2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)

XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines

Xiaohuang Huang[1], Christopher I. Rodrigues[1], Stephen Jones[2], Ian Buck[2], Wen-mei Hwu[1]

University of Illinois at Urbana-Champaign Urbana, IL 61801[1]
NVIDIA Corporation[2]
xhuang22@illinois.edu, cirodrig@illinois.edu, stjones@nvidia.com, IBuck@nvidia.com, hwu@illinois.edu

***Abstract*—There are two avenues for many-core machines to gain higher performance: increasing the number of processors, and increasing the number of vector units in one SIMD processor. A truly scalable algorithm should take advantage of both. However, most past research on scalable memory allocators scales well with the number of processors, but poorly with the number of vector units in one SIMD processor. As a result, they are not truly scalable on many-core architectures.**

In this work, we introduce our proposed solution through the design of XMalloc, a truly scalable, efficient lock-free memory allocator. We will present (1) our solution for transforming traditional atomic compare-and-swap based lock-free algorithm to scale on SIMD architectures, and (2) a hierarchical cache-like buffer solution to reduce the average latency of accesses to non-scalable or slow resources such as main memory in a many-core machine.

We implemented XMalloc as a memory allocator on an NVIDIA Tesla C1060 GPU with 240 processing units. Our experimental results show that XMalloc scales very well with growth in both the number of processors and the number of vector units in each SIMD processor. Our truly scalable lock-free solution achieves 211 times speedup compared to the common lock-free solution.

***Keywords*-malloc, CUDA, GPGPU, lock-free, async-signal-safe, availability, scalability**

I. Introduction

Dynamic memory allocators are widely used in sequential and multi-threaded applications. However, dynamic memory allocation is not available in parallel code in many-core programming models such as CUDA and OpenCL. This poses a barrier to supporting higher-level programming languages such as C++ that rely more heavily on memory allocation. For shared-memory programs with hundreds or thousands of concurrent threads, the lack of support for dynamic memory management often emerges as a programmability obstacle. Without a safe and scalable memory allocator, programmers are forced to estimate the dynamic memory usage and pre-allocate the memory space in advance, which can be error prone, inefficient, or both.

The XMalloc memory allocator scales with the number of processors and the number of vector units in a processor. We provide a solution to transform traditional CAS (compare-and-swap) based lock-free algorithms to scale with increasing SIMD processing unit count. In XMalloc, this solution is employed to aggregate SIMD-parallel memory allocation requests into one request. In this way, our algorithm scales with the number of vector units in each processor. Then we present the design of an atomic CAS-based lock-free algorithm to get excellent memory allocation scalability with increasing processor count.

II. Background and Related Work

On shared-memory multi-core or many-core machines, many processing units share the same data structure. To ensure the consistency of these concurrent objects, there is a need to synchronize their accesses. In such a machine the programmers typically have to use synchronization primitives such as semaphores, monitors, guarded statements, and mutex locks, but algorithms based on locks do not scale well. Consequently the operations of different processes on a shared data structure should appear to be serialized: if two operations execute simultaneously, the system guarantees the same result as if one of them is arbitrarily executed before the other [4].

Atomic instructions perform read-modify-write atomic operations on one word residing in memory. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can modify this address until the operation is complete. The widely used lock-free solutions [11] [10] [13] are based on atomic operation based.

Research into lock-free synchronization began over two decades ago. For example, Figure 1(a) is an example of novel CAS-based lock-free algorithm, attributed to early work by Lamport [7]. The impossibility and universality results of Herlihy [6] had significant influence on the theory and practice of lock-free synchronization, by showing that atomic instructions such as CAS and LL/SC (load-linked/store-conditional) are more powerful than others such as Test-and-Set, Swap, and Fetch-and-Add, in their ability to provide lock-free implementations of arbitrary object types. Michael and Scott [9] review practical lock-free algorithms for dynamic data structures in light of recent advances in lock-free memory management.

