PROCEEDINGS OF SCIENCE

A New Non-Blocking Approach on GPU Dynamical Memory Management

Yu-Shiang Lin [a], **Chun-Yuan Lin** [b], **Jon-Yu Lee**∗ [b]

[a] *Department of Computer Science, National TsingHua University, Hsinchu 300, Taiwan*

[b] *Department of Computer Science, Chang Gung University, Taoyuan 333, Taiwan*

E-mail: coldfunction@gmail.com, cyulin@mail.cgu.edu.tw, a5000ml@gmail.com

Dynamic memory allocation is a very important and basic technique implemented on modern computer architecture. In the massively parallel processor (MPP) architecture such as Graphics Processing Units (GPUs), many threads try to send allocation or deallocation requests to system in the same time, which could cause the issue of synchronization or race condition. In this paper, we design a new signal model with signal queue to handle the interaction of threads. Based on the signal model, we involve the concept of buddy memory to construct a non-blocking parallel buddy system. Our design have no synchronization problem and adopt a simpler structure implemented than before. Finally, we implement our model in real hardware and experimental results show that the model have better performance than other methods.

2013 International Workshop on Computational Science and Engineering,
14-17 October 2013
National Taiwan University, Taipei, Taiwan

∗Speaker.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike Licence.

1. Introduction

Dynamic memory allocation let processes to obtain a permitted address from memory pool, usually handled by a memory management. The mechanism of memory management shares or protects the content of memory using independently. There have been many research and development about memory management on single or multi-cores CPUs architecture in the past. In recent years, there are more and more high performance computing demand with varied applications. In addition to traditional CPUs architecture, the MPP architectures, such as GPU plays an important role of cooperator performing a high intensive computing. When we port the dynamic memory allocation from CPU to the massively parallel cores environment, it may suffer some performance issues in massive cores such like latency of memory transaction or threads synchronization that can be a bottleneck and reduce total computing power [1]. Therefore, we need a suitable memory management to handle dynamic memory allocation on MPP architecture.

There have been some researchers developed the allocator on MPP architecture for dynamic memory allocation [2, 3, 4, 5]. In the past, most developments of dynamic memory allocation on parallel computing architecture adopt free list based on superblock structures. In our work, we will first introduce a new signal model with signal queue to handle the interaction of threads. The buddy memory will be involved to the signal model as a non-blocking method. Finally, we implement **3L-Allocator** with our signal model on real GPU hardware and the experimental results shows that our model get better performance in MPP architecture than other methods.

This paper is organized as follows. In Section 2, we first discuss the related work on dynamic memory allocation for CPU and MPP architecture. In Section 3, we will introduce our signal model and the non-blocking parallel buddy system. Analysis and experimental results are shown in Section 4. Finally, Section 5 is conclusions.

2. Related Work

In recent years, some researchers have noted the importance of developing the dynamic memory allocator on MPP architecture. In 2010, Huang, *et al.* developed XMalloc [3], the first scalable memory allocator on GPU. In 2012, Steinberger, *et al.* propose the ScatterAlloc [4], the performance achieved 10 times faster than XMalloc. In 2013, Widmer, *et al.* [5] proposed a voting algorithm to increase SIMD scalability named FDGMalloc that claimed to get the best performance than before. Although these proposed methods have better performance than CUDA built-in allocator, studies also established under the concept of free list and superblock structures for chunks accessing.

There have been many methods proposed to handle the dynamic memory allocation, such as free list, fast fits, and buddy system. The buddy system has lesser external fragmentation in comparison to other methods and take a constant amortized time for allocating or deallocating memory block [7, 8]. Although many variant buddy system have been proposed on CPU environment, the parallel buddy system on MPP architecture haven't been proposed so far. Therefore, we construct a new signal model and involve an efficient buddy system on MPP architecture, we will discuss more details about our model in next section.

