Overall Effects of Offsetting Data-Intensive, General-Purpose Tasks to Graphics Processing Units

William R. Jones

For Graduation Requirements B.COSC

Algoma University College

Supervisor: Dr. George Townsend

March 31, 2008

Acknowledgements

I would like to thank Dr. George Townsend for his help in creating this thesis paper and advice throughout the semester as well as for being my first reader. I would also like to extend thanks to Dr. Yi Feng for her help as a second reader for this thesis.

As well, I would like to thank everyone else who may have lent a hand along the way.

Contents

List of Figures

List of Tables

Chapter One

Introduction

1.1 Purpose

The purpose of this thesis is to analyze the potential effects of offsetting different data-intensive, general purpose tasks to the GPU (Graphics Processing Unit). The focus will not be on whether or not improvements are attainable by using the GPU for general-purpose computations, as this is already a well-established fact. Instead, the focus will be on examining speed increases attainable by this method, and on determining the effects of parallelization, work efficiency, and bank conflict avoidance on the improvements. As well, any other factors which can contribute to, or hinder the speed at which these parallel computations can be done on the GPU will be discussed and analyzed.

1.2 Methods

The methods used in this paper to analyze the effects of offsetting data-intensive, general purpose tasks to graphics processing units will be two-fold. First, each test will be introduced and examined in order to determine how exactly it is constructed, as well as to attempt to understand the reasons why it is constructed as it is. This analysis will hopefully help in understanding some of the intricacies of GPGPU (General-Purpose [Computation on] Graphics Processing Units) programming. Second, test applications from the NVIDIA CUDA (Compute Unified Device Architecture) SDK, which will be slightly altered to suit the needs of this paper, will be run on randomized data sets of varied lengths in order to give a good overall picture of how each test performs on both smaller and larger sets of data.

Some tests will contain a comparison between the same application running on the CPU and on the GPU. Some tests will, instead, be used to compare the importance of creating work-efficient algorithms as well as evaluating the importance of avoiding memory bank conflicts which can arise during computation.

As well, an analysis of the results attained from said testing will be examined in an attempt to garner knowledge about the overall performance of GPUs for general purpose computing. Test results will be presented in both table and graph formats for ease of examination.

Chapter Two

General Purpose Computation on Graphics Processing Units

2.1 Generic GPGPU Background Information

This section will contain a brief review of general-purpose computation on graphics processing units. Instead of focusing on a particular vendor's implementation, such as NVIDIA's CUDA architecture, this section will look at the history of GPGPU in general, some early pitfalls of the GPGPU movement, and some recent GPGPU improvements which have triggered increased awareness of GPGPU in the technical sector as a whole.

2.1.1 GPGPU Beginnings

Due to the driving force of the computer game market, graphics processing units have evolved immensely over the last decade. Although early uses of graphics processing units for general-purpose tasks can be traced back to a paper written by Jed Lengyel in 1990 [1], using graphics cards for general purpose computations was overly-complicated, tedious work since the hardware lacked many of today's programming conveniences [2]. The rapid development of graphics processing units over the last 10 years has brought with it many luxuries of modern programming such as read/write memory (2001), floating-point abilities (2002), and conditional execution (2004) [3].

Early adopters of GPGPU programming had to deal with many limitations and hardships, including, but not limited to: Hardware limitations, the requirement of dealing with the low-level details of graphics APIs and the requirement of using somewhat unorthodox coding techniques in order to generate results on the GPU.

2.1.2 Early GPGPU Pitfalls

Original work with general-purpose computations on graphics processors had many limitations and constraints, which hindered wide-spread adoption of GPGPU and which made it incredibly cumbersome for even the simplest of tasks. Some of the more painful aspects of early GPGPU were: Having to deal directly with the graphics API, limited communication abilities, and the incredibly sharp learning curve required to code for the GPU if a person had no prior graphics training or experience.

Originally the graphics card had to be tricked into performing as a general-purpose computing device. This required the use of a graphics API such as Microsoft's DirectX or OpenGL. This "tricking" of the GPU was accomplished by using texture maps to hold data, and rendering some geometric shape numerous times in order to process this data. The result texture then had to be read back to the CPU. The necessary graphics setup just mentioned, which is required when dealing directly with the graphics API, could also cause significant overhead.

The speed at which data could be transferred between the CPU and the GPU on older video cards was a huge concern in early GPGPU implementations. Minimizing data transfers between the CPU and GPU was extremely important, and so logically, this made programming general purpose tasks for the GPU even more difficult. Now, the 16x PCI-Express bus, found in most of today's new computers, has a theoretical maximum speed of 8000MB/s (2-way 4000MB/s max). The theoretical maximum speed of older technologies such as 8x AGP and PCI are 2100MB/s and 132MB/s, respectively [4]. Due to these recent advances in hardware technology, the bus speed limitation is not nearly as significant of a concern as it was in the early era of GPGPU, although minimizing data transfers in general is still very important.

As well as the above limitations, another major pitfall was the limited input/output abilities of the graphics cards when using the graphics APIs. More specifically, memory accesses were done as pixels. Two types of oft used operations, gather and scatter, which will be illustrated in the following diagrams, were also somewhat restricted in early GPGPU implementations. The "gather" DMA input/output operation could read data from multiple pixels; however, there was no support for "scatter", so only one pixel could be written to during the write phase.

The following illustration (figure 2.1) produced by Hendrik Lensch [5], displays this gather/scatter limitation graphically and makes it somewhat easier to understand in terms of the GPU.

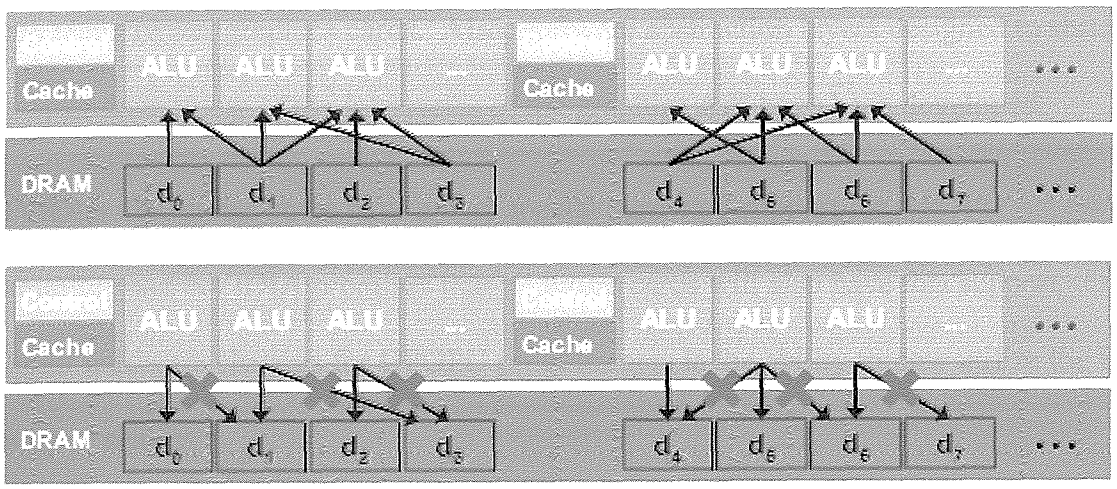


Figure 2.1: Limitations of Memory Accesses in Early GPGPU

The top half of figure 2.1 shows how the gather operation can read multiple pixels from the GPU DRAM; however, the bottom half of the figure illustrates the inability to utilize the I/O scatter technique, namely the inability to write data to more than a single pixel in DRAM.

Another hindrance of early GPGPU adoption and use was the steep learning curve associated with effectively coding for the GPU. People with little or no experience with graphics programming would be at an even greater disadvantage than those with this type of experience. In order to code what should be a relatively simple application, people with little or no experience with computer graphics APIs would be forced to first learn all about them, and before attempting to do the actual coding required.

2.1.3 Recent GPGPU Improvements

GPGPU programming has become easier over the last few years with the help of major universities and graphics card manufacturers. One major improvement in the GPGPU front of late has been the better hardware support for general-purpose computing which has been added to many newer graphics cards. Also, software abstraction layers designed to shield the programmer from many of the intricacies of the graphics API have been developed.

The NVIDIA G80 series chips have added special hardware support for GPGPU programming, and NVIDIA promises that all future cards will also have this support. One of the biggest hardware changes for GPGPU programmers is the ability to now utilize the I/O technique of scatter.

The following illustration (figure 2.2) produced by Hendrik Lensch [5], displays this scatter ability graphically.

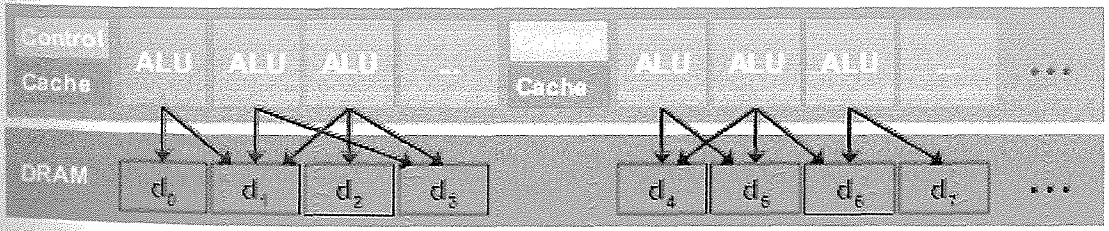


Figure 2.2: Support for Scatter I/O Operation in NVIDIA G80 Series+ Hardware

The ability of writing to more than one pixel opens up much more options for programmers, and helps eliminate one of the major hindrances of memory management which hurt early GPGPU adopters.

Another huge hardware improvement that was introduced by NVIDIA in the G80 series chips is on-chip shared memory, which can be used for sharing data between threads. This is extremely important, because, as mentioned above in the section about early GPGPU pitfalls, memory bandwidth between the host (CPU) and device (GPU) is a limiting factor and often a cause of processing bottlenecks. This bottleneck can cause data starvation (lack of data ready to be processed by open processors), which can seriously hinder overall performance and offset the benefit of parallelization. By properly utilizing this newly introduced on-chip memory, the amount of transfers required between host and device can be reduced, as can bottlenecks and processor data starvation [5].

The following illustration (figure 2.3) produced by Hendrik Lensch [5], gives a graphical depiction of the new, parallel, on-chip shared memory.

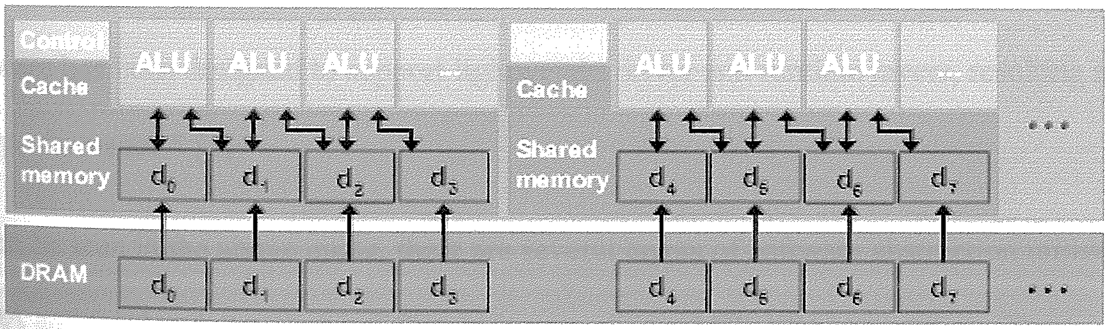


Figure 2.3: On-Chip Shared Memory on NVIDIA G80 Series+ Hardware

As can be seen in the above illustration, the processing units can utilize this new shared memory (which are limited to 16KB blocks in current generation NVIDIA cards) to share data between threads.

These mentioned hardware improvements introduced by NVIDIA are coupled with a relatively new software abstraction layer called NVIDIA CUDA. It allows the low-level details of the graphics API to remain hidden from the user, and it allows programmers to utilize the C programming language, with some CUDA extensions in order to develop programs for the GPU.

2.2 NVIDIA's Compute Unified Device Architecture

This section will outline the major components of NVIDIA's CUDA architecture and outline how it differs from the old ways of GPGPU programming.

The following statement is made by NVIDIA about the CUDA software architecture: "The CUDA software stack is composed of several layers [...] a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage, CUFFT and CUBLAS that are both described in separate documents. The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance." [6]

As can be seen in the above quote, there are several layers of software abstraction utilized in CUDA and this is what makes it different from traditional GPGPU approaches. The programmer need not be familiar with the details of graphics APIs any longer in order to utilize NVIDIA GPUs for general purpose computations. Although the process is still anything but simple, this is a major step forward, and a true testament to NVIDIA's dedication to the GPGPU front.

The following graphic, taken from the NVIDIA CUDA Programming Guide 1.1 [6], gives a visual depiction of the 3 main layers of software abstraction:

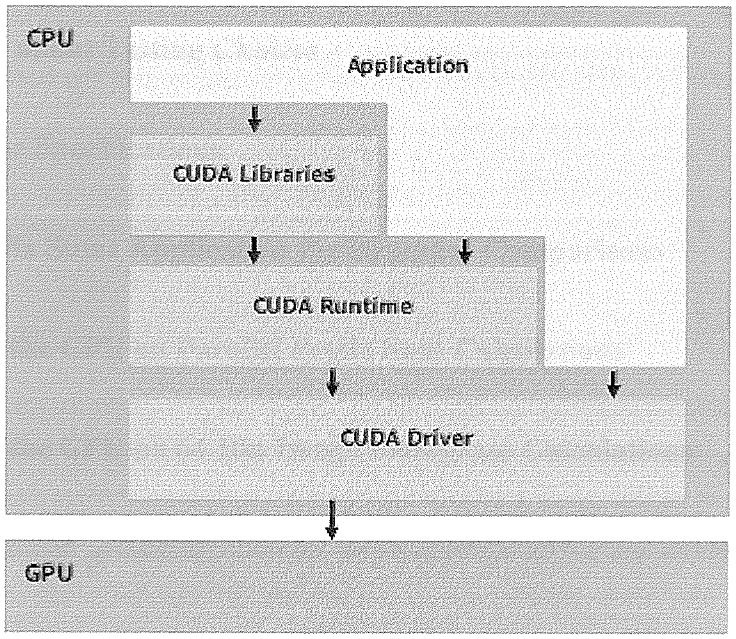


Figure 2.4: NVIDIA CUDA Software Abstraction Layers

Chapter Three

Testing and Comparisons of Various GPGPU Applications

3.1 Explanation about Testing Choices

The tests which have been chosen were provided by NVIDIA in their CUDA SDK [10]. Modifications were made to the code to control the number of elements used and the number of iterations which are done to garner an average result in order to analyze the effects. Many hours were spent on attempting to create applications specifically for this paper that would be sufficient for testing and comparison. Unfortunately, effectively coding on this platform is quite complicated for many reasons (many of which will follow in this section).

As well, GPU code cannot just be slightly altered and run on the CPU for testing and comparison purposes either, as this would skew the results in favor of the GPU, and vice-versa. Because of this restriction, completely different code, which has the same goal and produces the same results, needed to be developed and optimized for both the CPU and GPU separately. Since there are many sample programs on the NVIDIA CUDA developer site [10] which already have these characteristics, some of these demonstration programs have been chosen for the following tests.

3.2 Test Machine Specifications

The test machine used in all of the tests has the following specifications:

Motherboard

- ASUS P5W DH Deluxe v1.1
- Memory Bus Speed : 4x 166MHz (664MHz data rate)
- Maximum Memory Speed : 4x 200MHz (800MHz data rate)

CPU

- Intel Core 2 Duo 6600 @ 2.4GHz
- Cores: 2
- Speed : 2.40GHz
- Package : FC LGA775
- Rated Speed/FSB: 2400MHz / 4x 267MHz

Memory

- 2 1GB Sticks of OCZ DDR2-SDRAM
- Speed : PC2-6400U DDR2-400
- Dual Channel
- Timings : 5.0-5-5-15
- Technology : 16x(64Mx8) x 2
- Bus Width: 128-bit
- Effective Clock: 667 MHz

Video Card

- NVIDIA GeForce 8800 GT (GV-NX88T512H-B)
- PCIe Bus Width: x16 / x16
- Speed: 2.5Gb/s / 5.0Gb/s
- 112 Stream Processor
- 512 MB GDDR3 memory

3.3 Parallel Prefix Sums Application Performance Comparisons

This section will describe and analyze how the three simple parallel prefix sums applications from NVIDIA perform, and what the limitations of these applications are. Comparisons between the three simple scan implementations will be included. This will make it easy to examine what types of things affect the speed of the code on the GPU. All three algorithms are considered simple because they are limited to a single thread block, and as such, are limited to 512 simultaneous threads. Due to how this code is structured, this only allows a maximum of 512 elements in the array to be scanned. This is because the code is set to have exactly one thread for each element in the array. For this reason, only data set sizes which are powers of 2, and contain 512 or less elements can be used.

3.3.1 Information about the Chosen Test Applications

First this section will explain the all-prefix-sums application in a general way; not focusing on any particular implementation. According to the whitepaper located on the NVIDIA CUDA Developer website [10]:

"[The] all-prefix-sums [application] is a good example of a computation that seems inherently sequential, but for which there is an efficient parallel algorithm. The all-prefix-sums operation is defined as follows:

The all-prefix-sums operation takes a binary associative operator $\oplus$, and an array of n elements [a0, a1,..., an-1], and returns the array [a0, (a0 $\oplus$ a1),..., (a0 $\oplus$ a1 $\oplus$ ... $\oplus$ an-1)].

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array [3 1 7 0 4 1 6 3], would return: [3 4 11 11 14 16 22 25]." [7]

The sequential version of this algorithm performs one addition for every item in the array, and it has a work complexity of O(n). This is important because any code developed to run in a parallel fashion on the GPU must strive to have this same work complexity. Parallelized applications which do not have the same work complexity as the sequential versions are considered not to be work-efficient and on large data sets, the addition of this extra work complexity can seriously hinder performance.

The three simple versions of scan which will be examined are the naïve work-inefficient scan, the work-efficient scan which doesn't avoid memory bank conflicts, and the work-efficient scan which attempts to avoid bank conflicts. All three versions are included in the same C source code entitled "Scan" which can be located at the NVIDIA CUDA Developers website [10].

The first simple version of the all-prefix-sums application is what NVIDIA calls the "naïve scan". This algorithm for the parallel prefix sums operation, has a work complexity of $O(n \log_2 n)$. This work-inefficient algorithm is used as a benchmark against better versions of the code in order to determine the benefits of keeping the work complexity as low as the sequential version.

The next simple version of the all-prefix-sums application is work-efficient, having a work complexity of $O(n)$; however, there are many memory bank conflicts which force certain steps to be ran serially within the application and in turn, compromise the overall performance of this algorithm.

The last simple version of the all-prefix-sums application is both work-efficient and step-efficient. This means it has a work complexity of $O(n)$, just like the sequential approach, but this algorithm takes special precautions in order to avoid bank conflicts and therefore excess serialization does not occur, which lowers overall step count. This specific application can actually handle larger arrays and arrays which have a length which is not a power of 2;

however, since these applications are tested against each other, only arrays which the other applications support are tested in this section. Testing of larger arrays with this algorithm is done against the CPU in section 3.4.

3.3.2 Naive Parallel Prefix Sums

The naïve parallel prefix sums calculation used in this SDK application is relatively simplistic when compared to the other variants. According to NVIDIA, this version of the test was given the title "naïve" due to the fact that the algorithm is work-inefficient (has a higher work-complexity than the serial version), and also because it is limited to only using one thread block on the GPU. Due to the size of the thread blocks on the NVIDIA G80 series GPUs (512 threads) and because this algorithm only processes 1 element per thread, this limits the number of elements in the array which can be scanned to 512.

In the first step, the first item in the array obviously remains unaffected; however, every other element is summed with the element before it in the array. For example, the element with index one becomes the sum of the element with index 0 and itself, and so on. In the second step, this process is repeated; however, this time the sum is calculated between the current element and the element with an index which is two less than the current index. The index chosen at this point grows by a factor of two and the entire process is repeated log2 n times (n being the number of elements in the array). The number of elements at the beginning of the array which stay the same grow by a factor of two every iteration (because the calculation of sums up to that point is complete).

The following figure (3.1), taken from the application whitepaper [7] gives a graphical depiction of the steps taken by the naïve scan:

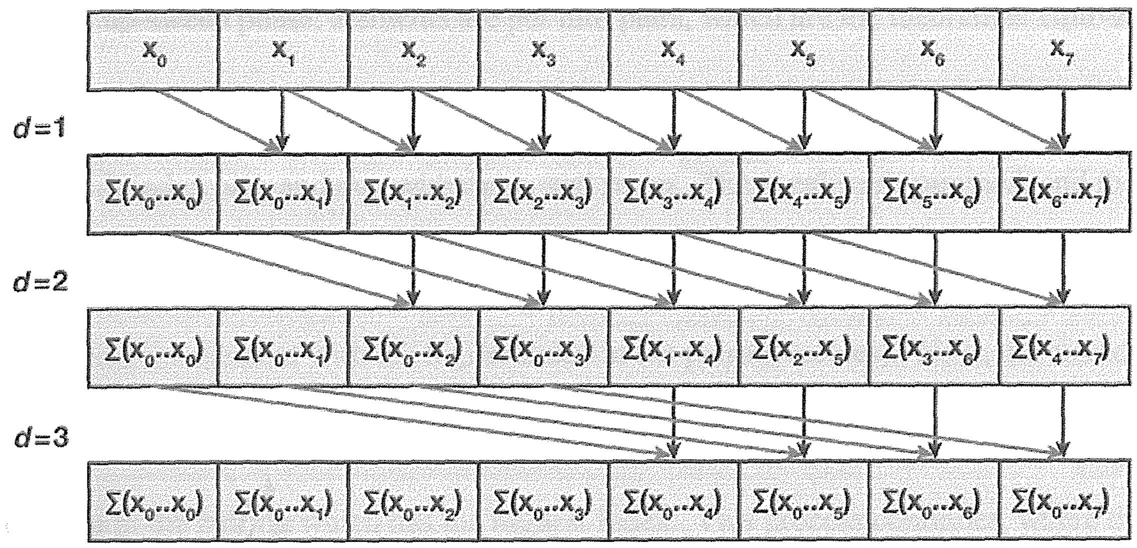

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
$d=1$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_1..x_2)$ $\Sigma(x_2..x_3)$ $\Sigma(x_3..x_4)$ $\Sigma(x_4..x_5)$ $\Sigma(x_5..x_6)$ $\Sigma(x_6..x_7)$
$d=2$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_1..x_4)$ $\Sigma(x_2..x_5)$ $\Sigma(x_3..x_6)$ $\Sigma(x_4..x_7)$
$d=3$
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$ $\Sigma(x_0..x_7)$

Figure 3.1: Depiction of a Scan of an 8 Element Array using the Naïve Approach

Of note here is the fact that this scan is performed on the same memory space, and therefore two temporary buffer arrays must be used for the calculations.

3.3.3 Work-Efficient Parallel Prefix Sums

The work-efficient parallel prefix sum application included in the NVIDIA CUDA SDK [10] attempts to minimize complexity of the scan process. This application, just as the one above, is limited in the size of the array which can be used, but with this algorithm, each thread processes two elements and so the maximum array size is 1024 elements. Also, just like the above algorithm, this one is limited to arrays which have a length that is a power of two.

This application utilizes a theoretical balanced binary tree approach to the scan. The process can be broken down into two main steps, which NVIDIA calls the up-sweep phase and the down-sweep phase [7].

