High Performance Computing using Graphics Processing Units (GPUs)

Simulating the Physics of Heavy Ion Collisions using GPUs

Kawthar Shafiekhorassani

Dr. Abhijit Majumder, Dr. Loren Schwiebert

Wayne State University

REU Summer Research 2017

Table of Contents

Abstract

Modern Graphics Processing Units(GPUs) have an inherent architecture that can be exploited for high performance computing in addition to their original purpose of rendering graphics. The GPU has many features that support parallel computing including its highly multithreaded architecture and high memory bandwidth. Parallel computing can be an optimal solution to performing multiple calculations at a high speed. The purpose of parallel computing is to reduce the delay between inputting a process and yielding an output and to increase the number of processes that can run through a system simultaneously. This is referred to as reducing the latency and improving the throughput of a program.

This paper examines various features of the GPU and how to optimize certain processes in order to exploit these features to render high computation speed. A code that simulates the physics of a heavy-ion collision was examined to port certain functions with high computations to the GPU. However, upon examining the code, many functions were recursive. While modern GPUs support recursion, recursive processes on the GPU may not take advantage of the massively parallel architecture by running in parallel. In order to eliminate recursion, a heap tree structure was used to process the recursive calls. The goal of this research was to remove recursion from the CPU version of code in order to utilize the highly multithreaded architecture of the GPU to optimize the performance.

Introduction

The highly multi-threaded architecture of a GPU can be utilized to significantly reduce the time it takes to process, and the number of processes that can run through a system simultaneously. Due to the multi-threaded architecture, parallel computing can be used and threads can be assigned a specific work load to process in parallel with other threads.

The central processing unit is the processor of the computer that executes the software instruction. It processes sequentially, which means every event is dependent on the completion of the prior event. It can be found in phones, dvd players, cars, washing machines, and etc. The graphics processing unit is generally known for rendering graphics and most often used for gaming. Different properties of the GPU make it suitable for performing computations at a much higher speed than the CPU. In order to utilize a GPU, the CPU can act as a co-processor. While some events can run more efficiently on a GPU, others are better handled by the CPU.

A CPU consists of a few cores optimized for sequential processing. The GPU has a massively parallel architecture consisting of thousands of smaller cores designed to handle multiple tasks simultaneously. The GPU also has high memory bandwidth, which refers to the rate data can be read from or stored in memory. Two characteristics of the GPU can be problematic; the first one is that the GPU and host memories are typically disjoint. This requires explicit data transfer between the two, whenever the CPU needs to collect data or the GPU needs to process it. The second characteristic is that the GPU does not adhere to the same accuracy standards as the CPU. Therefore, results from the GPU must always be cross-checked

with results from the CPU. This specific obstacle however, was addressed by Nvidia in newer versions of their releases. (Barlas 2015).

The main advantage of using the GPU is to utilize its ability to perform parallel computing, which allows for events to run simultaneously. This basically refers to the idea of running multiple events at the same time without requiring the need for prior events to be complete. The multi-threaded architecture of the GPU is what allows for the implementation of parallel computing. Multithreading refers to to the execution of a sequence of programmed instructions within threads that execute independently. The goal is to decrease the delay between inputting a process and having an output, and to increase the number of processes that can pass through a system simultaneously. This is also referred to as reducing the latency and improving throughput (Fanz, Kaufman, & Yoakum, 2004).

Compute Unified Device Architecture (CUDA)

The hardware architecture of the GPU requires an extension language to utilize it through an efficient programming methodology (Hemalatha & Kodada, 2013). Compute Unified Device Architecture (CUDA) is a general purpose parallel computing platform and programming model. CUDA is used as an extension to C programming. The most efficient way of exploiting the architecture of the GPU is to utilize a large number of threads that execute multiple processes concurrently. CUDA uses kernels and threads to execute and access data from memory. It can access data from multiple memory spaces during execution. CUDA organizes threads into blocks, which are then organized into threads. Every thread is given a specific position identified by the size of each block, the size of each grid, threadID (position of thread in

block), and blockID (position of block in grid). This allows a thread to recognize its workload or assignment (Sander, Kandrot, & Dongarra, 2015).

Methods

JETSCAPE

This research utilized a portion of code from the Jet Energy-loss Tomography with a Statistically and Computationally Advanced Program Envelope (JETSCAPE). It simulates the physics of a heavy ion collision. The Shower function starts with an initial parton. It then recursively splits, and generates a shower. The code for this simulation is designed to use information for a specific parton including the parent ID, location, and distance to predict the location, and distance of the partons that split from it. As this process occurs many times, it forms a shower.

High Computation Functions

Various functions in the code were analyzed for processes that require high computational power and the majority of the run-time. The shower function requires the highest computation time and performed the majority of the work during run-time. It initially begins with one parton then keeps splitting the parton until it cannot split anymore, which is determined by energy. This generates a shower. It is often easy to recognize a process that requires high-computation when it is iterative. However, the shower function was recursive. There were two recursive calls within the function that had to be removed in order to fully utilize the advantages of the GPU. A recursive process is dependent on the completion of another process to continually build on.

The purpose of the GPU is to perform multiple calculations that are independent of each other. In order to do that, the recursion must be removed and replaced with an iterative process. In this case, the parameters for the recursive call can be stored in an array which can be passed back into the function. Originally, it made sense to store the values of the recursive call in the S array. The S array stored the starting point for every parton in the shower. The struct of arrays represented by S stored various parameters including each partons parent ID, a counter, the current index of the array, the location, distance, and other values that were calculated according to the parton being processed. However, after extensive research on this process, it made more sense to define a new array of structs that would store the recursive calls. This array can then be copied to the original array being used. The recursive call was represented by the following:

shower_vac (c_line, pid_a, nu_s1, t0_in, z_g*z_g*t_g, nkx, nky, loc+distance, next_lead)

To store the parameters of this recursive call in an array, a struct of arrays was created with every parameter indicated. Additionally, another parameter (is_valid) was initialized to represent whether or not the current parton would be splitting in the next iteration. The value is initialized to false and only set to true when the code reaches the point where the values are stored in the array. The array, SV1, was initialized as show below:

```
SV1[i_call1].c_line = c_line;
SV1[i_call1].pid_a = pid_a;
SV1[i_call1].nu_s1 = nu_s1;
SV1[i_call1].t0_in = t0_in;
SV1[i_call1].z_g2t_g = z_g*z_g*t_g;
```

```
SV1[i_call1].nkx = nkx;
SV1[i_call1].nky = nky;
SV1[i_call1].locdist = loc+distance;
SV1[i_call1].next_lead = next_lead;
SV1[i_call1].is_valid = true;
```

In order to ensure that a process does not assign any values into the array when a parton stops splitting, the Boolean variable, is_valid was used. This variable keeps track of whether or not the current parton will split in the next iteration. A loop was defined to check if is_valid is false. If it is, it increments the counter then continues to the next iteration, skipping the assignments to the array.

Heap Tree Implementation

The two recursive calls in the function represent the two children of the parent from the original function call. With every recursive call to the function, the parton is splitting to form two children. This was implemented in an array with a heap tree representation. The formula to determine the children of the parent in a heap tree was used to define the location in the array to assign the children. For the first child, the parent index was multiplied by 2 and 1 was added to that. The parent index was multiplied by 2 and 2 was added to that in order to determine the location for the second child. This is represented in Figure 1, with the black filling showing the parent, and the grey filling representing the children. Node 1 is the parent of 3 and 4, which can be calculated using the formula mentioned earlier.

