MAM: A Memory Allocation Manager for GPUs

Can Aknesil

Computer Science and Engineering
Koç University
Istanbul, Turkey
caknesil13@ku.edu.tr

Didem Unat

Computer Science and Engineering
Koç University
Istanbul, Turkey
dunat@ku.edu.tr

***Abstract*—Nowadays, GPUs are used in all kinds of computing fields to accelerate computer programs. We observed that allocating memory on GPUs is much slower than that of allocating memory on the CPUs. In this study, we focus on decreasing the device memory allocation overhead of GPUs. The overhead becomes significantly larger as the size of the memory segment that is being allocated increases. In order to achieve the lowest possible overhead during device memory allocations in GPUs, we develop a thread safe memory management library called Memory Allocation Manager (MAM) for CUDA. Our library removes the allocation and the deallocation overheads occurring during the runtime, and makes the performance of CUDA programs independent from the device memory allocation size.**

***Index Terms*—GPU, CUDA, device memory allocation, performance improvement.**

I. Introduction

The current trend of computer system is to parallelize the hardware and the software programs running on it, rather than producing faster processor cores. With this trend, the usage of GPUs has been increasing in all areas of computing fields. Currently, GPUs are used for many purposes such as for graphics, machine learning, and high performance computing. Since GPUs are used extensively, it is very important to keep the performance of programs using GPUs as high as possible.

In this study, we focus on decreasing the device memory allocation overhead of GPUs. This allocation overhead is large, especially when the allocation size is large. Thus, applications requiring repetitive or large allocations may reduce the overall performance. As our measurements indicate, shown in Figure 1, the overhead associated to device memory allocations increases almost linearly for allocations larger than 1MB. The similar result can be observed from a study of a group from the University of Virginia [2].

We develop a thread safe memory management library, called Memory Allocation Manager (MAM), in order to remove the allocation overhead on GPU memory. Our library provides an abstraction layer between the programmer and the memory management module of CUDA [1] environment. In order to allocate and free memory using MAM, the programmer should call procedures defined in the MAM API rather than directly calling the regular `cudaMalloc()` and `cudaFree()` procedures. In this paper, we first introduce the MAM API, then its implementation. Lastly we present its performance and compare it against regular memory routines provided by CUDA.

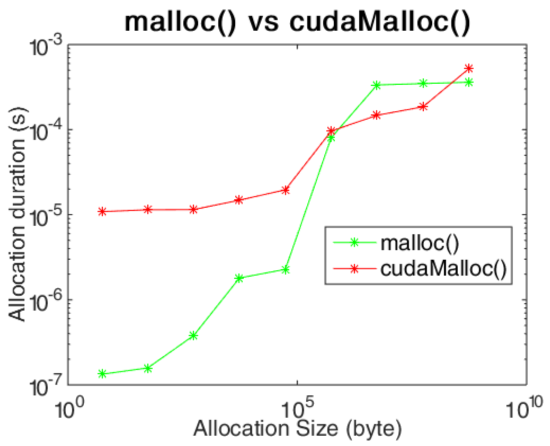

Fig. 1. `malloc()` and `cudaMalloc()` durations vs allocation size

II. Application Programming Interface

MAM API contains five procedures that a programmer can use for memory management. During the creation of the MAM environment, a large *chunk* of memory is allocated by MAM on the device memory, which will be explained in detail later.

- `MAM_Create(maxSize)`: Creates the MAM environment. Takes a parameter that defines the size of the chunk of memory that will allocated during the creation.
- `MAM_Create_auto()`: Creates the MAM environment. Allocates the largest possible chunk of memory during the creation.
- `MAM_Destroy()`: Destroys the MAM environment.
- `MAM_CudaMalloc(&ptr, size)`: Allocates specified size of device memory.
- `MAM_CudaFree(ptr)`: Frees the previously allocated device memory.

