Optimizing Memory Access on GPUs using Morton Order Indexing

Anthony E. Nocentino
Department of Computer and Information Science, University of Mississippi
662-915-7310
aen@cs.olemiss.edu

Philip J. Rhodes
Department of Computer and Information Science, University of Mississippi
662-915-7082
rhodes@cs.olemiss.edu

ABSTRACT

High performance computing environments are not freely available to every scientist or programmer. However, massively parallel computational devices are available in nearly every workstation class computer and laptop sold today. The programmable GPU gives immense computational power to a user in a standard office environment; however, programming a GPU to function efficiently is not a trivial task. An issue of primary concern is memory latency, if not managed properly it can cost the GPU in performance resulting in increased runtimes waiting for data. In this paper we describe an optimization of memory access methods on GPUs using Morton order indexing, sometimes referred to as Z-order index.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming — Parallel Programming

General Terms

Performance, Algorithms.

Keywords

Memory access, memory latency, GPGPU, Morton order, space filling curves.

1. INTRODUCTION

In today's high performance computing environments, general purpose computing on graphics processor units (GPGPU) is becoming increasingly popular[6]. Benefits include massively parallel computation in small, inexpensive and easily managed systems with devices most workstation class computers already have. GPGPU systems provide a highly parallel computation environment with high memory bandwidth, but it is important to minimize memory latency to maximize performance [1].

In this paper we present a method that overcomes memory latency when copying data to and from GPU global memory. We take advantage of the GPU's vast computational power to compute Morton[3] indexes that relax the requirements of memory coalescing. Due to their computational complexity, Morton order and other space-filling curve techniques are usually applied in secondary storage systems where the discrepancy in access times is much greater than that of CPU to main memory. Realizing the computational power of the GPU, we apply it to memory access.

ACMSE'10, April 15-17, 2010, Oxford, MS, USA.

Our eventual goal is to support applications on the GPU which exploit *spatial coherence* to reduce the total amount of processing required. For this reason, we prefer to divide the dataset domain into square blocks, since they have better locality than other choices. Unfortunately, they may also suffer from higher latency costs because of the way that memory is accessed. Morton order indexing allows us to address the latency problem while retaining the spatial advantages of square blocks.

We begin with some background information on programming for the GPU, including the CUDA programming model and the GPU memory hierarchy. We will also discuss coalesced memory access and its importance to high performance computing in a GPU program. Next, we describe the motivation and contributions for this project. After a discussion of our implementation, experiments and results we present our conclusions and future work.

2. GPU PROGRAMMING

In this section we discuss concepts of GPU programming, the hardware, the CUDA programming model, grouping threads into a block, and coalesced memory access.

2.1 GPU Memory Hierarchy

The GPU has several different types of memory available, each with different characteristics and management needs. *Global, local* and *texture* memory are the most plentiful but have the highest memory latency and that register, shared and constant memory are scarce but have the lowest memory latency. This contrast between slow and plentiful and fast and small is a critical factor in the success of a GPU program. Global memory is an uncached memory with size on the order of hundreds of megabytes. Shared memory is much smaller, on the order of kilobytes. It is shared between threads in a block, but it is not visible between blocks. For this reason transactions between shared and global memory occur frequently and therefore are a major factor for performance. Because of the significant performance discrepancy between global and shared memory, programmers must take great care to access memory in a manner that minimizes the effect of access latency, further discussed in section 2.4.

Shared memory is a parallel data cache that is shared by all scalar processor cores and is where the streaming multiprocessors' (SM)

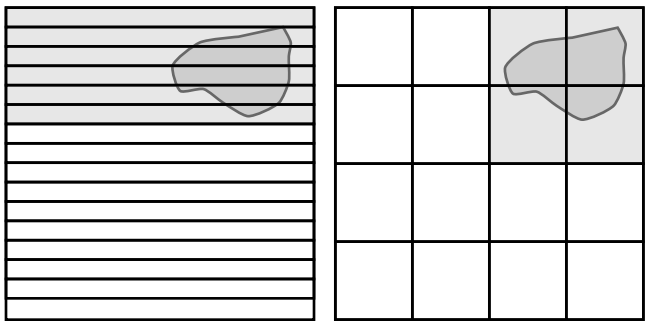

Figure 1: With a linear access pattern (left) each row much be processed. Using a square access (right) pattern we can reduce amount of area needing computation

shared memory space resides, it is often used as a data cache among threads on the same SM[1].

2.2 Typical GPU Program

The programming paradigm familiar to general purpose computer programmers shifts dramatically when programming for the GPU. GPU programs are massively parallel and one must carefully consider the computation and data management during program design. CUDA is an extension to C provided by NVidia for their graphics processors [1, 2]. CUDA programs have their beginning in the host; this is where memory is allocated and populated with program data. Program data must traverse the memory hierarchy to get from the main memory of the computer into memory that is accessible by the GPU's multiprocessors.

This difference changes the way data is managed so a GPU program usually follows this path:

1. Host process sends data into GPU global or texture memory
2. Partition the device's global memory data and transfer partitions into shared memory
3. Perform computation on the data in shared memory
4. Write the result from shared memory out to global memory
5. Host process reads results back to host memory

This paper relates to steps two and four. Focusing on the memory transactions between global and shared memory.

2.3 CUDA Programming Model

During processing, data is partitioned into a *grid*, a grid is partitioned into *blocks*, and blocks are sets of threads. The *execution configuration* consists of the grid dimension, the block dimension and shared memory allocation per thread. Blocks and threads can be uniquely identified by a numerical index; we refer to them as *blockID* and *threadID*. The memory access pattern is dictated by the *execution configuration*, which is discussed further in section 4.

A *warp* is a group of 32 threads that are scheduled in the GPU; a *half warp* is 16 threads. Accesses to global memory are scheduled in units of half warps. It is great of interest to us how a half warp accesses data. It is at this critical junction when data transfer takes place[1]. We will discuss this further in section 4.

2.4 Coalesced Memory Access

Memory access is *coalesced* when reads or writes for a half warp can be combined into a single global memory transaction; otherwise several separate memory transactions are performed. Because of the high latency of global memory access, coalesced memory access is the single most important performance consideration when programming for a GPU. Programs must meet the follow requirements for coalesced access [2]:

1. On devices with compute capability 1.1
   - The k^{th} thread in a half warp must access the k^{th} word in a segment aligned to 16 times the size of the elements accessed
   - Threads per block should be a multiple of 16, the half warp size
2. On devices with compute capability 1.2 or higher
   - Any access pattern that fits into a segment of size 32 bytes for 8-bit words or 64 bytes for 16-bit words or 128 bytes for 32-bit or 64-bit words

For compute capability 1.1, if a memory transaction is issued and these requirements are not met, 16 memory transactions are issued. For compute capability 1.2 and higher, the k^{th} thread can access any element. However, 1.2 and higher devices will issue memory transactions comprised of a number segments of fixed size. If only a portion of a segment contains requested data, than bandwidth is wasted resulting in a lower *load factor*.

Caching has long been used to improve performance for serial systems. While current GPUs support a *texture cache*, a 2D read-only cache of global memory, our technique addresses both read and write access for *n*-dimensional data. Relevant literature [19] indicates that next generation GPUs will have a coherent and unified L2 cache shared amongst the SMs, but these systems will not have a spatial view of the data. Our technique should prove effective as it diminishes the impact of cache misses and initial data loads from global memory into shared memory.

