Optimizing Parallel Reduction In Cuda To Reach GPU Peak Performance

[1]Adityo Mahardito
[2]Adang Suhendra
[3]Deni Tri Hasta

[1]Gunadarma University(adit@student.gunadarma.ac.id)
[2]Gunadarma University(adang@staff.gunadarma.ac.id)
[3]Gunadarma University(deni@student.gunadarma.ac.id)

Abstract

GPUs are massively multi threaded many core chips that have hundreds of cores and thousands of concurrent threads swith high performance and memory bandwidth. Now days, GPUs have already been used to accelerate some numerically intensive high performance computing applications in parallel not only used to graphic processing. This thesis aims primarily to demonstrate the programming model approaches that can be maximize the performance of GPUs. This is accomplished by a proof of maximum reach of bandwidth memory and get speed up from the GPU that used to process parallel computation. The programming environment that used is NVIDIA's CUDA, it is parallel architecture and model for general-purpose computing on a GPU.

1 Introduction

Currently, demand for hardware resources for processing data on computers is increasing. This can be proved by continuously increasing the minimum requirement of software applications. The phenomenon of the advent multi core CPUs and many core GPUs means that mainstream processor chips are now parallel systems. Many applications that process large data sets can use a data parallel programming model to speed up the computations. The application outside the field of computation itself are accelerated by data parallel processing, from general image rendering and processing, signal processing, physics simulation to computational finance or computational biology. However, to more support of data parallel processing, the GPU can be considered than the CPU, because there are very significant differences between the capability that are owned by the CPUs and GPUs to process sets large data on parallel computation. In the other side, GPUs are very efficient not only at manipulating computer graphics, but also at their highly parallel structure makes them more effective than general purpose CPUs for a range of complex algorithms [2].

The GPU is specialized for compute intensive, highly parallel computation and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. Many companies have produced GPUs under a number of brand names. Intel, NVIDIA, AMD/AT, S3 Graphics, VIA Technologies, and Matrox produce GPUs. But, NVIDIA and ATI control nearly 100% of the market. Recently, NVIDIA began releasing cards supporting an API extension to the C programming language CUDA ("Compute Unified Device Architecture"), which allows specified functions from a normal C program to run on the GPU's stream processors[8].

The performance offered by the GPU, in fact, depend-ing on the method approach that used while create a CUDA program, even though using same algorithm. Because in the parallel programming, especially on CUDA, the thinks that must be considered to obtain optimal results are memory usage and instruction usage. The main focus of this thesis is make optimization of parallel programming on the CUDA environment that running over GPU using parallel reduction algorithm. The parallel reduction algorithm that implemented is a summation algorithm that computes the sum of large arrays of values.

2 Literature Review

2.1 Graphic Processor Unit

A graphics processing unit or GPU is a processor attached to a graphics card dedicated to calculating floating point operations. GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth.

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute intensive, highly parallel computation and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.

This difference makes the CPU and GPU suited for different kinds of problems. In a simplified way, one can say that a CPU performs best when a few, small pieces of data are processed in a complex, but sequential way. This lets the CPU utilize the many transistors used for caching, branch prediction and instruction level parallelism. The GPU, on the other hand, need massively data parallel problems to work efficiently. The programming model most commonly used when programming a GPU is based on the stream programming model. In the stream programming model, input to and output from a computation comes in the form of streams. A stream is a collection of homogeneous data elements on which some operation, called a kernel, is to be performed, and the operation on one element is independent of the other elements in the stream.

Another important difference between a general purpose processor and a typical GPU is the memory bandwidth. Because of simpler memory models and no requirements from legacy operating systems, the GPU can support more than 100 GB/s of memory bandwidth, while the bandwidth of general-purpose processors is around 20 GB/s [7].

2.1.1 GPU Architecture

Start from Streaming processor (see at Figure 1), it is a fully pipelined, single-issue, in order microprocessor complete with two ALUs and a FPU. An SP doesn't have any cache, so it's not particularly great at anything other than cranking through tons of mathematical operations. Since an SP spends most of its time working on pixel or vertex data, the fact that it doesn't have a cache doesn't really matter, because if you add up enough of these cores of GPU technology you can start to get something productive given that graphics rendering is a highly parallelizable task[3].

The GPU has a many-core processor containing an array of Streaming Multiprocessors (SMs). A SM is an array of SPs, consists of 8 Streaming Processors (SPs), along with two more processors called Special Function Units (SFUs). Each SFU has four FP multiply units which are used for transcendental operations (e.g. sin, cosin) and interpolation, the latter being used in some of the calculations for things like anisotropic texture filtering. NVIDIA

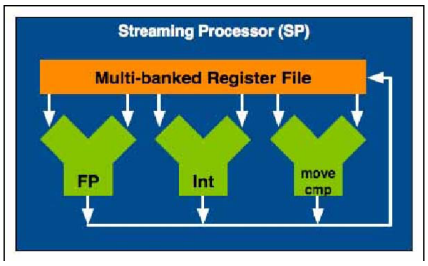


Figure 1: Streaming Processor (SP)

isn't specific in saying about SFU architecture, just assume that each SFU is also a fully pipelined, single issue, in order microprocessor. There's a MT issue unit that dispatches instructions to all of the SPs and SFUs in the group.

In addition to the processor cores in a SM, there's a very small instruction cache, a read only data cache and a 16KB read/write shared memory. These cache sizes are kept purposefully small because unlike a conventional desktop microprocessor, the datasets we're trying to cache here are small. Each SP ends up working on an individual pixel and despite the move to 32-bit floating point values, there's only so much data associated with a single pixel. The 16KB memory is akin to Cell's local stores in that it's not a cache, but a software managed data store so that latency is always predictable. With this many cores in a single SM, control and predictability and very important to making the whole thing work efficiently.

The next cluster is Texture Processor Cluster (TPC), shown by Figure 2. NVIDIA purposefully designed its GPU architecture to be modular, so a single TPC can be made up of any number of SMs. In the GT200 architecture it was made up of three SMs and it can be different with the other series [4].

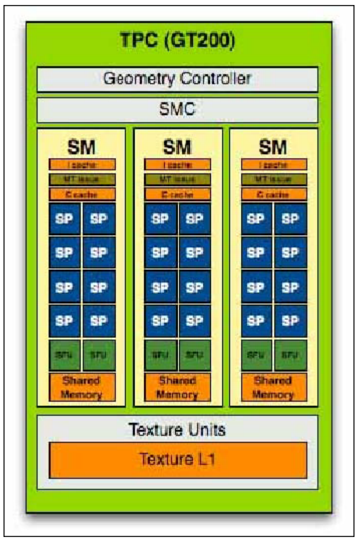


Figure 2: Texture Processor Cluster

The components of the TPC however haven't changed, a TPC is made up of SMs, some control logic and a texture block. Remember that a SM is a total of 8 SPs and 2 SFUs, so that brings the total up to 24 SPs and 6 SFUs per cluster in GT200. The texture block includes texture addressing and filtering logic as well as a L1 texture cache.

The modular theme continues with the Streaming Processor Array (SPA) that is composed of a number of TPCs, see Figure 3 [5].

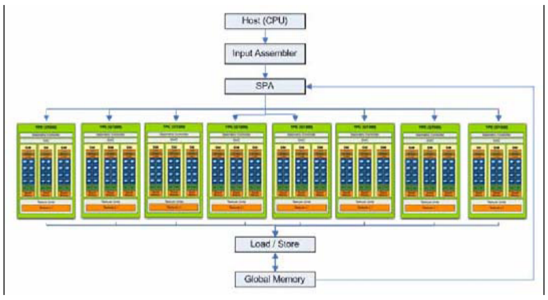


Figure 3: Streaming Processor Array

Similarly, the processed data elements need to be stored back in global memory for copying back to host memory. The task of effectively hiding the global memory access la-tency and managing the memory hierarchy is very crucial for obtaining maximal performance from the GPU. Each SM manages the creation, execution, synchronization and destruction of concurrent threads in hardware, with zero scheduling overhead, and this is one of the key factors in achieving very high execution throughput. Each parallel thread is mapped to an SP for execution, and each thread maintains it's own register state. All the SPs in an SM execute their threads in lock-step, according to the order of instructions issued by the per-SM instruction unit. The SM creates and manages threads in groups of 32, and each such group is called a warp [6].

