Throughput Computing on Future GPUs [1]

Rune J. HOVLAND [a] and Anne C. ELSTER [a]
[a] *Norwegian University of Science and Technology (NTNU), Trondheim, Norway*

Abstract. The focus on throughput and large data volumes separates Information Retrieval (IR) from scientific computing, since for IR it is critical to process large amounts of data efficiently, a task which the GPU currently does not excel at. Only recently has the IR community begun to explore the possibilities, and an implementation of a search engine for the GPU was published recently in April 2009. This paper analyzes how GPUs can be improved to better suit such large data volume applications. Current graphics cards have a bottleneck regarding the transfer of data between the host and the GPU. One approach to resolve this bottleneck is to include the host memory as part of the GPUsŠ memory hierarchy. Benchmarks from NVIDIA ION, 9800m and GTX 240 are included. Several suggestions for future GPU features are also prensented.

Introduction

While the GPU is gaining interest in the HPC community, others are more reluctant to embrace the GPU as a computational device. The field of Information Retrieval is a field with large data volumes and computationally lighter applications than traditional HPC. For this reason the GPU has not yet gained the status as a suited computational platform. However, the WestLab3 research group at the New York University has recently created a fully functional search engine [5] on the GPU with improved performance. While a search engine often contains complex ranking schemes and other calculations, it is still a application bound by the large data volumes stored in a search index.

Handling large data volumes on the GPU is not trivial on current GPUs. A data copy must be performed before the application can start, and the results from the GPU must be copied back to the host after the completed calculations. When handling large data volumes this can cause critical delays which seriously impair the GPUs ability to compete with CPU-only implementations. By introducing full streaming capabilities, the GPU might be able to remove most of these delays and efficiently handle large data volumes.

This paper will suggest improvements that can be made to enable future GPUs to efficiently support large data volumes, and ease the development process of such applications. The NVIDIA Tesla Architecture and the NVIDIA CUDA programming extension will be used as the representative of current GPUs. More detailed models and further references can be found in [3].

[1] The authors would like to thank NVIDIA for the donations of the graphic cards benchmarked in this study. This work was done while the first author was a master student at NTNU.

1. GPU performance characteristics

While the GPU allows general-purpose calculations to be performed, it is not a fully general-purpose processor, and thus has a bias towards graphics processing. This bias has made the architectural designers make certain tradeoffs with regard to performance to create the optimal GPU for what NVIDIA considers to be its main markets. This section will shed light on some of these performance characteristics.

1.1. Host to device transfers and GPU global memory

When using a GPU for computations, it usually requires data as input and in most cases produce output data. These data must be copied to and from the GPUs memory, since the GPU is unable to access the host memory while performing the calculations. This copy operation can be costly in many applications, especially for data intensive calculations.

NVIDIA Tesla GPUs have two levels of memory hierarchy [4]. The large storage capacity is provided by the global memory which is currently up to gigabytes. When data is copied from the host memory onto the GPU, it is copied into this memory. An access to this memory is slowed down by a latency between 400 and 600 clock cycles [2], and is thus not able to fulfill the role as high bandwidth memory. To improve performance, global memory allows memory access to adjacent addresses to be grouped together into one read or write operation. This approach is called coalesced read and write operations.

Paging and Direct Memory Access

Modern operating systems allow programs to use more memory than physically available though the use of paging [**?**]. Paging allows memory to be automatically swapped out to a hard-drive when it is not needed, and thereby freeing physical memory for other uses while still maintaining the integrity of the virtual memory. The main drawback with this technique is that only the operating system knows the exact location of a memory segment since it may be moved around due to memory swapping. In situations where an exact memory location is needed, page-locking can be used. This is mostly used when using Direct Memory Access (DMA), since DMA allows memory copies to be handled by a DMA handler instead of the CPU.

Data transfers to GPU

All copy operations between host memory and device memory on NVIDIA GPUs requires the use of DMA. Since most memory locations are not DMA accessible, there are two techniques which can be used. The first solution is to store all data which will be used on the GPU in page-locked memory locations. This is not always feasible since page-locked memory locations are a scarce resource. The other option is to copy the data to a page-locked memory location before copying it to the GPU. This approach requires an extra memory copy which can be costly. When to use which of the two techniques depends on the usage of the data. For data which will be copied often to the GPU it may be best to use dedicated page-locked memory, while for a one-time copy one may just as well use the second technique [4].

In CUDA, both techniques is supported automatically based on which type of memory location that is given to the copy-instruction [2]. By default, a pageable memory location is given when using the standard C/C++ command malloc, and if such a memory

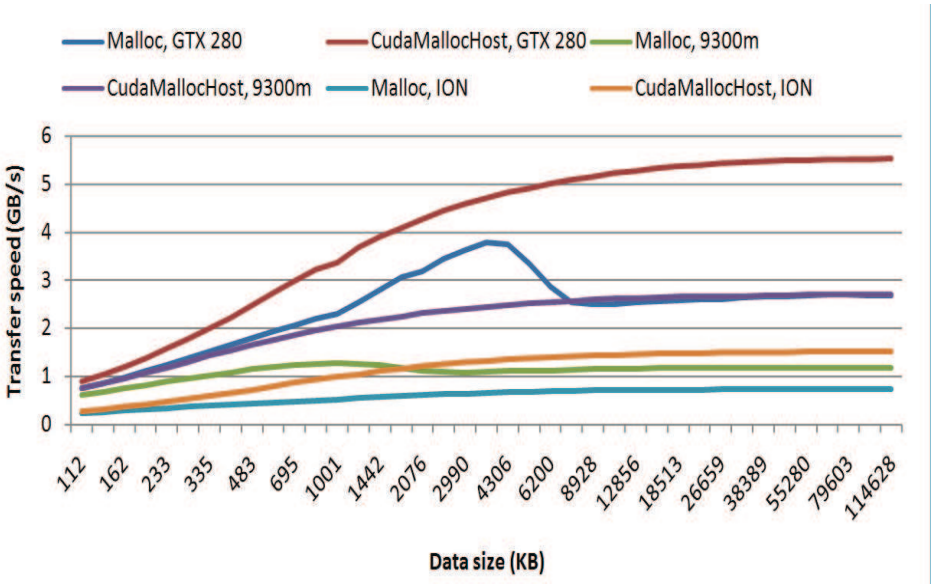

Figure 1. Host-device bandwidths for NVIDIA ION, 9300m, and GTX 280

location is given to the copy instruction in CUDA, it will copy the data to a page-locked memory location before copying it to the GPU. To allocate a page-locked memory location CUDA provides a method cudaMallocHost. Which of the two techniques to use may wary from application to application, but the measured bandwith when using the techniques in a simple test-case on the NVIDIA ION, 9300m adn GTX 280 can be seen in Figure 1.

1.2. Data Access

In large data volume applications, the data may be too large to fit in memory and hard-drives must be used. This means that any data which should be used must be read into main memory before it can be used. If this should be done on the GPU, it is even more cumbersome, since it must first be read into memory, before being copied onto the memory of the GPU. The results calculated at the GPU must also be copied back to the host-memory if it is to be used further by the CPU.

Modern hard-drives are considered to be the bottleneck of any application operating over large datasets. The highest transfer rate12 found for sustainedread for a hard-drive was 171MB/s13 which is fairly low considering the memory bandwidth of the new Intel Core i7 processor14 which is 25.6GB/s. The bandwidth of data transfers to and from the GPU which was found to be close to 6GB/s [4].

This time required can be expressed as a function of the hard-drive latency L_{HD}, size S, and hard-drive bandwidth B_{HD} of the data transfer as given in Equation 1:

$$T = L_{HD} + \frac{S}{B_{HD}} \tag{1}$$

This equation describes the simplest form of data access where the data is stored uncompressed on a single disk. Once read, it is stored directly in the location where it will be used.

2. Compression and GPU off-loading

A search engine [1] can be considered to focus on High-Throughput Computation (HTC) rather than a High-Performance Computation, as long as the latency of a single query is below a certain threshold. This query-latency is the time from the query is given until it is answered. There are many components of this latency, and the retrieval of the index-entry is one of them.

To reduce the impact of the hard-drive transfer rate, one can use compression thereby reducing the size of transferred data. This compression reduces the time required to transfer the data, but introduces a computational step which decodes the data and copies it over to the final memory location.

The new equation for the time required to access the entry which takes into account the time Ccomp required by the added computational step and the compression ratio Rcomp is given in Equation 2:

$$T = L_{HD} + S \cdot (\frac{R_{comp}}{B_{HD}} + C_{comp}) \tag{2}$$

By offloading the decompression to the GPU, the CPU is free to perform other tasks, and the overall throughput of the system may increase even if the time required to fetch an index-entry may increase.

2.1. Data copy hiding

Current GPUs are incapable of reading directly from the hard-drive, and the hard-drive is unable to write to the GPUs memory. However, it is possible to perform most of the transfer to the GPU parallel with the transfer from the hard-drive to system RAM. This is done by dividing the transfer into n parts, and start copying a part to the GPU asynchronously as soon as it is read from the hard-drive. By choosing the right size to partition the transfer into, the extra time needed to copy datato the GPU would only be equal to the time needed to copy the final part.

The operation of copying of the result to the host-memory is more cumbersome to remove, since it will be used by other parts of the system, and its lifespan may not be known. It is therefore best to free its memory location on the GPU and instead maintain it in host-memory which is more cost-efficient.

2.2. Memory Mapped Files

The last approach which can be considered is to use Memory Mapped Files which is specified in POSIX [20] through mmap(). This is a technique which creates a memory pointer to a Virtual Memory location in which the file is mapped. In this way, the file can be accessed as if it were residing in the memory, and the task of actually supplying the data is left to the operating system. By choosing this approach, the programmer can utilize efficient

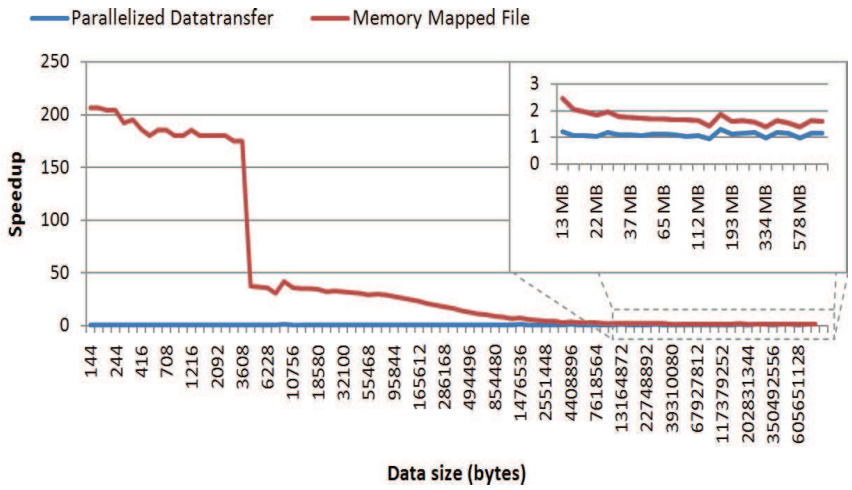

Figure 2. Memory mapped vs. parallel data transfers

2.3. Preliminary Performance Analysis

To determine if either of the two proposed improvements (data copy hiding and memory mapped files) would give improved performance over the basic approach, a test case was developed. By measuring the time each of the three approaches use to read data of various sizes from file and make it accessible on the GPU, an indication on the performance gain by the improved approaches could then be found. In Figure 2, the speedup of the two improvements with regard to the simple approach is given. As one can see, the memory mapped files obtains much better performance than the other two approaches. The hiding of data copy approach does not achieve noticeable speedups. This might be due to an added cost of initializing multiple file reads and data copies.