3. MOTIVATION AND CONTRIBUTION

Our current work is part of a larger effort to optimize spatial calculation on the GPU. Certain calculations can be optimized by early termination if appropriate conditions are met. We are interested in addressing the case where such conditions are *spatially coherent*. For example, a flood simulation application could be accelerated by early termination of computation in entirely dry areas of the terrain. The shaded irregular region in figure 1 shows the extent of a flood at a point in time. Our goal is to choose a block shape that maximizes the probability of all terrain elements in a block being dry. Due to the SIMD nature of GPU programming, if a block has even one wet element, all threads in the block must wait for that element to finish processing. For this reason, the thin horizontal blocks in figure 1a will perform poorly compared to the square blocks of similar area in figure 1b.

Although square blocks will take better advantage of spatial coherence, they will usually incur higher latency costs when used with linear order data. Considering the GPU programming paradigm—assigning threads to each data element and performing computation simultaneously—the motivation for minimizing latency costs is obvious. We do not want to waste processing time waiting for memory access to complete. *Chunking* is a well understood technique for addressing latency that would group the data needed by a block contiguously in memory, providing good performance [18]. Unfortunately, this technique would tightly couple the execution configuration with the data organization, requiring reorganization of the entire dataset if we change the block size. We instead chose to apply *Morton order indexing* [3]. Using this technique we are able to reduce the impact of memory latency by exploiting the vast computational power of the GPU to perform the Morton index calculations. Morton order indexing transforms our n-dimensional dataset into a one-dimensional

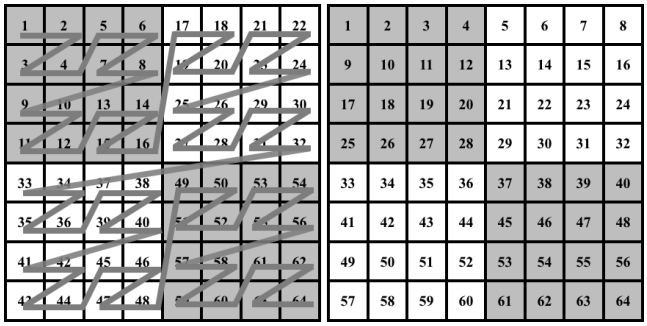

Figure 2a (left): Each large square is a block and its data, which is stored contiguously in memory, producing coalesced reads. Grid traversal is in Morton order. **Figure 2b (right):** Each shaded region is a block, containing a collection of numbered cells. The numbers in each cell indicate that block elements are not contiguous in the linear ordering, resulting in suboptimal performance.

representation. With a one dimensional linear representation of the data whenever our block size is greater than or equal to a half-warp (16) we will increase the number of coalesced memory transactions and for some cases be able to provide all coalesced memory transactions, thereby reducing the total number of memory transactions required for a program.

3.1 Morton Order

Morton order indexing is used to index an n-dimensional array in a manner that converts the Cartesian coordinates into a single index value by bit interleaving or *dilation* [3, 4]. Data in the array is accessed in the distinctive Z pattern. See Figure 2a.

4. IMPLEMENTATION

In this section we will discuss our implementation details for the Morton and linear indexing schemes. The details of our experiments will be discussed in the next section.

A Morton index is calculated for the X and Y coordinate of a blockID. Through testing we selected a dilation algorithm and implemented it as a function inside our CUDA kernel. The implementation of the Morton calculation algorithm we chose is described thoroughly by Raman and Wise[4]. As each thread processes its element, it must also calculate the Morton index for that element's block. Elements within a block are loaded using coalesced reads, and are then processed independently by each thread. We would like to emphasize that because our current tests do not perform a calculation that requires information from neighboring elements, we never have to compute the Morton index for individual elements.

Our method translates the two dimensional block into a one dimensional representation of the dataset. Additionally, memory access order is determined first by its location in the grid, by blockID and then by block dimension; independent of block dimension on the Y axis. This is in contrast to a linear access pattern in which memory access order is determined by blockID and block dimension in both the X and Y directions resulting in memory access patterns which are determined by the execution configuration. Accessing the one dimensional representation of the dataset using this Morton indexing for blocks we can achieve increased number of coalesced memory transactions (in some configurations all coalesced memory transactions) and a reduced number of total memory transactions. As shown in figure 2a, we have an 8x8 grid of 4 blocks; the block's data is accessed in parallel and the grid is traversed in Morton order. In this case access is coalesced since the read transaction is the same as the warp size. In contrast, in figure 2b, we have the same execution configuration but memory access will not be optimal. On a 1.1 device this is because the k^{th} thread is not accessing the k^{th} element this results in uncoalesced transactions. On a 1.2 or greater device this will results in transactions with lower load factor effectively reducing bandwidth.