A lot of memory allocation research in sequential or multi-threaded environments has been done before [12] [2] [14]. Ptmalloc [5], developed by Wolfram Gloger and based on Doug Lea's dlmalloc sequential allocator [8], Hoard

978-0-7695-4108-2/10 $26.00 © 2010 IEEE
DOI 10.1109/CIT.2010.206

[1], developed by Emery Berger are lock-based memory allocators. Mostly Lock-free malloc, designed by Dice and Garthwaite [3], proposes a partly lock-free allocator.

Hoard [1] uses multiple processor heaps in addition to a global heap. Each heap contains zero or more superblocks, and each superblock contains one or more blocks of the same size. Statistics are maintained individually for each superblock as well as collectively for the superblocks of each heap. When a processor heap is found to have too much available space, one of its superblocks is moved to the global heap. When a thread finds that its processor heap does not have available blocks of the desired size, it checks if any superblocks of the desired size are available in the global heap. Threads use their thread IDs to decide which processor heap to use for malloc. When a thread frees a memory block, it returns the block to its original superblock and updates the fullness statistics for the superblock as well as the heap that owns it. Typically, malloc and free require one and two lock acquisitions, respectively.

Michael [10] presents a completely lock-free memory allocator that is immune to deadlock and guarantees progress regardless of whether some threads are delayed or even killed. It uses atomic CAS instructions to implement a lock-free algorithm to allocate memory resources. It arranges some high-level structures from Hoard into 64-bit words in order to update them with atomic operations. It use heaps to maintain superblocks, and allocates fixed-size blocks from superblocks. In our XMalloc design, we maintain a similar data structure to Hoard's superblocks and we borrow the atomic CAS based lock-free idea from Michael's paper.

III. Overview of XMalloc's Design

The XMalloc memory allocator is designed to furnish a heavily multithreaded program with high-throughput access to a shared memory pool. Parallel allocators commonly use thread-private data structures to achieve scalability [1]. However, our implementation platform, the Tesla C1060, has very little space for per-thread data: when its 16 kilobytes of scratchpad memory is divided among of 1024 concurrent threads, only 16 bytes of local storage is available per thread. Consequently, XMalloc uses only globally shared data structures. Since global memory bandwidth is a limited resource, XMalloc sustains high allocation throughput by economizing on memory traffic as well as avoiding contention for shared data structures.

XMalloc's data structures are organized to reduce memory traffic. Metadata in memory blocks managed by XMalloc is stored in a four-byte or eight-byte (depending on the block's type) header that can be atomically modified, avoiding the extra memory traffic associated with acquiring and releasing a lock. XMalloc's free lists, which are its most heavily accessed data structure, are lock-free FIFOs requiring only two and four memory operations, respectively, for successfully enqueuing and dequeuing items.

To increase allocator throughput, XMalloc manages a hierarchical memory pool. The hierarchy amplifies parallelism by aggregating many parallel memory requests into fewer, larger requests that are allocated, subdivided, and distributed to consumers. Section IV describes how SIMD-parallel memory requests are aggregated. This step is performed local to an SIMD processor and reduces the number of memory requests that traverse global data structures. Another stage of the allocator amortizes the cost of heap allocation by allocating large blocks and subdividing them in anticipation of future memory requests. Section V describes this allocation strategy.

To use XMalloc, a program must call XMalloc's initialization function, which allocates and initializes a block of GPU global memory that will serve as the heap. Then, XMalloc's `malloc` and `free` functions can be called from CUDA kernels in exactly the same way as `malloc` and `free` functions in the C standard library.

IV. Achieving SIMD Scalability

Most many-core GPU architectures run parallel threads simultaneously on SIMD processors to achieve higher computing throughput. SIMD designs reduce hardware cost by exploiting data parallelism. However, SIMD-parallel operations that access the same resource must still be serialized. A lock-free algorithm will naturally serialize itself, but may induce severe slowdown. We observed this effect in a simple version of XMalloc. Our scalable allocation algorithm avoids this effect by combining memory requests, using one thread to allocate on behalf of the entire SIMD processor.