During the up-sweep phase, elements are put into pairs, which are the theoretical equivalent of leaves on the binary tree. For example, the element with index zero is paired with the element with index one, and two with three, and so on. These pairs are summed, and the result is written to the array in the place where the parent node would reside (which is located where the element with the higher index in the pairing resides). Next, the elements at index one and three, five and seven, etcetera, are summed, and again, the sum is written to the element in the pairing which has the highest index. This process continues, where the chosen indexes continue to rise by a factor of two, until the final element contains the sum of all elements before it in the array (so the root node of the conceptual binary tree now contains the sum of all the leaves). Also every second array element is now a partial sum. [7]

The following figure (3.2), taken from the application whitepaper [7] gives a graphical depiction of the steps taken by the up-sweep part of the work-efficient parallel prefix sum scan:

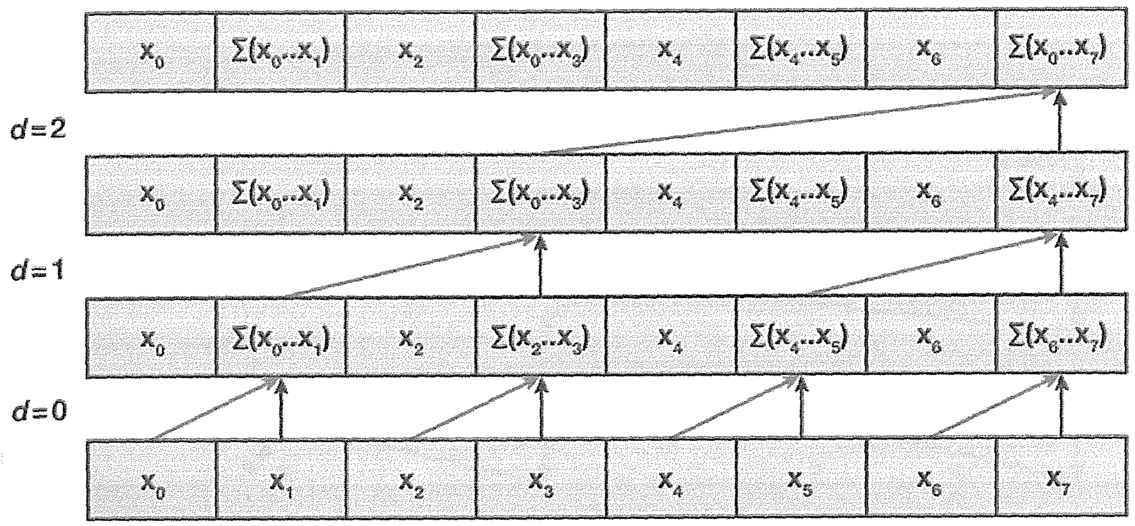
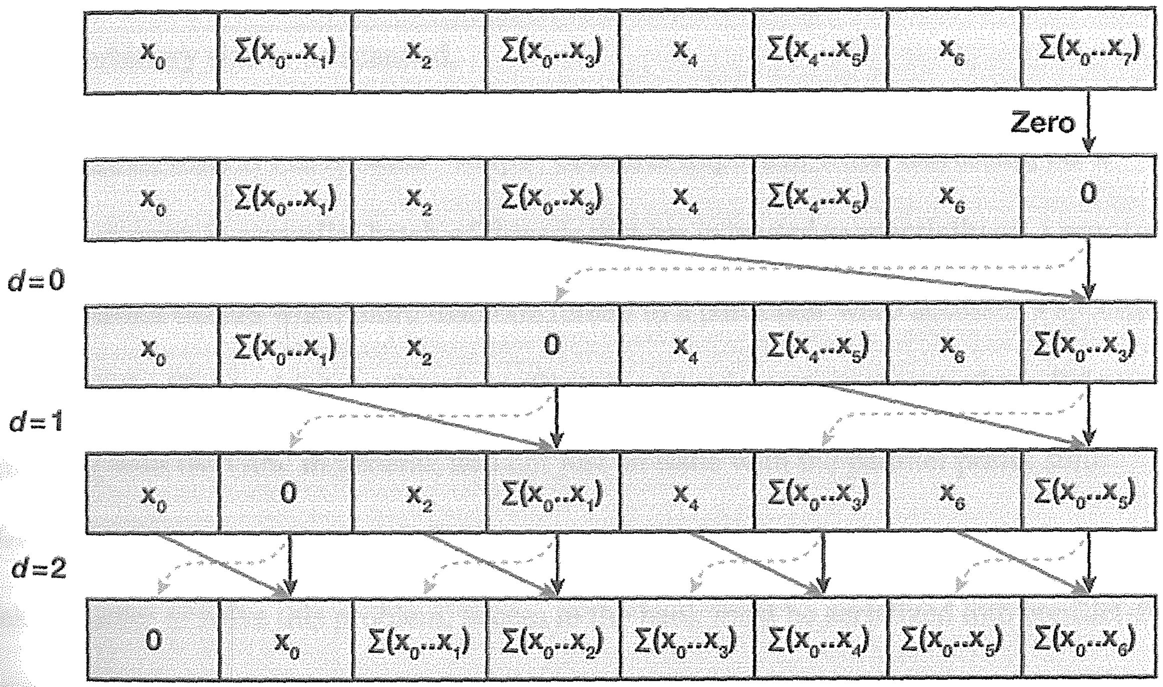


Figure 3.2: Depiction of the Up-Sweep Phase of the Work-Efficient Algorithm

The down-sweep phase is a bit more complex, and requires both summing and swapping of elements. The last element in the array is set to 0, since this is an exclusive prefix sum, and not an inclusive one. After this, the conceptual tree is traversed and the partial sums are gradually swapped into position, and in the last step, the elements in the array which have yet to be summed, are integrated and then replaced with the correct summed values. The 0, which replaced the last array element (the total sum), traverses the conceptual tree during the above process and at the completion of the algorithm, it resides in the first array index (leftmost leaf). This zero can simply be considered padding and is not used. [7]

The following figure (3.3), taken from the application whitepaper [7] gives a graphical depiction of the steps taken by the down-sweep part of the work-efficient parallel prefix sum scan:



x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_7)$
Zero
x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | 0
$d=0$
x_0 | $\Sigma(x_0..x_1)$ | x_2 | 0 | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_3)$
$d=1$
x_0 | 0 | x_2 | $\Sigma(x_0..x_1)$ | x_4 | $\Sigma(x_0..x_3)$ | x_6 | $\Sigma(x_0..x_5)$
$d=2$
0 | x_0 | $\Sigma(x_0..x_1)$ | $\Sigma(x_0..x_2)$ | $\Sigma(x_0..x_3)$ | $\Sigma(x_0..x_4)$ | $\Sigma(x_0..x_5)$ | $\Sigma(x_0..x_6)$

Figure 3.3: Depiction of the Down-Sweep Phase of the Work-Efficient Algorithm

As with the naïve scan, the work-efficient scan is performed in shared memory space; however with the work-efficient scan, no buffer arrays need to be utilized.

3.3.4 Work-Efficient Parallel Prefix Sums with Bank Conflict Avoidance

NVIDIA considers the work-efficient parallel prefix sum algorithm with bank conflict avoidance to be the best solution for this process. Because of this, the algorithm is also adapted to allow for arbitrarily large arrays, and arrays which do not have a length which is a power of two. Besides being more practical for actual use, NVIDIA suggests that this application should have much better performance, and since it will utilize more than one thread block, it should give a better true depiction of the actual processing power of the GPU. [7] This section will focus only on the differences between the regular work-efficient parallel prefix sum scan, and the improved one which avoids bank conflicts and allows arrays of arbitrary size to be scanned.

The concept of what NVIDIA calls “warps” becomes important when discussing bank conflicts. A warp is a parallel batch of threads that are executed sequentially on a processor. A bank conflict occurs when more than one thread in a particular warp accesses a specific memory bank. The avoidance of bank conflicts is a very important aspect of parallel computing with the GPU in general, and not just an issue with the parallel prefix sum calculation. The “degree” of a conflict is the number of threads in the warp which access the bank. In order to solve this problem, access to the bank must be serialized into conflict-free requests, and this can severely hurt overall performance of parallel algorithms. For example, on the NVIDIA G80 GPUs, the highest-degree bank conflict which can occur is a 16-degree conflict, and this will cause 16 serial accesses to this memory bank in order to finish the calculation and therefore 15 extra cycles must be wasted. [7]

In order to avoid these conflicts, changes to the code must be made. In particular, one must be cautious about how accesses to shared memory arrays are used. This must be done differently for each specific application since the way in which these bank accesses occur determines the reasons for conflicts.

According to NVIDIA, "[if one is to] add to the index the value of the index, divided by the number of shared memory banks", high-degree conflicts can be avoided [7].

The following illustration (figure 3.4) from the application whitepaper [7] gives a graphical depiction of the addressing technique used in the original work-efficient scan:

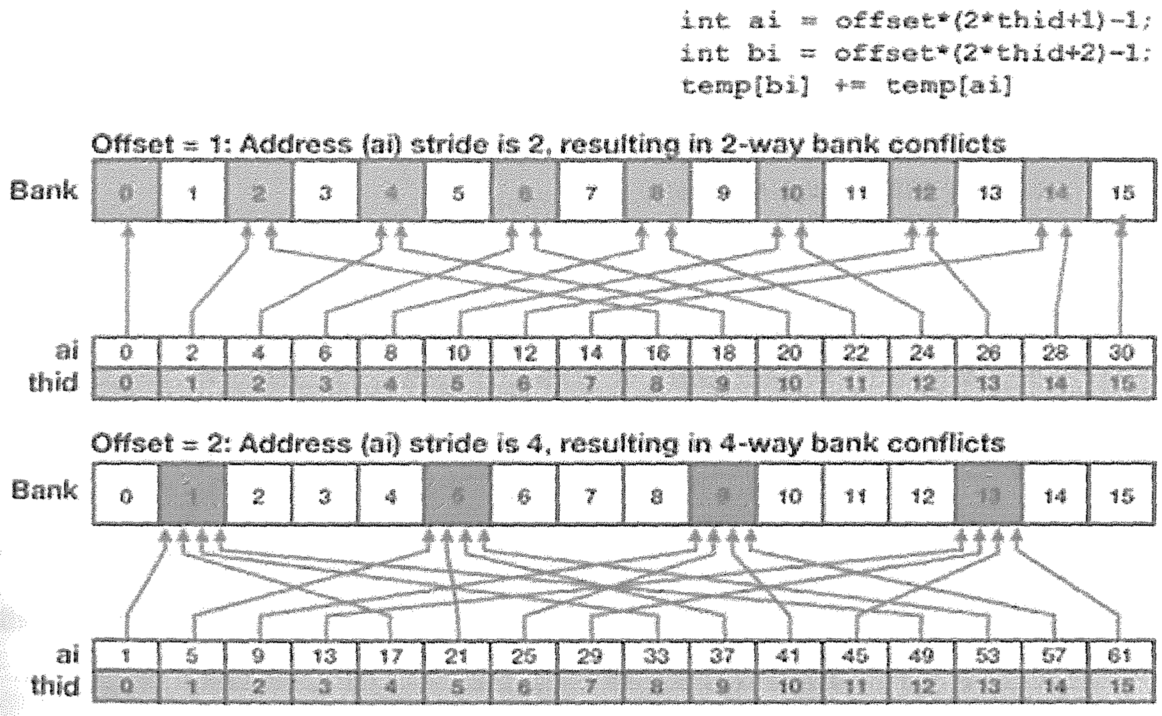

int ai = offset*(2*thid+1)-1;
int bi = offset*(2*thid+2)-1;
temp[bi] += temp[ai]
Offset = 1: Address (ai) stride is 2, resulting in 2-way bank conflicts
Bank
ai
thid
Offset = 2: Address (ai) stride is 4, resulting in 4-way bank conflicts
Bank
ai
thid

Figure 3.4: Depiction of Work-Efficient Algorithm Addressing Without Padding

As can be seen in the previous diagram, the tree-based approach of the work-efficient scan produces bank conflicts during traversal because as the theoretical tree is traversed, only banks (or array items) where the conceptual tree nodes would be located are used. So, when the leaves are accessed, every second array item is used, then every fourth, and so on. Due to the size of the warps on the NVIDIA G80 GPUs, bank-conflicts at the root node level can be as high as 16-degree.