5. EXPERIMENTS AND RESULTS

In this section we present our experimental design and results. We devised an experiment to test the number of memory transactions executed by the GPU using a test application. Additionally, we ran our experiments on devices with compute capability 1.1 and 1.3 to capture the effects of the different coalescing models.

We implemented a compositing (alpha blending) volume rendering technique[5]. This technique was chosen since it does not require neighbor information and is data parallel. We will address our concerns with neighbor calculation in the future work section.

Using a set of CT data, approximately 14MB in size consisting of 256x256 pixels and 113 slices, we developed a parallel algorithms to process this data[17]. The algorithm follows a ray through the data set performing the compositing function on each voxel along the ray generating a single image[5]. Our implementation loads the whole dataset into global memory and performs the processing on each slice from back to front using a loop.

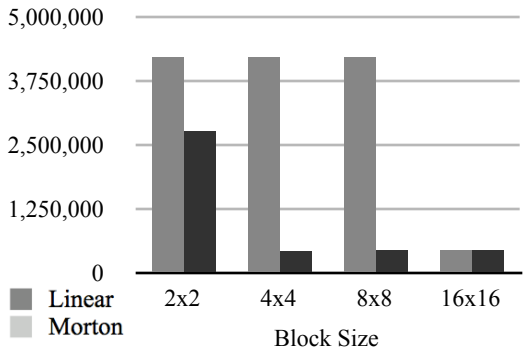

Figure 3: Total memory transactions for compositing on GPU with compute capability 1.1

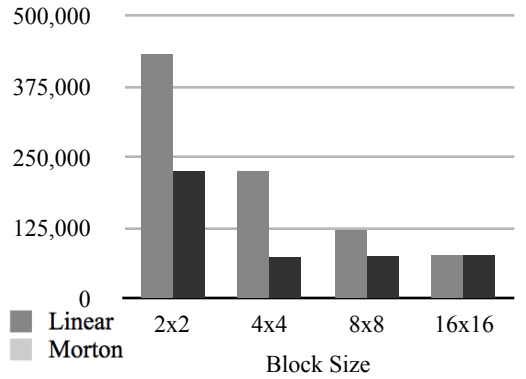

Figure 4: Total memory transactions for compositing on GPU with compute capability 1.3

The results presented in figures 3 and 4 are for square execution configurations. In each test case, we are using two dimensional Morton indexing for each plane in the dataset.

On devices with compute capability 1.1 and 1.3 using Morton indexing significantly reduces the number of memory transactions when compared to linear indexing. This is true for all cases but the 16x16 case. With the 16x16 case all memory transactions are coalesced because the row width is 16 and meets the requirements for coalesced memory access for both device types. This is true using either indexing method. Note that on devices with compute capability 1.3 the total number of transactions is reduced by a factor of 10. This is because it has looser coalescing requirements and the ability to combine smaller memory transactions into larger ones.

These results indicate that the Morton indexing method will prove effective for data access on the GPU.

6. RELATED WORKS

As the separation in speed in memory hierarchies increases due to faster clock speeds and multi-core CPU systems, Morton order indexing has emerged as an optimization to address the memory locality and latency issues which are compounded in today's multi-core systems[10]. This is extended into GPUs with their large number of streaming multiprocessors (SM) as programmers try to keep each SM sufficiently occupied a computation[1, 2, 10]. It is known that GPUs possess signification power[6, 7, 16] However, they were not initially designed for general purpose computation. Their memory hierarchies were built to maximize throughput rather than minimize memory access latency for threads[13].