A. The Problem with Atomic CAS-based Lock-free Algorithms

Many scalable parallel algorithms follow the pattern shown in Figure 1(a), where a critical section begins by reading a shared variable and ends by updating the variable with a CAS operation. If the update CAS fails, meaning that the variable was modified by another thread, then the section is repeated. When several threads execute the critical section simultaneously, one thread will finish first and update the variable. The lagging threads' CAS operations will all fail, and these threads will repeat the critical section.

SIMD execution of CAS-based critical sections produces pathological performance. Because all SIMD threads on one processor execute in lockstep, they execute the critical section simultaneously. One thread completes the critical section while the others fail and retry. An N-wide SIMD processor will loop through the critical section at least N times, as slow as if all SIMD threads ran serially. What is worse, because multiple SIMD threads are active in each iteration, the shared variable will be accessed by CAS instructions at least $\frac{(N+1)*N}{2}$ times, generating more memory traffic than serial execution would. When memory bandwidth is a bottleneck, this will slow execution. The

```
do { // Repeat until CAS succeeds
     oldvalue = *addr;
     // Critical section
     newvalue = f(oldvalue);
}while(CAS(addr, oldvalue, newvalue));
     (a)

do { // Repeat until CAS succeeds
     oldvalue = *addr;
     // Critical section
     newvalue = SIMD_reduce(f, oldvalue);
     if (ThreadID == 0)
         *success = CAS(addr, oldvalue, newvalue);
} while (!(*success));
     (b)
```

Figure 1. A typical CAS-based lock-free algorithm shown in part (a) is not scalable with SIMD width because most executions of the CAS operation wil not succeed. The SIMD scalable version in part (b) locally accumulates the effect of all CAS operations, then uses a single thread to commit the result.

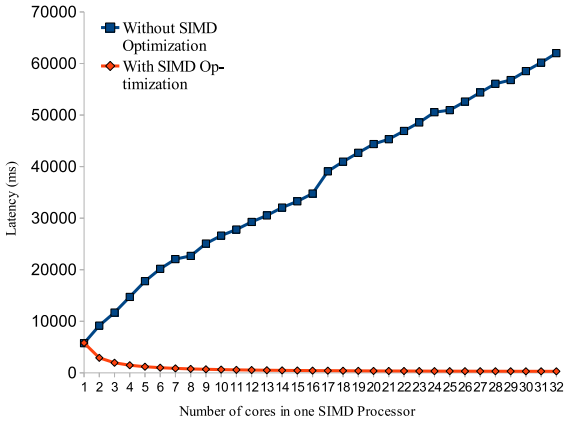

Figure 2. SIMD scalability comparsion between SIMD-scalable lock-free memory allocation and a SIMD-oblivious version.

more vector units each SIMD processor has, the slower execution becomes.

The blue curve in Figure 2 illustrates the issue, using our atomic CAS-based lock-free allocator as an example. In this benchmark, we generate 1 million threads, each of which makes one call to `malloc` and one call to `free`. We simulate varying SIMD width by running the computation on a subset of threads in each warp. The execution latency of the simple CAS-based lock-free algorithm rises rapidly with SIMD width. Its performance is about 11 times slower than serial execution if we use SIMD processors with 32 vector units. In contrast, the execution latency continues to decrease when we use our proposed scalable algorithm, as shown in the orange curve in Figure 2.

B. Scalability Transformation for SIMD Processors

The cause of poor SIMD scalability of the simple CAS code is that all SIMD threads in a processor execute a

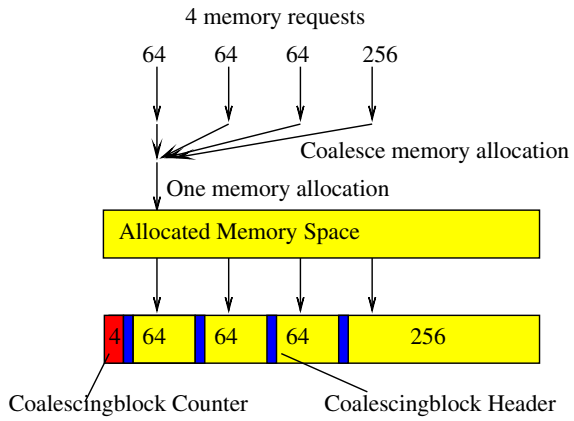

Figure 3. SIMD memory allocation coalescing. Allocated memory is shown in yellow; allocator-managed metadata is red or blue. In the figure, four threads request memory spaces of 64 bytes, 64 bytes, 64 bytes and 256 bytes simultaneously. XMalloc coalesces them into one memory request and obtains a memory space. All threads select and initialize their piece of the new memory space simultaneously.

critical section simultaneously, thus inducing worst-case contention for synchronization variables. For critical sections that consist of reads followed by a single CAS operation to a synchronization variable (Figure 1(a)), the problem can be solved by combining the transactions from all SIMD threads into one. Since all transactions update the same variable, they can be combined by making a local copy of the variable, applying all transactions sequentially to the local copy, and committing the local copy back to global storage by issuing a single CAS from one SIMD thread (Figure 1(b)).

The point of contention in our unscalable `malloc` function is a free list. The critical section is re-executed as one free memory block at a time is dequeued from the free list. Figure 3 illustrates how allocation is transformed to eliminate this problem, using four SIMD-parallel memory requests. The threads first cooperatively sum their requested memory sizes using a prefix sum scan. Then, one thread obtains a memory area of this size. This step involves a lock-free operation, but since it is performed by only one thread, there is no SIMD-level memory contention. The new memory area is then divided among the different threads, using the prefix partial sums to identify each thread's piece. We call each thread's piece of the original memory area a *Coalescingblock*. XMalloc reserves a header in each *Coalescingblock* and stores the number of *Coalescingblocks* at the beginning of the original memory block; these are used for deallocation as described in Section V-B. The SIMD scalable version runs up to 211 times faster than the SIMD-oblivious version.

V. Two Level Allocation Algorithm

The growth in core count of contemporary many-core machines has outpaced the growth in the throughput of

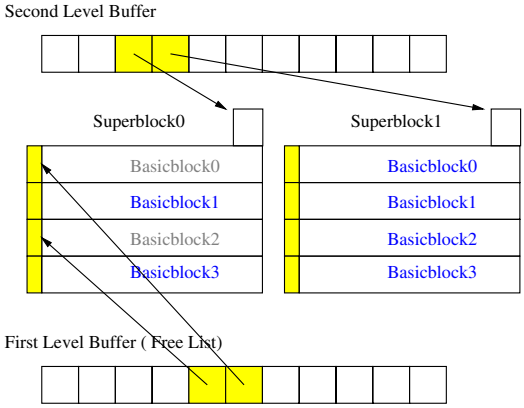

Figure 4. Example of two level buffer. The pointers of Basicblocks are stored in First Level Buffer and the pointers of non empty Superblocks are stored in the Second Level Buffer.

their shared memory systems. Computation on this class of machines can easily become bottlenecked by memory throughput, especially when a massive number of threads are performing the same task concurrently. After SIMD-level coalescing of memory requests, XMalloc uses a two-level allocation strategy to further reduce the average allocation overhead.

A. Allocation

In the common case, memory requests are satisfied from a free list that holds unused, previously allocated memory areas. These memory areas are called *Basicblocks*. All small memory requests are coerced to one of a set of statically determined sizes, each having its own free list. Large requests bypass this mechanism and are allocated directly from *Memoryblocks*, described later. These steps of allocation are performed by one thread in a SIMD thread group. In Figure 3, these steps are performed after coalescing and the allocated memory space is the body of a *Basicblock* or *Memoryblock*.

XMalloc's free lists are fixed-capacity, lock-free FIFOs optimized to minimize memory traffic. They are implemented with preallocated arrays. Allocating a *Basicblock* and returning one to a free list requires two and four memory operations, respectively, when memory contention is low and the free list is not empty or full.

Bursts of memory requests can quickly empty a free list. Performance under bursty allocation depends on how quickly it can be refilled. When no free *Basicblocks* are available, XMalloc allocates a large memory area called a *Superblock*, subdivides it into 32 equally sized *Basicblocks*, and fills these into the free list. Filling many *Basicblocks* at a time ensures that most requests are satisfied from the free list. Memory traffic is reduced compared to performing 32 individual allocations: creating each *Basicblock* inside the allocated *Superblock* requires only a single memory access

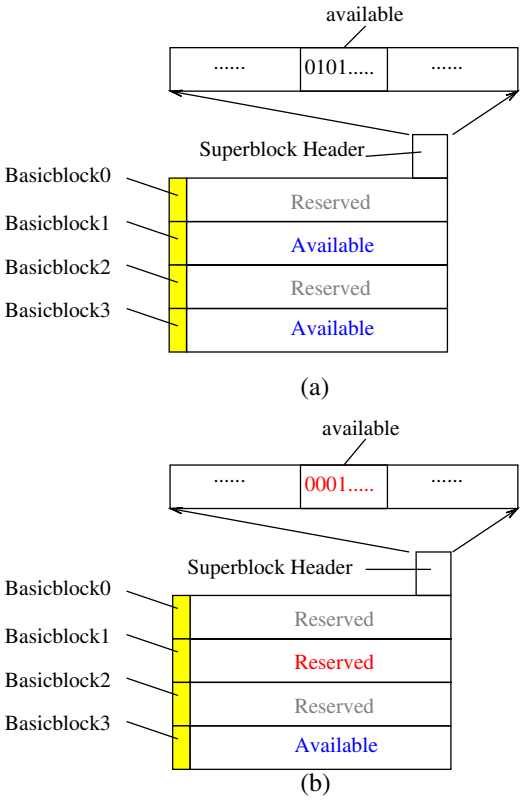

Figure 5. Superblock Structure: (a) This is a *Superblock* containing 4 *Basicblocks*. *Basicblocks* 1 and 3 are still available, so the first 4 bits of *available* are 0101. (b) Reserving *Basicblock* 1 from the *Superblock* causes the changes marked in red. We use one 64-bit atomic CAS to make this update.

to initialize the header. Moreover, memory contention for allocating new *Superblocks* from the free memory pool is lessened compared to performing 32 individual allocations. Creating *Basicblocks* from *Superblocks* constitutes the first level of allocation.

As *Basicblocks* are allocated and freed, a pool of partly-empty *Superblocks* will accumulate. These are stored in buffers that, like the free lists, each hold *Superblocks* of one size. Figure 4 shows these two structures. Free lists provide direct access to a small number of free *Basicblocks*, while the second-level buffer provides indirect access to all other free *Basicblocks*. If a free list is empty, the allocator first tries to refill it with free *Basicblocks* from a buffered *Superblock*. A new *Superblock* is allocated only if the buffer is empty. Figure 5 shows a *Superblock* from Figure 4 in more detail. A bit mask in the *Superblock*'s header records which *Basicblocks* are free but not already in a free list.