A warp is the smallest unit of scheduling within each SM. The GPU achieves efficiency by splitting it's work-load into multiple warps and multiplexing many warps onto the same SM.

2.2 CUDA Overview

CUDA, Compute Unified Device Architecture, is a general-purpose hardware interface designed to let pro-grammers use NVIDIA graphics hardware for purposes other than graphics in a more familiar way. In general, the hardware need not be a graphics related card at all, as there are cards designed specifically for general-purpose calculations. CUDA defines a programming model and a memory model that is consistent between all CUDA devices. The programming model describes how parallel code is writ-ten, launched and executed on a device and how threads are grouped into blocks. The memory model defines the different types of memories that are available to a CUDA program.

2.3 CUDA Programming Model

NVIDIA's Compute Unified Device Architecture (CUDA) is a programming model and hardware/software environment for GPGPU, which consists of the following components:

- An extension to the C programming language that al-lows programmers to define GPGPU functions called kernels which are executed by multiple threads on the GPU.
- A compiler and related tools for translating CUDA source code to GPU devicespecific binary
- A software stack consisting of GPGPU application li-braries, CUDA runtime libraries and CUDA device driver
- A CUDA enabled GPU device

2.3.1 CUDA Kernels

The CUDA programming language provides a means for programmers to express a problem that exhibits significant data-parallelism as a CUDA kernel. A CUDA kernel is a function that is executed in Single Program Multiple Data (SPMD) on a large set of data elements.

Kernel code is written on the thread level with access to built-in variables that identify the executing thread. A kernel is defined using the global declaration specifier and the number of CUDA threads for each call is specified using a new <<<...>>>syntax.

2.3.2 CUDA Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 4.

Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block.

2.3.3 Thread Hierarchy

The CUDA programming model organizes threads into a three-level hierarchy as shown in Figure 5. At the highest level of the hierarchy is the grid. A grid is a 2D array of thread blocks, and thread blocks are in turn 3D arrays of threads.

The size of the grid and the thread-blocks are determined by the programmer, according to the size

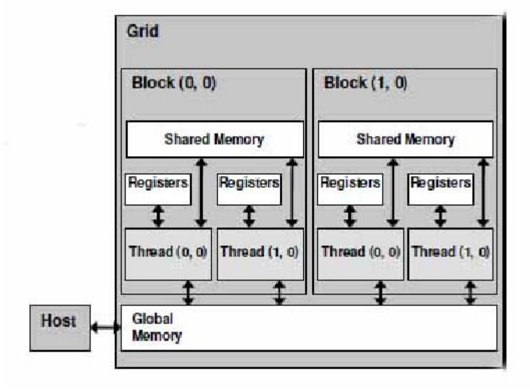


Figure 4: Memory Access

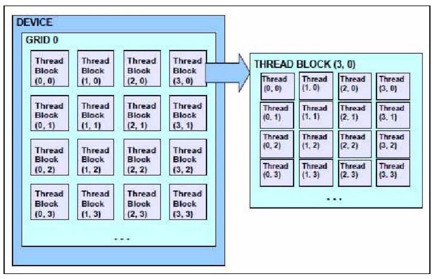


Figure 5: Hierarchy of threads in the CUDA Programming Model

of the problem be-ing solved and communicated to the driver at kernel launch time. Each thread-block in a grid has it's own unique identifier and each thread has a unique identifier within a block. Using a combination of block-id and thread-id, it is possible to distinguish each individual thread running on the entire device. Only a single grid of thread blocks can be launched on the GPU at once, and the hardware limits on the num-ber of thread blocks and threads vary across different GPU architectures.

A kernel is executed by a grid, which contain blocks. These blocks contain the threads. A thread block is a batch of threads that can cooperate in sharing data through shared memory and synchronizing their execution. A threads from different blocks operate independently.