NVIDIA offers a solution, which they suggest will work on many tree-based parallel computations and this address padding technique is illustrated in the following figure (3.5), taken from the application whitepaper [7]:

```
int ai = offset*(2*thid+1)-1;
int bi = offset*(2*thid+2)-1;
ai += ai / NUM_BANKS;
bi += bi / NUM_BANKS;
temp[bi] += temp[ai]
```

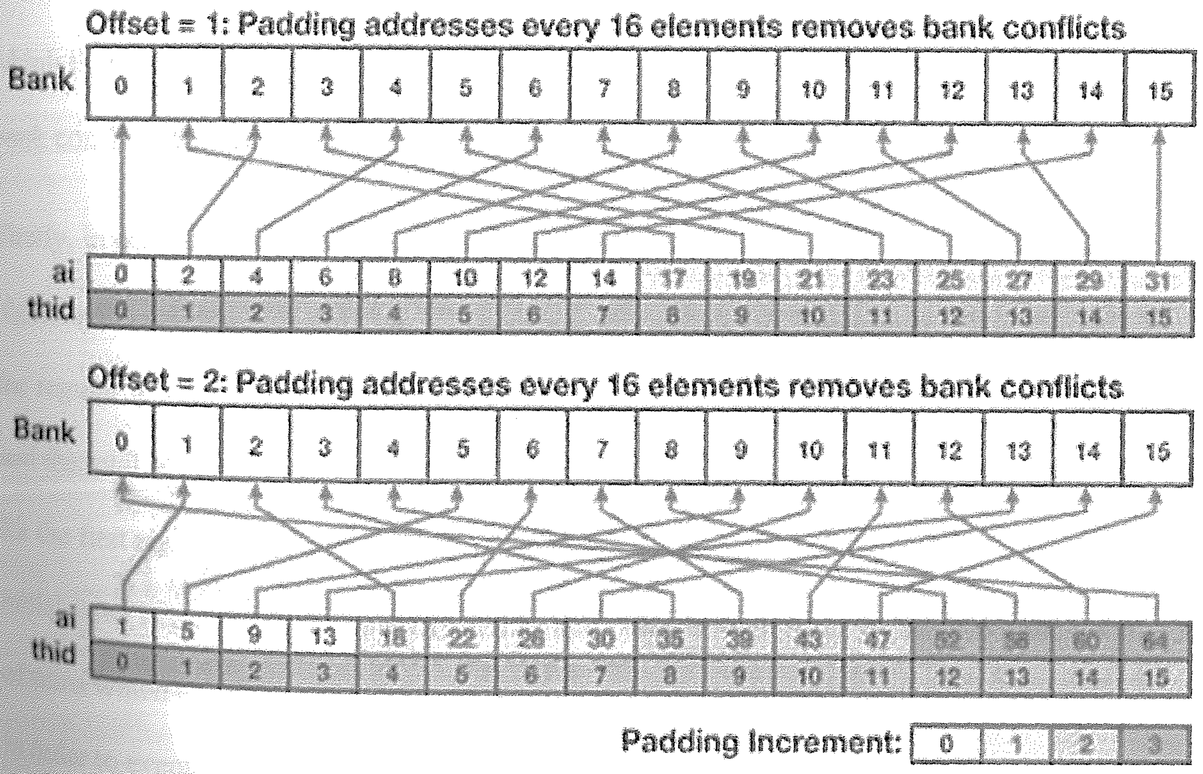


Figure 3.5: **Depiction of Work-Efficient Algorithm Addressing with Padding**

As can be seen in the above diagram, by padding the addresses every 16 elements, the bank conflicts shown in the previous diagram can be avoided. In the top half of the diagram, a padding of one was used which utilizes the memory banks which were not used in the previous diagram. Instead of having interleaved array items, which are then doubled-up, the entire array is used. In the bottom half of the diagram, the padding is incremented so as to effectively do the same thing (use the entire array, instead of having 4-degree conflicts across the entire array). This process is repeated all the way up the conceptual binary tree, and high-degree bank conflicts are avoided.

As earlier mentioned, the other two algorithms only allow scanning arrays of 512 elements, and 1024 elements respectively on NVIDIA G80 GPUs. Also, only arrays with a length which is a factor of two can be scanned properly. The work-efficient algorithm which avoids bank conflicts fixes these two limitations by adopting a more complex strategy.

Firstly, in order to overcome the array size limitation, the array to be scanned by this algorithm is spliced into separate blocks of no more than 1024 elements (the maximum that the regular work-efficient algorithm can support). After this, each separate block is scanned and the total of sum of each block is added to a new temporary array. For all blocks, except for the first one, the value of the sum of the blocks which come before it are added to the initial array item, and the final array can be recreated at this time. In other words, the array of sums which is created, is used to increment the next spliced block by the correct number in order to eventually generate the correct scan output which is regenerated in the final step.

The following figure (figure 3.6), taken from the application whitepaper [7] gives a graphical depiction of the above outlined process:

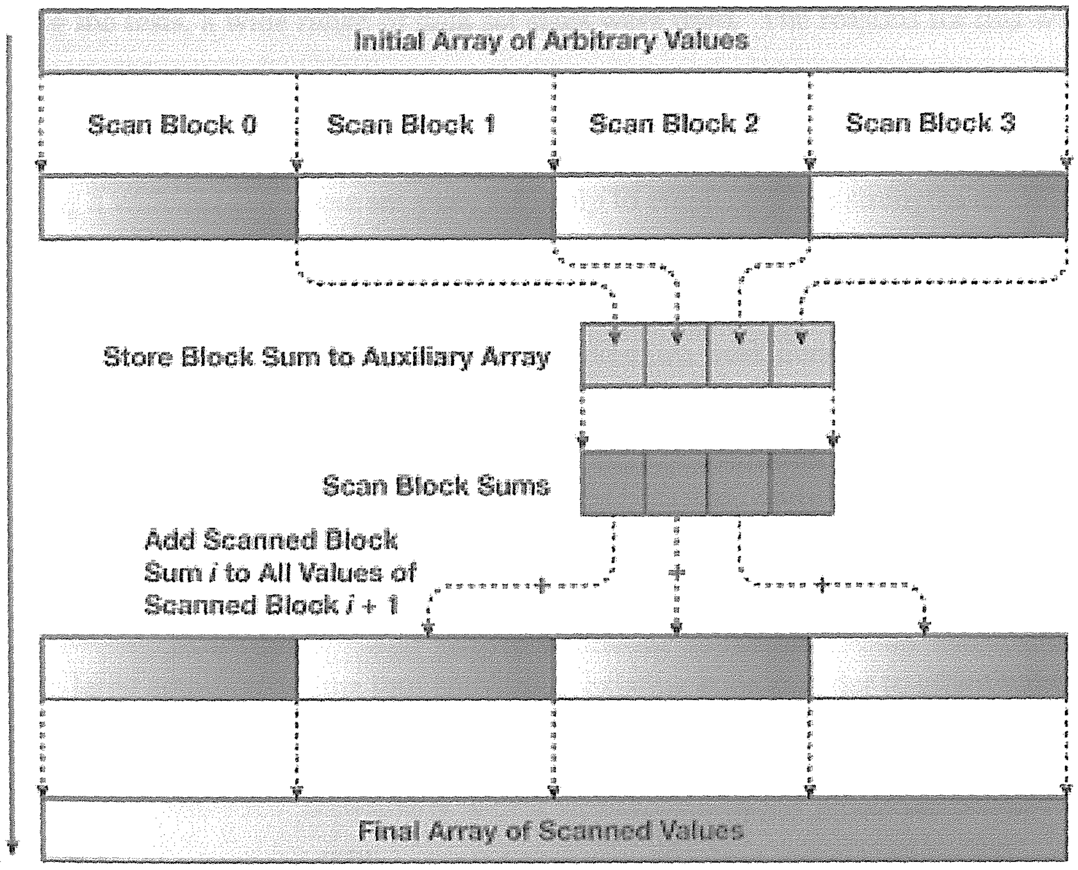

Initial Array of Arbitrary Values
Scan Block 0
Scan Block 1
Scan Block 2
Scan Block 3
Store Block Sum to Auxiliary Array
Scan Block Sums
Add Scanned Block Sum *i* to All Values of Scanned Block *i* + 1
Final Array of Scanned Values

Figure 3.6: Depiction of Method to Overcome Array Size Limitation

With only the above changes implemented, the application still could not handle arrays with a length which is not a power of two. NVIDIA says the following about overcoming this final limitation, "Handling non-power-of-two dimensions is easy. […] simply pad the array out to the next multiple of the block size B. The scan algorithm is not dependent on elements past the end of the array, so we don't have to use a special case for the last block." [7]

3.3.5 Test Results

Randomized tests data is automatically generated by the test applications in the NVIDIA SDK. This has not been altered because this will actually best simulate real-world situations. When doing the tests, a wide range of data set sizes were used. The reasons for this were two-fold.

Firstly, using a wide range of data set sizes will help to emphasize the strengths as well as the limitations of using the GPU for these types of computations. It is expected that when the number of elements is very small, the extra setup and extra work done by the more complex algorithms may actually result in worse performance than the naïve implementation.

Secondly, the use of a wide range of data set sizes should best simulate certain data-intensive processes in the real-world, specifically at my workplace. In order to truly rate the prospect of investing large amounts of time into converting existing code in order to effectively run on the GPU, data set sizes at all ends of the spectrum needed to be analyzed.

In order to ensure the best possibility of reliable test results, each test has been chosen to iterate 500 times and an average duration is calculated. As well, all non-essential applications on the test machine were terminated to eliminate any additional unnecessary load on the CPU.

The three parallel prefix sums applications were first tested on arrays with a length of two, and the length of the test array was incremented by powers of two up to a final array length of 512 elements.

Below is a table outlining the results of this test, which uses the same test data on all three applications of the parallel prefix sum scan:

Elements	Naive Scan	Work-Efficient	Avoiding Bank Conflicts
2	0.017 ms	0.021 ms	0.022 ms
4	0.016 ms	0.027 ms	0.029 ms
8	0.02 ms	0.04 ms	0.031 ms
16	0.024 ms	0.076 ms	0.037 ms
32	0.039 ms	0.133 ms	0.037 ms
64	0.062 ms	0.239 ms	0.055 ms
128	0.076 ms	0.234 ms	0.064 ms
256	0.157 ms	0.25 ms	0.103 ms
512	0.385 ms	0.587 ms	0.196 ms

Table 3.1: Results of Naïve, Work-Efficient and Bank Conflict Avoidance Algorithms on Parallel Prefix Sums Calculation

For simplicity in analyzing the data above, a graphical depiction now follows:

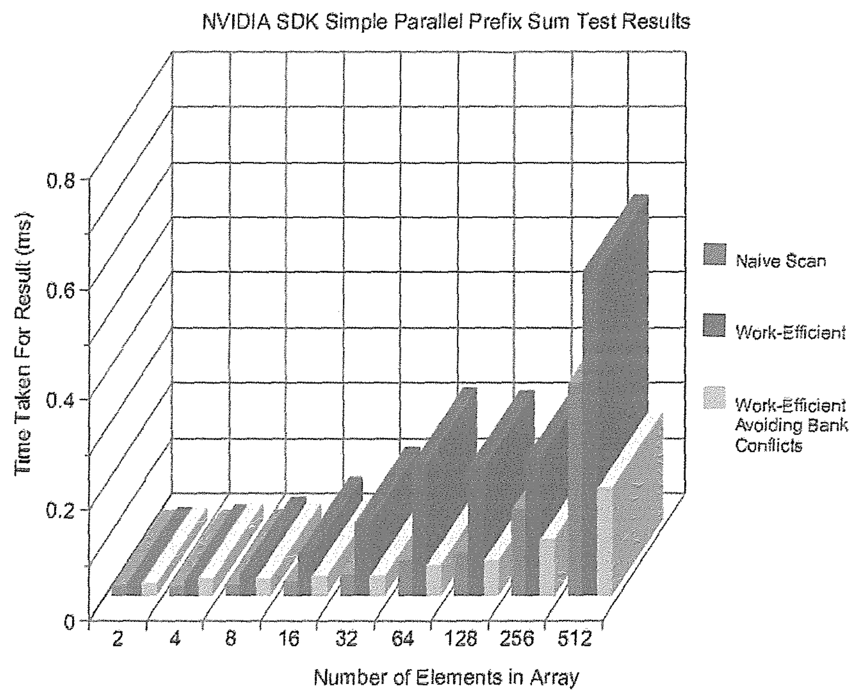


Figure 3.7: Graphical Results of Naïve, Work-Efficient and Bank Conflict Avoidance Algorithms on Parallel Prefix Sums Calculation

As can be seen in the above table and graph, when the number of elements was small (less than 32), the naïve scan actually outperforms both the work-efficient implementation as well as the work-efficient scan which avoids bank conflicts. This was somewhat expected as there is more setup and work required in these more complex algorithms than there is in the naïve scan. For very small sets (4 elements and under), the extra work done to avoid bank conflicts seems to make the application, which avoids bank conflicts, somewhat slower than the one which does not.

