5.3 Vectorized Memory Accesses

We explored the possibility of using vector reads to decrease the number of individual memory requests. OpenCL supports 4-wide vector types, which allow reading 128 bits at a time. We rewrote our algorithm to use vector loads to access the header and the first three references at the same time (and then read references in groups of four). This made it necessary to lay out the objects in such a way that headers are aligned to 128 bit boundaries, which we achieved by introducing additional padding to our reference graph.

Vector reads can lead to extra work, as the algorithm may read more references than necessary. Overall, however, we expected a speed-up due to the reduced number of memory requests.

5.4 Cut-off for Long, Narrow Heaps

As we show in our heap analysis (Section 3), some workloads exhibit very long narrow tails (stemming from e.g. linked lists). From a performance perspective, it is beneficial to detect such cases and return execution to the CPU. We believe that without the ability to saturate the memory pipe with many requests, the GPU loses out to the CPU due to the CPU's much lower average memory latency as a result of caching. The CPU benefits from any spatial locality of the memory graph that may exist, whereas the GPU does not. The CPU also benefits from the fact that the (very small) active section of the queue ends up in L1 cache.

We therefore implemented a mechanism that returns execution to the CPU once the size of the queue drops below a certain threshold. As a safeguard, we require a minimum number of iterations on the GPU to complete before returning.

Veldama and Philippsen identified a similar optimization, but in a different context: their discussion focuses on avoiding context switches to and from the GPU. To handle linked lists, their algorithm runs multiple iterations on the GPU without switching to the CPU. This optimization does not apply to our approach.

5.5 Multiple Compute Units

In order to achieve high throughput, it is desirable to leverage all of the GPU's compute units. For the purposes of our evaluation, we chose a naïve proof-of-concept approach to run the algorithm on the two compute units that our platform provides: we first divide the root set into two halves and hand one of them to each compute unit. Each compute unit then runs the algorithm independently, without any load balancing or synchronization. This approach has two drawbacks:

- If the initial partitioning results in an uneven distribution of work, one compute unit may be idle for most of the execution.
- We may perform redundant work in cases where the two compute units race to mark an object.

While this results in a negative performance impact, our approach is nonetheless correct: marking a node is an idempotent operation and can be performed multiple times without harm. Better results can be achieved by using dynamic load balancing between compute units – we discuss this aspect in Section 7.4.

6. Evaluation

In this section, we present the results of experiments we ran to evaluate the performance of our mark algorithm. We first describe our evaluation platform and then use microbenchmarks to highlight strengths and weaknesses of our algorithm and collector implementation. We then examine the performance of our implementation on real-world application benchmarks from the DaCapo 9.12 benchmark suite. We conclude with a brief discussion of additional overheads that were excluded from the previous subsections.

6.1 Test Platform

Our test platform was an AMD E-350 APU[6] which is one of the first chips that integrate a CPU and GPU into a single device (Intel's Sandy Bridge architecture has a similar integrated GPU, but it is not programmable). The E-350 targets low-end laptops and tablets.

The system was configured with 3.5 GB of DDR3 1066 RAM. The APU's "Bobcat" CPU is a dual core running at 1.6 GHz with a 512 KB L2 cache [2]. Its "Brazos" GPU is running at 492 MHz with 2 compute units, 16 stream processors, an 8 KB L1 cache per compute unit, and a 64 KB L2 cache per GPU [3]. Measurements show that the caches are disabled for accesses to local and shared memory. The CPU and GPU share memory and a single memory controller on which they compete for bandwidth; we experimentally determined that the GPU can only map 128 MB in any given kernel invocation. All experiments were conducted on Fedora Linux (kernel version 2.6.35.14-103).

6.2 Microbenchmark Results

To explain the performance of the baseline algorithm and explore potential optimizations, we used a set of simple microbenchmarks. These benchmarks were handwritten and do not run through the Jikes environment. This approach was chosen to get pure forms of the heap graphs; even a small Java program creates enough internal objects to obscure the microbenchmark results.

Table 1 presents the execution times of the microbenchmarks for a set of different configurations of the garbage collector.

[6] *Accelerated Processing Unit* (APU) is a term coined by AMD to describe their integrated CPU/GPU solution marketed as *AMD Fusion*.

	Size	CPU	GPU	GPU+P	GPU+V	GPU+F	GPU+D	GPU+VD	GPU+2CU
Single Item	28 bytes	0.001	0.027	0.027	0.028	0.028	0.027	0.028	0.028
Long Linked List	156 KB	0.151	94.604	74.400	83.451	0.360	96.279	85.412	84.173
256 Parallel Linked Lists	39 MB	129.723	140.465	192.823	118.982	140.783	142.556	120.984	102.092
2560 Parallel Linked Lists	117 MB	1074.920	415.553	572.153	350.523	416.389	421.589	356.986	194.317
Very Wide Object	3.92 KB	0.018	1.862	1.077	1.696	1.861	0.218	0.221	0.220
VP Linked List	4 MB	0.150	77.462	60.950	68.123	0.317	78.757	69.802	68.843
VP Array of Objects	20 MB	1.347	5.361	6.308	3.516	5.378	5.843	3.095	1.693

Table 1. Average mark times for microbenchmarks. All times are in ms and do not include overheads. CPU is a baseline CPU implementation. GPU is our baseline algorithm using the histogram approach. GPU+P is the variant using prefix-sum. GPU+V is the vectorization of that algorithm. GPU+F falls back to the CPU if a narrow tail is encountered. GPU+D has special support for large objects to prevent divergence. GPU+VD combines vector and divergence. GPU+2CU enables both compute units for the GPU+VD configuration.

Methodology We ran our microbenchmarks on the following configurations: *CPU* is our implementation of a serial single CPU mark phase. *GPU* is the baseline algorithm described previously. *GPU+V* is the vectorization of that algorithm (Section 5.3). *GPU+D* is the variant with special support for large objects to prevent divergence (Section 5.2). *GPU+F* is a variant which falls back to the CPU once the queue length drops below a threshold and a minimum number of iterations have run (Section 5.4); we use 20 as the threshold and 5 as the minimum number of iterations. We report the sum of the GPU and CPU runtime. *GPU+P* uses the prefix-sum approach instead of the histogram (Section 5.1), for comparison. *GPU+2CU* contains the first two optimizations but also uses both compute units.

Each configuration was run for 20 iterations and the average runtime is reported; variation between runs was extremely low.

Benchmark Descriptions For each benchmark, we also provide the overall size of the reference graph that is associated with it:

- **Single Item** (28 bytes) - This benchmark consists of a single item in the heap, with a corresponding pointer in the root set. The purpose of this benchmark is to measure the overhead (excluding copy overhead) of the algorithm. As would be expected, the startup cost for the GPU variants are similar. The CPU is an order of magnitude faster since the data is already in cache.
- **Long Linked List** (156 KB) - This benchmark consists of a single long linked list with 10,000 elements. This case is the worst for the GPU since it cannot exploit any parallelism in the graph. All of the GPU implementations perform badly, but the one with the option to fall back to the CPU fares best. It runs the minimum number of iterations on the GPU, then returns to the CPU for the majority of the execution. Unfortunately, the few iterations it does run on the GPU prove quite expensive.
- **256 Parallel Linked Lists** (39 MB) - This benchmark consists of 256 parallel linked lists with 10,000 elements each. The root set contains a pointer to each linked list. The effect of this is that each work-item within the workgroup can operate independently, which allows the GPU implementations to perform relatively well, some even beating the CPU by a small amount.
- **2560 Parallel Linked Lists** (117 MB) - This benchmark extends the previous by adding more linked lists. Due to our hardware's limited amount of mappable memory, we shortened each list to 3,000 elements each. This case can arguably be seen as the best for the GPU since there is abundant parallelism and little locality between objects in the queue. This microbenchmark is the only one where the GPU solidly outperforms the CPU.
- **Very Wide Object** (3.92 KB) - This benchmark consists of a single array containing 1,000 individual objects. This is an extreme case designed to illustrate the effects when SIMT divergence is not addressed.