Additionally, it is known that to achieve optimal performance on GPUs a major concern is managing global memory latency [1, 2 , 8]. Morton order indexing has been applied as a latency reduction technique [12]. We extend the latter into a more general case and envision our technique as part of an API for efficient memory access in GPUs. Other strategies have been suggested for latency reduction but we hope that our methods can apply to more general cases and reduce the burden on the programmer[8, 14, 15].

7. FUTURE WORK

As we work toward spatial applications like the flood simulation example discussed briefly in section 3, there are two important areas that we must investigate.

First, it became evident during our exploration for application scenarios that we needed an *undilation function*, the ability to transform a Morton index back into an X, Y pair. This will allow for easier access to neighbors around a data element, greatly increasing the range of applications we can support.

Second, we must extend our current implementation beyond square grids and square blocks. We will explore varying grid and block dimensions to measure the impact on performance. Preliminary experimentation comparing Morton indexing with linear indexing using non-square execution configurations shows there is an increase in the number of coalesced memory transactions thereby reducing the number of memory transactions.

8. CONCLUSION

We have presented a method that allows us to choose a square block shape while avoiding the substantial performance penalties normally associated with square blocks. As demonstrated in the results, we have verified that Morton order reduces the number of required memory transactions compared with linear order for a program using square blocks. As our work progresses, Morton order will allow us to exploit the desirable spatial properties of square blocks without incurring the latency penalties that are otherwise expected.

9.REFERENCES

[1] NVIDIA CUDA, Programming Guide. Version 2.3.1 2009.

[2] NVIDIA CUDA C Programming Best Practices Guide, CUDA Toolkit 2.3, July 2009.

[3] G. M. Morton, "A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing", Technical Report. 1966.

[4] Rajeev Raman, David Stephen Wise, "Converting to and from Dilated Integers". *IEEE Transactions on Computers*, pp. 567-573, April, 2007

[5] W. Shroeder, K. Martin and Bill Lorensen, *"The Visualization Toolkit"* 4th Edition. 2006. pp. 214-220.

[6] http://www.nvidia.com/object/cuda_home.html

[7] Owens et al. "A survey of general-purpose computation on graphics hardware". *Computer Graphics Forum* (2007) vol. 26 (1) pp. 80-113

[8] Ryoo et al. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (2008)

[9] http://www-graphics.stanford.edu/~seander/bithacks.html#InterleaveTableObvious

[10] Adams and Wise. "Fast additions on masked integers." *SIGPLAN Notices* (2006) vol. 41 (5)

[11] Lorton and Wise. "Analyzing block locality in Morton-order and Morton-hybrid matrices". *SIGARCH Computer Architecture News* (2007) vol. 35 (4)

[12] Aila and Laine. "Understanding the efficiency of ray traversal on GPUs". *HPG '09: Proceedings of the Conference on High Performance Graphics* 2009 (2009)

[13] Tarjan et al. "Increasing memory miss tolerance for SIMD cores." *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009)

[14] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K. 2008. "A performance study of general-purpose applications on graphics processors using CUDA". *J. Parallel Distrib. Comput*. 68, 10 (Oct. 2008), 1370-1380.

[15] Nickolls, J., Buck, I., Garland, M., and Skadron, K. 2008. "Scalable Parallel Programming with CUDA". *Queue* 6, 2 (Mar. 2008), 40-53

[16] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. 2008. "Parallel Computing Experiences with CUDA." *IEEE Micro* 28, 4 (Jul. 2008), 13-27.

[17] http://www.graphics.stanford.edu/data/voldata

[18] Sarawagi, S. and Stonebraker, M. 1994. "Efficient Organization of Large Multidimensional Arrays", In *Proceedings of the Tenth international Conference on Data Engineering* (February 14 - 18, 1994). IEEE Computer Society, Washington, DC, 328-336.

[19] NVIDIA CUDA, Programming Guide. Version 3.0 2010.