Also noteworthy is the fact that the work-efficient scan, which does not avoid bank conflicts, performs much more poorly than the naïve scan in every single case. In the final test, which used 512 elements, the work-efficient algorithm was actually approximately 65% slower than the naïve scan.

In addition, the scan algorithm which does employ techniques in order to avoid bank conflicts performs much better than both other applications when the data sets become large. On the test data set of 512 elements the work-efficient scan which avoids bank conflicts performed 2.99 times better than the work-efficient scan which does not avoid conflicts and 1.96 times faster than the naïve implementation.

3.3.6 Conclusions

As shown in the tests between the three parallel prefix sum algorithms, the cost of bank conflicts on this type of application is even more detrimental to overall performance than ensuring work-efficiency. When large amounts of bank conflicts occur, many cycles are wasted waiting for the serialized accesses to occur.

As well, due to the balanced binary tree approach of the work-efficient parallel prefix sum calculation, many bank conflicts are incurred. Because of this, the naïve scan, which of course does have more required steps in theory, actually significantly outperforms the work-efficient scan which does not employ bank conflict avoidance techniques.

Overall, these results make it apparent that with some types of parallel algorithms which can be designed to take advantage of the parallel nature of the GPU (such as the parallel prefix sums application), extreme care must be taken when accessing shared memory. This point can, at least sometimes, be even more important than ensuring that the work complexity of the parallel code matches that of the sequential code designed for use on a CPU. Also, if data sets are smaller, and they still must be calculated on the GPU for any specific reason, code which avoids bank conflicts and code which is work-efficient may actually not perform as well as a non-work-efficient implementation. With that in mind, it is difficult to foresee a situation where these small data sets could not simply be processed on a CPU instead of on a GPU since the CPU must always be operating as a host device for the GPU anyway.

All in all, it is quite obvious that since the GPU is only suggested for data-intensive applications in the first place, the work-efficient parallel prefix sum scan which avoids bank conflicts is, by far, the best implementation which has been tested.

3.4 GPU versus the CPU on Parallel Prefix Sums Calculations

In the previous section, it has been shown that the work-efficient parallel prefix sum scan which avoids bank conflicts is, by a significant margin, the best application which has been tested for use on the GPU for this purpose. Because of this, it will be used in this section to make some comparisons between the CPU and the GPU.

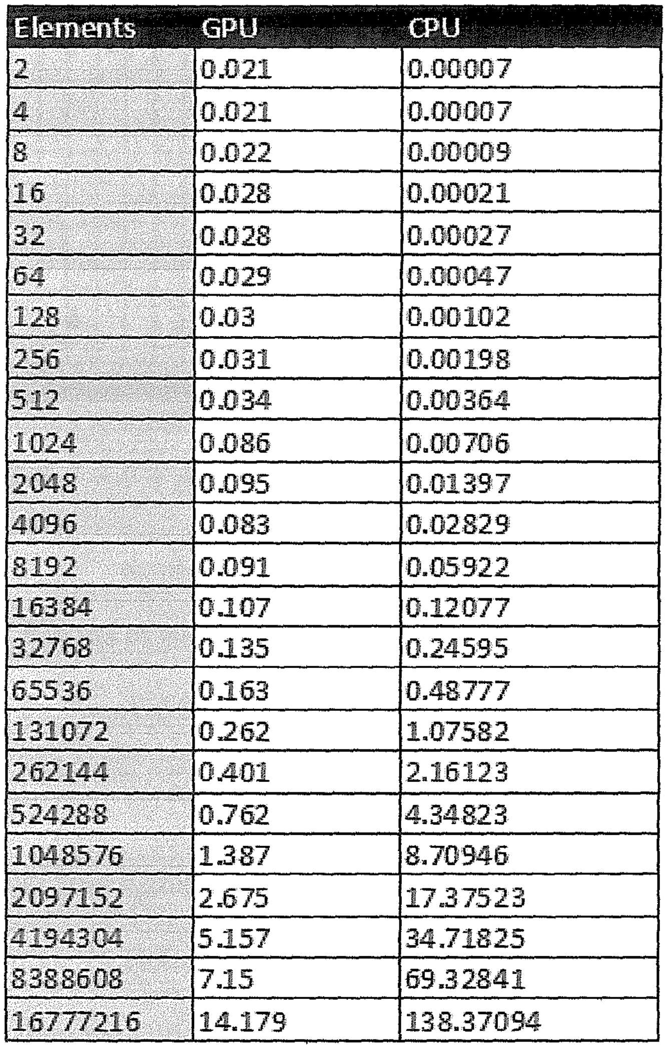
Optimized serial code which performs the exact same scan, designed for use on a CPU will be tested against this optimized GPU code in order to determine the speed increases possible when using the GPU for data-intensive general purpose computations. Of note here is that the code used for the CPU is sequential, and as such, it will not take advantage of the dual-core capabilities of the test hardware. The sequential code, which is offered by NVIDIA for the all-prefix-sums operation, is included in the code located on the NVIDIA CUDA Developers website [10].

3.4.1 Test Results

In these tests between the CPU and the GPU, randomized test data is generated by the code, just as it was in prior tests. Again, a very wide range of data set sizes have been tested, in order to determine the benefits and disadvantages of using the GPU for offsetting general purpose computations as opposed to the more traditional method of using the CPU for all calculations. Just as with the other tests in previous sections, the CPU and the GPU calculations are performed on the same randomized test data over 500 iterations, in order to generate reliable results.

In the first test, an array with only 2 elements is processed by both the CPU and the GPU, and from there the size of the tests array grows, by factors of two, until a final data set size of 16,777,216 is reached.

Below is a table outlining the results of this scan, using the GPU and the CPU:



Elements	GPU	CPU
2	0.021	0.00007
4	0.021	0.00007
8	0.022	0.00009
16	0.028	0.00021
32	0.028	0.00027
64	0.029	0.00047
128	0.03	0.00102
256	0.031	0.00198
512	0.034	0.00364
1024	0.086	0.00706
2048	0.095	0.01397
4096	0.083	0.02829
8192	0.091	0.05922
16384	0.107	0.12077
32768	0.135	0.24595
65536	0.163	0.48777
131072	0.262	1.07582
262144	0.401	2.16123
524288	0.762	4.34823
1048576	1.387	8.70946
2097152	2.675	17.37523
4194304	5.157	34.71825
8388608	7.15	69.32841
16777216	14.179	138.37094

Table 3.2: Results of Serialized CPU Code and Work-Efficient, Bank Conflict Avoidance Algorithms on Parallel Prefix Sums Calculation

For simplicity in analyzing the previous data, a graphical depiction now follows:

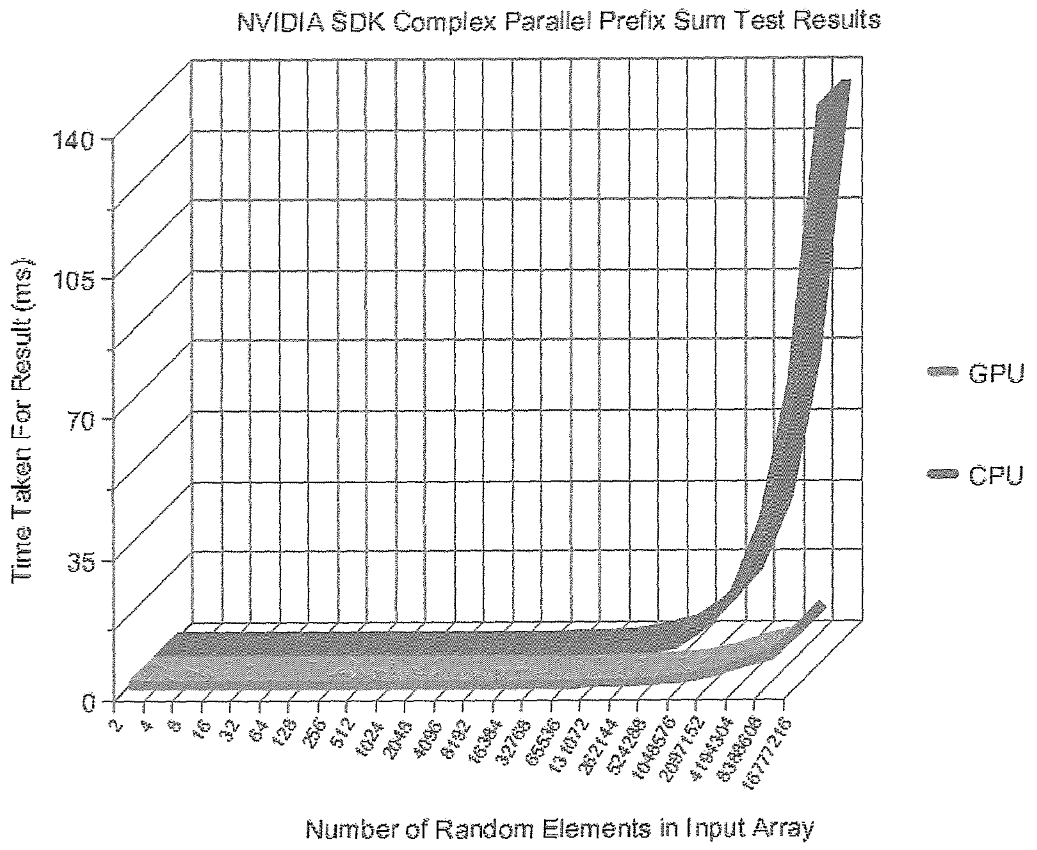


Figure 3.8: Graphical Results of Serialized CPU Code and Work-Efficient, Bank Conflict Avoidance Algorithms on Parallel Prefix Sums Calculation

As can be seen in the two above graphics, when smaller data sets are used, the CPU performs at least as well as the GPU. In fact, until the test of 16,384 array elements, the CPU performed much faster than the GPU and on a very small set (of 8 elements), the CPU performs approximately 314 times faster than the GPU.

Alternately, when very large sets of data are used, as in the test with 16,777,216 elements, the GPU far outperforms the CPU. In this specific test, the GPU performs approximately 9.76 times faster.

The following table, included in the application whitepaper [7], shows the results of the same test ran on slightly different hardware. The card used is an NVIDIA 8800 GTX (128 stream processors), and the CPU used is an Intel Core 2 Duo Extreme 2.93GHz.

# elements	CPU Scan (ms)	GPU Scan (ms)	Speedup
1024	0.002231	0.079492	0.03
32768	0.072663	0.106159	0.68
65536	0.146326	0.137006	1.07
131072	0.726429	0.200257	3.63
262144	1.454742	0.326900	4.45
524288	2.911067	0.624104	4.66
1048576	5.900097	1.118091	5.28
2097152	11.848376	2.099666	5.64
4194304	23.835931	4.062923	5.87
8388688	47.390906	7.987311	5.93
16777216	94.794598	15.854781	5.98

Table 3.3: Application Whitepaper Results on Different Hardware

A comparison of the results of table 3.3 and the results of table 3.2 are quite similar; however, of note here is the fact that the faster GPU used by NVIDIA in their test, seems to perform slightly worse than the one used in the testing for this paper on the array of 16,777,216 elements. One has to assume that NVIDIA's test was either performed while more graphical work was being done on the same computer, or their test was not averaged as thoroughly as the tests performed in this paper. For example, if the NVIDIA tests were performed using Microsoft Windows Vista, there would be more GPU resources wasted to display the Windows GUI than would have been used in the Windows XP environment used during the tests included in this paper. Alternately, it is also possible that less iterations were used in their test (the default is 100) and the code was slightly altered for the tests in this paper in order to iterate 500 times. This may have led to less accurate results in the NVIDIA test. Either way, the results are still quite similar even though a less powerful CPU and GPU were used.

Of note is the fact that when the CPU outperforms the GPU, although the margin is about 314 times faster, the time difference is only about 0.022 milliseconds. On the largest data set used in the test, the GPU is about 124.192 milliseconds faster than the CPU.

One more test was administered using the same code, but again with different hardware to illustrate the performance of the GPU and CPU. This time, a NVIDIA 8400 GS card was used which was also available for testing during the creation of this paper. The 8400 GS card is much less powerful than the main test card. It uses GDDR2 DRAM as opposed to the GDDR3 DRAM in the main test card. As well, it only has 256MB of this memory, compared to the 512MB available in the main test card. Lastly, and of probably the most consequence, is the fact that this card only has 16 stream processors compared to the 112 in the main test card, and the 128 available in the 8800 GTX card which was displayed above in table 3.3. Although this may be true, the card still performs relatively well on the largest data set, but falls short of outperforming the Intel Core 2 Duo 2.4GHz processor used on the test machine.

The 8400 GS result was a time of 158.45 milliseconds over an average taken after 500 iterations of the parallel prefix scan with bank conflict avoidance. This is approximately 20 milliseconds slower than the CPU for this specific test; however, as mentioned above, the 8400 GS only has 16 stream processors. This limitation seriously hinders the amount of parallelization which can be attained, which obviously hurts the overall result.

The following graph (figure 3.9) shows the average time taken to compute the parallel prefix sums operation on an array of length 16,777,216 using the NVIDIA 8800 GTS (results taken from the application whitepaper), the NVIDIA 8600 GT (results taken from table 3.2), the NVIDIA 8400 GS (results taken from test which was just mentioned), the Intel Core 2 Duo Extreme 2.93GHz CPU (results taken from application whitepaper) [5] and the Intel Core 2 Duo 2.4GHz CPU (results taken from table 3.2).

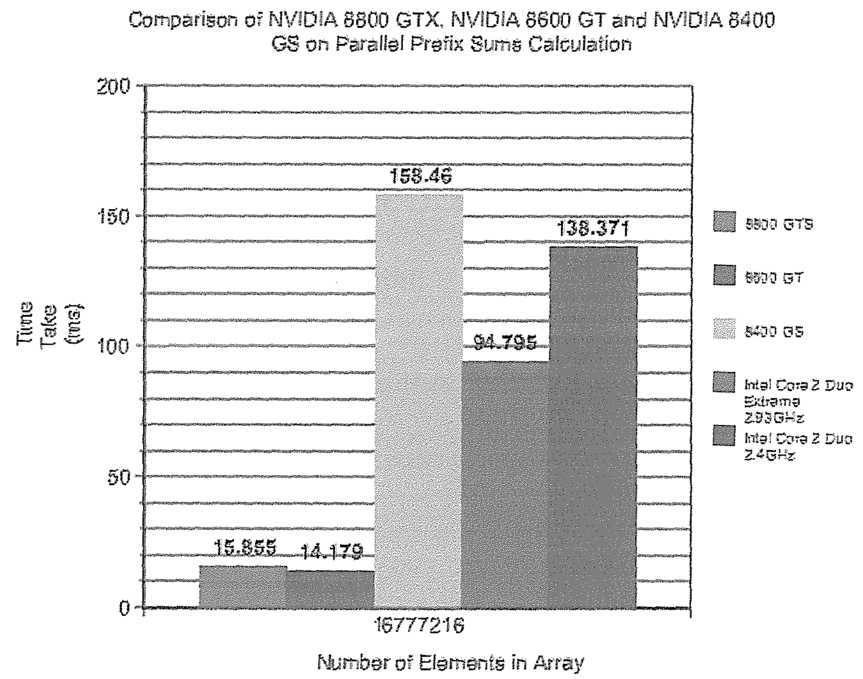


Figure 3.9: Graphical Depiction of Three Different NVIDIA Cards and Two Different Intel Processors on the Parallel Prefix Sums Calculation

As can be seen from figure 3.9, and as beforehand mentioned, the reason for the faster performance by the less powerful NVIDIA card (8600 GT), could come down to other operations (such as a more demanding GUI display) performed by the more expensive card, or it may come down to less accurate results due to less iterations. Either way, the GPUs with the above 100 stream processors seriously outperform the CPU and the less powerful 8400 GS at the parallel prefix sums calculation.

3.4.2 Conclusions

Calculating parallel prefix sums on the CPU and on the GPU gives interesting results. As expected, when the data set is small, the CPU outperforms the GPU by a large margin; however, as the size of the data set grows, the GPU overtakes the CPU in performance.

As mentioned in the test results section, the time difference between the GPU and the CPU when the data set is small, and CPU outperforms the GPU is much less significant than when the data set is large and the GPU does better. For this reason, when data sets vary in size, but there are enough very large data sets, the GPU is certainly the best choice for data processing similar to the parallel prefix sums calculation.

It is very clear by examining the test results that, at least with this specific application, the ability to offset data-intensive, general-purpose computations to the GPU can generate huge performance gains. If the size of the data sets are going to be small, this type of algorithm will not be effective on the GPU because the CPU performs the control operations much more efficiently than the GPU.

Another important note here is that it may be possible to do other things to utilize the GPUs parallel processing prowess for this type of calculation. For example, if the length of the data sets is known to be small beforehand, it is certainly feasible that an algorithm which combines many arrays and uses larger block transfers could possibly be created in order to still exploit the GPU effectively. This would depend on how often theses small sets are offset to the GPU and many other factors. However, with this specific NVIDIA CUDA code, it is clear that small data sets are processed inefficiently on the GPU.

The main reason for the inclusion of the test on the 8400 GS video card within the context of this paper was to display the fact that this seriously low-end G80 card can almost perform as well as the much higher priced CPU. In fact, the current market price for an 8400 GS card is under $40, and if there were multiple cards set up in a multi-card configuration, it could conceivably dwarf the CPU used in the test machine at this particular test. When one considers the price of the CPU used in the test machine is currently approximately $240, it begins to become clear as to how powerful these video cards can be, especially ones which have DDR3 DRAM and over 100 stream processors.

3.5 GPU versus the CPU on 64-Bin Image Histogram Calculations

In this section another application from the NVIDIA CUDA development kit (64-Bin Histogram) is analyzed and tested against varied data sets. Ramtin Shams and R. A. Kennedy say the following about histograms in general and about the implementation in NVIDIA CUDA in particular in their journal entitled "Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices": "The histogram is a non-parametric density estimator which provides a consistent estimate of the probability density function (pfd) of the data being analyzed. The histogram is a fundamental statistical tool for data analysis which is used as an integral part of many scientific computational algorithms. Recent advances in graphics processor units (GPUs), most notably the introduction of CUDA by NVIDIA, allows implementation of non-graphical and general purpose algorithms on the GPU. [...] Due to its architecture, the GPU is a natural candidate for implementation of many scientific applications, which require high-precision and efficient processing of large amounts of data." [8]

3.5.1 Information about the Chosen Test Application

In this section, the 64-Bin Histogram application, included in the NVIDIA CUDA SDK will be examined. Although the SDK also has a 256-Bin Histogram application, the code for that is much more complicated, and according to the whitepaper on both histogram applications, the performance and concepts are similar [8]. For these reasons, the 64-Bin Histogram was chosen over the 256-Bin variant. Also of note, is the fact that due to the complexity of the actual implementation details, only the factors relevant to the performance of the application will be examined (namely, bank conflict avoidance).

Histograms can be used for a number of applications, including but not limited to: Exposure calculations and automatic image adjustments. Since this is a graphical application, it may seem as though the GPU should perform very well on this application, but it should be noted that these calculations are typically done on a CPU.

The following figure, taken from the whitepaper [8] on the 64-Bin Histogram application in the NVIDIA SDK, shows an example of an image histogram. The horizontal axis shows the amplitude (colour), while the vertical axis depicts the number of pixels in the image.

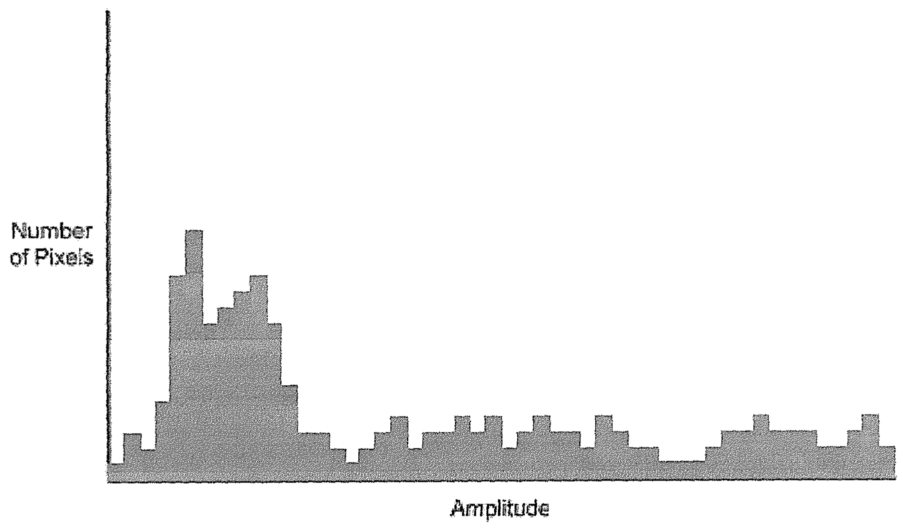


Figure 3.10: Graphical Depiction of Image Histogram

Parallelizing the data in order to efficiently process it on the GPU is relatively similar to the process used in the previous section on parallel prefix sums because in order for large data sets to be processed, they must be split up and then reorganized again at the end of the calculation.

According to NVIDIA, this process involves three steps: Subdividing the data between execution threads, processing these sub-histograms and merging the sub-histograms back into a single histogram. In actuality, depending on the size of the data set, the final merge step might actually involve two stages. The first stage is merging the thread sub-histograms into block sub-histograms, and then merging the blocks into a final histogram solution.

NVIDIA suggests that the most important factor in this application, as well as with the parallel prefix sum calculation is avoiding bank conflicts. The following graphic, from the application whitepaper [9], shows the memory layout:

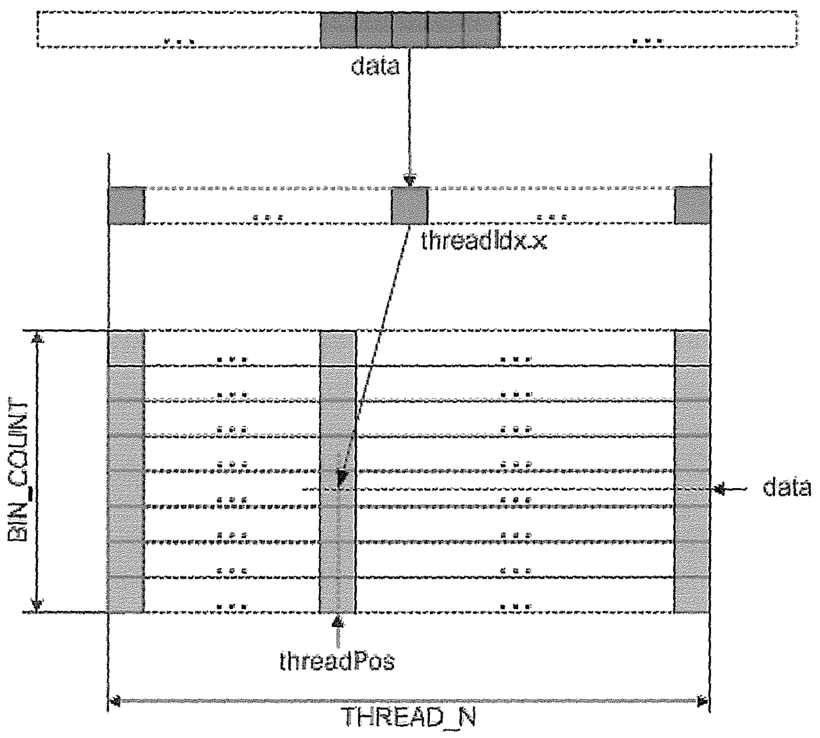


Figure 3.11: Graphical Depiction of Memory Layout of 64-Bin Histogram Application

The following is said about avoiding bank conflicts in the first portion of this code, "If we just set threadPos equal to threadIdx.x, all threads within a half-warp will access its own byte "lane", but these lanes will map to only 4 banks, thus introducing 4-way bank conflicts.

However, shuffling the [5 : 4] and [3 : 0] bit ranges of threadIdx.x will cause all threads within each warp to access the same byte within double words, stored in 16 different banks, thus completely avoiding bank conflicts." [9] In other words, by utilizing this shuffling trick, the code is able to completely avoid bank conflicts, and will perform much better than it otherwise would.

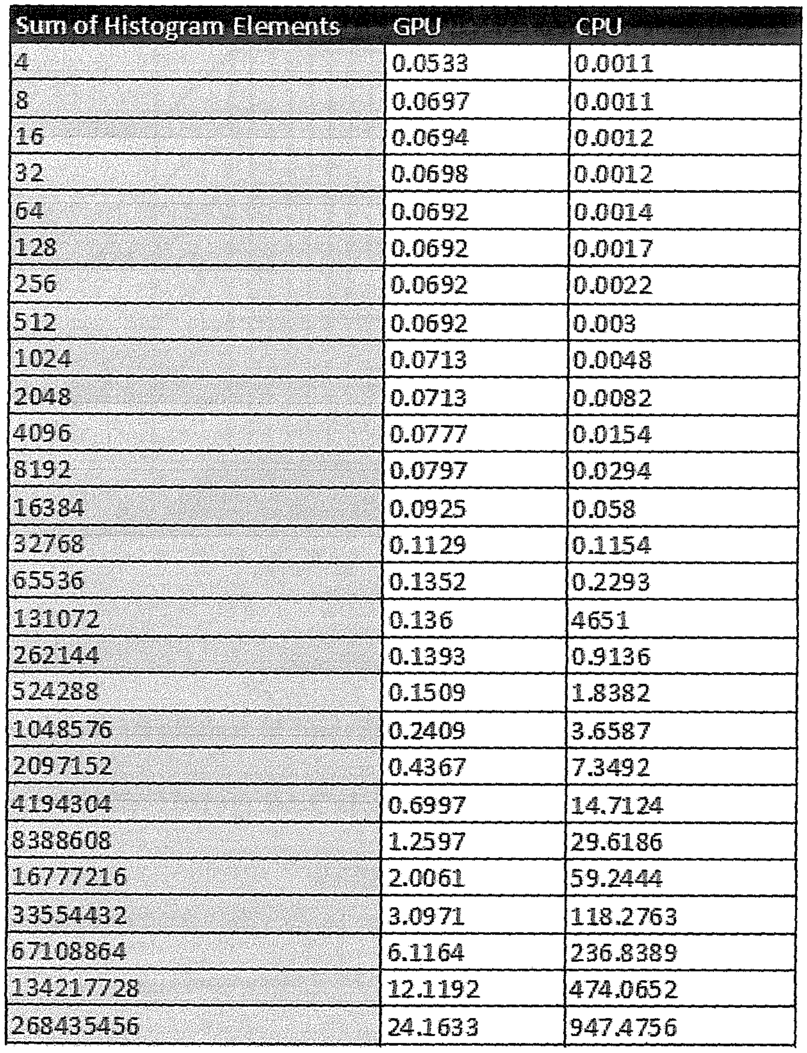
For a more detailed explanation of the exact operations of the 64-Bin Histogram, see the source code entitled "Histogram64" located at the NVIDIA CUDA Developers website. [10]

3.5.2 Test Results and Conclusions

The tests in this section are performed between the CPU and the GPU. Optimized serial code is used for the CPU, and optimized parallel code is utilized for the GPU. As in previous tests, the data sets generated are random, and a wide variety of data set sizes are used. Again, each test is ran through 500 iterations in order to improve accuracy of the results. These tests are done in order to examine the benefits of using the GPU for histogram calculations instead of the more traditional approach of using the CPU. Also of note is the fact that the data used for each test is identical between the CPU and the GPU.

The first test which was done started out with 4 histogram elements and this number was increased by a factor of two until reaching a final size of 268,435,456 histogram elements. Tests were done on both time taken (in milliseconds) and on memory bandwidth performance (in megabytes per second).

The table that follows shows the time taken in milliseconds to compute the 64-bin histogram on both the CPU and the GPU:



Sum of Histogram Elements	GPU	CPU
4	0.0533	0.0011
8	0.0697	0.0011
16	0.0694	0.0012
32	0.0698	0.0012
64	0.0692	0.0014
128	0.0692	0.0017
256	0.0692	0.0022
512	0.0692	0.003
1024	0.0713	0.0048
2048	0.0713	0.0082
4096	0.0777	0.0154
8192	0.0797	0.0294
16384	0.0925	0.058
32768	0.1129	0.1154
65536	0.1352	0.2293
131072	0.136	4651
262144	0.1393	0.9136
524288	0.1509	1.8382
1048576	0.2409	3.6587
2097152	0.4367	7.3492
4194304	0.6997	14.7124
8388608	1.2597	29.6186
16777216	2.0061	59.2444
33554432	3.0971	118.2763
67108864	6.1164	236.8389
134217728	12.1192	474.0652
268435456	24.1633	947.4756

Table 3.4: Speeds of Serialized CPU Code and Parallelized GPU Code Running a 64-Bin Histogram Calculation on the CPU and GPU respectively (in milliseconds)

For easy of examination, the following graph depicts the same information visually:

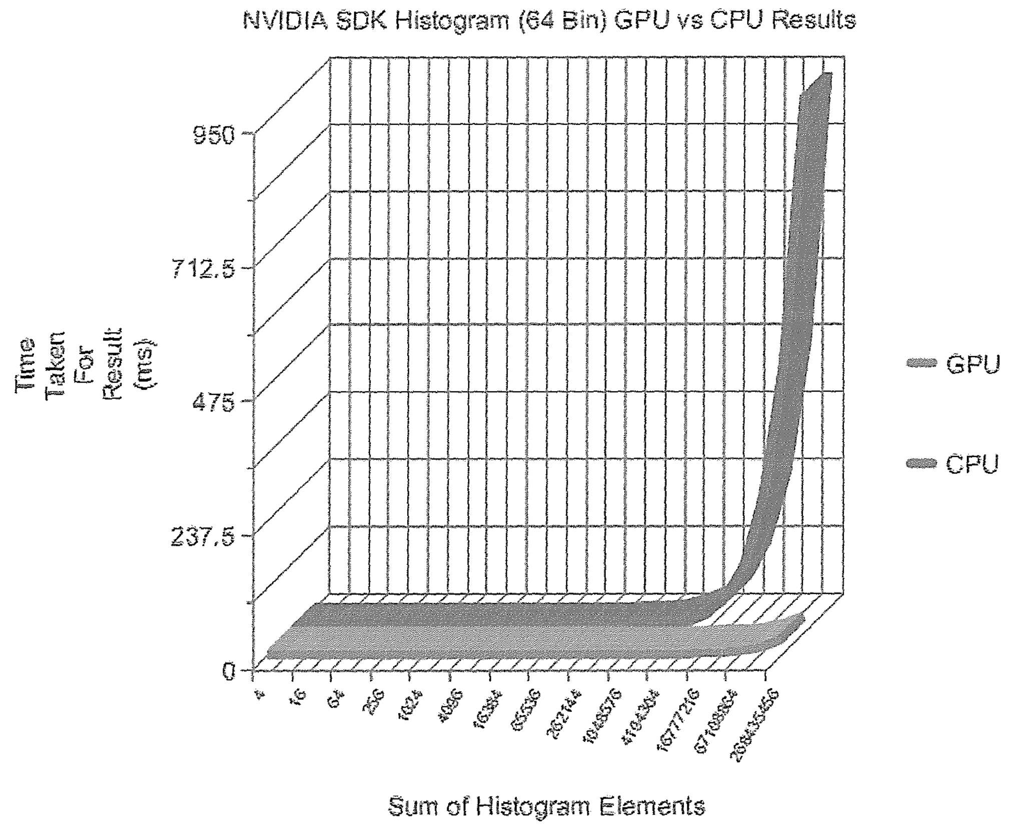
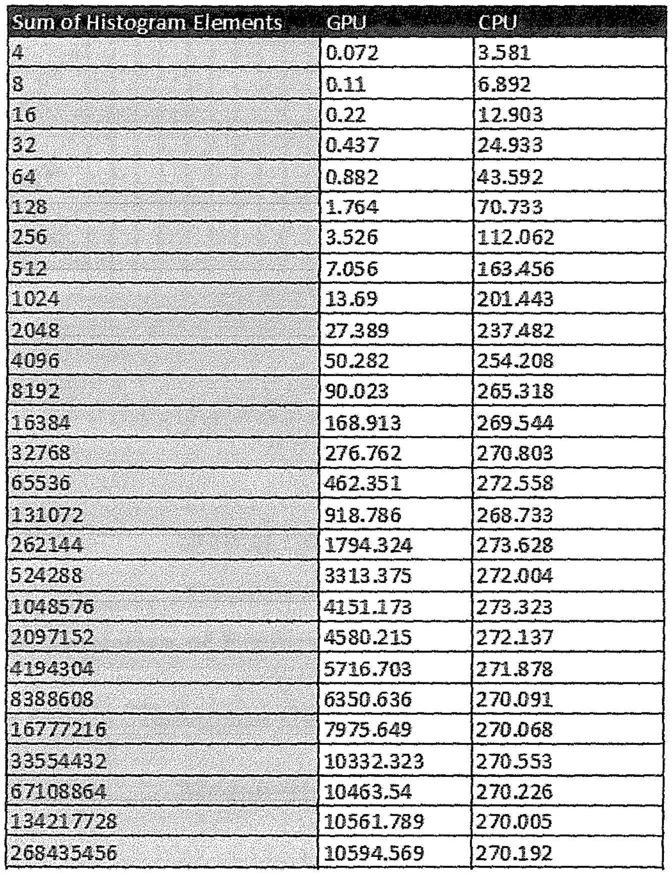


Figure 3.12: Graphical Depiction of Speeds of Serialized CPU Code and Parallelized GPU Code Running a 64-Bin Histogram Calculation on the CPU and GPU respectively (in milliseconds)

As can be seen in the above results, the CPU again performed much better than the GPU on very small data sets due to the high ratio of control operations. However, once the sum of the histogram elements reaches 32768, the GPU begins to perform better than the CPU, and on the largest data sets tested, the GPU performs the calculation almost 40 times faster than the CPU.