Discussion These benchmarks allow us to evaluate the impact of the optimizations discussed in the previous section.

- **Histogram** (Section 5.1) - Comparing the GPU and GPU+P results shows the difference in performance characteristics of the histogram and prefix styles of offset calculations. The prefix sum implementation performs well for cases in which a small subset of the work-items perform useful work, while the histogram fares better when many work-items are active. On real-world benchmarks (not presented), the histogram is clearly better, but it may be worth exploring a combination of both approaches (e.g. by switching dynamically between them).
- **Divergence Handling** (Section 5.2) - This causes a slight slowdown for those benchmarks that do not contain objects with large numbers of references. For benchmarks that do (such as *Very Wide Object* above), the performance improvement is substantial (a 89% improvement). For real workloads, we believe divergence handling to be a critical and necessary optimization.
- **Vectorization of Loads** (Section 5.3) - This optimization shows an improvement on most of the microbenchmarks we report. The improvements range from 12% to 40% for all benchmarks except the *Single Item* case. This case is hinting at a more general problem which is that vectorization can (and does) hurt performance in some cases: if the vectorization causes memory words to be read that are not used, and if memory bandwidth is already running at the hardware limit, vectorization can slow down the algorithm. However, from our experiences, this seems to be a rare case.
- **Falling back to the CPU for narrow tails** (Section 5.4) - Our implementation of fallback has a barely perceptible negative impact on performance for most benchmarks. However, for cases where the GPU would perform extremely poorly (such as the *Long Linked List* microbenchmark), it recovers some, but not all, of the performance lost. There is still a significant amount of time spent on the GPU to handle narrow sections before the cut-off is invoked; we believe this to be a necessary evil to prevent temporary drops in parallelism from triggering overly eager fall-back.
- **Multiple Compute Units** (Section 5.5) - Despite our naïve approach, we still obtain perceptible improvements by using both compute units. It is important to note that this improvement is not guaranteed: using a second compute unit can hurt performance if the first unit would otherwise get additional bandwidth and the second unit is performing only redundant work.

Comparison with related work The last two benchmarks are modeled closely after those presented by Veldema and Philippsen for evaluating their GPU mark algorithm. Unfortunately, the results are not directly comparable due to different experimental setups. We would like to note that their results were collected on a sig-

	Jikes MS	Serial CPU	Baseline GPU	Optimized GPU	Opt GPU + 2CU	GPU Slowdown	Opt Speedup
lusearch	1566.10	1084.18	11739.50	2404.18	1490.96	1.38	7.69
pmd	211.98	356.09	1651.66	634.24	357.49	1.69	4.55
sunflow	1422.54	401.36	3446.52	724.92	554.25	1.38	6.25
xalan	809.79	423.31	2836.33	1088.78	750.12	1.77	3.85

Table 2. Cumulative mark times for DaCapo benchmarks (default sizes). All times are in ms and do not include overheads. *Optimized GPU* uses vectorization and divergence handling. *Jikes MS* is the unmodified `MarkSweep` collector. *Serial CPU* is a CPU implementation using the reference graph. The last two columns show the slowdown over the best CPU implementation and the improvement from optimizations.

nificantly more powerful GPU. Nonetheless, our mark algorithm appears to fare well in comparison.

- **VP Linked List** (4MB) - This benchmark consists of 16 linked lists of 8,192 element each, of which all but one is immediately garbage. Only one of the linked lists is traced by the mark phase. As a result, this is structurally very similar to the *Long Linked List* benchmark above.
- **VP Arrays of Objects** (20MB) - This benchmark consists of 1,024 arrays, each containing exactly 1,024 objects. Only the first 64 arrays are retained. All others immediately become garbage and are not traced.

It should be noted that we do not report launch overheads, while Veldema and Philippsen report complete execution times. Furthermore, they perform 8 collector runs while we only measure one.

6.3 DaCapo Benchmarks

We measured the performance of our GPU-based collector for real-world application benchmarks from the DaCapo 9.12 benchmark suite. The results are shown in Table 2.