PoS(IWCSE 2013)071

2.1 Dynamic memory allocation on CPUs

Dynamic memory allocation is a classic problem of computer science, and has been used for many decades. In 1995 Wilson *et al.* [9] provide a complete introduction and overview of dynamic memory allocation in detail. The mechanism of memory allocation could be modified to get more efficient, if it port from single process to multiple processes on multi-cores with shared memory. That would be a challenge to solve a serious bottleneck when multiple processes try to allocate memory concurrently. When multi-processes or multi-threads access the global shared memory, for keeping the shared data consistent, the mutual exclusion mechanism required to handle. Hoard, a scalable memory allocator proposed by Berger *et al.* [10] maintains per-processor heaps and one global heap to avoid false sharing and increase scalability. Michael [11] proposed a lock-free allocator to operate the atomic instruction such like Compare-and-Swap (CAS) that complete avoid lock. The other lock-free method was proposed by Hudson *et al.* [12], they implemented the McRT-malloc, a non-blocking transaction aware memory allocator.

2.2 Dynamic memory allocation on MPP architectures

A massively parallel architecture, meaning that it possibly take many synchronization for different memory requests with threads. However, too much synchronization similar to access by serializes and cause the benefit of parallel processing decreasing. Therefore, memory management in MPP architecture need to redesign. The first introduced the dynamic memory allocation on MPP architecture is XMalloc [3]. XMalloc refers Hoard's method [10] to use the concept of superblocks and using the atomic CAS operation by Michael's lock-free allocator. Later, Steinberger *et al.* proposed the ScatterAlloc [4] to get better performance than XMalloc. They proposed a new method to reduce the overhead of memory requests causing collisions on hashing. FDGMalloc was recently proposed by Widmer *et al.* [5], base on the concept of superblocks, FDGMalloc try to solve the bottleneck of SIMD scalability, they reduce the amount of memory requests by their voting algorithm. In their results show that the FDGMalloc got the better performance than other allocator on MPP architecture in the past.

3. Signal Model

While threads execute *Allocate()* or *Free()* functions, threads need to access the free list. In our work, we regard the free list as a signal queue that can pass different signal values by threads. Each item of free list named a slot that the thread send or receive signal from a slot, and the slot points the state of memory block. We construct a finite-state automaton (shown in Fig. 1) to describe the *Allocate()* state of each thread wishes to access a slot in the free list. There have five mainly states to describe the behavior of threads negotiate with slots, they are INIT, EMPTY, CAPTURED, ADDRESS and OVERFLOW. These states are changed by different signals triggered. The thread claims a signal from a slot, the slot has signal values stored in SIG_EMPTY, SIG_CAPTURED, SIG_OVERFLOW and SIG_ADDRESS. We will discuss how to extend this signal model as non-blocking parallel buddy system in following paragraphs.