2.4 CUDA Optimization Strategy

In CUDA optimization performance strategy, there are 4 approaches that can be used, ie:

- Instruction Performance
- Number of thread per block
- Data transfer between host(CPU) and device(GPU)
- Texture Fetch versus Global or Constant Memory Read

In this case, the strategy that will used is instruction performance approach.

2.4.1 Instruction Performance

To process an instruction for a warp of threads, a multiprocessor must read the instruction operands for each thread of the warp, execute the instruction, then write the result for each thread of the warp.

Therefore, the effective instruction throughput depends on the nominal instruction throughput as well as the mem-ory latency and bandwidth. It is maximized by minimizing the use of instructions with low throughput, then maximiz-ing the use of the available memory bandwidth for each category of memory.

2.4.2 Instruction Throughput

- **Arithmetic Instruction**

  To issue one instruction for a warp, a multiprocessor takes 4 clock cycles for single-precision floating-point add, multiply, and multiply-add, integer add, bitwise operations, compare, min, max, type conversion in-struction. Integer division and modulo operation are particularly costly and should be avoided if possible or replaced with bitwise operations whenever possible. If n is a power of 2, (i/n) is equivalent to $(i >> log2(n))$ and $(i\%$is equivalent to $(i\&(n-1))$, the compiler will perform these conversions if n is literal.

- **Control Flow Instructions**

  Any flow control instruction (if, switch, do, for,while) can significantly impact the effective instruc-tion throughput by causing threads of the same warp to diverge, that is, to follow different execution paths. If this happens, the different executions paths have to be serialized, increasing the total number of instruc-tions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path. To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps.

- **Memory Instruction**

  Memory instructions include any instruction that reads from or writes to shared, local or global memory. A multiprocessor takes 4 clock cycles to issue one memory instruction for a warp. When accessing local or global memory, there are, in addition, 400 to 600 clock cycles of memory latency. Much of this global memory latency can be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete.

- **Synchronization**

  Instruction syncthreads takes 4 clock cycles to issue for a warp if no thread has to wait for any other threads.

2.4.3 Memory Bandwidth

The effective bandwidth of each memory space depends significantly on the memory access pattern. Since device memory is of much higher latency and lower bandwidth than on-chip memory, device memory accesses should be minimized. A typical programming pattern is to stage data coming from device memory into shared memory; in other words, to have each thread of a block:

1. Load data from device memory to shared memory
2. Synchronize with all the other threads of the block so that each thread can safely read shared memory loca-tions that were written by different threads
3. Process the data in shared memory
4. Synchronize again if necessary to make sure that shared memory has been updated with the results
5. Write the results back to device memory

2.5 Parallel Reduction

Parallel reduction is one of the kinds of Parallel Random Access Machine (PRAM) algorithms. A Parallel Random Access Machine (PRAM) is a shared memory abstract machine which is used by parallel algorithms designers to esti-mate the algorithm performance (like its time complexity). So, a parallel reduction is a process of PRAM in the manipulating data stored in the memory of global registers. There are several operations in parallel reduction i.e., the addition operation, subtraction operation, and multiplication operation. Parallel reduction can be described with a binary tree, a group of n log p value is added in addition steps in parallel.

2.5.1 Kahan Summation Algorithm

In numerical analysis, the Kahan summation algorithm (see algorithm 1) also known as compensated summation significantly reduces the numerical error in the total obtained by adding a sequence of finite precision floating point numbers, compared to the obvious approach. This is done by keeping a separate running compensation (a variable to accumulate small errors).

In particular, simply summing n numbers in sequence has a worst-case error that grows proportional to n, and a root mean square error that grows as $\sqrt{n}$ for random inputs (the roundoff errors form a random walk). With compensated summation, the worst-case error bound is independent of n, so a large number of values can be summed with an error that only depends on the floating-point precision.

Algorithm 1 Kahan Summation Algorithm

```
function kahanSum(input)
var sum ← input[i] A running compensation for lost-low order bits
var c ←0:0
i = 2 to input.length
y ← input[i] - c
t ← sum + y
c ← (t -sum) - y
sum← t
next i
return sum
```

2.5.2 Parallel Reduction Complexity and Cost

In the parallel processing, there are three kind of complexity, i.e. step complexity, work complexity, and time complexity.