Methodology In these results, the *Optimized GPU* implementation includes the vectorization and divergence handling optimizations (*Opt GPU+2CU* also uses both compute units). Both the optimized and unoptimized results use the histogram method for offset calculation. Neither version includes the long-tail cutoff (to avoid the issue of confusing what is actually running on the GPU). The *Jikes MS* column is an unmodified instance of Jikes' `MarkSweep` collector. The second column is a trivial CPU implementation which operates on the reference graph. We present these numbers to offset any minor locality advantage the reference graph structure may give us. The final two columns present the slowdown of the GPU over the best of the two CPU implementations and the improvement resulting from optimization of the GPU algorithm.

The Jikes RVM was configured with a maximum heap size of 192 MB - the largest we could map on the GPU even with the reference graph. We do not report collection times for `avrora` or `luindex` since neither consistently triggers a collection at the heap size we are using. All results were generated running the benchmarks with their default configurations and using the "converge" (`-C`) option provided by the suite. We report the cumulative time of all garbage collections conducted during the final iteration.

Discussion As can be seen from the results in Table 2, our GPU mark implementation is within a factor of two for all of the benchmarks we report. As a reminder to the reader, we are conservatively comparing against the better of Jikes' `MarkSweep` and our own CPU implementation working off the reference graph. When comparing only against the `MarkSweep` collector, our implementation fares significantly better; the GPU outperforms Jikes on 3 of 4 benchmarks. We consider this to be a highly encouraging result.

We would like to note that these performance results are extremely sensitive to the heap size. As the heap size increased, the relative performance of our GPU implementation to Jikes increased sharply. We present the largest heap sizes supported by our evaluation platform, but even those are small for real program heaps. We suspect that relative performance would continue to improve as the heap size increases.

6.4 Overheads of Our Implementation

In the preceeding discussion, we excluded the copy overhead and kernel launch overhead for any of the GPU configurations; we report kernel execution only[7].

Our reference graph implementation adds some additional overhead outside the mark phase. Allocating each object requires that a corresponding reference graph node be allocated as well; this introduced mutator overhead of approximately 40% in an allocation stress test microbenchmark. This overhead is less pronounced in the DaCapo results, but is still significant, varying between 7% and 25%. It should be mentioned that this overhead could presumably be reduced by adding this functionality through the compiler, rather than adding an extra function call.

Using the basic approach of filling in the entire reference graph before every collection adds a major overhead to each collection, taking several times as long as the mark phase on the CPU (arguably due to a highly untuned implementation). The double-write approach eliminates this at the cost of an additional 11% runtime overhead in the microbenchmark (for a cumulative total of 57%).

Some collector overhead is also added in copying markings from the reference graph back to the heap in preparation for running an unmodified Jikes sweep phase.

Let us emphasize that all overheads discussed in this subsection are artifacts of either the copying of data to the GPU (Section 4.4) or our need to reduce the size of the space being collected (Section 4.3). Neither is intrinsic to the problem and both are likely to be eliminated by hardware changes in the near future.

7. Discussion

While our numbers imply that our GPU-based garbage collector is 40-80% slower than our CPU-based collector and therefore not directly competitive in terms of performance, our experimental results nonetheless answer the questions we set out to investigate. We identified the key points for offloading garbage collection to the GPU, some of which are surprising in hindsight. We were also able to assess the suitability of today's GPUs for garbage collection, as well as making predictions on how future hardware will further improve the situation.

7.1 Lessons from the Mark Algorithm

Somewhat counter-intuitively, the primary goal for garbage collection on the GPU is not to parallelize the computational steps of the algorithm but to maximize the hardware's ability to schedule memory requests. The key challenge is to ensure that each work-item can effectively generate and handle memory requests. It is therefore crucially important to avoid serialization of execution (as en-

[7] For Jikes, we only report the mark phase, `scan` in Jikes terminology.

sured by our queue approach), but also to reduce divergence between threads (Section 5.2). The numbers presented in Section 3 confirm that common heap graphs exhibit the rare but long objects and arrays which cause this divergence.

