J Supercomput
DOI 10.1007/s11227-011-0680-7

Scalable SIMD-parallel memory allocation for many-core machines

Xiaohuang Huang · Christopher I. Rodrigues · Stephen Jones · Ian Buck · Wen-mei Hwu

Abstract Dynamic memory allocation is an important feature of modern programming systems. However, the cost of memory allocation in massively parallel execution environments such as CUDA has been too high for many types of kernels. This paper presents XMalloc, a high-throughput memory allocation mechanism that dramatically magnifies the allocation throughput of an underlying memory allocator. XMalloc embodies two key techniques: allocation coalescing and buffering using efficient queues. This paper describes these two techniques and presents our implementation of XMalloc as a memory allocator library. The library is designed to be called from kernels executed by massive numbers of threads. Our experimental results based on the NVIDIA G480 GPU show that XMalloc magnifies the allocation throughput of the underlying memory allocator by a factor of 48.

Keywords Malloc · CUDA · GPGPU

X. Huang · C.I. Rodrigues (✉) · W.-m. Hwu
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
e-mail: cirodrig@illinois.edu

X. Huang
e-mail: xhuang22@illinois.edu

W.-m. Hwu
e-mail: w-hwu@illinois.edu

S. Jones · I. Buck
NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, USA

S. Jones
e-mail: stjones@nvidia.com

I. Buck
e-mail: IBuck@nvidia.com

Published online: 23 September 2011

1 Introduction

Dynamic memory allocators are widely used in modern applications. Efficient memory allocation poses a special challenge in architectures that support a high degree of multithreading and SIMD-parallel execution. The NVIDIA Fermi GPU architecture, for instance, sustains thousands of concurrently running threads with a large number of processors each with a high degree of SIMD execution. To provide enough memory allocation throughput to satisfy such a large number of concurrent threads, the memory allocator should itself be highly parallel and avoid generating high memory traffic.

In versions 3.2 and later of the CUDA SDK, NVIDIA has included support for dynamic memory allocation in CUDA kernel functions. Programmers no longer need to preallocate all the memory for a kernel before launching the kernel for parallel execution. Based on our preliminary analysis, the throughput of the allocator is on the order of microseconds for the G480 GPU. This is still at least an order of magnitude too slow for many applications.

In this paper, we present two techniques that amplify the throughput of the memory allocator provided in the CUDA SDK. One technique, called allocation coalescing, aggregates memory allocation requests from SIMD-parallel threads into one request to be handled by the underlying memory allocator. The single allocated memory block is then subdivided and distributed to the individual requesting threads. This technique takes advantage of lock-step execution of groups of SIMD threads (what are called *warps* in CUDA) to reliably amplify the throughput of the underlying memory allocator.

The other technique, allocation buffering, buffers a limited number of free memory blocks for fast access. These buffers are used in conjunction with allocation coalescing. Allocation requests are processed in parallel by distributing the load among multiple buffers.

We have implemented both allocation coalescing and allocation buffering into XMalloc, a user-level memory allocation layer that can be used to magnify the throughput of an underlying, standard memory allocator. On an NVIDIA G480 GPU, XMalloc achieves up to 48× speedup for a microbenchmark that stresses the memory allocator.

2 Background and related work

Berger et al. survey parallel memory allocators [1]. Many parallel memory allocators [1, 5, 11] use *private heaps*. That is, multiple heaps are set up so that each processor or thread interacts predominantly with only one of the heaps. Different processors can interact with different heaps concurrently. In a cache-coherent system, a heap's data tends to reside in the cache of the processor that uses it most, so that heap management involves mostly processor-local memory traffic. Unfortunately, private heaps do not benefit the current generation of NVIDIA GPU architectures due to the absence of cache coherence.

Parallel allocators may also allow concurrent access to a single heap through concurrent data structures [2, 9]. Iyengar [8] uses multiple free lists for parallel allocation and deallocation.

An earlier version of XMalloc [7] was the first general-purpose memory allocator usable in computational GPU kernels. It used superblock allocation and CAS-based lock-free critical sections, similar to an allocator presented by Michael [11]. Due to the absence of writable caches in CUDA-capable GPUs at the time, all shared, writable data structures were assumed to reside in off-chip memory. XMalloc introduced allocation coalescing and an optimized FIFO-based free list to more efficiently utilize the limited-bandwidth, long-latency off-chip memory. We use the same allocation coalescing mechanism and FIFO data structure in this work. We also considered XMalloc's superblock allocation algorithm, but did not find its performance to be competitive with the CUDA SDK.