PoS(IWCSE 2013)071

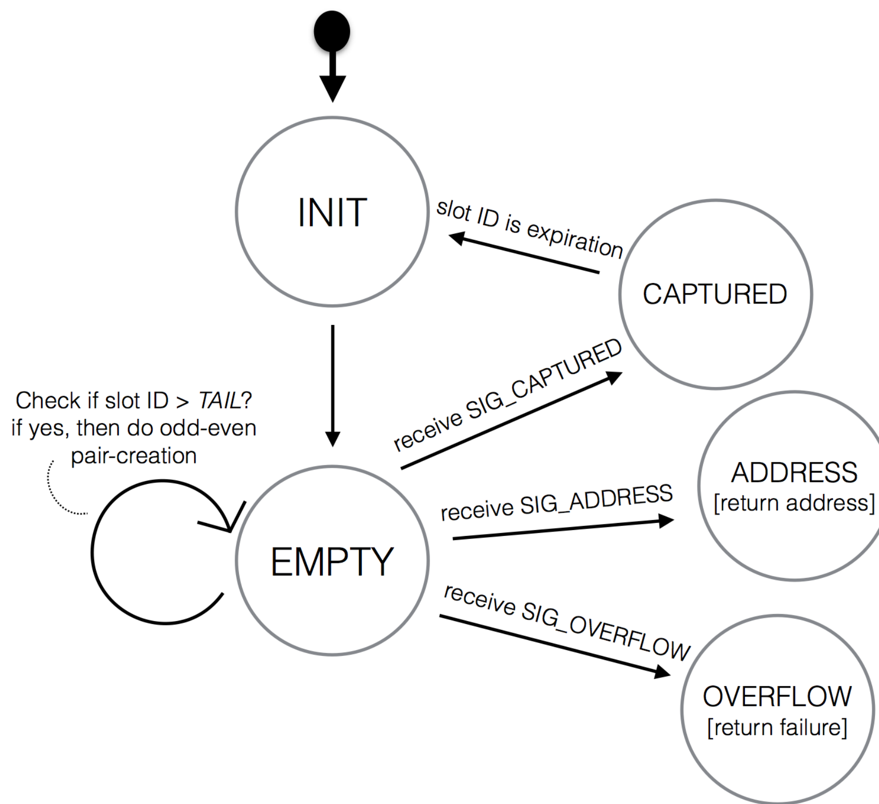

Figure 1: The finite-state automaton of allocation.

PoS(IWCSE 2013)071

3.1 Buddy Memory

First we introduce our buddy memory, in contrast to single free list, the buddy memory is a variable-size-block memory system, for managing on variable size memory system, a natural extension from fixed-size-block memory is constructed by multiple free lists. These free lists record the block size which grow up with 2^k, where k can start from specific positive integer to increase with different levels. We call the level k with 2^k block size point to memory pool in each item. The 2^k block size is current level, 2^{k+1} is upper level and 2^{k-1} is lower level. *Allocate()* and *Free()* in each free list of buddy memory inherited from the single free list, steps of functions we define as:
Allocate():

1. Find the first free list with record size larger than request size.

2. Take a item from the target free list.

3. Record the level of multiple free lists to the level table (LT).

4. If the target free list is empty, allocate two resources from upper list splits.

Free():

1. Find the target free list to free (lookup LT).

2. Return the released address to the target free list.
3. When an address have returned to free list from previous step, if its buddy is also in the free list, then combine them and return to upper level.

In buddy memory, the *Allocate()* take a arithmetic operation as *ceil(log(requested size))* to determine which level access. In the step 4 of *Allocate()* named pair-creation, if the pair-creation happened, it would take extra O(h) time in worst, where h equal log(maximum block size / minimum block size) is a constant. The *Free()* operation also take a constant time to find the target free list, when a address returned, we try to find their buddy which split from upper level by *Allocate()* before. The buddy address can be calculated by exclusive OR of address and block's size. While the buddy found, we combine both of them and free up to upper level.

3.2 Non-blocking Parallel Buddy System

Base on the signal model and buddy memory, we design a new model let multiple threads access the buddy memory. In our work, we define free list as a standard circular queue in our buddy memory, and we place *HEAD* and *TAIL* as global shared pointers to each level in buddy memory. Initially, every items in each free list are set to SIG_EMPTY except for the first level has a SIG_ADDRESS with the location *HEAD* points, SIG_ADDRESS is shown as an address value with the corresponding slot in our system. In addition to the last level, *HEAD* and *TAIL* pointers are start from the position 0 in each level. If the closed interval [*HEAD*, *TAIL*] exists, then threads will be allowed to receive a slot signal from it. If the thread received a SIG_ADDRESS value from a slot, then we claim that the thread successfully request a memory block and return the address for directly access in the future. If the thread release the memory block allocated before, then send the allocated address as SIG_ADDRESS to the target free list in the specific slot. The method of *Allocate()* and *Free()* works as follows.

Allocate(): The finite-state automaton is shown in Fig. 1, while threads start from INIT state, threads find the target free list and execute single atomicADD operation for booking a new slot with unique ID per thread, then the thread enter in EMPTY state and listen the signal from this slot. There are different paths while the thread receive different signals in EMPTY state:

- SIG_ADDRESS: The thread returns the SIG_ADDRESS value as a address in ADDRESS state (allocate successfully).
- SIG_OVERFLOW: The thread returns failure in OVERFLOW state (memory resources is not enough).
- SIG_CAPTURED: The slot ID is expiration and restart from INIT state (SIG_CAPTURED was sent by *Free()*).
- SIG_EMPTY: Check if the slot ID exceed TAIL? If yes, then execute the even-odd pair-creation.

The odd-even pair-creation for threads could be odd or even working thread in each round, see Fig. 2, odd threads execute single atomicADD let [*HEAD*, *TAIL*] add 2 new slots at the end, then odd

PoS(IWCSE 2013)071

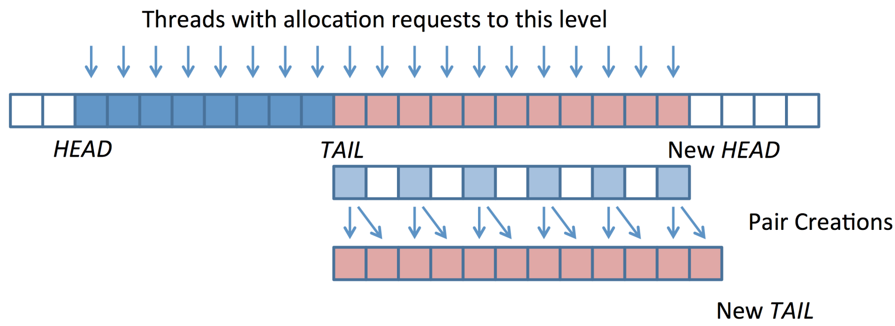

Figure 2: Threads in EMPTY state, if the slot ID > *TAIL*, then do odd-even pair-creations. Odd threads add new slots and eve threads wait to receive the address in single round.

threads recursively execute the *Allocate()* in upper levels until return two SIG_ADDRESS in new slots, while even threads wait other threads to send the SIG_ADDRESS to its new slot. If the odd thread receive SIG_EMPTY in this time, then keep in EMPTY state and change to even thread in next round and vice versa.

Free(): Assume the allocated address is held on the thread. The thread will try to send the address as SIG_ADDRESS to the slot when it execute *Free()*. The method works as follows steps:

1. Store address A into free list.
2. Calculate the buddy address with $B = A$ *XOR block's size*.
3. Check if B is in the free list, if not then return.
4. Try to remove min(A, B) from free list and set the corresponding slot with SIG_CAPTURED in single atomicCAS operation, if fail then return.
5. Try to remove max(A, B) from free list and set the corresponding slot with SIG_CAPTURED in single atomicCAS operation, if fail then set A = min(A, B) and go to step 1.
6. Free min(A, B) to upper level.

Follow above steps, we know the thread must ensure that the buddy address whether freed or not, and in the same time, the other thread may check its buddy address, too. If there exists two threads which one thread's freed address equal other thread's buddy of freed address in the same time, it may happen on one of them already freed address to upper level, but the other thread believe that it still hold on its address. To avoid this race condition happened, we design the order of address taken is consistent. In step 4, threads always remove the smaller address at first, then remove the other bigger in step 5, finally free the smaller to upper level recursively in step 6.

To handle the directly access in *Free()*, the memory pool content of buddy address record the slot id and the thread can directly use the slot ID to check the value of the slot. Therefore, we add a step 0 before step 1, the thread store the slot ID in memory pool. Note that we cannot arbitrarily change the order of step 0 and step 1. Because once the slot is set to SIG_ADDRESS,

PoS(ISCSE 2013)071

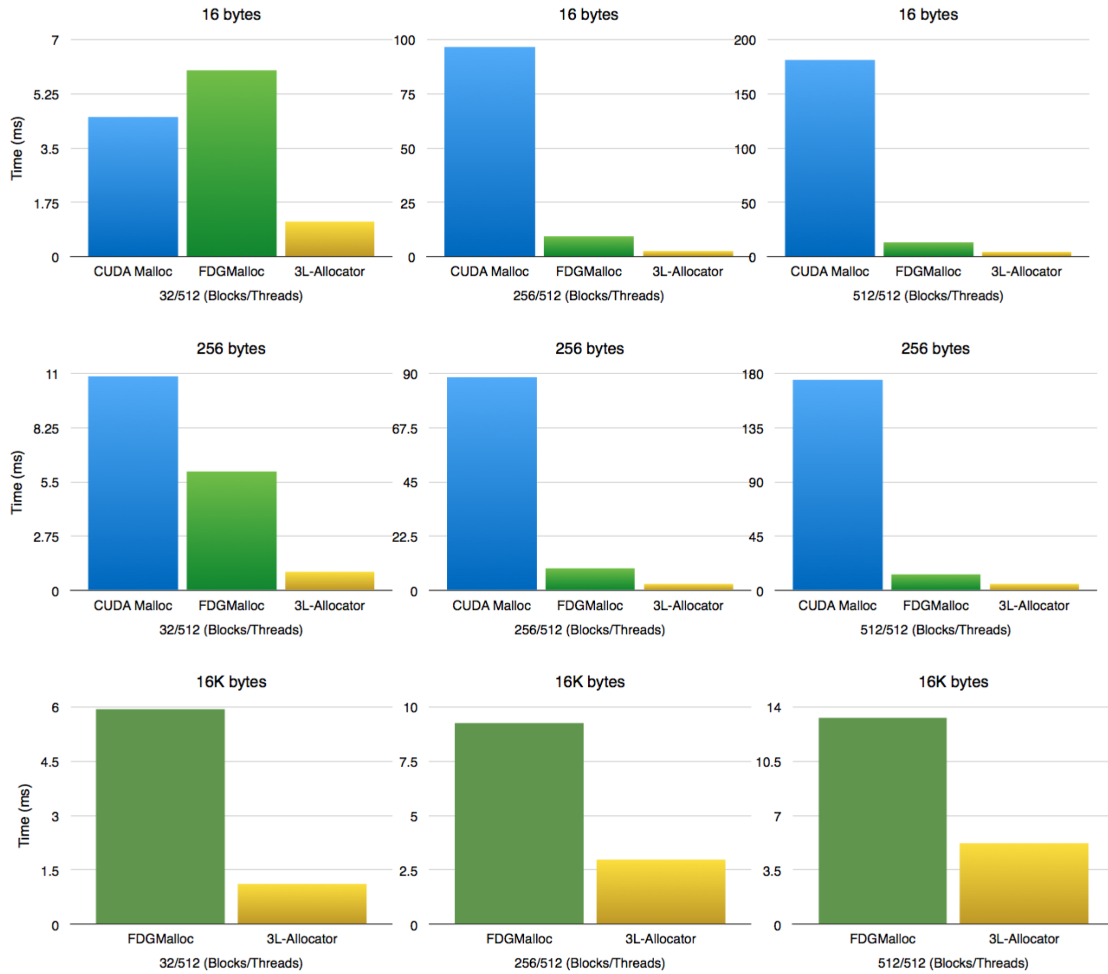

Figure 3: The figure shows the performance comparison of our proposed 3L-Allocator, CUDAMalloc, and FDGMalloc [5]. The execute time of each round of allocating and deallocating memory. In the test run with 16K bytes, the CUDAMalloc failed to execute, thus we omitted it in this case.

the SIG_ADDRESS may be moved to other slot by other thread, and then the corresponding slot ID stored in memory pool would be error.

4. Experimental Results

Our experiments environment were performed on a linux PC with Intel Core i7 920 CPU, 6G RAM, NVIDIA Tesla K20 with 2496 cores and 5GB GDDR5. We compare with the FDGMalloc [5] with different allocation sizes and different numbers of CUDA threads. In Fig. 3, the size of thread blocks fixed on 512 threads per block, different numbers of thread blocks from 32 to 512, and a allocation size set {16, 256, 16384} bytes. In this test, the kernel function call *Allocate()* and *Free()* functions in a single time, that is each thread executed their *Allocate()* and *Free()* once, then exited kernel immediately. The CUDA Malloc is a build-in CUDA toolkit allocator and the FDGMalloc claimed that have better performance than other parallel allocator proposed before, therefore we compare with FDGMalloc. In Fig. 3, all the execute time of our 3L-Allocator are