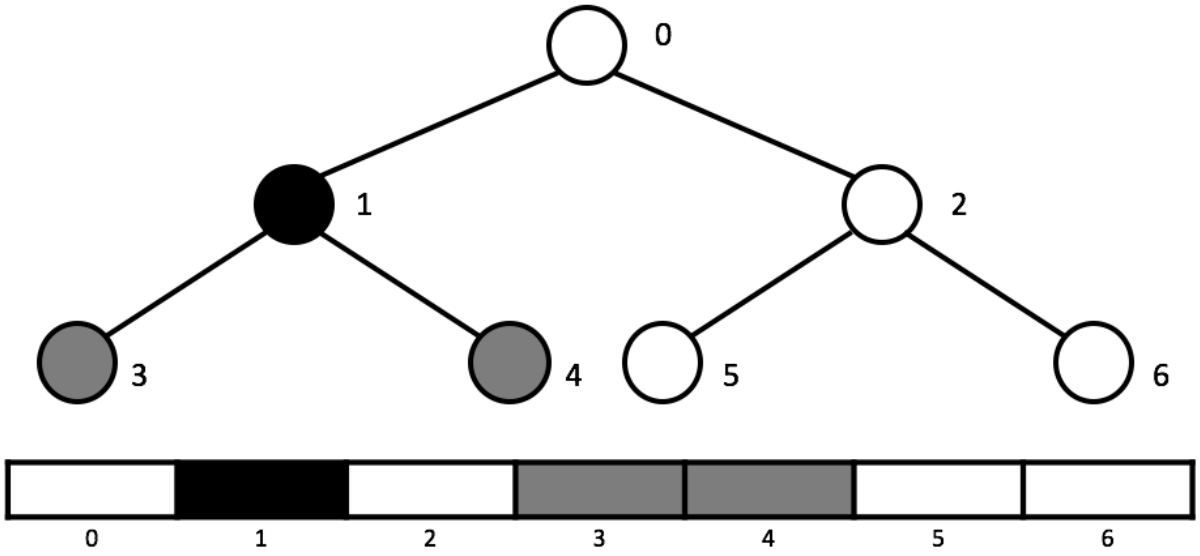

Figure 1: Heap Representation in an array. Node 1 is the parent of nodes 3 & 4 which are calculated using: parent ID *2 +1 & parent ID *2 +2.

The heap representation was implemented in order to have a process that would run efficiently when ported to the GPU. Recursion did not take advantage of multiple threads running simultaneously. However, with the new iterative method, every process that is in the same row of the heap would run in parallel. Every row in the heap structure includes the parents of the next row, which stores double the number of nodes. This means that as the heap grows, depending on the number of events run, more processes can be run in parallel, which will effect the run-time.

Array Compression

The recursive algorithm implements a pre-order traversal of the partons. This traversal does not keep track of the right child as the array fills up. However, when storing into an array through a counter that increments by 1 after every iteration, every position in the array is filled. Using the formula for a heap, every node is processed regardless of whether or not the parton

is splitting. This leaves many empty nodes and uses up much more memory than the recursive version. To deal with this issue, the array can be compressed to get rid of the empty nodes. This was implemented after the call to the shower function in the main function. The code loops through the array to find an empty node and assigns it, index i. It then continues to increment until it finds the next true node and assigns it index j. Then, index j is copied to index i and the process is repeated until the counter reaches the value of i_line, the counter for the SV array. This compresses the array by removing all the empty nodes and copying the full nodes to their positions. The following pseudo-code represents the iteration through the array to compress the nodes and update the printed size of the array.

```
for (k<= i_line)
{
        if the node is true (contains information)
                continue;
        else if the node is false (empty)
        {
                i = k+1;
                while i <= i_line the node is empty
                        i++
                if(i < i_line)
                        {
                        Assign S[i] values to S[k]
                        Assign F[i] values to F[k]

                        Set the i node in the heap to false
                        new_i_line = k+1;
                        }
        }
}
i_line = new_i_line
```

This also solves the issue of having a vastly different value for i_line in the recursive version and the iterative version. The value of i_line is used to calculate the mean number of shower

partons. After assigning i_line to a more accurate value, represented by index i, the mean number of shower partons was aligned with the original code. The array compression is represented in figure 2.

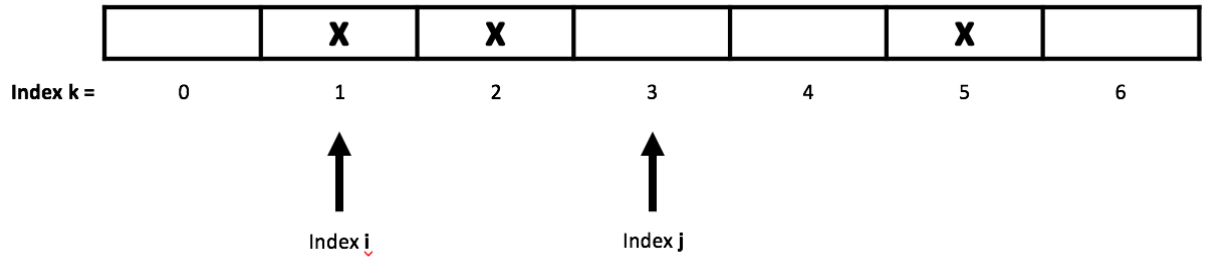

Figure 2: Array Compression – index j is copied to index i (an empty node)