In our algorithm, the number of outstanding requests is limited by the maximum size of a workgroup (which depends on the hardware). Notably, this is different from the number of streaming processors in the GPU: while the number of streaming processors limits the throughput in terms of instructions per cycle, the workgroup size limits the number of work-items that can be in-flight at a given time, and therefore the number of outstanding memory requests that can be issued. We already see this size increasing in high-end parts, implying that future generations will be increasingly good at memory-bound problems such as garbage collection. The same is true for the number of memory channels: While our APU features only two channels, high-end parts often provide eight.

As with many GPU algorithms, it is only feasible to run the mark algorithm on the GPU if the number of objects to mark is sufficiently large. For small collections, the launch-overhead dominates the entire collection time, in which case it is beneficial to run the collection on the CPU in the first place. Predicting the size of a collection is non-trivial, but heuristics could be applied.

7.2 Lessons from the Reference Graph

Our reference-graph approach is orthogonal to the problem of performing garbage collection on the GPU: we assume that in the near future, GPUs will be capable of mapping the entire heap, perhaps even cache-coherent with the CPU. However, we noticed that the reference graph gave us significantly better performance for a CPU collector: our untuned *Serial CPU* collector beat the optimized Jikes collector on several occasions, arguably due to increased cache locality. We therefore briefly discuss performance trade-offs for using the reference graph in a conventional GC.

Our numbers from Section 6.4 indicate that keeping the reference graph up-to-date when running the mutator seems to be the most promising approach. We believe that modifying the compiler to issue duplicate writes whenever a reference is written will lead to a significantly lower performance impact than we are incurring with our naïve, write-barrier based approach. An alternative approach consists of splitting each object into two parts, one only containing the references, the other containing the non-references. This avoids the need for duplicating data and substantially improves collector locality, at the cost of access locality.

We did not investigate this approach further but found it worth mentioning as we found the trade-offs intriguing.

7.3 Estimation of Performance Limits

To estimate the potential of our approach, we need to quantify how the performance of our implementation compares to the theoretical best case on the given hardware. To do so, we present two weak, but independent, constructions of a lower bound on execution time for the *2,560 Parallel Linked List* benchmark from Section 6. We then discuss performance measurements that lead us to believe that the actual bound is even tighter.

The first bound can be constructed by examining the minimum time required to touch every memory location in the reference graph exactly once. As constructed, the reference graph contains only the edges in the heap graph and some minor padding. While there may be a more compact representation, we believe that this is a reasonable first order approximation for a minimum-size representation of the heap graph. Using only the size of the benchmark (117 MB) and the peak memory bandwidth for our device (9 GB/s), we can establish a lower bound for GPU execution of ∼12.7 ms.

For the second bound, we can consider the minimum number of dependent loads from main memory and the stall latency implied by each. Without the presence of caches, each step of the list traversal requires at least one round trip to main memory. As a result, a lower bound on the run-time of the algorithm is given by $depth \times stall\_penalty\_in\_cycles \times 1/gpu\_frequency$. We benchmarked a stall latency of 256 cycles under load and the benchmark requires a minimum of 3,000 dependent loads (one per linked-list element). Taken together, this gives us a lower bound of $\sim$ 1.5 ms. For this benchmark, the bound is not particularly tight, but we present it nonetheless since it reflects structural features of the heap graph that cannot be avoided (Section 3).

Together, these two approaches gives us a bound that is about 15x better than our best measured performance on the GPU.

We also examined the sustained memory bandwidth achieved by our implementation over an entire execution of the mark phase and compare it against the peak memory bandwidth available on the device. For our benchmark, the optimized dual compute unit configuration achieves a sustained bandwidth of 3.016 GB/s, or roughly one third of peak. As expected, the single compute unit version of the same code achieves roughly $1/2$ of the bandwidth at 1.72 GB/s. It is worth noting that these are measurements of our actual implementation and thus may not reflect an actual bound due to errors in the implementation or missed optimizations. As an illustrative example, disabling the vectorization and divergence handling for the dual compute unit code gives a higher sustained bandwidth (4.317 GB/s), but lower overall performance. (We believe this to be due to the fact that the native memory request size is 2 words. In some cases, reading the two words separately can result in separate requests being issued and artificially inflate bandwidth). An additional caution is that the profiler is known by the vendor to provide unreliable results under some circumstances.[8]

Taking these points together, we believe our algorithm to be within a moderate constant factor of optimal on our hardware.