- Step complexity is Log(N) parallel steps, each step S does N/2 independent operation, so step complexity is O(log N).
- Work complexity because for $N = 2^{\text{D}}$ performs $\sum S_\epsilon[1..D]2^{D-S} = N - 1$ operations, so work complexity is $O(N)$.
- The time complexity is $O(N/P + logN)$ with P threads physically in parallel (P processor)

3 Methodology

3.1 General Scenario

A reduction is the process of combining elements of a vector or array to yield a single aggregate element. It is commonly used in scientific computations in parallel processing. Therefore, this research try to implementing this method to find a way optimizing the parallel device. Actually, the reduction have some operation that can be used, i.e. addition , multiplication, substraction, and division. The author chose the addition operation with consideration that it will not give the elements that consist negative value, decimal, or too large. Then, this reduction operation will be implemented to process large set elements of an array on GPU device. The case that used is sum of increment array elements on CUDA environment, then explore the CUDA programming model to get different implementation of each others, in an attempt to find an efficient and scalable implementation.

3.2 GPU Device

In this research, the GPU device that used to optimized is GeForce GT 240. This device is a desktop class, commonly used to play the games on the PC. It's have 96 CUDA cores to process on parallel with 54.4 GB/s memory bandwidth. The points that want to reach on GPU is the maximum of memory bandwidth and minimum of the time processing.

3.3 Research Method

There are some method that implemented in this research, i.e. :

1. Use recent of example program by Ian Buck research [1]
2. Find a weakness or problem from the program
3. Try to solve the problem then upgrade to better program
4. Test and compare between all of them

3.4 Program Implementation

3.4.1 Interleaved Addressing with Divergent Branching by Ian Buck

The program, for next called as kernel, doing reduction where each thread performs calculation on two elements that loaded from shared memory. It is an implementation of sum EREW PRAM reduction process, where the indexing method use a modulo operation for every iteration process.

This method is simple but has a weakness, that indexing thread is useless, because there is one hop between the thread and others, it will make highly divergent warps that affected inefficient process.

The solution is according to the strategy performance, this program can be revised with arithmetic instruction and control flow instruction approaches to replace if $(tid\%$ process to minimize the number of divergent branch and warps. It can be replaced with $if(index < blockDim.x)$ control flow, where $intindex = 2 \star s \star tid$.

3.4.2 Reduction kernel with bank conflict method

This method is using the solution of Interleaved addressing with divergent branching. Just replace the way on thread indexing with stride index and non divergent branch. So, the indexing of thread ID not based on anymore with the order of elements value in shared memory.

By using this method, the thread index has been sorted correctly without hop between each other, so there is no useless of thread usage.

But the new problem is shared memory bank conflicts. To achieve high memory bandwidth, shared memory is divided into equally sized memory modules, called banks, which can be accessed simultaneously. So, any memory read or write request made of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single module. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts. To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests so as to minimize bank conflicts.

For all threads of a warp (32 threads), accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads. So, the solution is using sequential addressing method for indexing thread ID.

3.4.3 Reduction kernel with sequential addressing

Sequential addressing method will avoid bank conflict problem, because in this method the thread ID indexing doing with sequential method, so every threads will not access the same address on memory bank, just point to one same shared memory address.

In the source code, the think that must replace is just a stride indexing in inner loop with reversed loop and thread ID based indexing.

But, this way still have a problem, half of the threads are idle on first loop iteration, so it make wasteful and of course will affected to performance. The solution can be get by doing reduction during the load element process from global memory to shared memory.

3.4.4 Reduction kernel with first add during load

To avoid idle of threads on sequential addressing method by doing the reduction when load elements from global memory to shared memory or can said first add during load method.

In this method, during loop process of loading element from global memory, also carried out the reduction process from element that have finished to load before finally writing on shared memory. So, the element that process in kernel function is the result of sum computation too. The point is replace single load with two loads on the load process and perform first add during load processed..