Again, as with the parallel prefix sum calculation, when the CPU is performing better than the GPU, the time difference is very inconsequential, whereas when the GPU outperforms the CPU, the time difference is much more substantial. This suggests that on data sets of random sizes, the GPU will likely still outperform the CPU by a large margin in terms of time spent overall. Of course, if the random data set sizes all happen to be small, this will not be the case, but again, the difference in overall time spent on calculations in that case will still be minimal.

The next table displays the overall performance rate (taken in megabytes per second) of the GPU and CPU on the same calculations:



Sum of Histogram Elements	GPU	CPU
4	0.072	3.581
8	0.11	6.892
16	0.22	12.903
32	0.437	24.933
64	0.882	43.592
128	1.764	70.733
256	3.526	112.062
512	7.056	163.456
1024	13.69	201.443
2048	27.389	237.482
4096	50.282	254.208
8192	90.023	265.318
16384	168.913	269.544
32768	276.762	270.803
65536	462.351	272.558
131072	918.786	268.733
262144	1794.324	273.628
524288	3313.375	272.004
1048576	4151.173	273.323
2097152	4580.215	272.137
4194304	5716.703	271.878
8388608	6350.636	270.091
16777216	7975.649	270.068
33554432	10332.323	270.553
67108864	10463.54	270.226
134217728	10561.789	270.005
268435456	10594.569	270.192

Table 3.5: Performance of Serialized CPU Code and Parallelized GPU Code Running a 64-Bin Histogram Calculation on the CPU and GPU respectively (in Megabytes per Second)

Again, for ease of examination, the same information is depicted visually in the following graph:

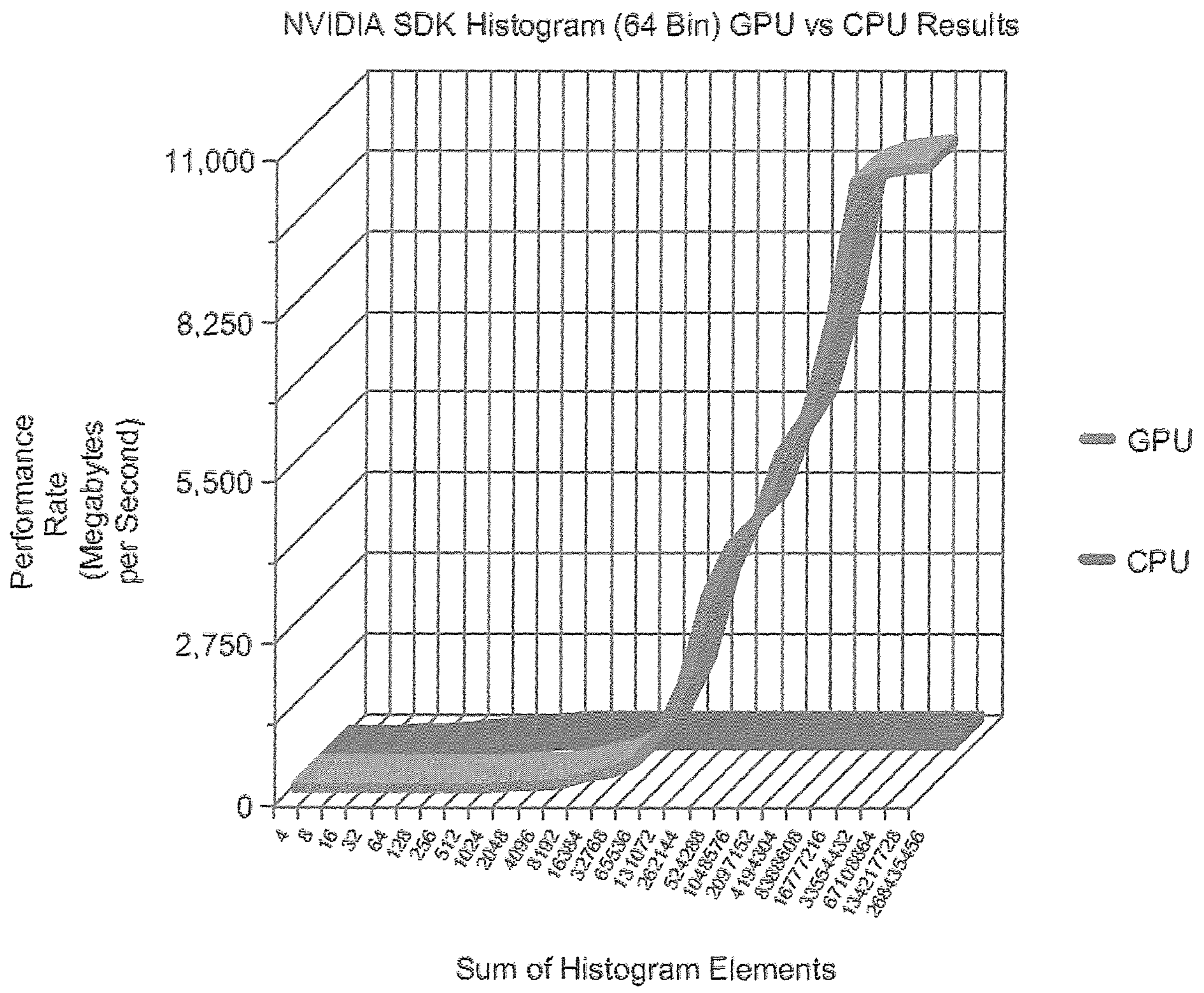


Figure 3.13: Graphical Depiction of Performance of Serialized CPU Code and Parallelized GPU Code Running a 64-Bin Histogram Calculation on the CPU and GPU respectively (in Megabytes per Second)

As just illustrated, the performance rate of the CPU is slightly higher than that of the GPU on smaller data set sizes. This difference is dwarfed, however, by the extreme difference in bandwidth shown in large data sizes which greatly favours the GPU. The performance rate on the GPU levels off at around 10,500 megabytes per second, and that appears to be the maximum rate this specific card can support when utilizing the 112 stream processors. Faster speeds could, however, be expected on higher-end NVIDIA G80 cards such as the 8800 GTX model which have 128 cores.

Chapter Four

Conclusions

4.1 Overall Conclusions about Offsetting General-Purpose, Data-Intensive Tasks to Graphics Processing Units

---

4.1 Overall Conclusions about Offsetting General-Purpose, Data-Intensive Tasks to Graphics Processing Units

In this section, some conclusions are rendered about the overall effects of offsetting general-purpose, data-intensive calculations to graphics processing units. All testing, results, and other information presented in this paper are analyzed and this is the basis for the said conclusions.

The tests in this paper ranged from ones which highlighted the significance of developing work-efficient applications which avoid bank conflicts, to ones which displayed the enormous processing potential of the GPU as a general-purpose processing apparatus.

By reviewing the results of the first set of tests, contained in section 3.3, it becomes clear that there are many things which determine the effectiveness of a parallel application on the GPU.

First of all, the application must strive for work-efficiency. This simply means that the GPU algorithm must attempt, if possible, to have the same work complexity as an efficient, serialized version of the code. Not doing so can seriously hurt the overall application performance, and this fact was prominently displayed in table 3.1 and figure 3.7.

Also displayed in these two result figures was the need for ensuring that bank conflicts be avoided. When care is not taken to avoid these conflicts, and serialization occurs, the exploitation of the parallel power of the GPU is severely crippled, and the performance degradation is extreme. In the tests shown in this paper, this degradation was even greater than when a work-inefficient algorithm was used since that work-inefficient algorithm happened to be created in a way which did not force many bank conflicts to occur.

By reviewing the remaining tests in this paper, it is apparent that the GPU is a great platform for parallel, data-intensive calculations, and it can seriously outperform even the most powerful current-generation CPUs in many applications. This was displayed in the various test result tables and figures throughout this text. The following major conclusions are both based on this observation, and extensions of it.

Firstly, the power which the GPU delivers, does not work for all applications, especially when the number of control operations is high in comparison to the number of computations being done. This is why the applications used on the GPU must be "data-intensive". Also, the number of data transfers from the device to host and from the host to the device must be minimal for best results.

Secondly, the difficulty in decomposing code into parallelized versions can be very difficult and can be an intimidating, time consuming task. Even operations such as the above outlined parallel prefix sums calculations, which are relatively trivial to design in a serialized manner become convoluted and difficult to grasp when parallelized.

All of the extra effort which must be exerted in order to truly optimize GPU code takes a very long time in comparison to traditional serial coding. People looking to port huge amounts of code to this type of architecture should only do so if the calculations they wish to process are very data-intensive and if they have the ability to take the time to do this conversion properly. It also needs to be determined if there is already a sufficient procedure or algorithm to achieve this task, while avoiding bank conflicts and keeping work complexity to a minimum, or if one can be determined. If all of the above conditions are true, or can be accomplished, then the GPU is an extremely powerful, viable alternative to CPUs for data-intensive calculations and can bring processing power, once only available to a select few, to the masses.

Chapter Five

Future Directions

5.1 Future Directions This Research Could Take

Anyone wishing to continue research in this area could do so by pursuing many avenues. Reasons for not doing so in this paper include lack of familiarity with the C programming language and with low-level graphics programming, lack of expensive testing equipment such as NVIDIA Tesla boxes, and time constraints.

Firstly, testing algorithms which enable the examiner to control threading, work efficiency, and bank conflicts at will could allow much more fine-grained testing. This would allow one to better determine the performance of massively multi-threaded applications against that of less multi-threaded implementations. This would also allow for a more accurate illustration of the true costs of bank conflicts and work efficiency since the testing in this paper only illustrate these differences in one specific balanced binary tree algorithm and not on the platform as a whole.

Secondly, if one were an experienced graphics programmer, it would be interesting to do some comparisons between straight low-level graphics code and code generated for the NVIDIA CUDA architecture. According to NVIDIA, code made for CUDA should outperform its lower-level counterpart on the GPU, and because of the added hardware support on the G80 chips, this is probable; however, it would still make for an interesting comparison.

Lastly, doing tests on the speed improvements possible while running the graphics cards in tandem (using SLI), or as part of a NVIDIA Tesla system as opposed to simply using a single-card configuration might also produce some interesting results.

Bibliography

[1] Jed Lengyel, Mark Reichert, Bruce R. Donald and Donald P. Greenberg, " Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware" *Computer Graphics*, vol. 24, no. 4, pp. 327--335, 1990.

[2] Mark J. Harris, Greg Coombe, Thomas Scheuermann and Anselmo Lastra, "Physically-Based Visual Simulation on Graphics Hardware" Graphics Hardware , Proc. 2002 SIGGRAPH / Eurographics Workshop on Graphics Hardware,pp.3,2002.

[3] "Aaron Lefohn" in *http://www.gpgpu.org/vis2004/A.lefohn.intro.pdf :* VIS 2004, 2004.

[4] "Lee Penrod" in *http://www.directron.com/expressguide.html:* What is PCI Express? A Layman's guide to high speed PCI-E technology.

[5] "Hendrik Lensch" in *http://www.mpi-inf.mpg.de/departments/d4/teaching/ws2007-2008/cg/slides/CG19-Cuda1.pdf:* Computer Graphics – CUDA Programming.

[6] "NVIDIA" in http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf: NVIDIA CUDA Programming Guide 1.1.

[7] "Mark Harris" in *http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf:* Parallel Prefix Sum (Scan) with CUDA.

[8] "Ramtin Shams and R. A. Kennedy" in http://users.rsise.anu.edu.au/~ramtin/papers/2007/ICSPCS_2007.pdf: Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices.

[9] "Victor Podlozhnyuk" in *http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram64/doc/histogram.pdf:* 64-bin Histogram.

[10] "NVIDIA" in http://www.nvidia.com/object/cuda_get.html: NVIDIA CUDA Developers Website.