3. Expanding the memory hierarchy on next-generation GPUs

This section describes how the memory hierarchy of future architecture should be expanded to improve the usability for developers of large data volume application. In this context, a number of suggestions for minor changes to the Tesla Architecture and CUDA are also provided.

3.1. Expanding the Memory Hierarchy

The current Tesla Architecture allows the GPU to have its own dedicated memory or using a part of the host memory as device memory. Any data which is to be used by the GPU must be copied into device memory before a kernel is initialized. Any results from the kernel must be copied back to host memory after execution. These requirements forces any CUDA application to contain three steps; data-copy to device, execution and data-copy from device.

When transferring large amounts of data between the host and device memory, it may be beneficial to start computations before all the data is located on the GPU. The

current Tesla Architecture allows this to be done by using streams, where both the data transfers and computations are divided into batches which are performed in an overlapping manner, thus hiding some of the data transfer cost. However, initializing multiple kernels is costly.

Direct host-memory access

Allowing the kernel to directly access the host-memory removes the need for host to device transfers in most cases. Syntactically this would be similar to accessing any other memory location on the GPU, but it would have a higher cost in terms of lower bandwidth and higher latencies. It would therefore be up to the developer to reduce the number of accesses to host-memory to a minimum by pre-fetching data into device memory such as global or shared memory.

For an application with large volumes of data, this possibility to access the host-memory directly would remove the need to divide the calculation into several kernel executions when the data exceeds the size of device memory. With the direct-access approach, the kernel can simply fetch more data into the device memory, while discarding used data without the need to return control to the CPU.

The dedicated GPU memory on the current high-end NVIDIA cards is significantly faster compared to the low-end NVIDIA ION system which uses older DDR2s and also a less powerfull CPU than typcially found on high-end PCs with separate graphics cards. Not surprisingly, the higher-end cards are therefore currently faster, despite the shared CPU-GPU RAM for the ION. This was seen in Figure 1 where the GTX 280 with dedicated memory outperformed the 9300m and ION which use local memory.

Customizing Memory Hierarchy

When allowing direct access to the host-memory from the kernel, it practically adds another level to the GPUs memory hierarchy. This added layer will change the usage of the device-memory since it would allow for more data access patterns. Through the added layer of GPU memory, the need for large device memory locations will be more individual. One can thus envision a more customizable device memory on the GPU to tune the size to the individual needs. Since the GDDR memory used in GPUs is more expensive than DDR memory used for local memory, this customization of memory hierarchy would allow for more cost-efficient installations. It may also allow the GPU access to more memory when computing.

Combining Dedicated and Shared Memory

If enabling customizations of the memory hierarchy is not feasible, there is another approach which seems easier to implement since most aspects needed exists in current GPUs. By allowing the GPU to use both dedicated and local memory as device memory, but dividing them into two distinct levels in the memory hierarchy, the same effect can almost be achieved. It would, however, require that the CPU also has write permissions to the host-memory used as device-memory, and that all data which the GPU will use is stored in this memory location.

Caching Problems

When using either of the two suggested approaches for kernel access to host-memory, there will be issues with CPU caching that must be resolved, since the data stored in the host-memory may not be the valid version due to cache and delayed write-back. These problems are assumed solvable in our study.

Zero-Copy in CUDA 2.2

The Zero-Copyfeature was recently introduced in CUDA version 2.2. Its main purpose is to allow the user to do, to some extent, what is suggested in this section. By allowing the user to access page-locked memory from the GPU, the need for copy prior to and after kernel execution can be eliminated. However, page-locked memory is a scarce resource, so the developer may still be required to do extra copy operations if there is extensive memory usage on the host.

3.2. Additional Improvements

Using CUDA to develop large data volume applications can be a cumbersome process. To increase the usability of CUDA for such applications, we present a number of improvements aimed at simplifying the development process and enable more compact and understandable code.