The heap from which new *Superblocks* and very large memory requests are allocated is segmented into free and allocated *Memoryblocks* (Figure 6). *Memoryblocks* are subdivided on demand when memory is allocated from this level. *Memoryblock* headers include boundary tags [14] that

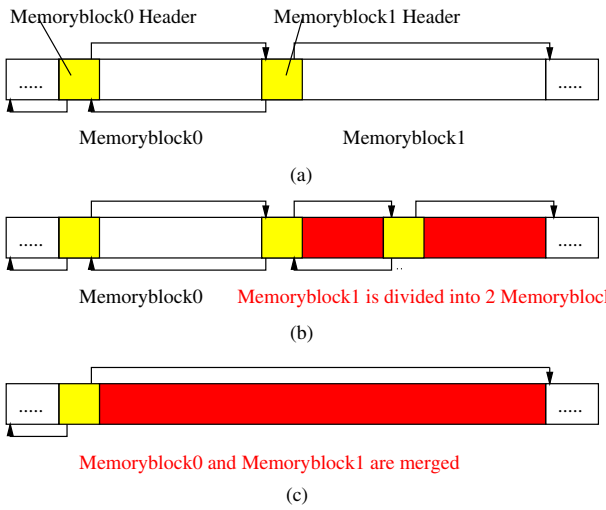

Figure 6. Memoryblock Structure: (a) The header of each *Memoryblock* has pointers to its two neighbor *Memoryblocks*; these pointers form a doubly-linked list. (b) *Memoryblock* 1 is subdivided by creating a new header within the block and updating the neighbor pointers of the old block and its successor. (c) *Memoryblocks* 0 and 1 are merged into one *Memoryblock* by updating neighbor pointers.

are used to find a *Memoryblock*'s immediate neighbors and merge free blocks. Allocating a *Memoryblock* is the second level of allocation, and is relatively slow: the allocator searches a list of free *Memoryblocks* to find a match, then updates or creates the boundary tags of the allocated block and its neighbors.

B. Deallocation

A memory space returned by a call to `malloc` typically resides in a *Coalescingblock*. It may also reside in a *Basicblock* or *Memoryblock* if SIMD coalescing or first-level allocation were skipped. When a program no longer needs the space, it deallocates the space with a call to `free`. In some cases, this empties the container's parent, which causes it to be freed as well. The action taken to free a block depends in its type, which is determined from a tag in its header. These actions are summarized below.

- A *Coalescingblock* is freed by updating the fill statistics of its parent block, as shown in Figure 7(a). The parent block contains a count of the number of in-use *Coalescingblocks* it contains. This count is decremented each time a *Coalescingblock* is freed. When it reaches zero, the parent block is freed and its children's storage is reclaimed (Figure 7(b)).
- A freed *Basicblock* is put into the free list for blocks of its size, if there is room. Otherwise, it is returned to its parent *Superblock* and the *Superblock* is enqueued into the *Superblock* buffer.
- A *Superblock* is freed by freeing its parent block.
- A *Memoryblock* is freed by marking its header as "free", merging it with adjacent free blocks, and returning it to the free *Memoryblock* list.

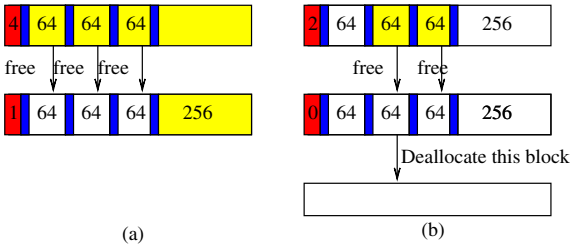

Figure 7. SIMD memory deallocation coalescing. Allocated memory is shown in yellow; free memory is white; allocator-managed metadata is red or blue. In the figure, (a) During memory de-allocation, if only parts of a block is de-allocated, we only decrease the value of *counter*. (b) If all parts of a block are deallocated, we will deallocate the whole block and return it to the FIFO buffer.

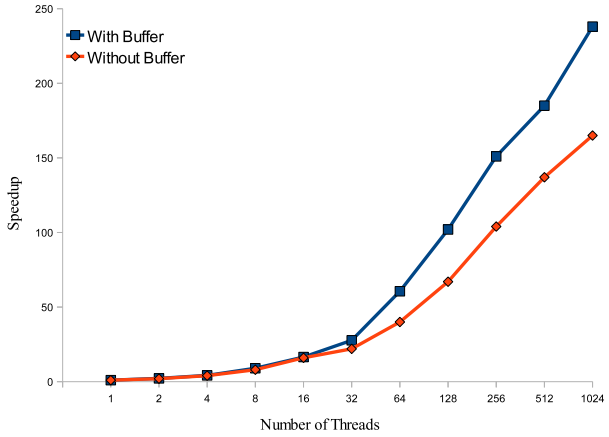

Figure 8. Allocation speedup for 1 million 8-byte memory requests with different numbers of threads in parallel. Speedup is normalized to the sequential execution time. The blue and orange lines show the performance scaling of XMalloc with and without two-level allocation, respectively.

VI. Experimental Results and Summary

Our experimental performance results were obtained on an NVIDIA Tesla C1060 using the 64-bit CUDA 3.0 Beta compiler in Linux. We use a benchmark *MilThreads*, which creates 1 million threads. Each thread allocates 8 bytes of memory, writes 8 bytes to that space, then deallocates it.

At first, we compare XMalloc's allocation latency with that of `cudaMalloc`, the function provided by NVIDIA's API to allocate memory space on GPU. Comparing to `cudaMalloc`, which is a sequential API, XMalloc even runs with only one thread on the GPU is faster, for example, for 4 kb memory allocation, the average latency of `cudaMalloc` is 0.166ms and XMalloc is 0.05ms.

As a measure of allocation throughput, we measure the average per-thread parallel allocation time while varying

the number of concurrently active threads from 1 to 1024. Each thread requests 8 bytes of memory. With 32 threads or fewer, a subset of SIMD threads in one processor is active. Above 32 threads, multiple SIMD processors are used with 32 active threads in each. One thread needs $40\,\mu$s to finish allocation. For 32 threads, the average allocation time per thread is $1.3\,\mu$s, and 1024 threads only need $0.15\,\mu$s per thread.

Figure 2 shows how latency varies with increasing number of vector units in each SIMD processor for the *MilThreads* benchmark. The orange line corresponds to the XMalloc SIMD-scalable solution. It becomes 20 times faster as the number of vector units per processor increases to 32. The blue line shows the SIMD-oblivious XMalloc solution for comparison. With 32 vector units per processor, it is 10 times *slower* than a serial processor. Overall, our SIMD scalable solution is 211 times faster than the traditional scalable version.

Figure 8 shows the speedup trend as the the number of threads increases. The orange line is the performance with *Superblock* allocation disabled. Without *Superblocks*, a new *Memoryblock* is allocated for each *Basicblock*. SIMD coalescing is used in both cases. From the test data, we can see that use of *Superblocks* is important to the scalability of the algorithm.

From the experimental results, our XMalloc algorithm scales both with the number of cores and the SIMD width. It is suitable for many-core architecture such as GPUs.

REFERENCES

[1] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2000.

[2] Jeff Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[3] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the 3rd international symposium on Memory management*, pages 163–174, New York, NY, USA, 2002. ACM.

[4] Hui Gao and Wim H. Hesselink. A formal reduction for lock-free parallel algorithms. In *Proceedings of the 16th International Conference on Computer-Aided Verification*, pages 44–56, 2004.

[5] Wolfram Gloger. Dynamic memory allocator implementations in linux system libraries. http://www.dent.med.uni-muenchen.de/ wmglo/.

[6] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[7] Leslie Lamport and Robert W. Taylor. Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems*, 8:305–310, 1990.

[8] Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[9] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30, New York, NY, USA, 2002. ACM.

[10] Maged M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, 2004.

[11] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[12] James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20:421–431, 1974.

[13] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 134–143, New York, NY, USA, 2001. ACM.

[14] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A critical review. In *Proceedings of the International Workshop on Memory Management*, volume 986 of *LNCS*. Springer Verlag, 1995.