Starting with version 3.2, the CUDA SDK [3] has incorporated a memory allocator. We use version 4.0 of the SDK to provide the memory allocation functionality in this work. To our knowledge, the design and performance characteristics of that allocator are not published. XMalloc is modified to magnify the throughput of the SDK's allocator. This use model allows one to easily port XMalloc to any CUDA or OpenCL platform that provides a base GPU memory allocator.

3 Shared-memory parallel programming in CUDA

The CUDA programming language encourages a task-parallel, bulk-synchronous programming style, in which a parallel workload consists of a large number of independent tasks, and each task consists of many parallel threads that share access to a scratchpad memory and can barrier-synchronize with one another. Yet CUDA code is not restricted to this programming style; synchronization and communication between arbitrary running threads is possible. Dynamic memory allocation uses these more general capabilities to manage a pool of free memory that is shared by all threads. To write race-free, deadlock-free, and efficient code in this more general programming model, one must mind details of the compiler and processor architecture that aren't part of the basic CUDA programming model. We review these details as they pertain to the Fermi GPU architecture, used to evaluate XMalloc in this work.

In CUDA, a task or *thread block* consists of a group of threads that are scheduled together onto a processor, called an *SM*. Once a task is initiated, all the threads of the task run on the SM until completion. The threads in a thread block can synchronize using a hardware-supported barrier operation. CUDA kernels that adhere to the task-parallel, bulk-synchronous style of programming use only the hardware-supported barrier for synchronization and communicate only between threads in the same thread block.

The threads of a thread block are organized into groups of 32, called *warps*, that execute in a SIMD-parallel manner: each thread executes the same instruction in one lane of a 32-wide SIMD processor. Nevertheless, each thread maintains its own flow of control. When threads in a warp follow different control flow paths, the hardware executes each control flow path separately, enabling only those SIMD lanes

```
#include "Xmalloc.h"

__global__ void kernel(void) {
  void *p = xmcMalloc(64);  // Allocate GPU memory
  /* ... use p ... */
  xmcFree(p);               // Free GPU memory
}

int main() {
  xmcInit();                // Initialize XMalloc heap
  kernel<<<1000, 256>>>();  // Run a GPU kernel
}
```

Fig. 1 Code example showing usage of XMalloc library functions

corresponding to threads on the active control flow path. Hardware multithreading is employed to interleave the execution of all warps on an SM.

The Fermi GPU architecture does not have cache coherence among its L1 caches. Consequently, an ordinary load from a shared data structure may not return the correct data, even in a properly synchronized program. Atomic operations, ordinary stores, and loads from `volatile` pointers appear to bypass the L1 cache and access the globally coherent L2 cache. Memory consistency can be achieved by avoiding non-volatile pointer reads and using memory fences at synchronization points.

Parallel programs typically employ synchronization, where one or more threads wait for some event to occur. However, synchronization of individual threads tends to be inefficient due to the GPU's scheduling policy. There is no way to suspend execution of an individual GPU thread. Although a thread may busy-wait for an event by looping until it detects that the event has occurred [10], it is costly to do so: since many threads are time-sliced onto one processor, busy-waiting consumes processor time and memory bandwidth that could have been used by other threads. Instead of using synchronization, we employ lock-free code [6] where communication is required. Lock-free code tolerates the GPU's indifferent thread scheduling policy by guaranteeing progress to whichever thread completes its critical section first.

4 Using XMalloc

XMalloc exports a memory management interface that is almost identical to the CUDA SDK's `malloc` and `free` functions. Before using XMalloc, a program must initialize the allocator's global data structures by calling `xmcInit()`. GPU heap memory is allocated within kernel code by calling `xmcMalloc` and freed by calling `xmcFree`. Allocated memory persists between kernel invocations. A program can use both `xmcMalloc` and `malloc` as long as `xmcFree` is only used for XMalloc objects and `free` is only used for other objects. Figure 1 summarizes the usage of XMalloc.

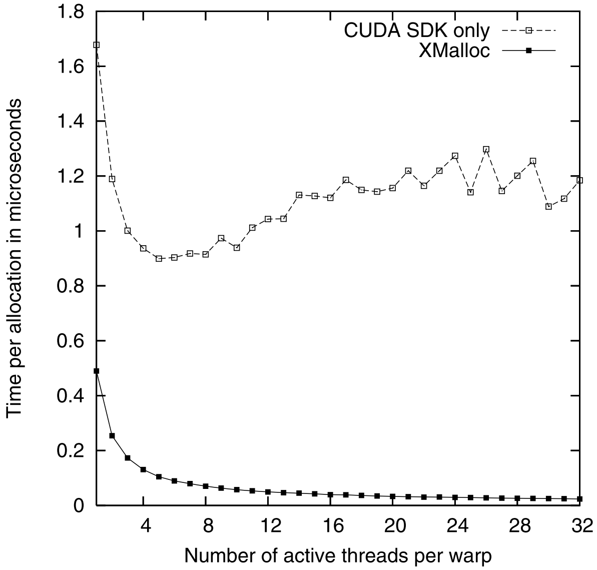


Fig. 2 Throughput comparison of XMalloc against using the CUDA SDK memory allocator directly

5 Implementation

Figure 2 compares the CUDA 4.0 SDK's memory allocation throughput (time per object allocated) against our enhanced allocator under a high load scenario on an NVIDIA G480 GPU. The measured times are the average of four runs of a microbenchmark consisting of one million worker threads that allocate an `int`, write to it, and then free it. A 500 megabyte heap was used. The heap was initially empty, aside from data structures created when XMalloc was initialized. On the graph's x-axis, we vary the number of worker threads per SIMD processor from one up to 32, where 32 is the SIMD width of the current NVIDIA GPU architectures. In cases where the number of worker threads is less than 32, the remaining SIMD lanes are occupied by nonworker threads that do nothing. The horizontal trend shows how well the allocators utilize SIMD parallelism. A flat line would indicate that SIMD-parallel calls are, effectively, processed serially with no overhead. The CUDA SDK appears to take advantage of a small amount of SIMD parallelism, evidenced by the downward slope of the left part of the curve, but the curve quickly levels out. In the best case, allocation takes 0.9 μs.

Our allocator uses two fast, high-throughput mechanisms to handle the majority of memory allocations and deallocations, effectively amplifying the throughput of the CUDA SDK's allocator. Allocation coalescing (Sect. 5.1) specifically targets SIMD-parallel memory allocation, combining multiple SIMD-parallel memory allocations into one. Buffering of free blocks (Sect. 5.2) allows us to satisfy frequent memory requests quickly and in parallel. With our enhancements, we are able to reduce the allocation time to as little as 0.024 μs. With 32 threads, we achieve a 48× speedup over the CUDA SDK.

Our allocation layer performs a constant amount of work per memory request, not counting the work performed by the underlying memory allocator. Thus, its performance is not directly affected by the heap size or the number of allocated objects.

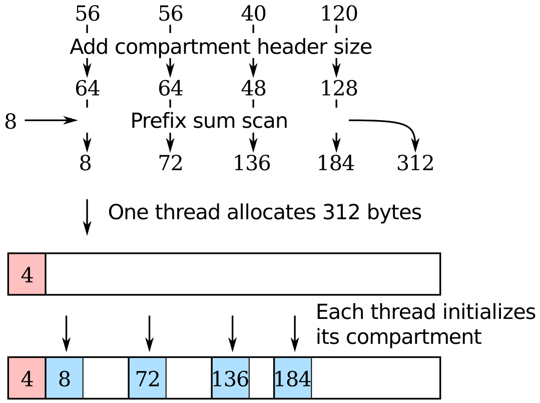

56
56
40
120
Add compartment header size
64
64
48
128
8
Prefix sum scan
8
72
136
184
312
One thread allocates 312 bytes
4
Each thread initializes its compartment
4
8
72
136
184

Fig. 3 SIMD memory allocation coalescing. *Boxes* represent allocated memory; *shaded areas* are allocator-managed metadata. In the figure, four threads request memory spaces of 56 bytes, 56 bytes, 40 bytes, and 120 bytes simultaneously. The total size and offset of each compartment is computed. These requests are converted into compartment sizes of 64 bytes, 64 bytes, 48 bytes, and 128 bytes assuming that each compartment header is 8 bytes. One thread allocates a memory space of this size. All threads select and initialize their compartment within the new memory space simultaneously

The underlying memory allocator's performance can still affect the performance of XMalloc. For example, filling the heap with 2^{17} one-kilobyte blocks (approximately one quarter of the heap) before testing slows down memory allocation both with and without XMalloc. The speedup of XMalloc with 32 threads diminishes in this case to just under 6×.

5.1 SIMD-level allocation coalescing

The first stage of our memory allocator combines multiple SIMD-parallel memory requests into one, which we call *allocation coalescing*. Allocation coalescing improves performance because subsequent stages of the allocator process only one memory allocation and deallocation per warp, whereas they may otherwise have to deal with up to 32 separate memory requests. Figure 3 illustrates the coalescing process in the case where four threads in the same warp request memory.

To keep track of allocated memory, additional space is requested when XMalloc makes a coalesced request on behalf of all the threads in a warp. As shown in Fig. 3, two kinds of metadata are added by XMalloc. First, a header for a coalesced memory area tracks bookkeeping information for the area, including the number of compartments in the coalesced area. During allocation, the number of compartments is initialized to be the number of threads in the warp that participated in the coalesced memory allocation request. In Fig. 3, there are four compartments.

Each compartment contains a header along with the memory area allocated to each thread. A compartment header contains the byte offset of the beginning of the memory area and a tag value. The tag is used during deallocation to distinguish a coalesced memory compartment from a memory area that was allocated without coalescing. Internally, XMalloc frees a coalesced memory area after all its compartments have been freed by the client application. To free a compartment, the client retrieves the offset from the compartment's header and uses it to find the beginning of the coalesced memory area. Then it decrements the count of compartments that is stored in

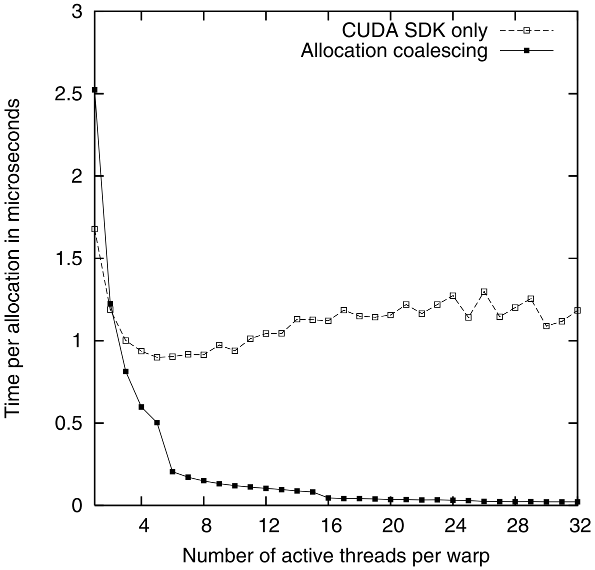
```
   void *xmcMalloc(unsigned int request_size)
   {
     unsigned int alloc_size;      // Number of bytes this thread will allocate
     __shared__ unsigned int allloc_sizes[MAX_WARPS];

     char *allocated_ptr;         // Pointer returned by malloc
     __shared__ char *allocated_ptrs[MAX_WARPS];

1    unsigned int warpId = getWarpIndex();

     // Number of threads in the current warp that are allocating
2    unsigned int num_threads = __popc(__ballot(1));

3    alloc_sizes[warpId] = sizeof(CoalesceHeader);

     // Determine how much memory to allocate
4    unsigned int offset =
       atomicAdd(&alloc_sizes[warpId], compartmentSize(request_size));

     // Pick one thread to perform allocation
5    alloc_size = offset == sizeof(CoalescedHeader) ? alloc_sizes[warpId] : 0;

     // Allocate memory (if alloc_size is nonzero)
6    allocated_ptr = buffered_malloc(alloc_size);

7    if (alloc_size) {
       // Send the address to all threads and initialize the coalesced block
8      allocated_ptrs[warpId] = allocated_ptr;

9      if (allocated_pointer != NULL)
         *(CoalesceHeader *)allocated_ptr = (CoalesceHeader){.count = num_threads};
    }

    // Get the address and initialize this thread's compartment
10  allocated_ptr = allocated_ptrs[warpId];
11  if (allocated_ptr == NULL) return NULL;

12  char *compartment = allocated_ptr + offset;
13  *(CompartmentHeader *)compartment =
      (CompartmentHeader){.offset = offset, .tag = COMPARTMENT};

    // Return the payload
14  return (void *)(compartment + sizeof(CompartmentHeader));
  }
```

Fig. 4 Implementation of memory allocation coalescing in XMalloc. For clarity, we omit the code that chooses whether to perform allocation coalescing

the header. The client that frees the last compartment, reducing the counter to zero, proceeds to call `free` on the coalesced memory area to return it to the free memory pool.

The XMalloc code is shown in Fig. 4. When called by client threads, XMalloc first calculates the number of threads that are participating in the allocation request in the current warp (line 2 in Fig. 4). XMalloc then calculates the compartment size for each thread (line 4). The `compartmentSize` function returns a size that is big enough for the compartment header and the requested memory, rounded up to a multiple of 8 bytes to maintain pointer alignment. Figure 3 assumes that there are four threads that participate in the allocation in the current warp and their requested memory sizes are

Fig. 5 Allocation throughput with and without allocation coalescing



56, 56, 40, and 120 bytes. The current XMalloc implementation uses 8 bytes for each compartment header. Therefore, the compartment sizes for these four threads are 64, 64, 48, and 128, as shown in Fig. 3.

The threads cooperatively sum their compartment sizes using atomic adds to a shared variable (line 4 in Fig. 4). Line 3 initializes the shared variable to be the size of the header for the coalesced memory area, 8 (bytes) in the current XMalloc implementation. This is illustrated as the number 8 feeding into the summation process in Fig. 3. The final value in this shared variable is the total amount of memory, including the headers, to allocate on behalf of all four threads. This is shown as 312 (bytes) in Fig. 3.

During the summation, each thread receives the old contents of the variable it adds to, i.e., the partial sum of memory requests so far, which determines the offset of each thread's compartment within the coalesced memory block. Then the thread whose compartment offset equals the size of the coalesced memory header (the thread whose compartment is the first in the coalesced memory area) calls the next stage of the allocator to obtain a coalesced memory area (lines 5 and 6 of Fig. 4).

The newly allocated memory area is then distributed to all the threads (lines 9–11), which initialize their own compartment within the memory area (line 12).

Coalescing is skipped when only one SIMD thread requests memory, as well as for large memory requests. In the former case, only one SIMD thread is active so coalescing offers no benefit. The latter case ensures that large memory areas are reclaimed immediately upon being freed. Uncoalesced memory areas also have a header created by XMalloc, containing a tag value to distinguish them from coalesced memory areas. The code is in the XMalloc implementation but omitted from Fig. 4 for simplicity.

Figure 5 compares the performance of memory allocation with and without allocation coalescing, with all other features disabled. Coalescing incurs overhead for deciding whether to coalesce and for managing header fields. The overhead results in a slowdown when only one thread allocates. Because coalescing n threads' allocations reduces the number of `malloc` calls, and thus the work performed, by a factor of n, we should expect the allocation time to be proportional to n^{-1}. The curve deviates from the expected shape due to variations in how fast the CUDA SDK's allocator processes different memory request sizes. Adding buffers (Sect. 5.2) insulates us from the SDK's performance variability and produces the expected performance curve. When all SIMD threads are allocating, allocation coalescing increases performance by a factor of 50.

5.2 Raising thread-level throughput with buffering

Even after coalescing SIMD-parallel memory allocation requests, the CUDA SDK's memory allocator can still be bottlenecked by memory requests. We further improve the allocator's performance by buffering some allocated memory blocks. Buffering is performed after allocation coalescing (in the call to `buffered_malloc` in Fig. 4). Buffering is like using multiple free lists [8] in the sense that it enables allocation and deallocation without going through a centralized data structure.

In order to find a memory request of the desired size, allocated blocks are classified into discrete size classes, and we create one or more buffers for each size class. Requested memory sizes are rounded up to the nearest size class. Memory requests larger than the largest size class are not buffered. A buffer is simply a fixed-capacity FIFO that holds pointers to free memory blocks. Memory allocations are satisfied from a buffer unless it is empty, in which case the CUDA SDK is invoked. Likewise, freed blocks are returned to a buffer unless it is full, in which case they are is freed through the CUDA SDK.

Space is maintained for 256 total free memory blocks in each size class, divided among one or more buffers. Figure 6 shows how throughput varies with the number of buffers. The graph shows the time per memory request that is processed by the buffering stage of the allocator. Since a coalesced memory request is processed as a single request, regardless of how many threads were involved, the speedup from processing fewer memory requests is not represented in this graph. The solid line is the same data as the solid line in Fig. 5. The curve is broken into several nearly-flat intervals. Vertical steps in the curve reflect differences in the CUDA SDK's throughput for different memory request sizes. Using a single buffer per size class yields a significant slowdown. By using a centralized FIFO, we have introduced a serialization point that slows down allocation. We can increase the allocation throughput by creating multiple buffer instances per size class. To load-balance memory requests across buffers, threads use a hash of their thread index, block index, and GPU clock to choose a buffer to access.

