# GMP implementation on CUDA - A Backward Compatible Design With Performance Tuning

Hao Jun Liu, Chu Tong
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
haojun.liu@utoronto.ca, chu.tong@utoronto.ca

## I. Introduction

The goal of this project is to implement the GMP library in CUDA and evaluate its performance. GMP (GNU Multiple Precision) is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface. The main target applications for GMP are cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc. GMP was carefully designed to be as fast as possible, both for small operands and for huge operands. The speed is originally achieved by using fullwords as the basic arithmetic type, by using fast algorithms, with highly optimized assembly code for the most common inner loops for a lot of CPUs, and by a general emphasis on speed. [1] With the emerging of relatively cheap and high performance vector processors, ie. the NVIDIA CUDA, we are able to implement some of the operations supported in GMP using CUDA while attempt to maintain the similar programming interface of the original library.

The project implements addiction, multiplication, bit manipulation, bitwise logic, subtraction for integers. The function prototypes have been modified on the part necessary to ensure both backward compatibility while maintaining best performance of the library. We conducted extensive functional and performance tests on our library and the result is presented.

## II. Related Work

### A. GMP library using CUDA

Based on our research, there has been request on GMP forum to implement GMP on CUDA. However, there has been no published implementation so far. One of the reasons of this status is that CUDA is still a relative new architecture and only few people can program it with good performance. Another reason is that CUDA architecture only works with data size that are sufficiently large.[2]

### B. GMP library using CELL

IBM's CELL$^{TM}$architecture is one implementation of SIMD machine and is somewhat similar to CUDA. There has been some work that implements an arbitrary precision library and achieved a performance improvement of 2X using CELL Processor.[3]

### C. FFT Multiplication

There is an existing large number multiplication implementation on CUDA by K. Zhao. [4] However, there is no performance comparison with GNU GMP Library in the work.

## III. Methodology

This section describes the algorithm and detailed description of the implementation.

### A. Data Structures

The data structures used to store operands and results has two components including the number of integers used and a pointer to an array of unsigned integers which is also named as the limb. The following code gives the data structure used to store multiple precision numbers in GNU GMP.

```
typedef struct
{
    /* Number of *limbs* allocated and pointed
       to by the _mp_d field.  */
    int _mp_alloc;

    /* abs(_mp_size) is the number of limbs the
       last field points to.  If _mp_size is
       negative this is a negative number.  */
    int _mp_size;

    /* Pointer to the limbs.  */
    mp_limb_t *_mp_d;
} cpu_mpz;
```

The following code gives the data structure used to store multiple precision numbers in our implementation of GNU GMP.

```
typedef struct
{
   // number of limbs allocated on CPU side
   int cpu_alloc;

   // number of limbs currently used on CPU side
   int cpu_used;

   // number of limbs allocated on GPU side
   int gpu_alloc;

   // number of limbs currently used on GPU side
   int gpu_used;

   // limbs for the number on the CPU side
   unsigned int * cpu_limb;

   // limbs for the number on the GPU side
   unsigned int * gpu_limb;

   // current state of the number
   enum MEMSTAT state;
} gpu_mpz;
```

The difference between is that more entries are added to the mpz data structure to store multiple precision numbers for GPU calculation.

### B. Memory Management

In order to minimize the number of memory transactions, we modified the structure of GMP operands by adding a new field to keep the GPU version of the data and a state variable about whether the most updated copy of the operand is on the CPU or the GPU. Whenever a computation takes place, the state variable is queried to determine if any memory copy is necessary and GPU calculation is performed afterwards. The overall performance is dramatically enhanced by this technique because of significantly decreased number of memory transactions in a series of calculations. However, one major disadvantage of this approach is that there is no mechanism of free inactive memory on GPU automatically.

### C. Bitwise Logic

The bitwise logic includes AND, OR and XOR.

*1) Algorithm:* To compute bitwise operations efficiently on the GPU, we employ the idea of using every thread operates on two corresponding limbs of the operands in parallel and store its result back to the result buffer after the computation.

*2) Implementation:* To implement bitwise operations, two operands represented in the format described in *Data Structures Section* are passed into CUDA kernel. Based on the thread ID, bitwise operations are performed on corresponding limbs of the operands and result buffer is updated during the execution. For the AND operation, if one operand has more limbs than the other, the extra limbs are simply ignored because of the fact that any number logically AND with zero equals zero. In the case of OR and XOR operations, the extra limbs are directly copied from the larger operand to the result as any number logically OR or XOR zero produce itself.

### D. Bit Manipulation Operation

The bit manipulation operations include shift, set and pop count.

*1) Algorithm:* To shift an operand by some number of bits, the algorithm has two parts. The first part of the algorithm performs shift operations on every limbs of the input and records the result in a result buffer and it put the truncated bits into a carry buffer. The second part of the algorithm does an addition between the result buffer and carry buffer to produce the final result.

To set a number, the algorithm use a number of threads to assign different limbs of the operands to result in parallel.

To count the number of 1-bits (Pop Count) in an operand, the algorithm finds the 1-bits in each limbs of the input in parallel and sum them up upon completion to obtain the result.

*2) Implementation:* For the shift operation, we first find the integer quotient of the number of bits to shift divide by 32 and use it as the offset of memory copy. Then, we shift and add to get the result.

For pop count operation, due to the huge performance slowdown of atomic operation on the GPU, we use shared memory and perform reduction to find number of 1-bits of limbs in the same block and do the sum-up on CPU after kernel. While counting the number of 1-bits from each limb, instead of using SHIFT, AND and ADD operations, we also tried to use lookup table to find the number of 1-bits from each part of the operand, however, due to the memory transaction between CPU and GPU to build up the lookup table, no performance improvement is acquired.

### E. Arithmetic Operation

The arithmetic operations include addition and subtraction.

*1) Algorithm:* The algorithms we use to implement addition and subtraction operations are divided into two parts. In the first part of the algorithm, we do arithmetic operation for every corresponding limbs of the two operands in parallel. The result and carry of the computation are stored into a result buffer and a carry buffer respectively. Upon the completion of the first part, a second CUDA kernel is invoked to locate the carries based on thread ID and propagate them to higher significant bits until no more propagation is necessary. The fact that arithmetic operations between two corresponding limbs can not carry more than once during the arithmetic operations, therefore, all race conditions are eliminated. The algorithm has one inherited drawback in the case of a carry bit is needed to move from the least significant bit (LSB) to the most significant bit (MSB) (i.e.: 0xFFFFFFFFFFFFFFFF + 1), the performance of the GPU is severely degraded as only one thread is active to move the carry bit while all other threads are idle. We have not been able to formulate a better solution to solve this issue, but due to its extremity, the overall performance enhancement introduced by our solution is still quite feasible.

*2) Implementation:* After breaking down the arithmetic operations into sub-problems. A single thread only operates on the two corresponding limbs of the operands chosen based on the thread ID, The implementation of the arithmetic operations require the ability to detect carry bit of arithmetic operation between two limbs and propagate the carry to its final position. As a solution to the first part of problem, we use the concept that for an arithmetic operation to have carry bit, the result of the operations between two operands must be smaller than either of them in the case of addition and bigger than the minuend in the case of subtraction to find carries and record them in a temporary carry buffer. For the second part of the problem, in the case of addition, propagation is done to ensure that first zero bit is found and increased by one starting from the position of the carry bit while bits between these two boundaries are zeroed out. On the other hand, the propagation of subtraction behaves in exactly the opposite way; The first non-zero bit is found and set to zero while bits between are set to one.

### F. Multiplication

It is known that GPU operations run faster only when the input data is large enough. There are different ways to implement multiple precision multiplication. For this project, We only implement multiplication using FFT and the idea is proposed by REF.

*1) Algorithm:* Multiplication of two numbers can be expressed as:

$$a = \sum_{i=0}^{m} 2^{wi} a_i \tag{1}$$

$$b = \sum_{j=0}^{m} 2^{wj} b_j \tag{2}$$

$$\begin{aligned} ab &= \sum_{i=0}^{m}\sum_{j=0}^{m} 2^{w(i+j)} a_i b_j \\ &= \sum_{k=0}^{2m} 2^{wk} \sum_{i=0}^{k} a_i b_{k-i} \\ &= \sum_{k=0}^{2m} 2^{wk} c_k \end{aligned} \tag{3}$$

In fact, multiplication of a and b is the sum of pairwise multiplication of the limbs of a and b. By the definition of convolution, multiplication of a and b is the convolution of a's limbs by b's limbs. Moreover, according to some signal and system concepts, convolution in time domain can be achieved using multiplications in frequency domain. The complexity of this algorithm is O(nlogn(loglogn)).

*2) Implementation:* We have implemented a large number multiplication using this above idea. There are two major limitations of the current implementation.

- The result will contain errors. There are inherent rounding issues in the FFT we used. This is unavoidable since all FFT algorithm use floating point. Using number-theoretic transforms instead of discrete Fourier transforms avoids rounding error problems by using modular arithmetic instead of complex numbers. However, this is far beyond the scope of this project.
- This implementation consumes significant amount of memory since there are several sophisticate intermediate procedures in our implementation. Moreover, CUDA FFT library only works accurately and efficiently when the input is power of 2. This further increases the memory required to perform multiplication.

## IV. Evaluation

This section presents experimental result shows the performance of GMP implementation on GPU outscores GMP implementation on CPU if the data size is sufficiently large. Performance is defined as the running time in this context when the data and data size is fixed. This section first gives the threshold of equal performance of CUDA implementation and CPU implementation given that the inputs are same. Then it describe the CUDA implementation's performance with different data size. The running time is measured in ms. Finally it gives further optimization can be performed and expected performance increase when running on the new CUDA architecture Fermi™.

### A. Experimental Methodology

The experimental data presented in this report were collected using randomly generated data with different size. We conducted functional testing and performance measurement.

Functional testing is achieved by running random data with random data size. We have access to 25 CUDA capable machines and but not all of them are available. Given the time allowed for this work, $22 \times 10000$ iterations have been run for every operation via different machines to ensure the expected functionality of our implementation.

Performance measurement contains the following:

1) Effect of block size and grid size on performance when data and data size are constant
2) Threshold of equal performance of CPU and CUDA by varying data and data size
3) Effect of different data and different data size by using the optimum configuration find in Effect.

### B. Bitwise

*1) Result:* We first vary the block size and grid size to find their effect on the performance. The data size is $256 \times 33000$ limbs and the data is generated randomly in this test. From Fig. 1 on the following page, the performance is best when the block size is 256 and grid size is 12000.

We find the relative performance increase depends heavily on the input data size. Fig. 2 on the next page gives the relative speed up at different data size. The CUDA performs better only when the data size is large than 262144.

There is no one execution configuration that works well on all ranges of data size. In general, the block size should be as large as possible, ie 512 when the input is large. The grid size tend of perform best when it is half of the data size. This is what we expected. If the grid size is large, it takes long to start all the threads and then execute. If the grid size is small, it takes many times to start all the kernels and the performance is bad.

*2) Possible Improvements:* The bitwise operation is very easy in its' underlying implementation. Since it only utilize the input data once so it is impossible to optimize by using shared memory. It only processes a maximum of 7.89GB data per second. One possible optimization is adaptive execution configuration at runtime based on the input size. Since our time is limited for this project, we did not have the chance to implement it. This optimization can only keep the GPU get similar performance improvement compared with CPU at all data size that are significantly large. To determine the threshold CPU and GPU performance, more testings have to run at different configurations.

### C. Addition

*1) Result:* We first vary the block size and grid size to find their effect on the performance. The data size is $256 \times 21000$ limbs and the data is generated randomly in this test. From Fig. 3 on the following page, the performance is best when the block size is 64 and grid size is 12000.

We find the relative performance increase depends heavily on the input data size. Fig. 4 on page 5 gives the relative

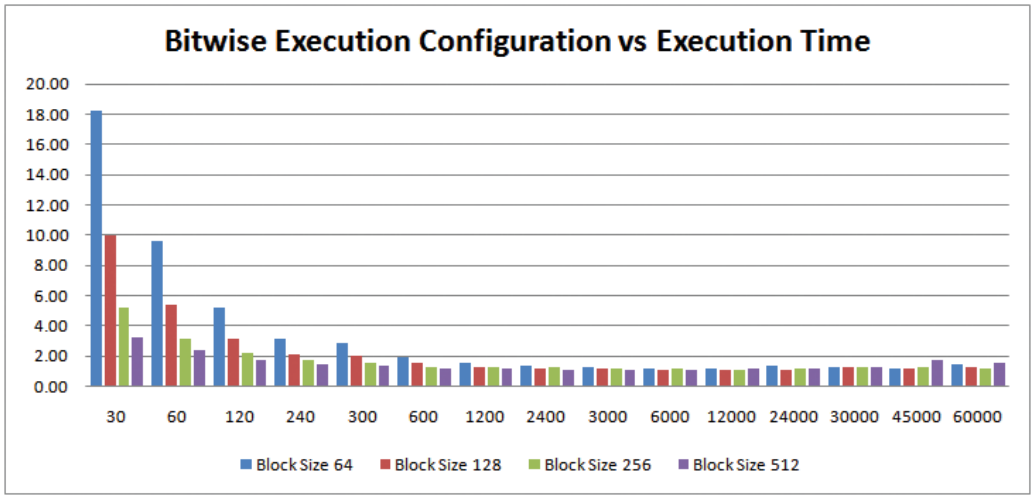


Fig. 1. Bitwise Block Size and Grid Size Effect on Performance

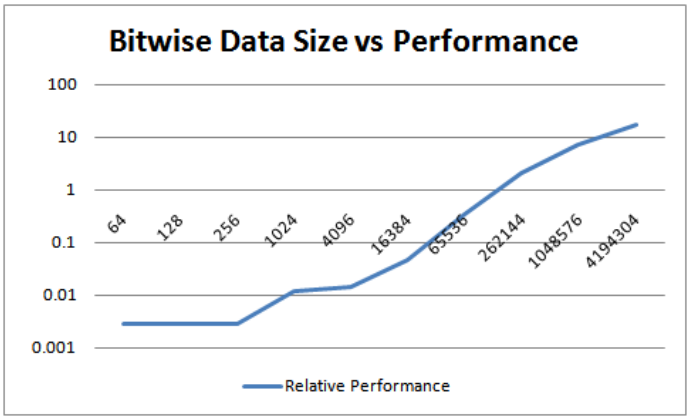


Fig. 2. Bitwise Data Size Effect on Performance

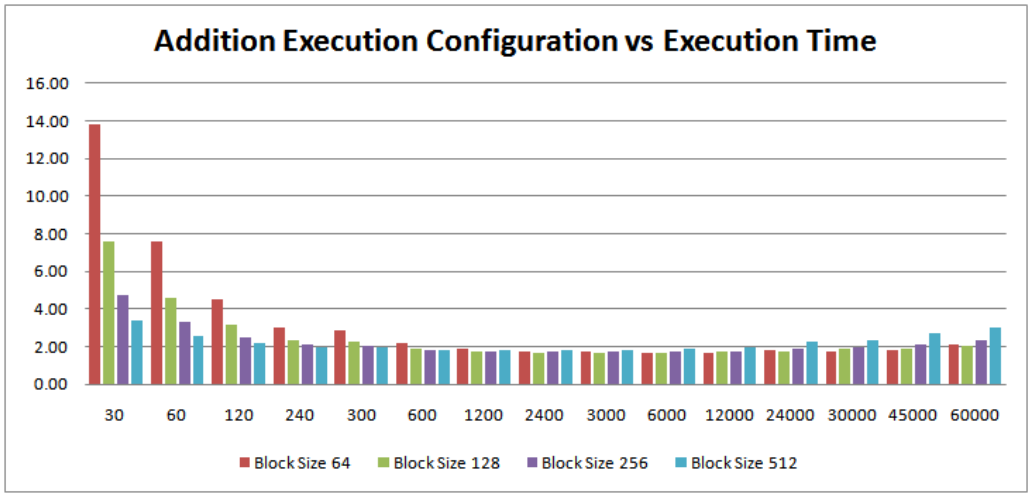


Fig. 3. Addition Block Size and Grid Size Effect on Performance

speed up at different data size. The CUDA performs better only when the data size is large than 65536.

There is no one execution configuration that works well on all ranges of data size. In general, the block size is better when the block size is small, ie 64. The grid size tend of perform best when it is half of the data size. This is what we expected. If the grid size is large, it takes long to start all the threads and then execute. If the grid size is small, it takes many times to start all the kernels and the performance is bad.

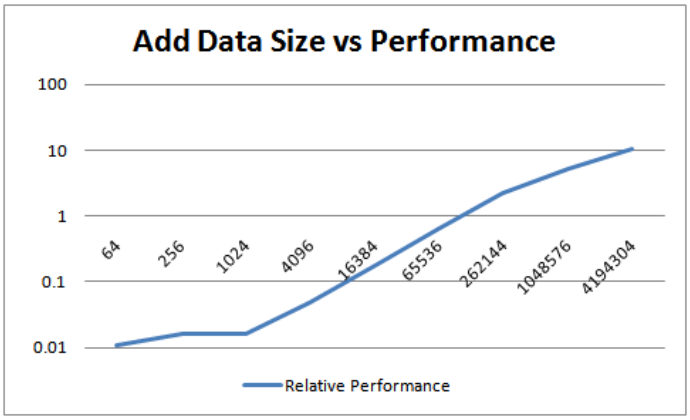


Fig. 4. Add Data Size Effect on Performance

*2) Possible Improvements:* The effective memory bandwidth is about 10.9GB per seconds. There are two major limitations of the implementation of addition. First, the carry bit has to be propagated and most of the threads are just idle during this process. Since only one kernel can execute at the same time, most of the hardware are just idle. This issue can be improved when the Fermi™becomes available since it can support multiple kernels execute at the same time. The second limitation is it is far from reaching the maximum memory bandwidth. This is indeed the same issue as the first one. Overall, the limitations is caused by the algorithm to perform addition. The carry propagate from the lowest bit to highest bit is a sequential process and cannot be parallelized.

### D. Subtraction

*1) Result:* We first vary the block size and grid size to find their effect on the performance. The data size is $256 \times 21000$ limbs and the data is generated randomly in this test. From Fig. 3 on the preceding page, the performance is best when the block size is 64 and grid size is 12000.

We find the relative performance increase depends heavily on the input data size. Fig. 6 on the next page gives the relative speed up at different data size. The CUDA performs better only when the data size is large than 65536.

There is no one execution configuration that works well on all ranges of data size. In general, the block size works best at 128. The grid size tend of perform best when the grid size 45000. This is different from what we expects. We have to perform more investigation on this.

*2) Possible Improvement:* The effective memory bandwidth is about 8.99GB per seconds. The are three major limitations of the implementation of subtraction. First, the carry bit has to be propagated and most of the threads are just idling. Since only one kernel can execute at the same time, most of the hardware are just idling. This issue can be improved when the Fermi™becomes available since it can support multiple kernels execute at the same time. The second limitation is it is far from reaching the maximum memory bandwidth. This is indeed the same issue as the first one. The third limitation is that there is an additional step compared with addition. The subtraction is performed by adding the two's complement of the subtracter Overall, the limitations is caused by the algorithm to perform addition in the end. The carry propagate from the lowest bit to highest bit is a sequential process and cannot be parallelized.

### E. Pop Count

*1) Result:* We first vary the block size and grid size to find their effect on the performance. The data size is $256 \times 45000$ limbs and the data is generated randomly in this test. From Fig. 7 on the following page, the performance is best when the block size is 64 and grid size is 12000.

We find the relative performance increase depends heavily on the input data size. Fig. 8 on page 7 gives the relative speed up at different data size. The CUDA performs better only when the data size is large than 65536.

There is no one execution configuration that works well on all ranges of data size. In general, the block size should be as large as possible, ie 512 when the input is large. The grid size tend of perform best when it is half of the data size. This is what we expected. If the grid size is large, it takes long to start all the threads and then execute. If the grid size is small, it takes many times to start all the kernels and the performance is bad.

*2) Possible Improvement:* To improve the performance of Pop Count operation further, we propose that by unrolling the the CUDA kernel of pop count to let each thread to pop several limbs of the operands may help as less kernel will be invoked in this scenario and overhead of invoking kernels is eliminated.

### F. Set

*1) Result:* We first vary the block size and grid size to find their effect on the performance. The data size is $256 \times 45000$ limbs and the data is a constant in this test. From Fig. 9 on page 7, the performance is best when the block size is 256 and grid size is 24000.

We find the relative performance increase depends heavily on the input data size. Fig. 8 on page 7 gives the relative speed up at different data size. The CUDA performs better only when the data size is large than 65536.

There is no one execution configuration that works well on all ranges of data size. The grid size tend of perform best

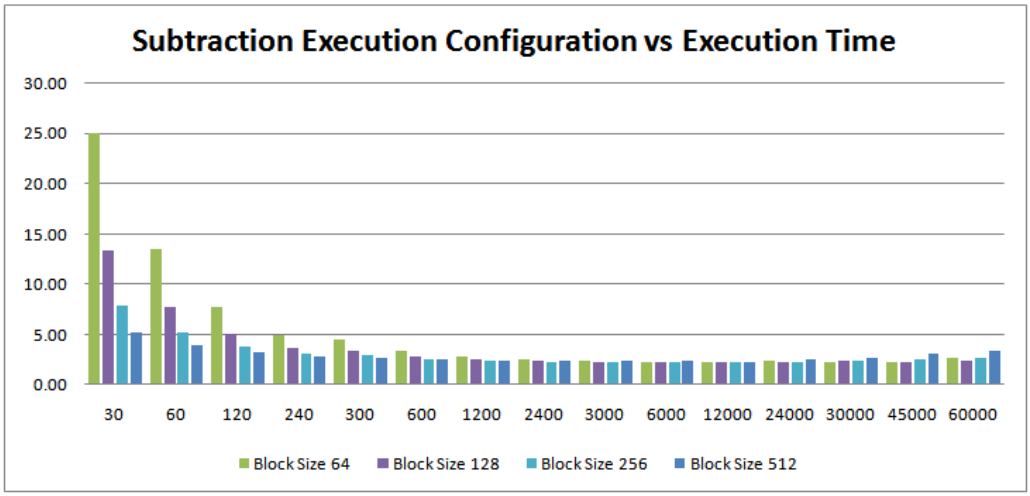


Fig. 5. Subtraction Block Size and Grid Size Effect on Performance

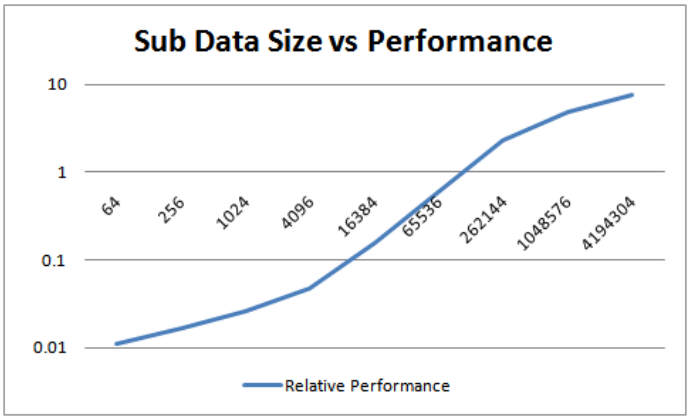


Fig. 6. Sub Data Size Effect on Performance

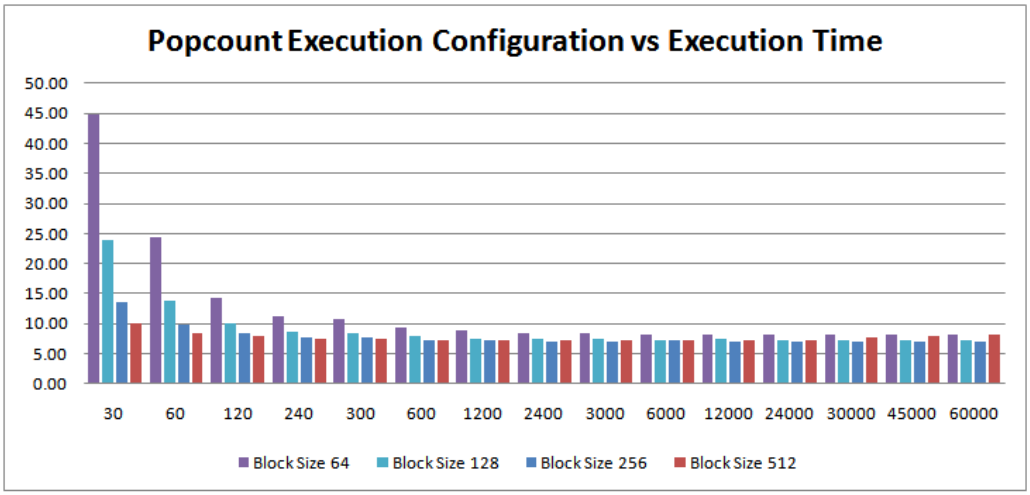


Fig. 7. Pop Count Block Size and Grid Size Effect on Performance

when it is half of the data size. This is what we expected. If the grid size is large, it takes long to start all the threads and then execute. If the grid size is small, it takes many times to start all the kernels and the performance is bad.

*2) Possible Improvement:* There is not much of performance improvement that can be done on the set function. Also, there is one limitation of this function, all the limbs have to be set to the same value. It is more useful for clearing all the values or initialize everything to 1.

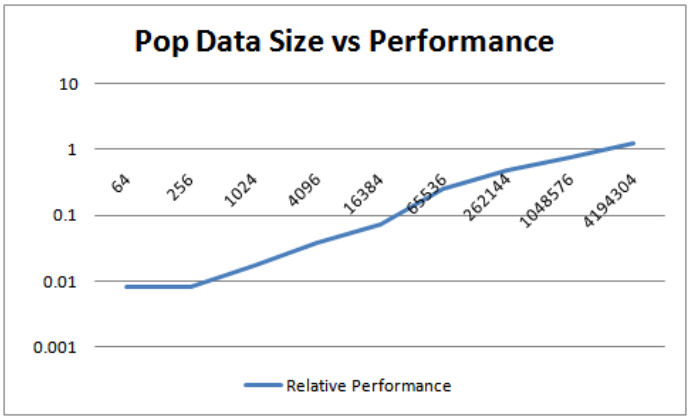


Fig. 8. Popcount Data Size Effect on Performance

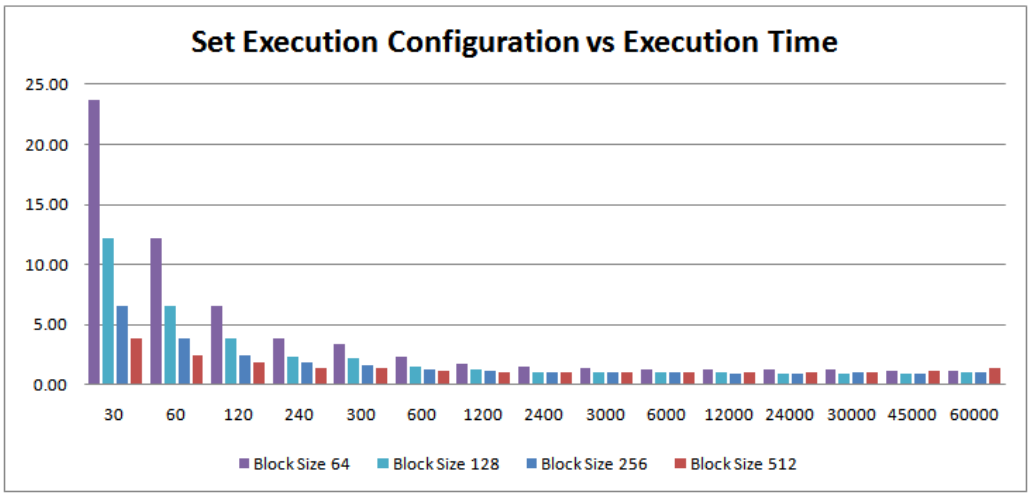


Fig. 9. Set Block Size and Grid Size Effect on Performance

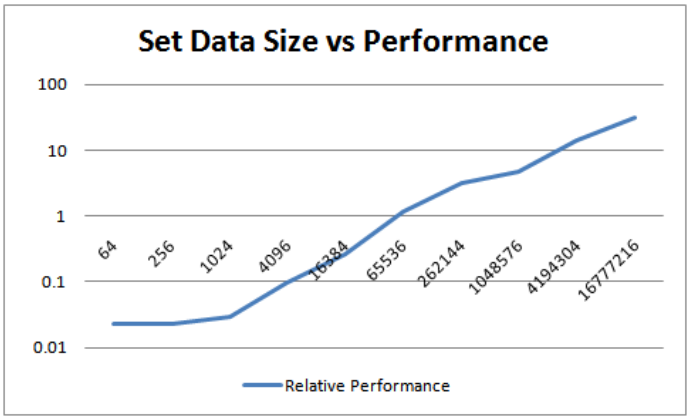


Fig. 10. Set Data Size Effect on Performance

### G. Multiplication

*1) Result:* We first vary the block size and grid size to find their effect on the performance. The data size is $256 \times 1024$ limbs and the data is generated randomly in this test. From Fig. 11 on the following page, the performance is best when the block size is 128 and grid size is 2400.

We find the relative performance increase depends heavily on the input data size. Fig. 12 on the next page gives the relative speed up at different data size. The CUDA performs better only when the data size is large than 65536.

*2) Possible Improvement:* The are three major limitations of the implementation of multiplication. First, the multiplication algorithm has to be improved for better accuracy of the result. Second, the current GPU memory size is too small. Although there is 1GB of memory on GTX280, the available memory

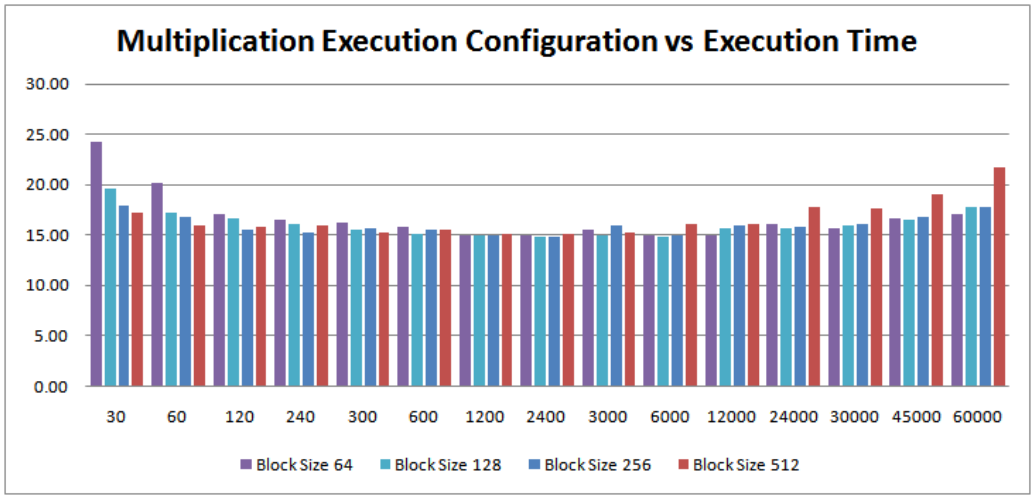


Fig. 11. Mul Block Size and Grid Size Effect on Performance

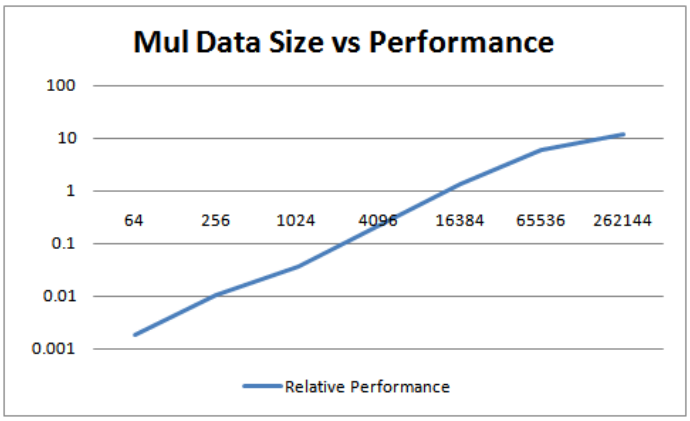


Fig. 12. Mul Data Size Effect on Performance

to the user is very small and is about 400MB. We cannot test larger data size on the hardware that we have. The new Fermi Tesla platform may help since it has 4GB or even 6GB of memory and they are all available to the user since there is no display function on the Fermi Tesla platform.

## V. Conclusion and Future Work

We have successfully implemented part of the GMP library on CUDA platform and have done some testing on it. We get a performance improvement about 10X in cases where the data size is large enough. This is lower than what expected. Depends on the user data size, this implementation of GMP may not be very widely used. The Fermi architecture may perform better since it can support multiple kernels running at the same time, in such way, each operation can have a smaller data size but the performance will remain or improved.

We also find that parallel algorithm is the most important thing when writing CUDA application. Without good algorithm, it is impossible to write performance code on CUDA.

## References

[1] "What is gmp?" [Online]. Available: http://gmplib.org/
[2] "Any luck on gmp cuda implementation."
[3] B. S. M. Belzner, M. Rohr, "Arbitrary precision integers on the cell processor," Tech. Rep.
[4] K. Zhao, "Implementation of multiple-precision modular multiplication on gpu," Tech. Rep.