It should be the best method that can used to reach GPU peak performance, but in fact the reach of memory bandwidth is still far from bandwidth bound, it may caused by instruction bottleneck or instruction latency. An instruction's latency is the number of clock cycles it takes for the instruction to pass through the pipeline. To avoid this condition, the strategy that can used is unroll loops, because of the reduction has low arithmetic intensity.

3.4.5 Reduction kernel with unroll last warp

Sometimes, the compiler may unroll loops or optimize out if or switch statements by using branch predication instead.In these cases, no warp can ever diverge. The programmer can also control loop unrolling.

As reduction proceeds, the number of active threads decreases. In GPUs architecture, a kernel or an instruction are SIMD synchronous within a warp. Warp is a group of threads executed physically in parallel, and each group of warp consist of 32 thread. That means when ”si=32”, no longer need to _syncthreads_() and no longer need to ”if(tidis)” because it doesn't save any work, so it can be control loop unrolling. Let's unroll the last 6 iteration of the inner loop.

This saves useless work in all warps, not just the last one. Without unrolling, all warps execute every iteration of the for loop and if statement.

3.4.6 Reduction kernel with completely unrolled

If the number of iterations at compile time is known, it could completely unroll the reduction. Luckily, the block size is limited by the GPU to 512 threads and also sticking to power of 2 block size. So it can easily unroll for fixed block size, then replace the iterative process ”for” with completely unroll.

3.4.7 Reduction kernel with multiple elements per thread

Given a total number of threads per grid, the number of threads per block, or equivalently the number of blocks, should be chosen to maximize the utilization of the available computing resources. This means that there should be at least as many blocks as there are multiprocessors in the device.

Furthermore, running only one block per multiprocessor will force the multiprocessor to idle during thread synchronization and also during device memory reads if there are not enough threads per block to cover the load latency. It is therefore usually better to allow for two or more blocks to be active on each multiprocessor to allow overlap between blocks that wait and blocks that can run. For this to happen, not only should there be at least twice as many blocks as there are multiprocessors in the device, but also the amount of allocated shared memory per block should be at most half the total amount of shared memory available per multiprocessor.

4 Result and Comparison

4.1 Parameter

Before start to run all of program from each method, the parameter that used on CUDA for all method is:

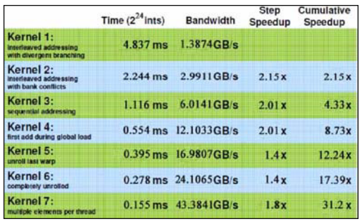
- Number of elements array is 2^{17} to 2^{25}
- Number of thread per block is 512
- Number of block is 512

4.2 Result and Comparison

- Result for 2^{24} elements array

  Result of running all of method, compared by memory bandwidth and time consuming shown by Figure 6. It's just described the performance by each method to process 2^{24} elements of array, where ”Time” column is time process that needed in millisecond, ”Bandwidth” column is rate at which data can be read from or stored into a memory. Memory bandwidth is usually expressed in units of bytes/second, ”Step Speedup” column is value of current kernel time divided by previous kernel time, and ”Cumulative Speedup” column shown the value of current kernel time divided by kernel 1 (divergent branch method) time.

- Result for 2^{17} to 2^{25} elements array



	Time (2^{24} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	4.837 ms	1.3874GB/s		
Kernel 2: interleaved addressing with bank conflicts	2.244 ms	2.9911GB/s	2.15x	2.15x
Kernel 3: sequential addressing	1.116 ms	6.0141GB/s	2.01x	4.33x
Kernel 4: first add during global load	0.554 ms	12.1033GB/s	2.01x	8.73x
Kernel 5: unroll last warp	0.395 ms	16.9807GB/s	1.4x	12.24x
Kernel 6: completely unrolled	0.278 ms	24.1065GB/s	1.4x	17.39x
Kernel 7: multiple elements per thread	0.155 ms	43.3841GB/s	1.8x	31.2 x

Figure 6: Result performance by all of method

Figure 7, show performance comparison by time on graphical. On that figure can see that from 131.072 - 1.048.576 (2^{17} to 2^{20}) elements array, all of method still reliable to used. At 2.097.152 - 16.777.216 (2^{21} to 2^{24}) elements array, the method of Divergent Branching, Bank Conflict, and Sequential Addressing was not reliable to used, because they need a time over 1.0 second. Furthermore, at 2^{25} show that just four methods that still reliable to used. In the end of the result, the best method is Multiple Element per thread because it consumes more less time than the others.

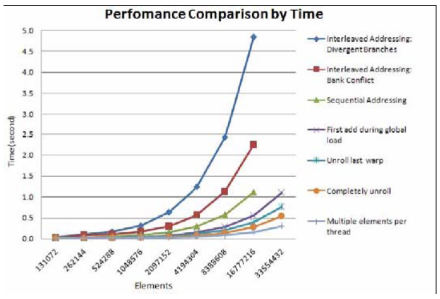


Figure 7: Result Comparison by time

Figure 8, show performance comparison by bandwidth per seconds on graphical, its means that the best method is that can reach highest bandwidth or closer with the GPU bandwidth boundary, at 54.4 GB/s. On that figure show performance from 131.072 - 33.554.432 (2^{17} to2^{25}) elements.

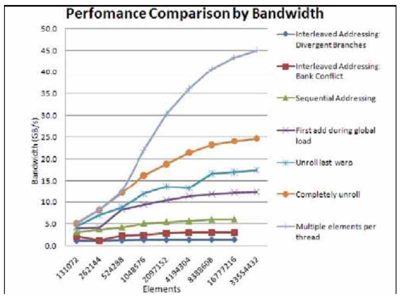


Figure 8: Result Comparison by bandwidth

5 Conclusions

5.1 Conclusion

According to the result, the GPU peak performance has reached. It proved by acquisition memory bandwidth value (at 43.3 GB/s) with memory bandwidth bound (54.4 GB/s) on GPU device (Geforce GT 240).

To optimize the GPU for parallel reduction process by keep the multiprocessors on the device as busy as possible. A device in which work is poorly balanced across the multiprocessors will deliver suboptimal performance. Hence, it's important to design your application to use threads and blocks in a way that maximizes hardware utilization and to limit practices that impede the free distribution of work. One of the keys to good performance is Another important concept is the management of system resources allocated for a particular task.

CUDA optimization strategy can be used to maximizes hardware utilization. In this case of reduction process, it can be use divergent branching, bank conflicts, memory coalescing with sequential addressing and latency hiding with unroll loop.

Actually, so many type of approaches that can be used to optimize performance on GPU, but especial for CUDA, there two type strategy, ie instruction throughput and memory bandwidth.

5.2 FutureWork

The author have an wish to continue the research about GPU performance on future work, wish to try with some different thinks,i.e.: Use different type of GPU device, more high specification to process more large data Thread and block parameter that used, to find ideal portion of shared memory usage Algorithm processing, to break through the boundary of memory bandwidth.

References

[1] Ian Buck. Parallel programming with cuda. NVIDIA, 2008.

[2] NVIDIA. Nvidia cuda compute unified device architecture. In *NVIDIA CUDA Programming Guide 2.0*, volume 2, page 1. 2008.

[3] Anand Lal Shimpi and Derek Wilson. Nvidia's 1.4 billion transistor gpu. page 2, 2008. http://www.anandtech.com/show/2549/2, accessed on August 2010.

[4] Anand Lal Shimpi and Derek Wilson. Nvidia's 1.4 billion transistor gpu. 2008. http://www.anandtech.com/show/2549/4, access on April 2010.

[5] Anand Lal Shimpi and Derek Wilson. Nvidia's 1.4 billion transistor gpu. page 4, 2008. http://www.anandtech.com/show/2549/4, access on April 2010.

[6] Anand Lal Shimpi and Derek Wilson. Nvidia's 1.4 billion transistor gpu. page 2, 2008. http://www.anandtech.com/show/2549/2, access on April 2010.

[7] wikipedia. *gpgpu*, 2010. http://en.wikipedia.org/wiki/GPGPU.

[8] Wikipedia. Graphics processing unit. 2010. http://en.wikipedia.org/wiki/Graphics_ processing_unit, accessed on August 2010.