Using 16 buffers, we increase the allocator's throughput by about 5× when a single thread per warp is active, but incur a slight slowdown with 32 threads. In particular, whereas allocation coalescing alone performed poorly when 1 or 2 threads in a warp were allocating, the combination of coalescing and buffering improves performance for any number of allocating threads.

Fig. 6 Allocation throughput with different degrees of buffering

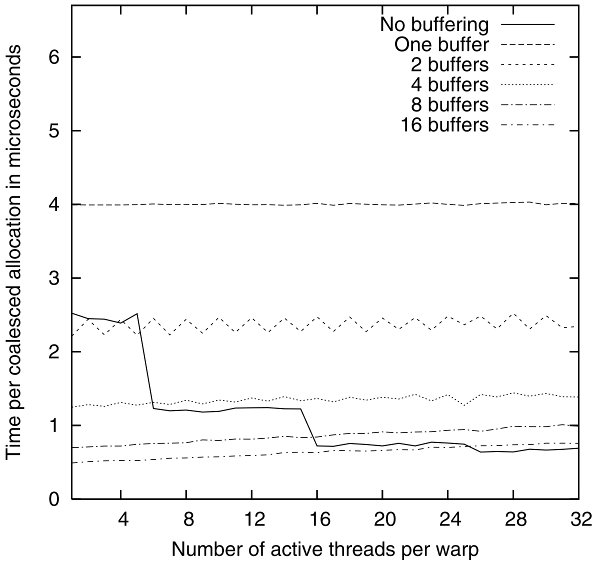
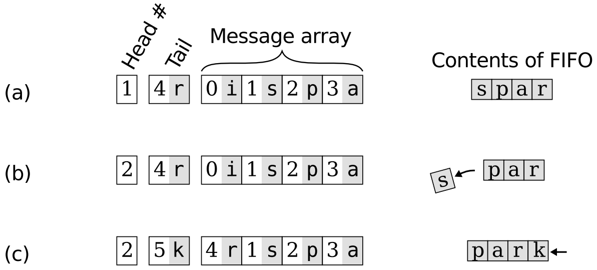


Fig. 7 FIFO data structure operations. Part (**a**) shows the state of a FIFO containing 4 elements. In part (**b**), an element is dequeued by incrementing the head number. In part (**c**), a new element is enqueued, updating the tail and one message in the array



5.2.1 *Buffer design*

Since the buffers are so heavily used, their design greatly affects the overall allocator performance. The buffers are fixed-capacity, lock-free FIFOs optimized to minimize memory traffic. In Fermi, this means minimizing the number of memory operations, all of which must go to the shared L2 cache; this yields a different design than for cache-coherent multiprocessors, where the cache coherence traffic is what matters [12]. A FIFO dequeue or enqueue requires four or five memory operations, respectively, when memory contention is low and the FIFO is not empty or full.

Rather than pointers or indices, messages are referenced using their *sequence number*. The nth message inserted into a FIFO has sequence number n. The number may wrap around modulo the 32-bit integer size. Sequence numbers identify where a message is stored, and also serve as pseudo-unique tags to avoid the ABA problem [4].

A FIFO data structure (Fig. 7) holds the sequence number of the head message, the entire tail message, and an array that holds up to 2^s messages for some fixed s. Each message contains its own sequence number and a 32-bit payload holding a pointer to a free block. To store 64-bit pointers in the payload field, we represent them as 32-bit

```
do { // Repeat until queue is empty or successfully updated
  head_num = queue->headNum;
  tail = queue->tail;

  // Fail if queue is empty
  if (tail.num - head_num + 1 == 0) return NULL;

  // Read queue head
  index = head_num % QUEUE_SIZE;
  return_value = head_num == tail.num ? tail.payload
                                      : queue->messages[index].payload;

  // Write new head index
} while (!CAS(&queue->headNum, head_num, head_num + 1));
```

Fig. 8 Pseudocode for a lock-free buffer dequeue operation

offsets relative to a reference address. A message with sequence number n resides at array index $n \bmod 2^s$. Note that the number of elements in the queue can be computed from the sequence numbers in the head and tail.

The FIFO manipulation functions ensure that threads observe a consistent state of the queue's head sequence number and tail value. Given those, it is possible to determine the sequence numbers and locations of messages in the queue. Threads verify that they are accessing the correct version of a message by checking its sequence number. All updates to a queue use compare-and-swap operations to replace a message or the queue head sequence number.

To dequeue a message (Fig. 8), a thread first reads the head number and the tail message. If the FIFO is empty, the dequeue operation fails. If the FIFO contains more than one element, then the head message is read from the array; otherwise the tail message (which has already been read) is the head message. Finally, the head number is atomically incremented. If the increment fails, then some other thread has dequeued the head, and the process restarts.

To enqueue a message (Fig. 9), a thread first reads the head number and the tail message. If the FIFO is full, the enqueue operation fails. Otherwise, the old tail message is transferred to the array. If the transfer fails, it is ignored; some other thread has already transferred the data. Finally the new tail message is written. If this write fails, then some other thread has enqueued a new tail, and the process restarts.

6 Conclusion

We have presented two memory allocator enhancements that dramatically magnify GPU dynamic memory allocation throughput. One enhancement utilizes SIMD-parallel thread execution to combine simultaneous memory requests. The other enhancement parallelizes allocation of frequently used blocks through the use of replicated buffers. These enhancements enable an allocator to keep up with concurrent memory requests generated from thousands of concurrently running threads.

```
do { // Repeat until queue is full or successfully updated
  head_num = queue->headNum;
  tail = queue->tail;

  // Fail if queue is full
  if (tail.num - head_num + 1 == QUEUE_SIZE) return false;

  // Write old tail into the message array
  index = tail.num % QUEUE_SIZE;
  old_message = queue->messages[index];
  if (old_message.num + QUEUE_SIZE == tail.num)
    CAS(&queue->messages[index], old_message, tail);

  // Write new tail
  new_tail.payload = value;
  new_tail.num = tail.num + 1;
} while (!CAS(&queue->tail, tail, new_tail));
```

Fig. 9 Pseudocode for a lock-free buffer enqueue operation

By making dynamic memory allocation fast, XMalloc reduces one of the barriers to its adoption in GPU programming. Dynamic allocation is still a new and rarely used feature on GPUs. Nevertheless, there is a trend toward more complex GPU codes requiring more complete support for general-purpose programming. We anticipate that, as this trend continues, GPU programmers will likely begin to take advantage of memory allocation capabilities.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. NSF-OCI07-25070.

References

1. Berger E, McKinley K, Blumofe R, Wilson P (2000) Hoard: a scalable memory allocator for multithreaded applications. In: Proceedings of the 9th international conference on architectural support for programming languages and operating systems, pp 117–128
2. Bigler B, Allan S, Oldehoeft R (1985) Parallel dynamic storage allocation. In: Proceedings of the international conference on parallel processing, pp 272–275
3. NVIDIA Corporation (2010) NVIDIA CUDA C programming guide
4. Dechev D, Pirkelbauer P, Stroustrup B (2010) Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In: Proceedings of the 13th IEEE international symposium on object/component/service-oriented real-time distributed computing, pp 185–192
5. Dice D, Garthwaite A (2002) Mostly lock-free malloc. In: Proceedings of the 3rd international symposium on memory management. ACM, New York, pp 163–174
6. Herlihy M (1991) Wait-free synchronization. ACM Trans Program Lang Syst 13(1):124–149
7. Huang X, Rodrigues C, Jones S, Buck I, Hwu W-M (2010) XMalloc: A scalable lock-free dynamic memory allocator for many-core machines. In: Proceedings of the 10th IEEE international conference on computer and information technology, pp 1134–1139
8. Iyengar A (1993) Parallel dynamic storage allocation algorithms. In: Proceedings of the 5th IEEE symposium on parallel and distributed processing, pp 82–91
9. Johnson T, Davis T (1992) Space efficient parallel buddy memory management. In: Proceedings of the 1992 international conference on computing and information, pp 128–132

10. Mellor-Crummey J, Scott M (1991) Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans Comput Syst 9(1):21–65
11. Michael M (2004) Scalable lock-free dynamic memory allocation. In: Proceedings of the ACM SIGPLAN 2004 conference on programming language design and implementation
12. Tsigas P, Zhang Y (2001) A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In: Proceedings of the 13th Annual ACM symposium on parallel algorithms and architectures. ACM, New York, pp 134–143