7.4 Load Balancing on Multiple Compute Units

As explained in Section 5.5, our current implementation statically distributes the load between the two compute units on the device. We believe that static load balancing will not suffice for a real implementation (or even our own implementation on a device with more than two compute units). Given that we expect to see the number of compute units grow in future-generation parts, this is an urgent concern. With this in mind, we experimented with a number of options for synchronization between compute units.

Today, GPUs are primarily used for regular numeric computations. The traditional approach to irregular (imbalanced) computations has been to either pre-partition data into regular components or to defer irregular work to the CPU. Synchronization and load balancing between compute units is an underexplored area.

Graph traversal is a highly irregular computation. The analogy of pre-partitioning (and re-partitioning) for graph traversal is to stop the GPU kernel after regular intervals, have the CPU inspects all queues, load balance if necessary, and then relaunch the kernels. This is related to the option chosen by Veldema and Philippsen [21]. As they showed, it can be used effectively, but incurs significant overhead since kernel launch and termination are expensive synchronization actions (see Section 4.4 for discussion of launch overheads). Additionally, this solution would interfere with our goal of leaving the CPU available for other processing. Potential alternatives include:

- Using global atomics to synchronize through shared memory. As discussed previously and documented by Elteir et al. [7], global operations are prohibitively expensive on AMD hardware. It may be viable on hardware from other vendors or future generations of GPUs.

[8] As noted in the Developer Release Notes for AMD APP SDK v2.6.

- Using on-device hardware counters to construct a fast software lock. After a trial implementation, we were forced to conclude that the counters were not appropriate for our goals.
- Having each compute unit copy content from the other compute unit's queue into its own if its queue length drops below its number of work-items. This scheme does not use any form of synchronization and thus cannot update the source queue safely. As a result, redundant work can and will be performed. From preliminary results, it appears that the overhead caused by the inspection outweighs any benefit provided by the load balancing. We did not explore this idea further.

Based on our investigation, the only dynamic load balancing scheme that seems currently viable is to use the CPU for coordination as suggested by Veldema and Philippsen [21]. This is unsatisfactory and we see a need for future work in this area.

7.5 Assessment of Garbage Collection on Current GPUs

Our results show that it is possible, with a significant overhead, to build a GPU-based garbage collector on current hardware. The numbers from the microbenchmarks show that an optimized GPU mark algorithm can, for the best case, significantly outperform a mark algorithm running on the CPU. However, our results for the DaCapo benchmarks show that the mark phase for real-world workloads is 40-80% slower than on the CPU (but sometimes outperforms Jikes' `MarkSweep` collector).

We noted that the copy-overheads for the heap (or reference graph) can quickly reach or exceed the order of magnitude of an actual collection on the CPU, even if CPU and GPU are on a single chip. We therefore argue that, to make GPU-based garbage collection feasible, we have to wait for architectures (or, in our case, drivers) that support zero-copy mapping between the two devices. However, these devices are appearing at the moment.

At the same time, the overhead from maintaining the reference graph will have to be reduced as well. We expect that GPUs will soon allow mapping enough memory to store the entire heap, so that minimal changes to Jikes' object model should be sufficient to run our collector without the reference graph.

Once these overheads disappear, there is no intrinsic reason why GPUs could not be used for garbage collection in the near future. A particularly interesting application area is the use in concurrent garbage collectors: if it is possible to generate a snapshot of a part of the heap (which may well have the form of our reference graph), it could be offloaded to the GPU and collected in isolation from the mutators running on the main CPU.

7.6 Future Directions in Hardware

During our work, we discovered a number of situations where we were severely limited by the capabilities of the hardware. Oftentimes these were "quirks" rather than fundamental limitations of GPUs, and we believe that, as more non-numeric workloads (such as GC) appear, vendors could quickly address them.