MAM can be used in three different ways:

1) By specifying the chunk size during its creation:

```
MAM_Create(maxSize);
MAM_CudaMalloc(&ptr, size);
...
MAM_CudaFree(ptr);
```

```
MAM_Destroy();
```

2) Without specifying the size of the chunk during its creation: In this case, the largest possible size is used. The largest possible size is allocated by performing multiple allocation operations by decreasing the allocation size exponentially starting from the size of the device memory until one of the allocations succeeds. We take this approach because it is not possible to allocate entire device memory.

```
MAM_Create_auto();
MAM_CudaMalloc(&ptr, size);
...
MAM_CudaFree(ptr);
MAM_Destroy();
```

3) Without explicit creation: In this case lazy creation occurs. `MAM_Create_auto()` is called automatically when `MAM_CudaMalloc()` is first called. When all the memory allocated using MAM API is freed, MAM automatically destroys itself.

```
MAM_CudaMalloc(&ptr, size);
...
MAM_CudaFree(ptr);
```

III. Implementation

During the creation of MAM, a large and continuous *chunk* of memory is allocated on the device memory. The size of the chunk is expected to be equal or smaller than the maximum size of the device memory that will be used by the CUDA program at a time instance. The pointers to the segments of this large chunk of memory will be returned by MAM during the allocation process. Every object existing in MAM environment other than the *chunk* live in the host memory.

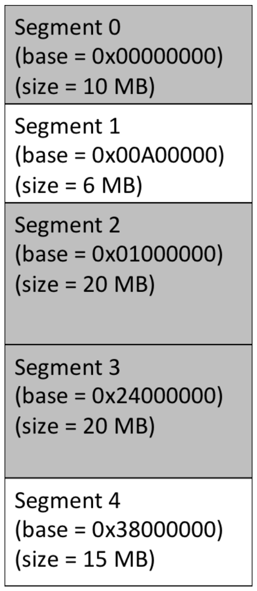

Fig. 2. An example of the chunk

A chunk is divided into segments that are either being used or not being used (empty) by the programmer. Figure 2 represents an example of the chunk at a time instance. The example chunk is a continuous memory and consists of 5 segments.

In the MAM environment, each segment is represented by a `segment struct` instance in the host memory. The `segment struct` contains mainly, a pointer to the beginning of the physical segment located in the device memory, a size attribute, and a flag indicating whether it is being used by the program or not. The `segment struct` declaration is as follows:

```
struct segment {
        void *basePtr;
        size_t size;
        char isEmpty;
        /* attributes related to data
        structures */
        ...
};
```

A. Internal Data Structures

In MAM, there are two data structures that store the `segment struct` instances. The first data structure is a tree that stores all the segments. It is sorted according to the base pointer of each segment that points to the beginning of the represented physical memory. It is used when the programmer calls `MAM_CudaFree(*void)` in order to find the corresponding segment using the pointer parameter.

The second data structure is a tree-dictionary that stores only the empty segments and it is sorted according to their size attribute. It is used to find an empty segment at an equal or greater size than the desired allocation size during the `MAM_CudaMalloc(**void, size_t)` call. In both data structures, a red-black tree is used since it is a balanced tree.

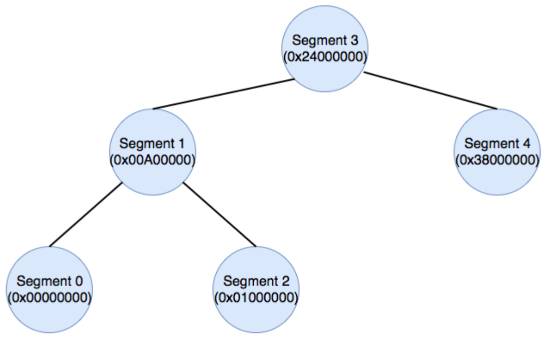

Fig. 3. Pointer tree

Figure 3 and Figure 4 show the corresponding data structures for the example chunk shown in Figure 2. At that instance, there are 3 segments that are allocated by the user (Segment 0, 2, and 3), and 2 segments that are not (Segment

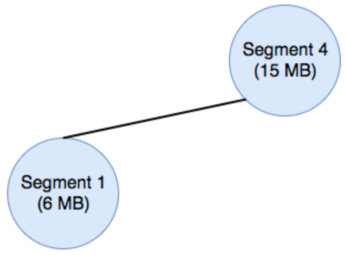

Fig. 4. Size tree-dictionary

1, and 4). Figure 3 shows the time instance of the pointer-tree. It contains all the segments and it is sorted by the base pointers of each segment. Figure 4 shows the time instance of the size-tree dictionary that contains all the empty segments. It is sorted according to the sizes of each segments.

IV. Memory Management

Allocation and deallocation calls to MAM API respectively starts and ends the usage of segments located in the chunk, which was previously allocated. Since the total physical memory that will be used is allocated as a large chunk during the creation of MAM environment, `MAM_CudaMalloc()` and `MAM_CudaFree()` calls do not actually allocate or free any physical memory but imitate the process. This is the main reason why MAM introduces much less overhead than the CUDA memory management module. The initialization of the MAM environment is slow but the initialization is performed once at the beginning; once MAM is created, all the memory management calls are faster. Next, we will discuss the allocation and deallocation implementations in MAM.

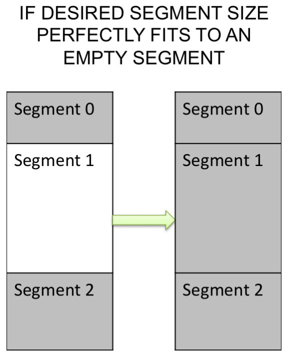

Fig. 5. Allocation diagram 1

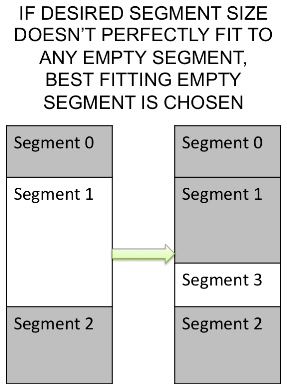
IF DESIRED SEGMENT SIZE DOESN'T PERFECTLY FIT TO ANY EMPTY SEGMENT, BEST FITTING EMPTY SEGMENT IS CHOSEN
Segment 0
Segment 1
Segment 2
Segment 0
Segment 1
Segment 3
Segment 2

Fig. 6. Allocation diagram 2

A. Allocation

When the programmer calls `MAM_CudaMalloc()`, MAM searches the smallest empty segment whose size is equal or greater than the desired segment using the size tree-dictionary. If there is an empty segment with the same size, MAM marks it as filled. If the segment that is found is larger then the desired segment, a new segment that represents the non-allocated empty part is created. This procedure is illustrated in Figure 5 and Figure 6. In Figure 6, Segment 3 is a newly created segment.

Algorithm 1 MAM Allocation Algorithm - $\mathcal{O}(\log n)$

```
 1: procedure ALLOCATE
 2:     Find a best-fitting empty segment from the tree-dictionary $\mathcal{O}(\log n)$
 3:     Mark the segment as filled $\mathcal{O}(1)$
 4:     if The segment perfectly fits $\mathcal{O}(1)$ then
 5:         Remove segment from tree-dictionary $\mathcal{O}(\log n)$
 6:     else
 7:         Resize it $\mathcal{O}(1)$
 8:         Remove it from tree-dictionary $\mathcal{O}(\log n)$
 9:         Create a new empty segment $\mathcal{O}(1)$
10:         Insert it in pointer-tree & tree-dictionary $\mathcal{O}(\log n)$
11:     end if
12:     Return the base pointer of filled segment $\mathcal{O}(1)$
13: end procedure
```

The algorithm of MAM allocation is shown in Algorithm 1. The complexities of all steps in the algorithm is shown at the end of each step. The overall complexity of this allocation algorithm is $\mathcal{O}(\log n)$, where n is the number of segments existing in the chunk.

B. Deallocation

When the programmer calls `MAM_CudaFree()`, MAM first marks the segment that is being freed as empty. Then merges the empty segment with previous and next segments if they are also empty. This procedure is illustrated in Figure 7.

The algorithm of MAM deallocation is shown in Algorithm 2. The overall completely of the deallocation algorithm is also $\mathcal{O}(\log n)$, where n is the number of the segments in the chunk.

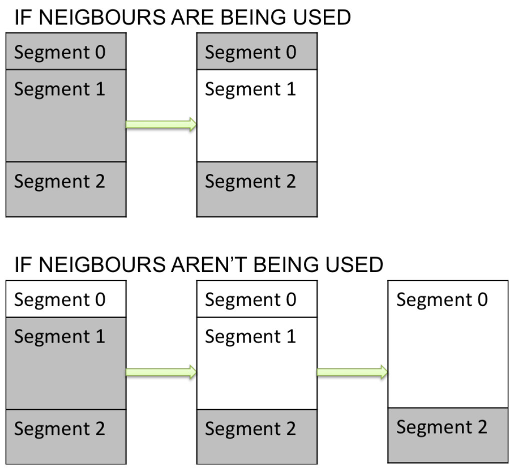

Fig. 7. Deallocation diagram

The allocation and deallocation algorithms are used in the implementation of MAM API, respectively in the procedures `MAM_CudaMalloc()` and `MAM_CudaFree()`. Thus, the

Algorithm 2 MAM Deallocation Algorithm - $\mathcal{O}(\log n)$

```
1: procedure DEALLOCATE
2:     Find the segment in the pointer-tree $\mathcal{O}(\log n)$
3:     Mark the segment as empty $\mathcal{O}(1)$
4:     Get previous and next segments $\mathcal{O}(\log n)$
5:     if the previous segment is empty $\mathcal{O}(1)$ then
6:         Remove the segment being newly emptied from pointer-tree and tree-dictionary $\mathcal{O}(\log n)$
7:         Destroy the segment being newly emptied $\mathcal{O}(1)$
8:         Resize previous segment $\mathcal{O}(1)$
9:         Replace it in tree-dictionary $\mathcal{O}(\log n)$
10:        Assign it to the variable stored the destroyed segment $\mathcal{O}(\log n)$
11:    end if
12:    //repeat the similar procedure for next segment.
13: end procedure
```

complexities of both allocation and deallocation are $\mathcal{O}(\log n)$ in terms of the number of segments.

V. Performance Evaluation

We demonstrate the performance of MAM in two ways: in terms of the allocation size, and in terms of the number of previously allocated segments. We used Tesla K20m as the GPU testbed, Linux 2.6.32-431.11.2.el6.x86_64 as the kernel and NVCC 7.0, V7.0.27 as CUDA Compilation Tools in all of our tests.

In order to measure the performance in terms of allocation size, we created a histogram that stores the time elapsed during allocation for different allocation sizes from 1Byte to 1GigaByte. We filled the histogram by allocating the device memory parts of random sizes over and over again until there is no more space.

Figure 8 and Figure 9 show the performance comparison between regular `cudaMalloc()` and `MAM_CudaMalloc()`, and `cudaFree()` and `MAM_CudaFree()`, respectively, in terms of allocation size.