Results

The code was run to compare the average values with those of the original code, in order to determine whether or not the changes made maintained accurate results. Every time the code is run, a random number generator is used to process the events. Therefore, there is no specific measure of how similar the data is. The results were just tested for similar values. Figure 3 depicts the different results for the recursive and the iterative version. The time taken to process 100,000 events with the iterative version took about 0.1 minutes longer. This could be a result of compressing the array after processing the shower function. However, decreasing the amount of memory required to store the information for every parton compensates for the difference in time. The code calculated the mean number of shower partons, detected partons, energy and virtuality. In both versions of the code, these averages were not more than approximately 1 value apart.

When the terminal processes the code, an error appears continuously until all the events are processed. The error reads, Issues with z_g in corrector. This refers to an issue with the S array assignment. The corrector looks through the S array and looks for the children that match up with every parent. Since the iterative version uses a different formula for assigning the children, the array was compressed to get rid of this warning. However, despite the array compression, the corrector continues to complain about the assignment in the S array. This error exists because when the array is compressed, the parent IDs are not properly updates. Therefore, the code is looking for the parent in the wrong location.

	Recursive Version	**Iterative Version**
Mean no. of shower partons =	11.5158	12.7654
Mean no. of detected partons =	2.72871	2.73125
Mean energy of detected partons =	75.2037	75.0583
Mean virtuality of detected partons =	10.7708	11.3777
Mean energy outside cone of one radian =	0	0
Mean_max_p =	45.5764	45.5523
Mean max virt =	1.9568	1.95718
Time taken in minutes :	52.8901	52.9103
Average time per event in seconds :	0.0317341	0.0317462
Average # of events / second :	31.5119	31.4999

Figure 3: Results for running 100,000 Events

Conclusion

Due to the time constraints in conducting this research, and the number of obstacles encountered, the results were limited. The results of removing the recursion from the CPU proved to be consistent with the results of the original code. Since the results are based on a random variable and the accuracy is dependent on the number of events run, running one hundred thousand events with very similar averages means the results were consistent.

In the future, this research will be applied to other functions within the code. The recursion will be removed from the function being called in the shower call. This recursive process may be removed through an implementation of Simpson's rule to approximate the integration. The array compression will also be optimized to properly compress each array after removing recursion. Eventually, the code will be made ready to be ported to the GPU. This will require more planning and research to be done in order to determine the most efficient way of using the blocks on the GPU to optimize the code. While the code can be transferred, specifying each threads ID and determining the number of blocks that will be used will also have an affect on how much speedup can be obtained.

References

Bakhoda, A., Yuan, G. L., Fung, W. W., Wong, H., & Aamodt, T. M. (2009, April). Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (pp. 163-174). IEEE.

Barlas, G. (2015). *Multicore and GPU Programming: an Integrated Approach*. Waltham, MA: Elsevier.

Fan, Z., Qiu, F., Kaufman, A., & Yoakum-Stover, S. (2004, November). GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (p. 47). IEEE Computer Society.

Hemalatha, V., & Kodada, B. B. (2013). High-Performance Computing using GPUs.

Kindratenko, V. V., Enos, J. J., Shi, G., Showerman, M. T., Arnold, G. W., Stone, J. E., ... & Hwu, W. M. (2009, August). GPU clusters for high-performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops* (pp. 1-8). IEEE.

Kirk, D. (2007, October). NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM* (Vol. 7, pp. 103-104).

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & Hwu, W. M. W. (2008, February). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (pp. 73-82). ACM.

Sander, J., Kandrot, E., & Dongarra, J. J. (2015). *CUDA by example: an introduction to general-purpose GPU programming.* Upper Saddle River: Addison-Wesley/Perason Education.

Shi, L., Chen, H., Sun, J., & Li, K. (2012). vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, *61*(6), 804-816.

Vakulin, N., Shaw, R., & Livingston, T. (2013). High Performance Computing with GPUs.