PoS(IWCSE 2013)071

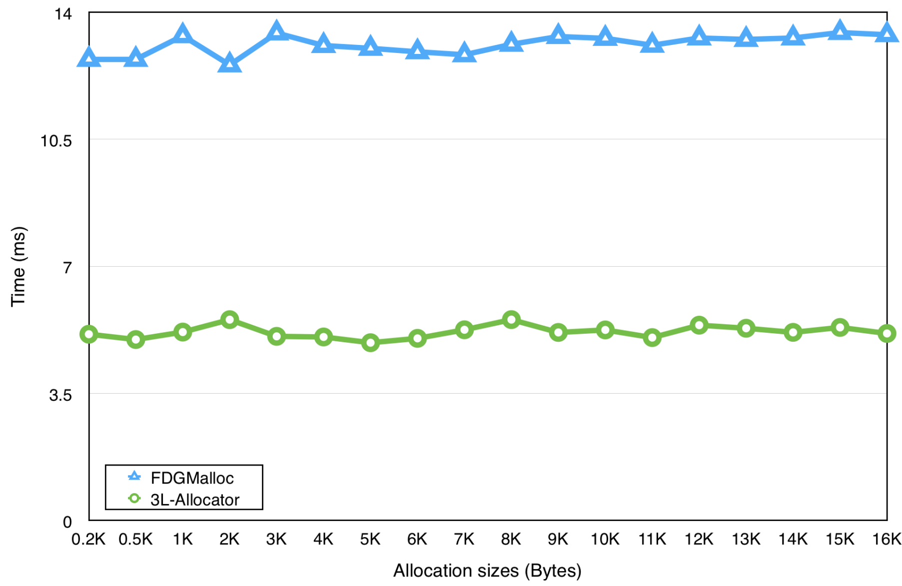

Figure 4: The figure shows that 3L-Allocator spend lesser time to execute *Allocate()* and *Free()* operations in single period. The kernel function with 512 threads per block × 512 thread blocks and allocation size from 0.2K bytes to 16K bytes.

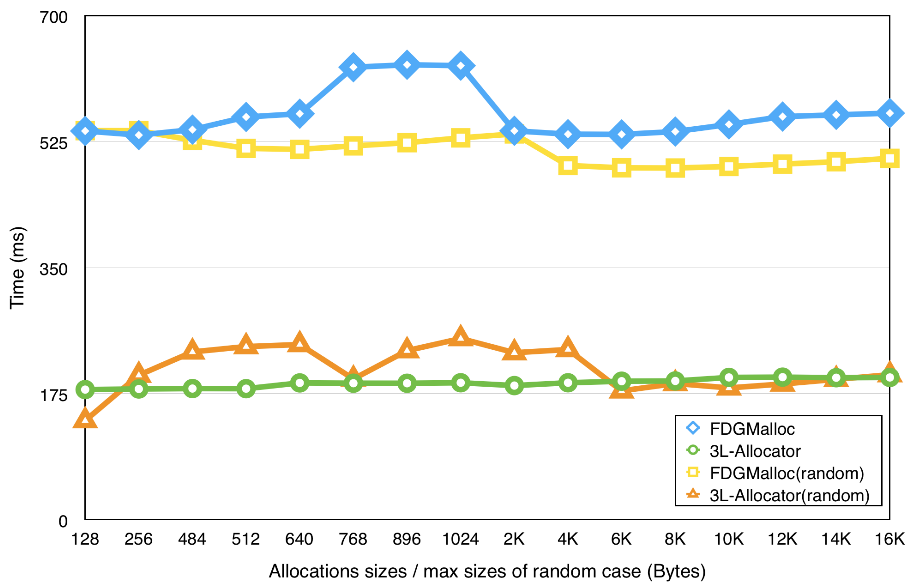

Figure 5: The figure shows results of two case between 3L-Allocator and FDGMalloc, one is for executing all the same allocations sizes with each thread, the other is for random generate sizes of memory requests. The kernel function is launched by 512 threads per thread block and total 512 thread blocks. All threads execute 100 times in the kernel function with *Allocate()* and *Free()* concurrently.

PoS(IWCSE 2013)071

lesser than CUDA Malloc and FDGMalloc, and the average speedup is approximately 29 times faster than CUDA Malloc and up to 3.3 times faster than FDGMalloc in these cases.