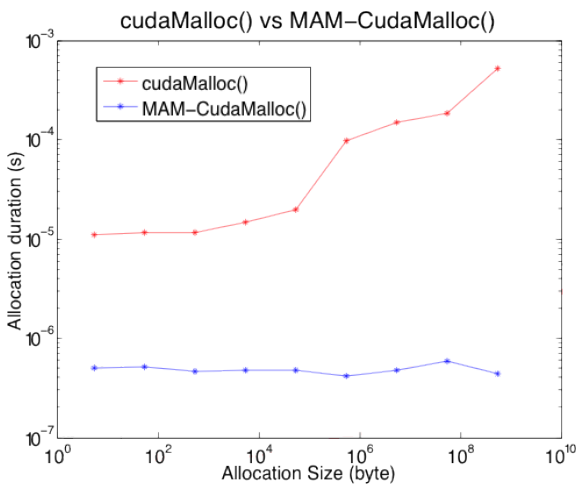

Fig. 8. `cudaMalloc()` vs `MAM_CudaMalloc()` comparison

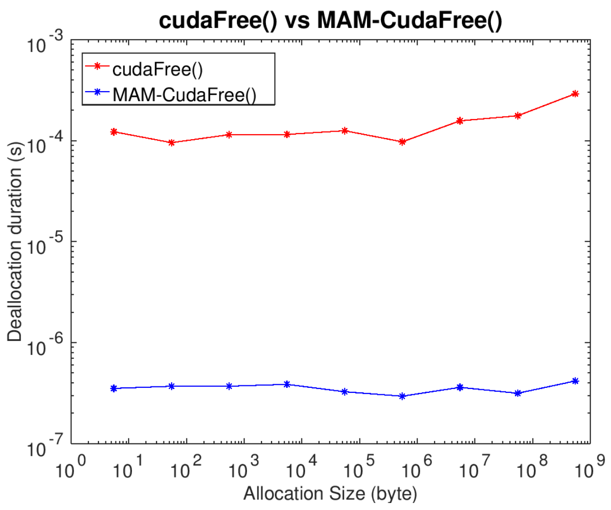

Fig. 9. `cudaFree()` vs `MAM_CudaFree()` comparison

As shown in these figures, while allocation duration of `cudaMalloc()` increases swiftly, the duration of `MAM_CudaMalloc()` stays almost constant. MAM removes the allocation and deallocation overhead and makes the performance of allocations independent from the allocation size. This result was expected because MAM moves the entire physical memory allocation overhead to the creation of MAM environment from individual allocations. Even though the initialization of MAM is slow, once it is initialized, there is no significant overhead caused by memory allocations or deallocations. Because, there is no physical memory allocation after the creation of MAM and the allocation size has no effect on the complexity of MAM.

The second performance measurement is based on the total number of existing segments during allocation or deallocation. This is meaningful because the size of data structures used in the MAM environment increases with the number of segments. In order to measure the performance in terms of the number of previously allocated segments, we measured the time elapsed during the first allocation after allocated variable number of segments. In this measurement, the allocation size was random between 1Byte to 10Bytes, sufficiently small so that we could make large number of allocations up to 10^7 before the device memory is full. Figure 10 shows the performance comparison between regular `cudaMalloc()` and `MAM_CudaMalloc()` in terms of the number of previously allocated segments.

According to this performance measurement, MAM is faster than CUDA and the duration of MAM allocation increases more slowly than actual CUDA allocation for the number of previously allocated segments larger than 100. This is the result of the fact that allocation algorithm of MAM is $\mathcal{O}(\log n)$, since the red-black tree used in MAM environment is a balanced tree.

We should also mention that when the programmer makes a very large number of small device memory allocations,

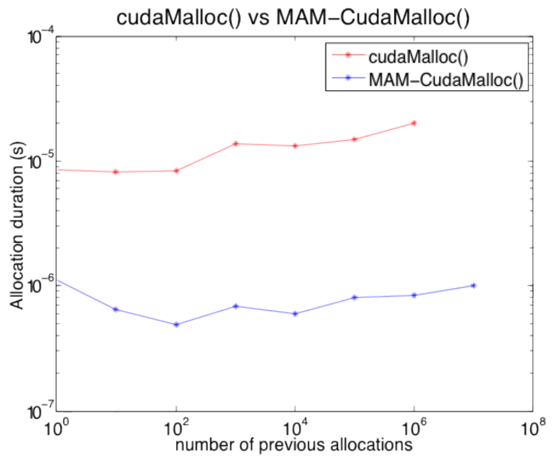

Fig. 10. `cudaMalloc()` vs `MAM_CudaMalloc()` comparison according to number of previous allocations

MAM uses lots of host memory, since a `segment struct` instance is created for each segment.

VI. Discussion & Related Work

This study only covers the performance comparison of MAM with CUDA device memory management. However, MAM is completely applicable to any other environment that involves allocation and deallocation of a contiguous space of any kind, such as pinned memory allocation of CUDA [7] or host memory allocation. MAM will work exactly the same way with any of these environments since it does not depend on the actual, or physical allocation procedure once it is created.

In the literature, there is a group that also focusses on GPU memory alloation and deallocation overhead [10]. They compare current GPU memory allocators and propse a new one that is register efficient. There are a lot of studies [7], [9], [8] about GPU memory management, mainly focusing on reducing data transfer overhead between the host and device memory. A study deals with the effective usage of relatively small GPU memory by using it as a cache for the host memory and transferring data between the two memories during runtime [3]. A second study that also focuses on small device memory size decreases data transfer overhead between device and host memory by directly connecting a Solid State Disk (SSD) to a GPU [4]. A group has developed a tool to manage device memory so that multiple applications can use the GPU without any problems [5]. Another study integrated GPU as a first-class resource to the operating system [6].

To our knowledge, there is no study focusing specifically on solving GPU memory allocation overhead. Programmers generally write their own memory manager for their specific application when it is needed. MAM offers a generalized solution, is independent of an applications, and provides efficient data structures to keep the overhead low.

VII. Conclusion

In this study, we focused on reducing the memory allocation overhead in GPUs and we developed MAM, which is a library for CUDA. This library abstracts the CUDA memory management module from the program and succeeds to remove the overhead by moving all the overhead to the beginning of the program. MAM currently offers a solution for the memory allocation problem of CUDA but it can be easily extended to be used in other platforms. Our future work will extend this work to Intel Xeon Phi architectures and other GPU programming models.

References

[1] ”CUDA Toolkit”, NVIDIA Developer, 2017. [Online]. Available: https://developer.nvidia.com/cuda-toolkit. [Accessed: 10- Jul- 2017].

[2] CUDA Memory Management Overhead. [Online]. Available: https://www.cs.virginia.edu/ mwb7w/cuda_support/memory_management_overhead.html [Accessed: 14-Oct-2016].

[3] Y. Kim, J. Lee, and J. Kim, ”GPUdmm: A high-performance and memoryoblivious GPU architecture using dynamic memory management,” in Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA), Feb. 2014, pp. 546-557.

[4] J. Zhang, D. Donofrio, J. Shalf, M. Kandemir, and M. Jung. Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. PACT 2015, 2015.

[5] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang, ”Gdm: Device memory management for gpgpu computing,” in The 2014 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS, (New York, NY, USA), pp. 533-545, ACM, 2014.

[6] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt. Gdev: First-class gpu resource management in the operating system. In USENIX Annual Technical Conference, 2012.

[7] B. Bastem, D. Unat, W. Zhang, A. Almgren, and J. Shalf. Overlapping Data Transfers with Computation on GPU with Tiles, The 46th International Conference on Parallel Processing, ICPP 2017

[8] Mehmet E. Belviranli, Farzad Khorasani, Laxmi N. Bhuyan, and Rajiv Gupta. 2016. CuMAS: Data Transfer Aware Multi-Application Scheduling for Shared GPUs. In Proceedings of the 2016 International Conference on Supercomputing (ICS ’16). ACM, New York, NY, USA, Article 31, 12 pages.

[9] T. Gysi, J. Bar and T. Hoefler, dCUDA: Hardware Supported Overlap of Computation and Communication, SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, 2016, pp. 609-620.

[10] M. Vinkler and V. Havran, ”Register Efficient Dynamic Memory Allocator for GPUs”, Computer Graphics Forum, vol. 34, no. 8, pp. 143-154, 2015.