Caching

While caching is currently not supported for normal data on current Tesla Architecture, there is still a way to have the GPU handle caching. By claiming that the data is a texture, the GPU seizes control of the data access and caches the data using the Texture-memory. One thing that must be noted is that by marking the data as a texture, it is read-only since there is no write-back on the caching. If cached data are altered the result when accessing the data is undefined. While this limitation is unacceptable in many situations, there are applications for which this does not impose a problem. To use data as a texture, it is only necessary to instruct the GPU to treat the data as a texture. Any data may be handled in this manner. However, the syntax for doing so does not resemble the normal way to handle data access, and it may be confusing to use. Therefore CUDA should include functionality to enable caching for data without referencing textures since this may easily be implemented as a syntactic sugar without any alterations to hardware.

Extended Host-Device Synchronization

The CPU and GPU have different objectives and will therefore continue to have different characteristics. To utilize the computational system optimally, calculations should be performed on the processor that gives the best overall performance of the system. This would in many cases require rapid changes between CPU and GPU calculations and data exchange between them. With CUDA, as it is now, this can only be done by stopping the kernel each time the CPU should perform a calculation that the GPU depends on. While this is a solution that enables interaction between CPU and GPU, it is a cumbersome process which complicates the development process. A better solution would be to allow halting the CUDA kernels by synchronizing with the CPU. In this way, there would be a more intuitive interaction between host and device. This can be solved by either

actually implementing synchronization in the architecture, or by adding the functionality as syntactic sugar which hides the process of dividing execution into multiple kernels. To efficiently implement the second approach, the cost of initializing a kernel must be reduced so that rapid control changes between GPU and CPU do not affect performance in to large extent. An possibility here is to implement it as syntactical sugar first to see if the developers will use it, and if so implement it in hardware. An approach like this will be less costly as hardware changes are more expensive

3.2.1. Allow File Access

Both search engines and many other applications require large data volumes which are stored on disk. In the current CUDA environment, the GPU cannot access files directly. The CPU must thus regain control and access the file, and copy the data to the GPU before restarting the kernel. Enabling file access from the GPU directly can be difficult, as it would require handling IO between the GPU and the disk. An easier approach could be implemented if local memory access is enabled as described in Section 3.1. This approach is to use memory-mapped files, which enables file access from the GPU by masking the file as a memory location, and giving this memory location to the GPU. By doing so, the operating system ensures that the data in the file is accessible to the GPU trough the virtual memory address that the file is mapped to.

4. Conclusion and Future Work

This paper took a close look at the current NVIDIA GPUs and pointed out the need for certain features which would improve performance for large data volume applications. By including the host-memory in the memory hierarchy of the GPU, like seen on the low-end NVIDIA ION, new ways to access data during calculations can be developed. Other benefits which reduce the complex code of large data volume applications have also been suggested. Further reflections on how likely these features are to be realized, and what work lies ahead in the process of providing these features can be found in [3].

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. 1st ed. Addison Wesley Longman Limited, 1999.

[2] *NVIDIA CUDA 2.1 Programming Guide*. NVIDIA Corporation. http://www.nvidia.com/object/cuda_develop.html.

[3] R. J. Hovland. *Throughput Computing on Future GPUs*. Master thesis, Norwegian University of Science and Technology, July 2009. http://www.idi.ntnu.no/ elster/master-studs/runejoho/rune-hovland-master-ntnu.pdf

[4] R. J. Hovland. *Latency and Bandwidth Impact on GPU-systems* Projet report, December 2008, Department of Computer and Information Science, Norwegian University of Science and Technology. http://www.idi.ntnu.no/ elster/master-studs/runejoho/ms-proj-gpgpu-latency-bandwidth.pdf

[5] S. Ding, J. He, H. Yan, and T. Suel. ŞUsing Graphics Processors for High Performance IR Query Processing,Ť in *Proceedings of the World Wide Web Conference 2009*, Madrid, Spain, April 2009, pp. 421Ű430.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2007.