For clearly prove that 3L-Allocator have better performance, in Fig. 4, shown the different allocation sizes from 0.2 KB to 16 KB and launch the kernel function with 512 threads per block $\times$ 512 thread blocks. All the *Allocate()* and *Free()* operations only execute once of each thread in the experiment of Figure 4. Because we allocate a fixed size memory with each memory request, 3L-Allocator always access the same level of buddy memory. While many pair-creations are finished in EMPTY state, threads get the allocated address. Threads make memory requests in constant time and release it also in directly access.

For measuring the impact of serial allocate and deallocate memory during time, we let threads executing *Allocate()* and *Free()* operations in many times of once kernel launch. In Fig. 5, we design two cases, case one allocate the same fixed-size memory with each thread during 100 times iteratively. Each time of iteration execute *Allocate()* and *Free()* operations, individually. The case two also let each thread execute 100 times iterations of *Allocate()* and *Free()*, the difference is the same fixed-size memory changed to create the variable-size of randomly generate between each thread. The randomly generator in kernel function is applied by CUDA CURAND Library [13]. In Fig. 5, 3L-Allocator is faster than FDGMalloc in these two cases, the average speedup is approximately 2.9 times in case one, and 2.5 times in case two. Because the feature of defragmentation in buddy system, the system only take little overhead for compaction of memory that keep the *Allocate()* and *Free()* in constant time. In Fig. 5, we find that some random set of memory requests could cause some rise and fall of the curve in 3L-Allocator, the surmise is in many distributed random requests may cause the different recursively times for pair-creation or reduce the times of restart when threads allocate.

5. Conclusion

In this paper, we develop a new signal model to handle the dynamic memory allocation on MPP architecture. In our model, we take the item of free list as a slot. Threads pass the signal by the slot, that is a non-blocking send/receive communication between threads. Base on the non-blocking signal model, we involve the buddy system as a non-blocking parallel buddy system. We implement the non-blocking parallel buddy system as 3L-Allocator and compare with CUDA Malloc and FDGMalloc on GPU. In comparable cases, 3L-Allocator is approximately 20 to 60 times speedup faster than CUDA Malloc and 2 to 3 times speedup faster than FDGMalloc. In these results of our experiments show the 3L-Allocator with well scalability and keeps a small range of execute time in many cases. In addition to benefits of performance, our model implement is simpler than other allocator and without complex data structure and algorithm.

References

[1] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," *in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 296-307.

[2] NVIDIA CUDA Tookit 3.2 download page: https://developer.nvidia.com/cuda-toolkit-32- downloads.

PoS(IWCSE 2013)071

[3] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu, “Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines,” *in Proc. IEEE ICCIT*, 2010.

[4] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, “Scatteralloc: Massively parallel dynamic memory allocation for the GPU,” *in Proc. InPar*, 2012.

[5] S. Widmer, D. Wodniok, N. Weber, M. Goesele, “Fast Dynamic Memory Allocator for Massively Parallel Architectures,” *in GPGPU-6 Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013, pp. 120-126

[6] A. Gidenstam, M. Papatriantafilou, and P. Tsigas, “Allocating Memory in a Lock-Free Manner,” *Proc. 13th Ann. European Symp. Algorithms (ESA ’05)*, vol. 3669, pp. 242-329, Oct. 2005.

[7] G.S. Brodal, E.D. Demaine, J.I. Munro, “Fast allocation and deallocation with an improved buddy system,” *Acta Inform*. 41 (4) (2005) 273-291

[8] D.C. Defoe, S.R. Cholleti, R.K. Cytron, “Upper bound for defragmenting buddy heaps,” *in Proc. of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005, pp. 222-229.

[9] P.R. Wilson, M.S. Johnstone, M. Neely, D. Boles, “Dynamic storage allocation: A survey and critical review,” *in Proceedings of the International Workshop on Memory Management*, 1995.

[10] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. “Hoard: a scalable memory allocator for multithreaded applications,” *SIGPLAN Not*., 2000.

[11] M. M. Michael. “Scalable lock-free dynamic memory allocation,” *in Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35-46, 2004.

[12] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. “Mcrt-malloc: a scalable transactional memory allocator,” *in Proc. ISMM*, 2006

[13] CUDA CURAND Library: http://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/CURAND_Library.pdf

PoS(IWCSE 2013)071