As discussed is Section 7.4, support for dynamic load balancing is a particular problematic area. Given the ongoing efforts by GPU vendors to generalize their applicability, we expect that better support will be forthcoming in future revisions of hardware and software. We also note that the specific issues we encountered were vendor-specific and might not apply to other vendors' devices.

Another area of potential improvement is the provisioning of dedicated memory-bandwidth for the individual components of integrated GPUs; AMD Fusion APUs have the disadvantage that the performance of an application running on the CPU is directly tied to the memory behavior of the corresponding code on the GPU and vice versa. While some dynamic provisioning is certainly desirable, a priority reservation per device would help to improve performance isolation.

A different aspect that causes problems on current-generation hardware is the vastly different model in AMD and NVIDIA GPUs, which requires fundamentally different mechanisms, optimizations and trade-offs (e.g. memory coalescing is much more important on NVIDIA GPUs than it is on AMD hardware). With a more serious entrance from Intel into the GPGPU market, this may well leave us with three fundamentally different GPU models. While it would be possible to auto-tune a collector to the individual platform before running it, a broad adaption of GPU-assisted garbage collection would require a more unified programming model.

Somewhat orthogonally, we also believe that GPU-like support for parallelism might become increasingly integrated into the CPU itself. This would make it easier to implement an approach such as ours directly on this parallel hardware on the CPU. However, current-generation CPUs lack support for `scatter` and `gather` instructions, which would be crucial for such an approach.

Overall, we believe that hardware is heading in a direction that is beneficial to our approach: the number of supported work-items per workgroup is increasing, more memory is becoming available to the GPU, CPU/GPU integration is becoming more common (eliminating copy-overhead) and cache-coherence between CPUs and GPUs is on the horizon. We therefore believe that the work we show in this paper will be particularly relevant for the next generation of hardware and may show an appealing application of those devices beyond graphics-intense and special-purpose workloads.

8. Related Work

The idea of performing garbage collection on the GPU is not a contribution of this paper. Jiva and Frost describe the basic approach in a patent application in 2010 [13], while Sun and Ricci [20] describe the idea as part of a larger vision of using GPUs to speed-up a variety of traditional operating system tasks. However, to our knowledge, none of them has published an appropriate algorithm or publicly disclosed a working GPU-based collector.

While the recent work by Veldema and Philippsen [21] explores the implementation of a mark & sweep garbage collector on the GPU, their work differs from ours in a number of important points. First and foremost, their goal was not to use the GPU to accelerate garbage collection for programs running on the CPU, but to provide garbage collection facilities for CUDA-like programs written in a Java dialect and running on the GPU. Additionally, our mark algorithm bears little resemblance to theirs.

There have been a few recent papers proposing potential non-numeric applications for GPUs. Naghmouchi et al. [18] investigated using GPUs for regular expression matching. Smith et al. [19] evaluate GPUs as a platform for network packet inspection.

Several groups have investigated efficient algorithms for performing breadth-first-search on a GPU. One of the first publications in this space was the work by Harish and Narayanan [10] who presented the first algorithm to perform an efficient breadth-first-search on the GPU. However, this approach was based on visiting every node at every iteration, and was less efficient than the most efficient CPU implementation at that time. Their approach was improved by Luo et al. [16] who used an approach based on hierarchical queues to achieve better performance. Recent work by Hong et al. [12] improved the performance even further. Veldema and Philippsen's [21] approach resembles the work by Harish and Narayanan [10], whereas ours takes the approach of Luo et al. [16] and Hong et al. [12]

Another body of work that is related to ours describes the use of other parallel architectures or heterogeneous platforms to perform garbage collection. An example for this is the work by Cher and Gschwind which demonstrates how to use the Cell processor to accelerate garbage collection [22]. Barabash and Petrank cover the problem of garbage collection on highly parallel platforms from a more general perspective and perform a heap analysis similar to ours [5]. An early paper by Appel and Bendiksen [4] covers garbage collection on vector processors and our approach has been influenced by some of their ideas.

There is, of course, a multitude of work in the general space of garbage collection. A general introduction can be found in [14]. While we are focusing on mark & sweep garbage collection, the state of the art collectors are usually parallel generational-copying collectors. A good example for such a collector is given in [17].

9. Conclusion

GPUs are often underutilized when not executing graphics-intense or special-purpose numerical computations. We showed that it is possible to offload garbage collection workloads to the GPU, to use these otherwise unused cycles.

We presented and evaluated a prototype of a GPU-based collector for real-world Java programs. We first examined heap graphs from the DaCapo benchmark suite to show that there are no structural features that would prevent the effective parallelization that is required by a GPU. We then implemented a queue-based mark algorithm on the GPU, as well as a number of optimizations. We integrated this algorithm into a collector for the Jikes RVM.

Reflecting the direction of current hardware trends, we used an integrated GPU/CPU device as our evaluation platform. With minor adjustments to reflect the limits of current-generation parts, we showed that a GPU implementation of the mark phase is nearly performance-competitive with a tuned CPU implementation.

We identified two hardware features which are essential for garbage collection on GPUs: eliminating copy overhead (zero-copy) and enabling the GPU to access the entire physical address space of the CPU. We also highlight fast synchronization between compute units on the GPU and the memory subsystem as areas where hardware changes would be profitable to better support garbage collection. Current hardware trends indicate that each of these areas is likely to improve rapidly in the near future.

Acknowledgments

We would like to thank David Sheffield for providing us with insights to developments in hardware architectures and GPUs in particular. Thanks is also owed to Eric Brewer for his feedback on early drafts of this paper, and to our colleagues at UC Berkeley for proof-reading and advice. Last but not least, we would like to thank the anonymous reviewers for their comments.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[2] AMD. AMD Embedded G-Series Platform: The world's first combination of low-power CPU and advanced GPU integrated into a single embedded device. `http://www.amd.com/us/Documents/49282_G-Series_platform_brief.pdf`.

[3] AMD. AMD Accelerated Parallel Processing (APP) SDK OpenCL Programming Guide. `http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf`.

[4] A. W. Appel and A. Bendiksen. Vectorized garbage collection. *The Journal of Supercomputing*, 3:151–160, 1989.

[5] K. Barabash and E. Petrank. Tracing garbage collection on highly parallel platforms. *SIGPLAN Not.*, 45:1–10, June 2010.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN Not.*, 41:169–190, October 2006.

[7] M. Elteir, H. Lin, and W.-C. Feng. Performance Characterization and Optimization of Atomic Operations on AMD GPUs. In *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 234 –243, Sept 2011.

[8] E. M. Gagnon and L. J. Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *In Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 27–40, 2000.

[9] R. J. Garner, S. M. Blackburn, and D. Frampton. A comprehensive evaluation of object scanning techniques. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 33–42, New York, NY, USA, 2011.

[10] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. *Technology*, 4873:197–208, 2007.

[11] M. Harris. Parallel Prefix Sum (Scan) with CUDA. *GPU Gems*, 3 (April):851–876, 2007.

[12] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011.

[13] A. S. Jiva and G. R. Frost. GPU Assisted Garbage Collection, 04 2010. URL `http://www.patentlens.net/patentlens/patent/US_2010_0082930_A1/en/`.

[14] R. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Sept. 1996.

[15] Khronos Group. OpenCL 1.2 Specification. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[16] L. Luo, M. Wong, and W.-m. Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010.

[17] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 11–20, New York, NY, USA, 2008.

[18] J. Naghmouchi, D. P. Scarpazza, and M. Berekovic. Small-ruleset regular expression matching on GPGPUs: quantitative performance analysis and optimization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 337–348, New York, NY, USA, 2010.

[19] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for network packet signature matching. In *International Symposium on Performance Analysis of Systems and Software, 2009. ISPASS 2009*, pages 175 –184, April 2009.

[20] W. Sun and R. Ricci. Augmenting Operating Systems With the GPU. Technical report, University of Utah, 2010.

[21] R. Veldema and M. Philippsen. Iterative data-parallel mark & sweep on a GPU. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 1–10, New York, NY, USA, 2011.

[22] C. yong Cher and M. Gschwind. Cell GC: using the Cell synergistic processor as a garbage collection coprocessor. In *VEE '08: Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 141–150. ACM, 2008.