Enabling Dynamic Memory Management Support for MTC on NVIDIA GPUs

Benjamin Grimmer, Scott Krieder, Ioan Raicu
Department of Computer Science
Illinois Institute of Technology
Chicago, IL USA
{bgrimmer,skrieder}@hawk.iit.edu, iraicu@cs.iit.edu

***Abstract*— Many-Task Computing emphasizes utilizing many computational tasks over a short period of time. Tasks can be either dependent or independent and are arranged as directed acyclical graphs (DAGs) [1]. Until recently there had been no support for MTC workloads on accelerators, but the development of GeMTC [4] enables Many-Task Computing to run efficiently on NVIDIA GPUs. One major complication with enabling MTC on NVIDIA GPUs is due to the memory management system. Due to overheads, the standard had been for applications to perform all allocations once at the beginning, reducing the importance of efficient memory management. To support MTC applications on NVIDIA GPUs, we need efficient memory management throughout the lifetime of the application. This paper presents a dynamic memory management system, which allows for efficient dynamic memory operations. We compare our results to the default CUDA approach; preliminary results highlight the ability to perform memory operations 8x faster than the default CUDA memory management system.**

I. Introduction

Many-Task Computing workloads have unique memory allocation needs that CUDA's memory management is not designed to handle. MTC workloads are composed of many different tasks, which will often need at least one dynamic memory allocation each to handle individual parameters and results. This does not follow the standard CUDA practice of allocating all memory at the beginning of the program. [2]

The existing CUDA memory management system was not designed for dynamic memory management. This results in poor performance on workloads that required a large number of memory allocations. To evaluate the existing memory management, we measured the average time taken to allocate many small amounts of device memory, and to allocate and deallocate many small amounts of memory. These benchmarks were run on an AMD 6 Core Workstation with 8 GB RAM and a GTX 670(1344 cuda cores), and show the default memory management doesn't scale when after many un-freed memory allocations (See Figure 1). Even in the best case, it takes more than 100µsec to execute a malloc and free through CUDA, and the cost grows linearly with additional mallocs exceeding milliseconds with tens of thousands of malloc operations. This is an unreasonable amount of time when the time to read/write to device memory from the host is O(10 µsec).

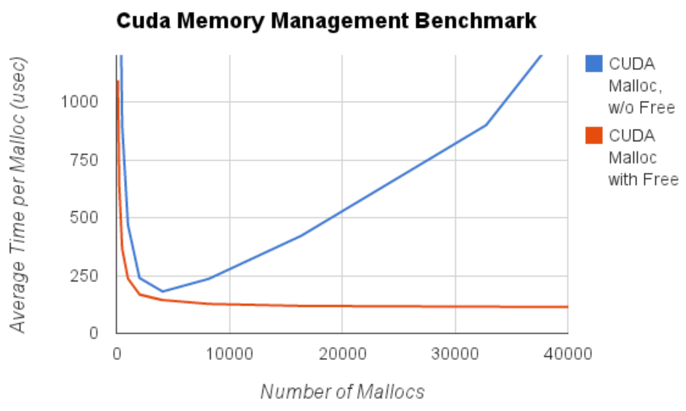

Figure 1 - CUDA memory allocation and deallocation execution times

II. Dynamic Memory Management

To improve malloc's scalability and reduce the cost of allocating device memory, we have implemented a sub-allocator designed to efficiently handle many requests for dynamic allocation. It uses the existing CUDA malloc to allocate large contiguous pieces of device memory, allocating more as needed. Then pointers to these free chunks and their sizes are stored in a circular linked list on the CPU (See Figure 2). This list is ordered by increasing device address to allow for easy memory coalescing.

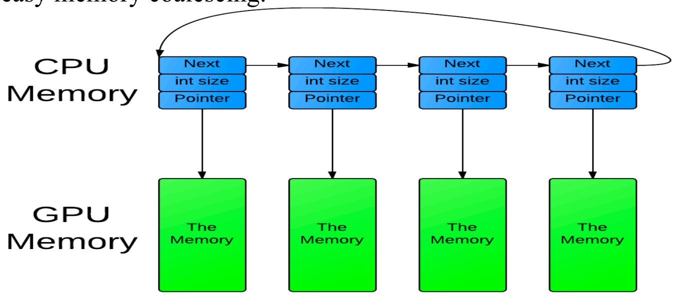

Figure 2 – Circular Linked List of free memory on the device

Upon memory allocation requests, the sub-allocator will find a large enough chuck of free device memory by searching its list. Then, it will write a header to the device memory indicating the size of the newly allocated chunk immediately before it and reduce the size of the original free chunk (See Figure 3). This operation takes roughly the same time as a memory copy to device.

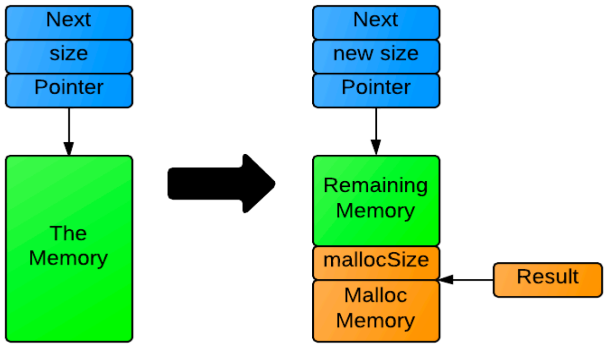

Figure 3 – Result of our malloc on free memory list

When device memory is freed, the header is read to identify the size of the chunk, and it is added to the list of free memory in its correct location. The chunk will be coalesced with an existing chunk if they are consecutive in memory. Freeing device memory takes roughly the same amount of time as a memory copy from the device.

Both malloc and free on our sub-allocator run in O(n), where n is the length of our free memory list. The size of the list is proportional to the amount of memory fragmentation, because each element is a separate chunk of memory. However, malloc and free both need to write and read, respectively, to GPU memory, which is typically more time consuming than searching the list. Despite this, workloads with high fragmentation will result in poor scaling of our memory operations.

Our current use case of supporting many-task computing workloads does not result in high fragmentation. We are running many independent tasks that each malloc at launch and free upon completion. As long as these tasks take approximately the same length of time, we will have memory allocated at approximately the same time being released at approximately the same time. Our free will coalesce each freed chunk with adjacent free chunks, resulting in decreased fragmentation.

To evaluate the performance of our sub-allocator, we measured the average time to allocate many small amounts of memory and to allocate and deallocate many small amounts of memory in the same environment as before. These results are compared with those from CUDA in Figure 4. CUDA's malloc without free results are omitted from the graph due to their poor scaling. Our sub-allocator offers an 8x speedup over CUDA for highly dynamic workloads with large amounts of allocation and deallocation. Our memory management also provides constant scaling under conditions where the default malloc scales linearly, enabling more than 30x speedups after ten thousand memory allocations, 100x speedup at 30,000 mallocs.

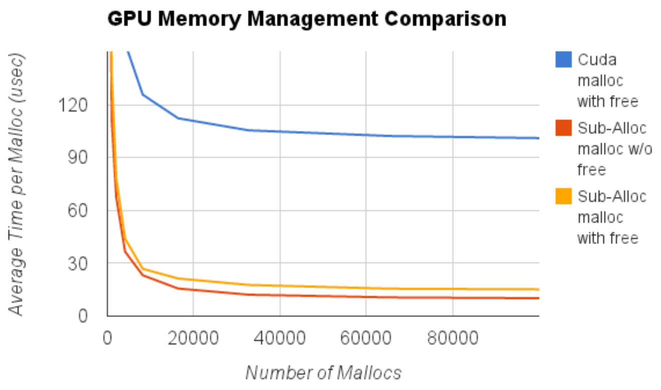

Figure 4 – Sub-Allocator and CUDA malloc comparison

III. Related Work

GCC standard allocator: This is the sub-allocator that is used by the standard c malloc and free. Its structure is similar to our sub-allocator's structure, utilizing a circular linked list of free memory and writing headers before allocated memory [3].

ScatterAlloc: This is a memory management module that runs on the GPU to handle many dynamic requests from within a CUDA kernel. This work was able to achieve speedups of 10-100x compared to CUDAs default allocation from kernels [5].

IV. Conclusions & Future Work

In conclusion we have presented a dynamic memory management module for enabling efficient support of many dynamic allocations and de-allocations of memory on NVIDIA GPUs. Our preliminary results demonstrate our ability to perform memory operations 8x faster than the default CUDA memory management system under a typical use case, and reaching 30x and 100x speedup after 10,000 and 30,000 mallocs, respectively.

We plan to improve the sub-allocator design by replacing the linked list with a data structure offering O(log n) insertion and search, rather than the current O(n). This should improve our worst-case time complexity with slowing down our typical use case.

References

[1] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, *et al.*, "Toward loosely coupled programming on petascale systems," presented at the Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Austin, Texas, 2008.

[2] CW, TB, JV, GZ, "Cuda C Best Practices Guide", version 4.1, January 2012

[3] B. Kernighan , D. Ritchie, "The C Programming Language", 1988

[4] S. Krieder and I. Raicu, "Towards the Support for Many-Task Computing on Many-Core Computing Platforms," Doctoral Showcase, SC 2012

[5] Markus Steinberger, Michael Kenzel, Bernhard Kainz, Dieter Schmalstieg "ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU"