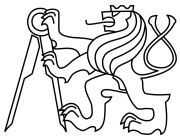
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	CUDA implementation of the GMP library
Student:	Bc. Petr Petrouš
Supervisor:	Ing. Ivan Šimeček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2016/17

Instructions

1) Analyse the previous solution [1] and try to find weak spots and performance reserves. As a result make a decision if it will be better to reuse and improve the existing solution or to start over and implement a new library.

2) Implement methods for large integer arithmetic, especially addition, subtraction, and multiplication. And also add a support for bitwise operations AND, OR, XOR and SHIFT. For multiplication, use the FFT (Fast Fourier Transform) algorithm. During implementation apply improvement suggestions mentioned in [2].

3) The resulting solution will be compared and evaluated with the GMP library working on CPU only. Main focus will be on execution performance.

References

[1] DP Kamil Šnajdr, FIT, 2013
[2] GMP implementation on CUDA - A Backward Compatible Design With Performance Tuning (http://individual.utoronto.ca/haojunliu/courses/ECE1724_Report.pdf)

L.S.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 5, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Master's thesis

CUDA implementation of the GMP library

Bc. Petr Petrouš

Supervisor: Ing. Ivan Šimeček, Ph.D.

10th May 2016

Acknowledgements

I would like to thank to my supervisor for valuable feedback during writing of this thesis. Furthermore I would like to thank to my girlfriend for emotional support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for nonprofit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 10th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Petrouš, Petr. *CUDA implementation of the GMP library.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Cílem této práce je zhodnotit použitelnost CUDA technologie pro práci s velkými čísly. Byla implementována knihovna podobná knihovně GMP, ale místo na CPU probíhají výpočty na grafické kartě. Mezi podporované operace patří sčítání, odčítání a násobení velkých čísel a dále bitový posun a operace AND, OR a XOR. Násobení probíhá pomocí algoritmu využívajícího rychlé Fourierovy transformace. Knihovna byla porovnána s GMP knihovnou z hlediska přesnosti a zejména z hlediska výkonu.

Klíčová slova CUDA, GMP, FFT.

Abstract

Main goal of this thesis is to asses CUDA technology for large integer arithmetic. A library similar to GMP was developed, supporting addition, subtraction, multiplication of large integers and also bitwise shift, AND, OR and XOR operations. Multiplication was implemented using fast Fourier transform algorithm. Library was compared to GMP library, mainly from performance and precision perspective.

Keywords CUDA, GMP, FFT.

Contents

List of Figures

List of Tables

Introduction

Modern computers can generally work with 64-bit integers. Arithmetic operations with numbers of this precision can be done natively in hardware component called arithmetic logic unit (ALU). But what if we want to compute with numbers larger than that?

Several solutions to this problem exist today. Our main contender will be GNU Multiple Precision Arithmetic Library (also called GMP). How big integers can this library handle? Authors claim that there is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on [1].

GMP is around since 1991 and it's main target applications are cryptography applications and research, security applications, algebra systems and computational algebra research. GMP is carefully designed to be as fast as possible. It is even capable of using multiple algorithms for the same operation, choosing the right one based on operand sizes.

On the other hand, there is always the possibility to go further. GMP does all of it's computations single-threaded, even though we have many core CPUs today. Apart from that, there are several claims that, with good parallel algorithm, using GPU as a parallel computing platform can provide considerable speedup against GMP (stated in articles [2] and [3]).

Goals

Main goal of this thesis is to asses the suitability of CUDA technology to do large integer arithmetic. At first I want to review existing parallel algorithms, their precision and if I will be able to use them with CUDA. Then I will implement chosen algorithms, mainly for addition, subtraction, multiplication and bitwise operations like SHIFT, AND, OR and XOR. At last I will focus on performance optimizations to be able to show that CUDA can be considerably faster than existing CPU-based solutions.

For performance comparison I will use GMP as a benchmark as it is one of the most common library to do large integer arithmetic on CPU.

The results will be publicly available on github, so anyone can improve, extend or validate existing solution. I believe this to be an important step in moving the research around CUDA technology further and I encourage all future researchers to build on it.

Thesis structure

First part of this thesis is dedicated to analysis of the necessary algorithms, theoretical background and overview of existing work. Separate chapter is also dedicated to CUDA technology where I would like to make readers familiar with the basic concepts of GPGPU computing in context of CUDA.

Next chapter is called design. There I discuss various architectural decisions, explain used algorithms and application structure. This chapter might be useful for someone trying to go through the code for contribution, review or other usage.

Following chapter is dedicated to realization details - what has been done and how it's been done. I try to describe all the implementation details and performance optimizations made.

In the last chapter I evaluate performance of the new library, sum up achievements and try to explain what further limitations exist. Then I try to outline possible further research in this topic. Things I believe would be interesting to implement and to evaluate against existing work and against GMP as well.

After conclusion there is also bibliography, contact to the author, installation instructions and contents of the attached media.

Chapter 1

Analysis

In this chapter I am going through the necessary algorithms, problem definition, theoretical background and overview of existing work. I will discuss Fourier transform as a mean of multiplication. I will also explain how large integers are stored in memory.

1.1 Storing large numbers in memory

As discussed in introduction, I am going to deal with integer numbers almost as large as available system memory. This requires an efficient way of storing such numbers in memory to allow fast arithmetic operations to be done with them.

As I will try to mimic GMP interface in operations implemented in my library, I will also adopt a concept of limbs. A limb means the part of a multi-precision number that fits in a single machine word. Word limb has been chosen because a limb of the human body is analogous to a digit of a number (only larger, and containing several digits). In our case, limb is 64 bits of data stored as unsigned integer [4].

1.2 Theoretical background

As far as large integer arithmetic is concerned, there are some concepts, algorithms and mathematical theory needed to go through, especially when effective parallelization is in question.

1.2.1 Addition operation

Parallel addition is basically very simple - it is enough to add corresponding limbs together, in parallel. The result, however, can overflow and therefore produce carry.

1.2.1.1 Carry propagation

To propagate carry appropriately there are number of ways. We could for example take that carry and propagate it in the thread that detected it - that means to increment the more significant result limb. That operation of incrementing other result limb can however also produce carry. But the main problem is, that the more significant limb might not be computed at the time - we would therefore end up with a race conditions and data dependencies.

To solve this problem without data dependencies, I propose to originate carry detection on the target. That means, looking in the least significant direction. To be able to detect, that carry could not affect the limb in question, we have to determine one of three states on each limb.

The obvious one is *carry* - that happens when a limb produces carry. Second one is usually called *propagate* - that is a special state that would produce carry, if carry would come from lesser significant limb. This state happens only when resulting limb is equal to maximum value, limb can hold - adding one therefore produces carry. The third is possible state is obvious - *nothing*. If carry would propagate from lesser significant bit, it would not propagate any further.

That means, the process of looking for carry can safely stop when it reaches other than *propagate* state. If it reaches *carry*, it increments it's own limb. If it reaches *nothing* state or the least significant limb, it just stops. Example of this, shown on single-bit hardware adder, is presented at chapter [5], section Carry look-ahead adder.

To summarize, what each thread does:

- Save its own state (carry/propagate/nothing) to carry buffer
- Look in the least significant direction

  if *nothing* is found, stop

  if *propagate* is found, continue in least significant direction

  if *carry* is found, add 1 to its result

1.2.2 Multiplication operation

For multiplication, GMP actually uses several different algorithms as stated in [6]. It chooses the right algorithm based on the operands sizes. Those algorithms are (from smallest to biggest operands):

- Basecase (schoolbook)
- Karatsuba
- Toom-Cook

- FFT

It is my main ambition to improve the performance on really large operands. That means my main target is to implement FFT multiplication as it is asymptotically fastest algorithm, that is commonly used.

Multiplication using FFT is made possible because of Convolution theorem as explained in section 1.2.2.5.

1.2.2.1 Fourier transform

The Fourier transform decomposes a function of time into the frequencies that make it up. The result of such transform is a complex valued function of frequency, whose absolute value represent the amount of that frequency in the original function (input signal) and whose complex argument is the phase offset of the sinusoid representing that frequency. We can therefore say that Fourier transform is frequency domain representation of time domain function.

1.2.2.2 Discrete Fourier transform

The discrete Fourier transform (DFT) converts a finite sequence of equally spaced samples of a function into the list of coefficients of a finite combination of complex sinusoids, ordered by their frequencies, that has those same sample values.

The input samples are complex numbers (in practice, usually real numbers as in our case), and the output coefficients are complex as well.

1.2.2.3 Fast Fourier transform

The fast Fourier transform is a discrete Fourier transform algorithm which reduces the number of computations needed for n elements from $O(n^2)$ to $O(n \log n)$[7].

Several algorithms exists for FFT. As part of CUDA Toolkit there is cuFFT library, that uses Cooley-Tukey algorithm as stated in documentation [8], [9]. It is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets. It is one of the most important and widely used numerical algorithms in computational physics and general signal processing. Internally it uses single or double precision floating point complex numbers in frequency domain. It's internals are discussed a bit closer in section 2.4.

1.2.2.4 Cooley-Tukey

As nicely explained in paper [10], the elegance of the Cooley-Tukey algorithm lies in its divide-and-conquer nature - lets explain how a polynomial can be

split up into two polynomials of half the original degree, and how we can combine their results to compute the results of the original polynomial.

If n is even, we can divide an $n-1$ degree polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

into two $(n/2-1)$ degree polynomials

$$\begin{aligned} p_{even}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ p_{odd}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned} \tag{1.1}$$

which we can combine into p using the equation

$$p(x) = p_{even}(x^2) + x p_{odd}(x^2). \tag{1.2}$$

We can easily verify that the above formula for combining the two polynomials is correct by entering x^2 into the formulas of p_{even} and p_{odd}. In the following, let "all n powers of ω" denote the set of values ω^k, with k ranging from 0 to $n-1$. We will also assume that n is a power of 2.

Let our recursive FFT procedure be declared with two parameters, one specifying the $n-1$ degree polynomial p, and the other, which we call ω, specifying the nth primitive root of unity. So by definition of the DFT, what we have to do in this procedure is evaluate p at all n powers of ω. Ignoring the base case for now, we first split the polynomial up into two polynomials p_{even} and p_{odd}, according to equation 1.1.

Now we perform a recursive call to our FFT procedure for these two polynomials, passing ω^2 as second parameter, which is justified by the reduction property, which tells us that ω^2 is a primitive $n/2$th root of unity and since n is a power of two, p_{even} and p_{odd} are both of degree $n/2-1$ (since they are half the size of the original polynomial, by definition of even and odd).

So by these recursive calls, we now have the values for p_{even} and p_{odd}, evaluated at all the $n/2$ powers of ω^2. Since we know by the reduction property that all $n/2$ powers of ω^2 are a subset of the powers of the n powers of ω, and to be more specific, we know that they match exactly half the values of the n powers of ω, we can combine their results using equation 1.2, which will give us evaluations of p for half the values in the set of n powers of ω. The other half, we can derive from these values by using the reflective property. This idea is formalized in equation 1.3:

$$\begin{aligned} p(\omega^k) &= p_{even}(\omega^{2k}) + \omega^k p_{odd}(\omega^{2k}) \\ p(\omega^{k+n/2}) &= p_{even}(\omega^{2k}) - \omega^k p_{odd}(\omega^{2k}) \end{aligned} \tag{1.3}$$

The first line of equation 1.3 is just equation 1.1 filled in for ω^k. Note that this use of the formula is valid, since we have called p_{even} and p_{odd} with the

value ω^2, their values at the kth index are actually their evaluations at the kth power of ω^2, $((\omega^2)^k)$. Furthermore, equation 1.2 specifies that p_{even} and p_{odd} be called with a square, which they are, since $(\omega^2)^k = (\omega^k)^2 = \omega^{2k}$.

Now since we have $n/2$ values in which p_{even} and p_{odd} are evaluated, we can loop equation 1.3 $n/2$ times, with k ranging from 0 to $n/2 - 1$. Recall that this will give us evaluations of p for exactly half the values (the lower half) in the set of n powers of ω in which we have to evaluate p. The other half is calculated in the second line of equation 1.3.

The second line of equation 1.3 is justified by the reflective property, which we recall telling us that $\omega^{k+n/2} = -\omega^k$. So by this property, we have to flip a sign wherever ω^k occurs. Note that the only actual difference between the first and second line of equation 1.3 is a single minus sign. This is because the other two occurrences of ω^k are squared, and we know that for every number a, real or complex, $a^2 = (-a)^2$, so we do not need to change these products.

1.2.2.5 Convolution theorem

Convolution theorem states that the Fourier transform of a convolution is the pointwise product of Fourier transforms. In other words, convolution in time domain equals point-wise multiplication in the frequency domain. Let $\mathcal{F}$ be a Fourier transform, $*$ convolution operator and $\cdot$ pointwise multiplication operator. Then we can say that:

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$$

If we take that one step further it means, that

$$f * g = \mathcal{F}^{-1}\big\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\big\}$$

In the above equations f and g are polynomials (in time domain). $\mathcal{F}\{f\}$ and $\mathcal{F}\{g\}$ are those polynomials represented in frequency domain. Normally, multiplication of two polynomials (convolution) would take $O(n^2)$ steps for polynomials of length n. Pointwise multiplication in frequency domain on the other hand takes only $O(n)$ steps.

1.3 Existing work

There have been a number of works done on the topic of large integer arithmetic, on polynomial multiplication etc. In the following chapter I will try to give an idea of things I have used in my design, things that I have had to modify and things that I have done differently.

1.3.1 Source code availability

I have not been able to find source codes for almost none of the papers and works I have studied in preparation for this thesis. That said, paper [2] at least contains some code samples on memory structures. In article [11], there are some samples of how the cuFFT library is used during the process of multiplication, which is helpful to begin with.

But in general I am disappointed that I couldn't dig inside the code behind those articles, that describe sometimes quite complex algorithms. This lead me to decision to publish my source codes on github - I expect that to help someone trying to extend research in the area of CUDA computing.

1.3.2 Performance and precision

In Masters Thesis [12], the main goal is to implement as many algorithms as possible. The performance results on the other hand are not as impressive as in other works. For example multiplication, even using Schönhage-Strassen algorithm, which is basically FFT multiplication, is approximately 30 times slower than GMP library for the largest operands tested. Interestingly enough, this implementation is even a little bit slower than GMP doing addition, but the results are at least comparable.

I suspect that the main problem of this implementation is, that the operands are mostly kept in host memory only. That means, the arithmetic operation itself has to do all the memory transfers to and from CUDA device, which can hurt performance a lot. On the other hand, this should not affect multiplication so much, as the main calculation should take considerably more time than the memory operations itself. It seems, that the main performance problem is with FFT implementation and probably with selection of recursive multiplication algorithm. That said, this implementation of multiplication probably does not suffer from FFT rounding errors.

More promising performance results come in paper [2]. This project has performance evaluation of CUDA technology as a main goal and it compares with GMP library. It achieves around 10 times the performance of GMP for large enough operands. On the other hand, implementation details are discussed very briefly, I suspect for example that they use cuFFT library, but it does not say it clearly. This is also one of the projects that mentions rounding errors and possible loss of precision doing FFT multiplication. That said, there is no evaluation of this, so I do not know how big error can be expected.

Another, very valuable, source is paper [3]. In this paper, the author is very clear about multiplication implementation details, it uses cuFFT library and it helped me a lot to understand some key parts of the algorithm. As for performance, this project achieved approximately twice the speed of GMP. It

does not deal with other methods than multiplication and rounding errors due to the use of FFT are not mentioned at all.

1.3.3 Integer representation

Weird thing that surprised me on paper [3] is using very different concept to store integers in memory. For polynomial representation this project uses base 10 and base 100. I have not seen any benefit in this, it is probably due to use of some library that stores numbers this way internally. I see this as a performance bottleneck as it prevents usage of bit masking and bit shifting for polynomial representation and overflow propagation. Instead, it has to use multiplication and division.

Interestingly, the same concept of base 10^k appeared also in paper [10]. This work does not use GPU at all, but it discusses rounding error in detail and does provide some measurements of those errors. Those measurements, on the other hand, are not comparison to the correct results, but it just detects and sums up rounding error on the way as the computation goes.

1.3.4 Number Theoretic Transforms and Fast Convolutions

There is also a paper [13] proposing usage of Number Theoretic Transforms (NTT) with FFT transform. This can be viewed as FFT over finite field. That eliminates rounding errors, but adds quite a lot of complexity. For result extraction, there is Chinese Remainder Theorem used. I have discovered this work late in the process of implementation, but I propose ideas presented in the future work section 6.3.

1.3.5 Conclusion

As there was no implementation with similar targets (mainly performance), that I could build on, available, I have decided to implement my own library from scratch.

To provide reasonable comparison with GMP, I decided to try to implement the interface as similar as possible. Operations will have the same function names. Data types will be the same or will extend the current ones.

Results will be published together with source code, so that future researchers can more easily implement more advanced functions, hopefully being able to reuse most of the logic that I will implement.

Chapter 2

CUDA technology

In this chapter I would like to introduce CUDA technology, which I will be using to implement large integer arithmetic with parallel algorithms.

CUDA is a parallel computing platform and programming model invented by NVIDIA. It allows to use GPU as a parallel co-processor. It offers significant speed-up in performance when good parallel algorithm is available. On the other hand, it introduces several additional responsibilities to the programmer like managing two separate memory address spaces.

To distinguish memory and operations in GPU from those in CPU and main computer memory, we will call GPU with its memory the **device** and CPU with its RAM the **host**.

2.1 Kernels

CUDA C extends C language by functions called *kernels*, which, when called, are executed N times in parallel by N different CUDA threads.

That means that instead of writing one for-loop to do a vector addition like in Source code 1, we would write simple function to add one element of the vector and then we execute as many threads as there are elements of the vectors - as presented in Source code 2.

Source code 1 Vector addition using traditional for-loop.

```
for (int i = 0; i < size; i++)
{
    c[i] = a[i] + b[i];
}
```

Source code 2 Vector addition using kernel run in parallel on CUDA.

```
__global__ void VectorAdd(float* a, float* b, float* c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main()
{
    ...
    VectorAdd<<<size / blockSize + 1, blockSize>>>(A, B, C);
}
```

2.2 Threads, blocks and grids - CUDA Thread hierarchy

Inside each kernel we usually start with determination of thread number - which is similar to iterator (i) inside a for-loop. There is some complexity to it, since there are multiple levels of thread execution in CUDA technology as illustrated in Figure 2.1.

At the lowest level, there is a thread. Each thread has it's own memory - very fast, but also very small - can be compared to L1 memory inside the CPU. We are declaring per-thread local memory inside a kernel the same way as we would create local variable inside a standard C function. In Source code 2 we do use this memory to store thread id (int i).

Each thread also can be uniquely identified. To do so, we have several built-in variables, for thread number it's threadIdx. Since CUDA runs on 3D graphics accelerator, those variables are 3-component vectors and can therefore have separate identificators for up to three dimensions. In our case we will manage with one dimension only.

Threads are then grouped together to blocks. Each block can consist of up to 1024 threads. Block, same as a thread, is not only a logical concept, but also has it's own memory area called shared memory. It is also very fast and it is shared across all threads inside each block, but each block has it's own. To synchronize access to shared memory, there is __syncthreads(); intrinsic function, which acts like a barrier. That means each thread stops execution until all threads from one block reach the barrier. Shared memory is declared inside a kernel with __shared__ prefix.

To identify thread inside a block, there are two more built-in variables. BlockIdx identifies a block inside a grid. BlockDim specifies the number of threads inside each block.

Blocks are then grouped inside a grid. Grid has no private memory area.

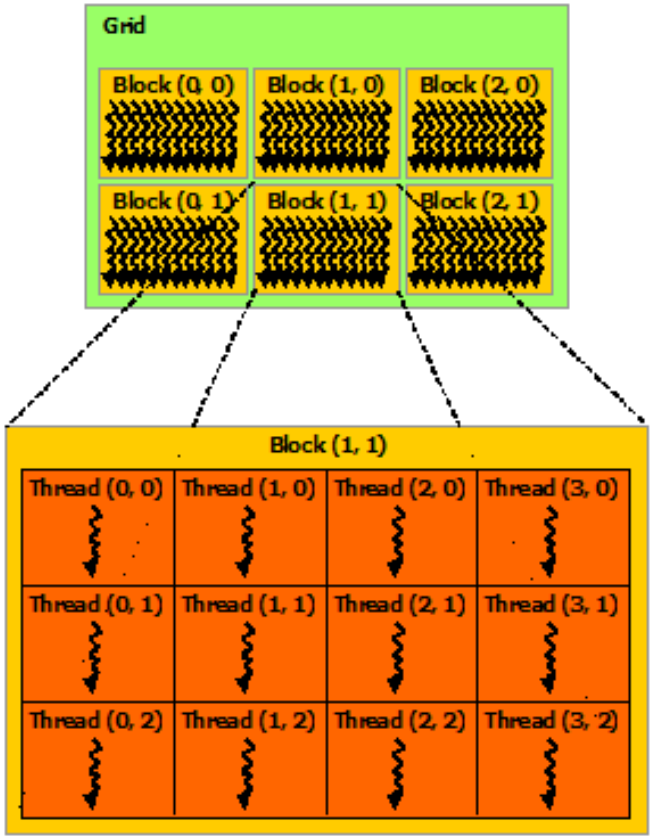


Figure 2.1: CUDA Thread hierarchy

Number of threads inside each block (blockDim) and number of blocks is determined upon kernel execution inside <<<...>>>syntax. Inside there are two numbers - number of blocks and number of thread per block.

2.2.1 Thread execution - warps

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. This architecture is called SIMT - single instruction, multiple threads. Further description is available in CUDA documentation [14].

2.2.2 Device memory hierarchy

Apart from per-thread local memory and per-block shared memory, there is yet another area called global memory. This memory area is persistent not even through the life of one grid, but even through multiple kernel executions. Memory hierarchy is nicely illustrated in Figure 2.2.

Memory chunks are usually allocated using cudaMalloc API call from the host. Returned memory pointers are usually passed to the threads using kernel arguments.

2.2.3 Heterogeneous programming

As illustrated by Figure 2.3, the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a co-processor to the host running the C program. The kernels execute on a GPU and the rest of the C program executes on a CPU.

The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the memory spaces visible to kernels through calls to the CUDA runtime (described in 2.3). This includes device memory allocation and deallocation as well as data transfer between host and device memory.

2.3 CUDA Programming Interface

The runtime is implemented in the cudart library, which is linked to the application, either statically via cudart.lib or dynamically via cudart.dll. All its entry points are prefixed with cuda - for example cudaMalloc, cudaMemcpy etc.

There is some initialization of CUDA runtime going on, but this all happens automatically after first CUDA API call is made. CUDA context is then created and kernel code is compiled using Just-in-Time compilation scheme and loaded into device memory. This context then lives until cudaDeviceReset

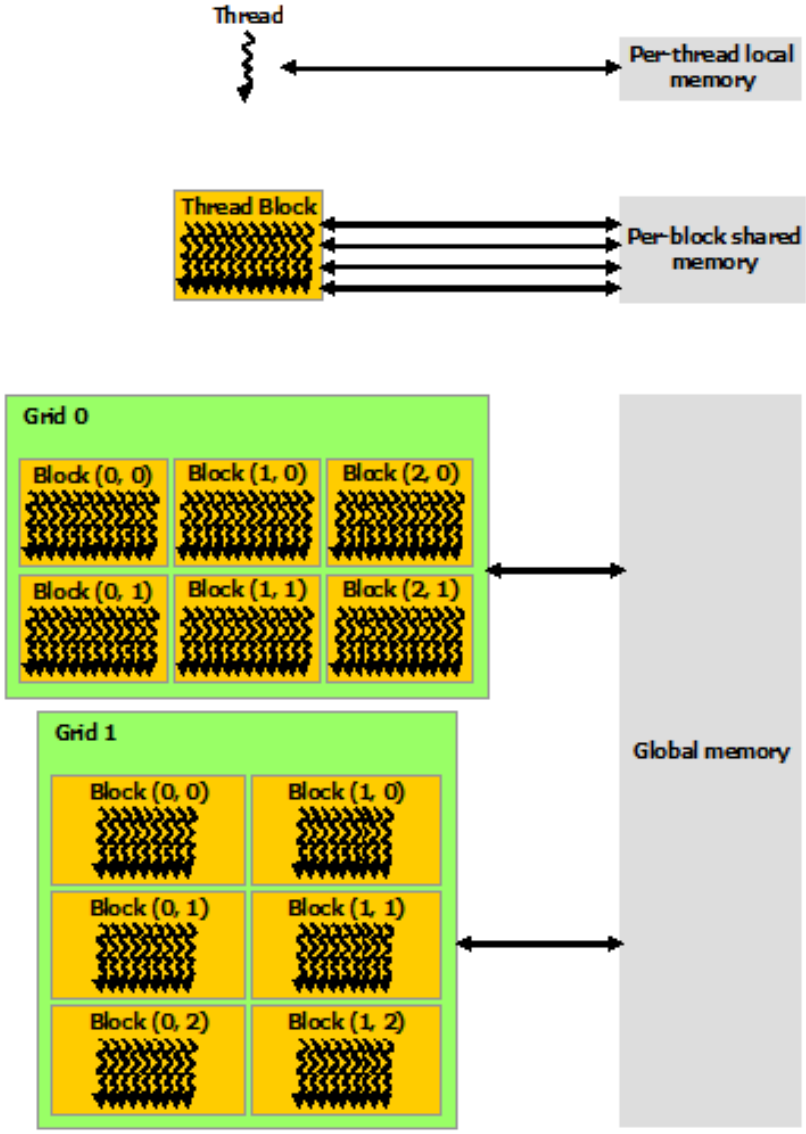


Figure 2.2: CUDA Memory hierarchy

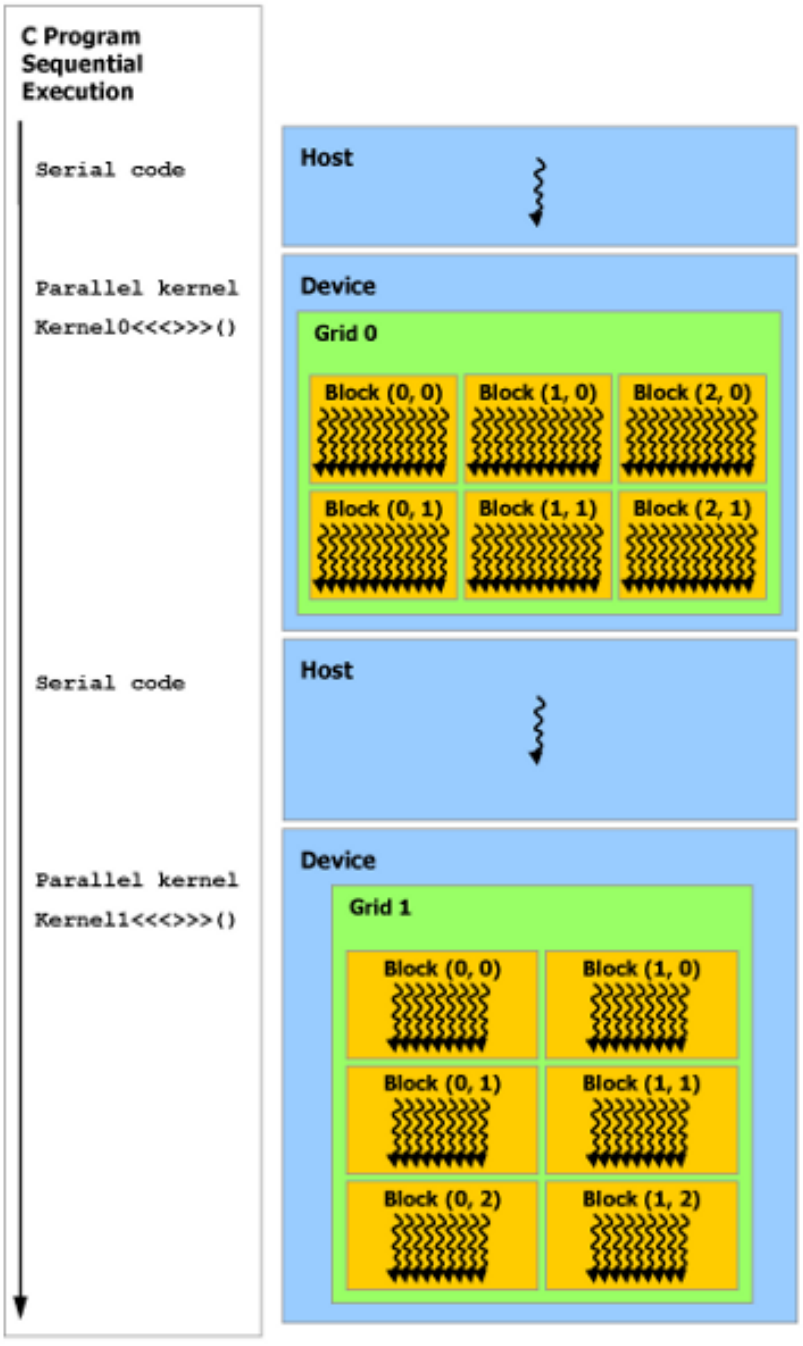


Figure 2.3: CUDA Heterogeneous programming

is called. This function clears all state information and destroys the context until other CUDA API call is executed.

2.3.1 Memory management

As stated before, host and device both have separate memory. In order to make parallel calculations inside CUDA device, there is usually the need to copy the data to the device. After the calculation, there is the need to copy results back to the host memory.

Typical example of such operations can be nicely illustrated on example with vector addition in Source code 3 - allocate memory for operands using cudaMalloc, copy operands using cudaMemcpy, execute addition kernel in parallel on CUDA, copy result back from device to host, free the allocated memory using cudaFree.

As explained in section 2.2.2, global device memory (allocated using cudaMalloc) is persistent. Therefore our operands can be initialized and copied to the device in completely different part of the code. This way we can separate operand initialization from the main calculation in performance measurement 5.1.

2.4 cuFFT library

CuFFT stands for NVIDIA CUDA Fast Fourier Transform library. It consist of two libraries actually. CuFFT is designed to provide high performance on CUDA GPUs. CuFFTW is to provide an easy way to start using GPU to FFTW users. FFTW is a common FFT open-source library.

That said, I will be using performance optimised cuFFT version. This library is highly performance optimized, it allows transfer sizes of up to 512 million elements in single precision and up to 256 million elements in double precision floating points. It actually allows execution on multiple GPUs simultaneously. It is nicely documented in the manual [15].

The algorithm that cuFFT library uses to do FFT transform is named, after the two authors, Cooley-Tukey. It is one of the most common FFT algorithms (explained in section 1.2.2.4).

2.4.1 Plans and memory consumption

CuFFT provides a simple configuration mechanism called a plan that uses internal building blocks to optimize the transform for the given configuration and the particular GPU hardware selected. Then, when the execution function is called, the actual transform takes place following the plan of execution. The advantage of this approach is that once the user creates a plan, the library retains whatever state is needed to execute the plan multiple times without recalculation of the configuration. This model works well for

Source code 3 Transferring memory to and from CUDA device - example.

```
__global__ void VectorAdd(float* a, float* b, float* c)
...

int main()
{
    ...
    size_t arr_size = sizeof(float) * size;
    float *dev_a, *dev_b, *dev_c;
    cudaMalloc((void**)&dev_a, arr_size);
    cudaMalloc((void**)&dev_b, arr_size);
    cudaMalloc((void**)&dev_c, arr_size);

    cudaMemcpy(host_a, dev_a, arr_size, cudaMemcpyHostToDevice);
    cudaMemcpy(host_b, dev_b, arr_size, cudaMemcpyHostToDevice);

    VectorAdd<<<size / blockSize + 1, blockSize>>>(dev_a, dev_b, dev_c);

    cudaMemcpy(dev_c, host_c, arr_size, cudaMemcpyDeviceToHost);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

cuFFT because different kinds of FFTs require different thread configurations and GPU resources, and the plan interface provides a simple way of reusing configurations.[15]

In the process of plan creation, cuFFT allocates work area to make FFT operations fast. To get an idea about memory size this work area consumes, functions like cufftEstimate and cufftGetSize1D are available.

2.4.2 Complex numbers

CuFFT supports two internal data types for complex numbers. CuFFTComplex consists of two single precision floating point numbers - real and imaginary, named x and y. CuFFTDoubleComplex is the same, apart from floating point precision, which is double (64bit).

Transforms using single precision floating points are faster, but I will be using double precision integers, as I will be able to store larger parts of the limbs to the complex numbers without losing precision.

Single precision floating point can store up to 24 bit long integer without

losing precision. Double precision floating point can store up to 53 bit long integer without losing precision.

After forward transform, point-wise multiplication and inverse transform I expect results to be twice as long as the inputs - that means I can store maximum of 26 bit elements on the input. For single precision this shrinks to 12 bits. More on this topic discussed in section 3.4.1.

Chapter 3

Design

In this chapter I will discuss various architecture decisions, go through application structure and also explain used algorithms in detail. I will explain how the CUDA related code with all arithmetic functions is separated in a static library, that for the purpose of this thesis will be used in a testing application, but could be also used by any other application in the future.

3.1 Addition operation

Parallel addition of 64-bit long limbs can be in general viewed similarly as multiple 1-bit hardware adders combined together via carry propagation. For better understanding this is illustrated in figure 3.1 (picture taken from [5]).

In fact, after addition of two limbs (regardless of their size) only three scenarios can happen. Result overflows, result would overflow if carry has been added to it (also known as carry propagate) and the third option - result fits into the limb without carry or carry propagate.

If we want to detect that addition of two limbs overflows and therefore

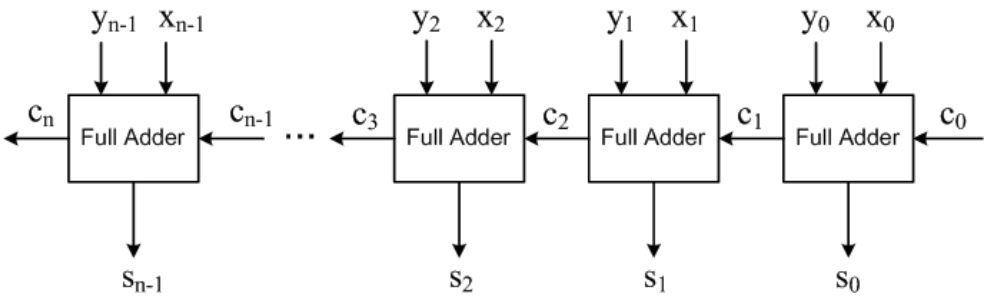


Figure 3.1: Simple hardware adder

produce carry, we only have to check that result of addition operation is smaller than one of the operands. In code that looks like this:

$$carry = (result < a)$$

Detection of carry propagate state is even easier, because limb would propagate carry only if all of its bits are set to one. In case of unsigned variables, this value is also equal to minus one. That said, carry propagate would be checked like this:

$$carry\_propagate = (result == -1)$$

In case of C implementation, we only have to make sure that -1 is the same data type as the result (uint64_t in our case). The resulting code therefore looks like this (adding llu modifier after the number constant):

$$carry\_propagate = (result == -1llu)$$

All that is left is to efficiently propagate carry to the right limbs. This to be done efficiently has to be done in parallel. I have made an assumption that carry would in real life not propagate through large number of limbs. Even though in order to make this reasonably fast I have decided to make a carry buffer to store if corresponding limb is in carry, carry_propagate or nothing state. Each limb then, on its own, detects if it should add 1 to itself. This is done by scanning the carry buffer in the right direction. While there is carry_propagate, scanning continues with the next element. If there is carry reached, add 1 and stop. If there is nothing state reached (or end of carry buffer), do nothing.

In the worst case scenario, there could be one CUDA thread scanning the whole carry buffer (n steps over char array), but profiling over randomly generated numbers didn't indicate this to be a performance problem. Carry propagation usually takes less than half the time that addition kernel does. In theory this could be sped up employing parallel prefix sum algorithm, but it seems not worth adding the complexity to the code. This possibility is in detail discussed in future work, section 6.1.

3.2 Subtraction operation

Subtraction process is very similar to addition. Only difference is, that this time instead of carry, I will propagate so called borrow. To be able to propagate borrow (subtracting one from the more significant limb), I always have to subtract smaller operand from the larger one using following formula:

$$a - b = -(b - a)$$

That means if b is larger than a, I swap the operands and then multiply the result by -1 (simply by changing the sign).

The other difference is, that borrow occurs when:

$$borrow = (result > a)$$

Borrow could be only propagated over limbs that contain value of 0:

$$borrow\_propagate = (result == 0)$$

3.3 Bit shifting operation

Again, bit shifting can be seen as a simple operation. Only performance drawback is that to avoid possible synchronization issues, this operation cannot work in-place on CUDA or any other parallel architecture. This means that for the result there have to be enough limbs allocated. This is not a problem for normal measurements as we can allocate in advance (discussed in section 3.6.2.1) but it is a problem while using it as a part of other operation - which I do for example in multiplication operation.

Apart from that issue it is quite easy. I have split this into two parts. At first, shift whole limbs (if shifting more bits than the size of one limb). Second, shift inside each limb + adding overflow to the more significant limb.

3.4 Multiplication operation

As proposed in articles [2] and [3] I am going to use Convolution theorem (explained in section 1.2.2.5) and Fast Fourier Transform (explained in section 1.2.2.1) for large integer multiplication.

3.4.1 Representing integer as polynomial

Convolution technically is a polynomial multiplication, but we want to multiply integers. To be able to do so, we need to represent our large integers as polynomials. To do so we only need to choose a base in which the number will be converted. To illustrate this process, let's say we want to represent number 12345 in base 10:

$$12345 = 1 \cdot 10^4 + 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$$

The resulting polynomial p with base $b = 10$ would then be:

$$p = b^4 + 2b^3 + 3b^2 + 4b + 5$$

We can now choose the width of polynomial base as we want it, for example with base 100:

$$12345 = 1 \cdot 100^2 + 23 \cdot 100^1 + 45 \cdot 100^0$$

And polynomial p in base $b = 100$ is:

$$p = b^2 + 23b + 45$$

As I am dealing with binary represented number and want to be able to make polynomial representation using simple operations like shifts and logical ANDs, I will choose base as a power of 2. I could actually use 64 bit limbs directly with base 2^{64} - that way I could skip this step entirely. Only drawback is that cuFFT library uses only float or double precision floating point numbers to store operands. Only way to pass arguments to cuFFT without losing precision is to encode them to mantissa. In case of double precision floating-point mantissa can handle up to 53 most significant bits (as stated in article [16]).

To make things as simple as possible I propose to choose base in a way that results in splitting limbs to parts, that means that base b must divide limb without remainder. In our case (64 bit limbs):

$$2^{64} \equiv 0 \mod b$$

Making polynomial representation this way is very straightforward and can be easily parallelized on CUDA device. Implementation details will be discussed in section 4.5.2.

3.4.2 Pointwise multiplication

The next step after converting operand polynomials into frequency domain via Fourier transform is pointwise multiplication. This is a simple step containing only multiplication of two corresponding complex numbers.

Multiplying two complex numbers ($x = a + ib$ and $y = c + id$) means computing:

$$x \cdot y = (a + ib) \cdot (c + id)$$

This can be simplified as [17]:

$$x \cdot y = ac + ibc + iad - bd$$

$$x \cdot y = (ac - bd) + i(ad + bc)$$

If we have L limbs, we divide each limb in k smaller parts (explained in section 3.4.1), we need to do only $k \cdot L$ times the above computation in the process of FFT multiplication. All those computations can be also done in parallel as they do not depend on each other - which is the main advantage of this method for parallelization.

3.4.3 Result normalization

As stated in [15]:

> "cuFFT performs un-normalized FFTs; that is, performing a forward FFT on an input data set followed by an inverse FFT on the resulting set yields data that is equal to the input, scaled by the number of elements. Scaling either transform by the reciprocal of the size of the data set is left for the user to perform as seen fit."

That means the results of forward and then inverse Fourier transform are upscaled by a number of elements used in transforms. To deal with this, I do divide each resulting limb by the number of elements. This can also be all done in parallel so it doesn't hurt performance a lot.

This should not affect precision, as the division is in fact only affecting floating point exponent, leaving mantissa intact.

3.4.4 Carry propagation

After performing result normalization there is still one step left before correct result can be extracted. As we divided limbs into smaller parts in section 3.4.1, we now have, as a result of multiplication, almost twice as big limbs in the result variable. To avoid losing data, some sort of carry propagation has to be done. This time, however, it is not zero or one bit carry. If we split limbs in 16 bit parts in the beginning, after multiplication we can have up to 32 bit numbers.

This basically leads to another addition operation. As that operation is also made in cuGMP library, I decided to reuse it. To do so, there are two result variables allocated - one for most significant bits and one for least significant bits. Each part of the result is split in two parts and those are added to the result variables.

The most significant bits variable is then shifted left by the size of one part (in this example by 16 bits then) and both variables are added together via standard addition explained in section 3.1. The process is in detail explained in chapter 4.5.5, and illustrated on Figure 4.1.

3.4.5 Multiplication process summary

To sum up the whole multiplication process:

1. Represent integers as polynomials = split limbs into smaller parts according to the base selected.

2. Perform forward Fourier transform on both polynomial-form integers.

3. Pointwise multiply transformed integers in frequency domain = multiplication of complex numbers.

4. Perform inverse Fourier transform on resulting integer to transfer it from frequency domain back to time domain - still polynomial form.

5. Normalize the result - divide by the size of Fourier transform.

6. Extract result - split normalized result variable into two standard 64-bit limb variables and add those together to propagate carry properly.

All those parts are done effectively in parallel and on CUDA device. This differs from article [3], where for polynomial representation base 10^k has been selected and polynomial representation has been done in CPU and RAM and after that it was copied to CUDA device. In the end the carry propagation has been done also in CPU to avoid concurrency problems. The scheme I have proposed relies solely on operands initialized straight in CUDA and therefore should perform better especially with larger operands.

3.4.6 Multiplication precision

There is one drawback to the method presented above and that is precision. CuFFT library has been probably designed primarily with signal processing in mind - therefore 100% precision may not be absolutely required. Internally it uses double precision floating points to store complex numbers. Those are by design prone to rounding errors. That means the result of multiplication can slightly differ from the correct result.

Unfortunately I have not been able to determine how big difference to expect, because all of my sources just mentioned that there is some error in the result, but did not specify how much of an error. My intention is to analyze this a bit more and to measure the average multiplication error based on operand size. I think it is worth knowing that number for two reasons. First - someone considering this method as good enough may thoroughly evaluate if this method is good enough. Second - anyone trying to implement similar solution may easily compare his results to mine and as a result some improvement in precision may come out.

It is actually possible to multiply large integers using Fourier transform without error in the result. Unfortunately, it requires a lot more pre-computation to be done and therefore it is probably reasonably fast only for astronomically large numbers. Secondly, it requires recursive multiplications of large integers in some cases, which adds complexity. And most importantly it requires Fourier transform to support so called number-theoretic transform. With that, Fourier transform works over finite field, using modular arithmetic and integers instead of floating point complex numbers. Those operations are then error-free.

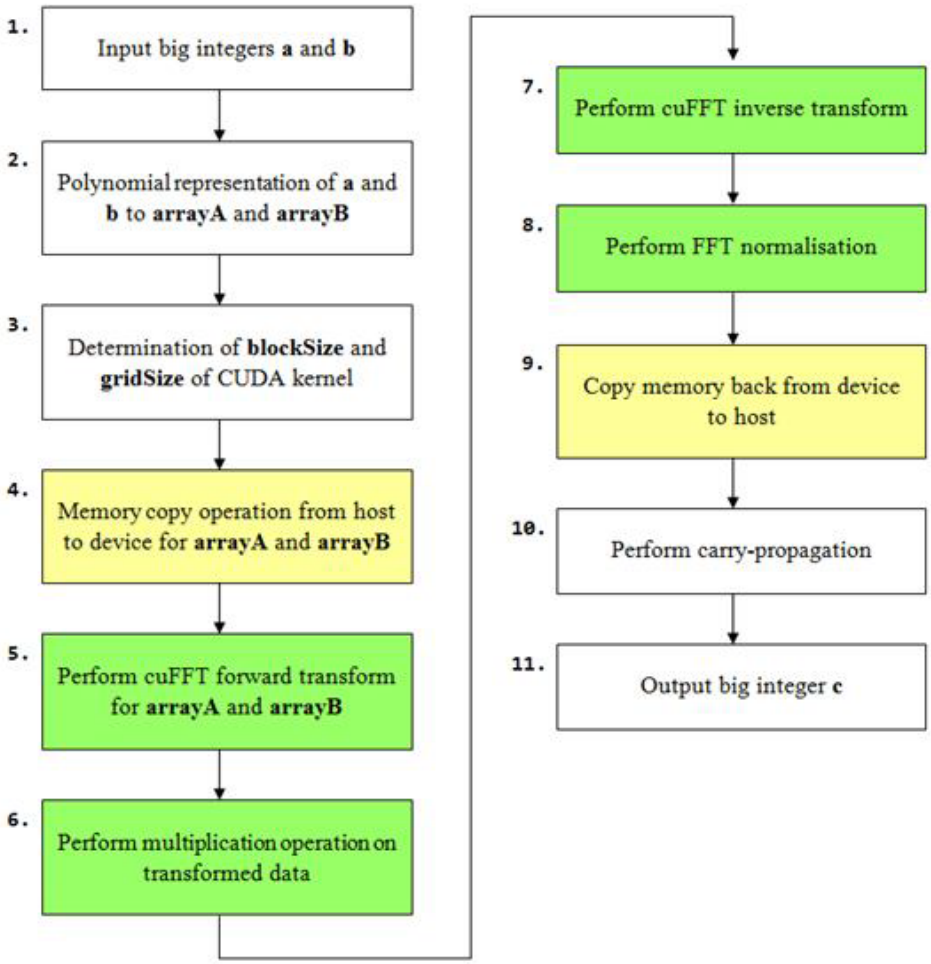


Figure 3.2: Multiplication process summary

That would unfortunately mean not using cuFFT library for fast Fourier transforms, as it can work only over complex floating point numbers. I have proposed implementation of FFT over finite fields on parallel architecture in future work chapter, section 6.3.

3.5 Multiplication - design optimizations

The multiplication process is discussed in section 3.4.5 and illustrated in Figure 3.2. The figure is taken from article [3]. In the figure, white background means host (CPU) code, yellow means memory transfer to and from device and green illustrates computation on GPU device.

My design is greatly inspired by this diagram, but I propose several optimizations to it.

At first, input and output (parsing, storing as string etc.) will not be a part of multiplication operations. This is very similar to how GMP library works - initialization, generation and loading is separate operation.

Another important difference is that operands are loaded into device memory right after initialization (parsing from string). Therefore, memory operations are avoided and all operations are done right inside the GPU. This is exceptionally useful in multiplication example as in my design, there are some immediate results, that need to be added together (carry propagation discussed in section 3.4.4). These big numbers never copy from device to host, because they are already in the device. Only thing that is copied to host is the pointer to operand data (limbs) and the size of the number (count of limbs).

To sum up - memory operations are avoided and polynomial representation and carry propagation is also done in the device, in parallel, using one kernel each.

3.6 Application structure

There are two main parts of my work in this project - large integer arithmetic functions implemented in CUDA framework and with those functions there is a lot of measurement, testing and evaluation. To leave the possibility to use the main arithmetic functions in a different way and to let others to extend this part of my work I have two main projects in my solution.

First and most important is called cuGMP. It is a static C++ library, contains only CUDA source files (*.cu), depends solely on nvcc compiler to create cuGMP.lib file. This project depends heavily on CUDA framework using it's header files, dynamic libraries etc. All CUDA kernels, CUDA API calls, memory management (for both CUDA device and host storage of large numbers) and integer initialization are implemented here. Idea is that to make a simple application that uses all this should be as easy as possible, ideally without many things to configure in a C++ project.

Second part is a simple console application called TestApp. Main responsibility of this application is to evaluate both libraries against each other regarding result correctness and computation time (eg. performance). Other purpose is to demonstrate how to use cuGMP library.

3.6.1 Storing the large integer in memory

For storing large integers in memory I've adopted a concept of so called limbs, used also throughout the GMP library. One limb is just a normal unsigned integer - 32 or 64 bits long.

On modern CPUs there is no reason not to use the bigger, 64-bit, integer to store a limb. On CUDA on the other hand, there might be, because

CUDA does not support native 64-bit integer arithmetic. Operations with 64-bit operands are actually split in several instructions on 32-bit parts in the compilation process. Yet I have chosen to use 64-bits for each limb in cuGMP.

This is an arguable decision since it might hurt performance and performance is a main goal in cuGMP. On the other hand it allow me to address larger operands and I do believe that this won't hurt performance at all because even though every operation splits into a bit more instructions in the assembly, these are all done in parallel without collisions. The main arithmetic operation with two limbs is actually expected to be just a small portion of the overall computation. For example in addition operation I would have to deal with twice as many possible overflows, I would have to allocate considerably more memory for FFT calculations in multiplication operation etc.

To evaluate how much time is spent with the main operations (kernels), I have used NSIGHT profiler that shows exactly (in fractions of micro seconds) how long it took to launch each kernel and CUDA API call. Measurements confirm that kernels with the main arithmetic operation are usually not the performance bottleneck. Exact results are discussed in section 5.4.1 for addition operation.

3.6.1.1 64-bit integers on Windows

As stated in [18] Microsoft Windows operating system and its C++ library uses 32-bit large long. This does not go against C language specification as it says that long is at least 32-bit long. On the other hand, most of the other 64bit operating systems like Linux use 64-bit longs. One solution would be to use unsigned long longs, as it's an integer that's at least 64 bits long (stated in C standard [19]).

For better compatibility across variety of systems, I have decided to use uint64_t, that requires including stdint.h header file. This, on Windows, uses unsigned long longs anyway, but should ease the process of porting this library to other operating systems in the future.

This differs from GMP library as it uses plain unsigned longs. This is one of the reasons GMP is not supported on Windows very much. It is also a reason why MPIR (Multiple Precision Integers and Rationals) was created as a fork of GMP. For testing purposes on Windows, I will use MPIR. On Linux I will use GMP. Both libraries are interchangeable as MPIR exposes almost identical gmp.h header file and also almost exactly the same implementation. For performance comparison I will use GMP running on Linux system.

3.6.2 Data types

For storing large integers into limbs I've adopted existing C-style structure from GMP library called **__mpz_struct**. It consists of two integers and one array pointer, that point to an array of limbs. All items are:

- _mp_alloc - number of limbs allocated in memory already
- _mp_size - number of limbs actually used
- _mp_d - pointer to array of limbs

If the _mp_size is negative, the whole number is negative. All limbs are unsigned and store an absolute value of the number.

The problem I had to face was that to compute on CUDA, I have to copy all limbs to CUDA. At first I wanted to store limbs only on CUDA device, but first I had to parse the numbers from string and copy it to CUDA limb by limb. It then turned out that calling cudaMemcpy repeatedly on small amount of data is very slow. To avoid this issue I have decided to split operand storage in two separate structs - keeping pointer to the device structure in the host __mpz_struct.

This allowed me to simplify the process of initializing numbers, reading results and copying data to and from CUDA device. Now when number gets initialized on a host it all happens in host memory and then only once cudaMemcpy is called to copy all limbs to CUDA device. Second cudaMemcpy is then required to copy details about operand size to the device and most importantly to copy pointer to those limbs to the device.

As a result we have two structs:

Source code 4 __dev_mpz_struct

```
typedef struct
{
    int _mp_alloc;
    int _mp_size;
    mp_limb_t *_mp_d;
} __dev_mpz_struct;
```

Source code 5 __mpz_struct

```
typedef struct
{
    int _mp_alloc;
    int _mp_size;
    mp_limb_t *_mp_d;
    __dev_mpz_struct *_dev_mp_struct;
} __mpz_struct;
```

The device structure (Source code 4) is exactly the same as in GMP library. The host structure (Source code 5) has been extended by a pointer to

__dev_mpz_struct. This allows programmers using cuGMP to work only with one pointer to __mpz_struct as an operand, all else (eg. copying data to CUDA device) is hidden inside cuGMP logic.

3.6.2.1 Initialization

All mpz_t type variables has to be initialized before usage. There are two main options of doing that - both are in detail discussed in GMP manual [20].

Most common option is using mpz_init(mpz_ptr x) function. This function initializes variable of type __mpz_struct to size 0 and allocates one limb.

Second option is to use mpz_init2(mpz_ptr x, mp_bitcnt_t bits) which can allocate larger amount of limbs straight away. This can be helpful to eliminate allocation from performance measurement.

The same set of functions is implemented in cuGMP library - only significant difference is that in cuGMP there is also immediate allocation on CUDA device.

In cuGMP library there are two additional functions to initialize CUDA device. First is called cudaInit() and it calls cudaSetDevice(0) from CUDA API. This is to be changed on systems with multiple GPUs. Second function here is called cudaReset(). It is optional to use this function, but it ensures that CUDA clears all state variables after computation, therefore it is guaranteed not to affect following computations.

All those functions are implemented in file mpz/init.cu.

3.6.2.2 Set operand from string

To accurately test that cuGMP library produce correct results I have implemented a set of string functions.

At first there is mpz_set_str(mpz_ptr x, const char *str, int base) - again the definition is taken directly from GMP library. Purpose of this function is to take a number represented by string in some base and convert it to __mpz_struct. In GMP this function accepts all bases in the range from 2 to 62. In cuGMP I have decided to support only binary and hexa-decimal bases. The reason for this is simple - strings in those two common bases can be easily divided into the right sized parts that fit exactly one limb. This means that the whole operation of setting operand from string is to divide an input string into parts and then use standard functions to convert string to integer.

I originally wanted to support base 10 since it is a bit more human-readable. I have checked source code of the GMP library and it is quite obvious that it would mean first implementing several arithmetic operations before being able to set operand from string. I then considered calling existing functions in GMP library directly, but that would mean the new library would be entirely dependent on GMP and it would also mean a great deal of trouble trying to expose the same interface as GMP because of the naming conflicts.

In the end it seems that for our purposes base 2 and 16 are more than enough - usage of this functionality is discussed in section 5.2.3.1.

3.6.2.3 Get string from operand

As an inverse to set string (section 3.6.2.2) there is also mpz_get_str(char * str, const int base, const mpz_ptr x). This function converts existing operand back to a string representation. Again, supported bases are 2 and 16. This is also heavily used for testing purposes as it means we can check the result of computation.

3.6.3 Memory management

As working with memory is a common place to make programming mistakes, I propose an unified way of working with memory through a simple set of functions.

First, there is the function *allocate_memory*, that is responsible for allocating limbs to the existing operand (initialized in section 3.6.2.1). What is does is check if there is enough memory allocated based on the requested size in the parameter. If not, existing limbs are freed and new are allocated, both in host and the device memory. Both operand structures are updated (value of _mp_alloc and _mp_size) in host and in device. If sufficient memory is already allocated, nothing happens.

There is also a function *copy_operand_data_without_limbs* that unifies copying of operand (type mpz_ptr) from and to device. It's main use is to propagate changes in size and number of limbs allocated. For example, when result size is determined on the device, operand needs to be copied back to the host after. It accepts only two arguments - mpz_ptr and the direction - to or from device. Elegance of this is, that it is always called from host and in mpz_ptr there is a pointer to the device store of the operand. This is a difference to the GMP library, but it is only extension, all the things from GMP remains untouched. It also does not have to be used outside the cuGMP library.

Similar function is *copy_operand_data_with_limbs*, accepting the same arguments. Only difference is, that this function also copies limbs. It is usually used during operand parsing or converting result back to the string representation (operations explained in section 3.6.2.2).

The primary storage for all limbs is the CUDA device. Host storage is usually used in set and get string phase. This way the overall performance is improved as all calculations are primarily done on CUDA, without the need to transfer limbs back and forth.

From the programmer point of view, this saves a lot of programming errors and mistakes as it seems to me that this is a common place to make mistakes with pointers and references, not remembering to copy some properties etc.

Last thing that there is about memory management is freeing up the memory allocated. This is also done in a standard GMP way. That means, utilizing *mpz_clear* frees up host and device limbs as well as host and device descriptors. For every *mpz_init*, there should be one *mpz_clear*.

3.6.4 Library interface

To provide interface familiar to current GMP users I exposed the main arithmetic functions in cuGMP.h header file with mostly the same function definitions as GMP library exposes through gmp.h header file. This ensures the possibility to use exactly the same test program on both libraries and then see the performance differences without any doubt. On the other hand, implementing all the functionality of GMP library certainly isn't in the scope of this project - some limitations might occur.

3.6.5 Example usage - TestApp

To test the library, make performance measurements and to demonstrate how to use cuGMP library I have implemented TestApp console application. Detailed structure and implementation characteristics are discussed in results chapter, section 5.2.1. This application provides an easy and automated way to process large batches of performance tests storing the results in unified way similar to CSV file format. This enables us to tweak various parameters like CUDA block size, multiplication limb size the best possible way. This is also needed to provide representable data for performance comparison discussed in chapter 5.4.

Chapter 4

Realization

In this chapter I will discuss implementation details, optimizations and overview of the cuFFT library usage. I will also describe porting this project from Visual Studio on Windows platform to Makefile on Linux platform.

The main functionality is implemented in a static C++ library called cuGMP as an acronym for CUDA and GMP. This library contains only *.cu files (discussed in chapter 2). There is also a console application using this library to make all the testing and measurement, that is called TestApp.

4.1 Limitations of cuGMP

As GMP supports a huge number of public functions and even larger amount of internal procedures, cuGMP is very young and does not have many functions implemented yet. For example it does not support random number generation, so in order to test it against GMP, the usual scenario is to generate random number in GMP, convert it to a string representation and then transfer it to CUDA device.

Initialization of large integers in cuGMP is also a bit limited. As support for all meaningful bases in which integers can be expressed as a string would require a handful of arithmetic functions I have decided to support only binary and hexadecimal bases - the reasoning is explained in section 3.6.2.2. On the other hand, as a way of validating results and transferring operands from one library to the other this proved to be sufficient.

4.2 Get and set string

This section provides implementation details about functions, that are responsible for conversions between numbers in string representation and internal representation stored in limbs.

First there is mpz_set_str(mpz_ptr x, const char *str, int base). It works simply by splitting the input string into parts, that correspond to one limb. For base 2, as 1 character represents one bit, it takes 64 characters from input string to initialize one limb. In base 16, one character represents 4 bits, so one limb corresponds to 16 characters. With those strings, I utilize standard C function to convert from string to unsigned 64 bit integer - strtoull.

Apart from that there is memory allocation to ensure there is enough space allocated to write those limbs.

The second function, mpz_get_str(char *str, const int base, const mpz_ptr x), works in similar way. Operand is copied from device to host. For each limb _ui64toa is called to convert 64 bit integer to string. Those strings are then concatenated and returned as a result.

One difficulty, that I had to deal with - string returned by _ui64toa are not always the same length as they do not contain the leading zeroes. To deal with that I start by allocating the result string with all characters set to '0'. Destination of the converted limb is computed based on string length. As a last step, leading zeroes are stripped simply by copying the string to the left. I have verified that this procedure returns exactly the same strings as GMP does - this testing is implemented in TestApp - section 5.2.3.2.

4.2.1 Optimization - get and set string

At first I wanted to have CUDA global memory area as the only storage for all the limbs, as all calculations take place in the device anyway. This lead to repeated cudaMemcpy API call for every limb that got parsed. I have examined results from NSIGHT profiler and discovered, that repeated call to copy small block of memory (8 bytes in this particular case) is significantly slower than one call to copy the same memory all at once.

To shorten the time to transfer operands from GMP to cuGMP in TestApp, I have added another limb storage to the host side. This increases memory consumption a little bit (not drastically compared to the string representation of the numbers), but speeds up set string operation significantly (approximately 5 times).

4.3 Addition operation

Addition is pretty straightforward process, explained in detail in design 3.1.

AddKernel does the hard work adding corresponding limbs together, detecting overflow and carry propagate states. Those states are then written to carry buffer (array of unsigned chars).

CarryPropagateKernel then examines this carry buffer. Each thread starts at its corresponding limb and ends execution when carry is propagated or *nothing* state is reached.

Result of addition can have the same amount of limbs as the larger of the addition operands, or it can have one limb more. *CalculateResultSize* is used to decide about the result size. It has to be decided inside a CUDA kernel, because operand is in the device memory. This kernel is executed on one thread only.

4.3.1 Optimization - memory allocation

As for the carry buffer, there is cudaMalloc once called. For smaller operands, this takes more time than the actual computation (approximately hundreds of micro seconds). This does not affect larger operands much.

Possible optimization would be to avoid this buffer at all. Each thread would then, inside carryPropagationKernel, have to compute the state on its own. This, being operation done on 64 bit integers, could be relatively slow.

As this does not affect larger operands and provides only constant time improvement, I have found it not promising enough to implement. I have mentioned this idea in future work chapter, section 6.1.1.

4.4 Subtraction operation

In order to detect which operand is larger, I had to implement a simple kernel, that is always run in single thread only, called *subSwapOperandsKernel*. The result is stored in device variable and then copied to the host using cudaMemcpyFromSymbol - this way I have avoided one cudaMemcpy and cudaFree API call.

Result size (number of active limbs) cannot be guessed as in case of addition operation. To compute the result size in parallel fashion, I have used CUDA function atomicMax inside calculateSubResultSizeKernel. Each thread looks if its own limb has a value greater than zero and also if value of the more significant limb is zero.

If such thing occur, it then stores the index of, presumably, the first empty limb to the result operand size. This scenario, however, can also occur in the middle of active limbs. The value I want to store is the maximum of all those indexes detected. AtomicMax function guarantees that no race condition will occur during this process and the maximum of the two values will be set atomically.

Downside is, that atomic functions were introduced in compute capability 2.0. That means, subtraction will not work on early CUDA cards with compute capability less then 2.0.

4.5 Multiplication operation

There is a number of variables and memory allocations going on inside *mpz_mul* operation. First, there are arrays that represent integer as polynomial - those are of cufftDoubleComplex type. As each limb splits into four parts and each part consist effectively of two doubles, we end up with considerable memory footprint. For each 64 bit limb, there are at least 512 bits of data. This increases even further as all operands have to be padded to the same size as the result - which is usually around double the size of operands. And we have 3 of those arrays - one for each operand and one for multiplication result.

As an estimation, lets consider two 1MB operands. We have 3 arrays, 2 times the size of the operands, using 2 times bigger data type, each limb splitting in 4 parts.

$$size = op\_size \cdot 3 \cdot 2 \cdot 2 \cdot 4$$

For 1MB operand, we get 48MB only in FFT buffers.

Another 3 allocations are used for partial results (*res_msb* and *res_lsb*, 4.5.5). These are all the same size as the result - in the above example - $3 \cdot 2$MB. Another memory is consumed by cuFFT plan.

All and all we can expect memory consumption of around 60-100 times the size of one operand. This corresponds nicely with my test results as the largest operands I have been able to process on GPU with 2GB of memory were 16MB each. That should take around a 1000-1500MB.

This memory consumption is in my opinion unavoidable. There are different cuFFT transport types, that use real instead of complex data types on the input, but they also produce complex results. As the transformation is done in-place, this does not provide much benefit.

4.5.1 Using cuFFT library

CuFFT library has been optimized for repeated transforms of the same size. It is therefore configured using so called plans. Plan has to be created before any cufftExec is invoked. Plan contains information about transform size (number of elements) and type (single or double precision etc.). Calling *cufftPlan1d* initialize some internal cuFFT memory structures. Plan has to be destroyed after all computations are done - using *cufftDestroy*.

To transform operands from time to frequency domain there is *cufftExecZ2Z* function. Z2Z suffix means it is used with cufftDoubleComplex input. If input and output pointers are the same, cuFFT performs the transform in-place without the need of additional memory - and I do it this way to allow processing of larger operands.

After computation in frequency domain has been completed, I call *cufftExecZ2Z* once more, but this time with CUFFT_INVERSE direction parameter instead of CUFFT_FORWARD.

As we can see, using cuFFT API is not very complicated.

4.5.1.1 Optimization - cuFFT transform size

As stated in [8]:

> "The cuFFT library has highly optimized kernels for transforms whose dimensions have these prime factors. In general the best performance occurs when using powers of 2, followed by powers of 3, then 5, 7."

To be able to exploit this optimization I rounded transform size to the next bigger power of 2. This can increase memory consumption - in worst case scenario to almost double the size required without this optimization. But then this is not only performance, but also precision optimization as the documentation in [8] continues:

> "When the length cannot be decomposed as multiples of powers of primes from 2 to 127, Bluestein's algorithm is used. Since the Bluestein implementation requires more computations per output point than the Cooley-Tukey implementation, the accuracy of the Cooley-Tukey algorithm is better. The pure Cooley-Tukey implementation has excellent accuracy, with the relative error growing proportionally to $\log_2(N)$, where N is the transform size in points."

That means not using this optimization would affect both performance and precision for some operand sizes.

4.5.1.2 Optimization - cuFFT initialization

Due to the fact that I have used memory leak detector regularly during development (not only cuda-memcheck, but also host memory leak detector - Visual Leak Detector in my case), I have detected a strange behavior of cuFFT initialization. The library gets initialized first time an API call is made. As stated in [21]:

> "The cuFFT library is initialized upon the first invocation of an API function, and cuFFT shuts down automatically when all user-created FFT plans are destroyed."

The first call to cuFFT API (usually during plan creation - cufftPlan1d) with memory leak detector took around 10 seconds. Without memory leak detector it took around 500ms. This still is significant amount of time.

I have exposed a simple function, cuFFT_Init, accepting no parameters. It creates cuFFT plan and immediately destroys it. My measurements suggest, that cuFFT documentation [21] is not accurate in this. From the documentation I would assume, that after destroying the plan, cuFFT would shut down. Even though, when I execute measurement after creating and destroying cuFFT plan, I get better results (approximately by 500ms).

I believe that this corresponds to initialization as discussed in section 2.3. After first cuFFT API call, the library probably gets JIT compiled and loaded into the device. By initializing it ahead of measurement, I get more stable results.

4.5.2 Polynomial representation

To extract polynomial representation from limbs directly, without much computation, I do only splitting of the limbs to smaller parts.

This is done using only logical ANDs and SHIFTs inside polynomialRepresentationKernel. There is a parameter defined, called MUL_BASE_BITS, that controls the size of resulting parts. To experiment with this parameter, it is sufficient to change only the define value. Supported values are 2^k for k in range $< 1, 5 >$, that means 32, 16, 8, 4, 2.

Each thread is responsible for one source limb to be split in parts. Let the MUL_BASE_BITS be set to 16 bits. Each thread then takes mask of 0xFFFF, shifts this mask to the left to the appropriate part, ANDs it with the source limb and the result is then shifted back to the right. Resulting value is set to the x coordinate of cufftDoubleComplex type. The y coordinate is set to zero as this number does not have an imaginary part in this step. This is repeated 4 times to cover all 64 bits of the source limb.

4.5.3 Pointwise multiplication

Pointwise multiplication of complex values, that are stored in Cartesian coordinates (which is the way cuFFT stores complex numbers) is done in the way proposed in paper [3], chapter VI. Multiplication of cufftComplex values a and b goes like this:

$$c.x = a.x \cdot b.x - a.y \cdot b.y;$$

$$c.y = a.x \cdot b.y + a.y \cdot b.x;$$

This is done for each pair of values in parallel in pointwiseMultiplicationKernel.

4.5.4 Result normalization

As discussed in section 3.4.3, I need to divide result by the size of the transform. Again, this is a very simple kernel, called resultNormalisationKernel. Each resulting complex number is divided by the size of the transform.

In fact, only the x coordinate is divided, as the y coordinate is not used in the result extraction.

4.5.5 Result extraction

In *extractResultKernel*, double precision floating point is converted into 64 bit unsigned integer limb. If no rounding errors occur during FFT transform and pointwise multiplication, the floating point contains whole-number. As rounding error do occur, I have used *llrint* function to round the result to the nearest 64 bit integer.

After that, the integer is split into two parts - least significant bits are assigned to appropriate limb in *res_lsb*, most significant bits are first shifted right and then assigned to *res_msb*. This shift right is a bit confusing, but I do that to avoid race conditions and data dependencies - this way, lsb and msb parts end up in the same limb index of res_lsb and res_msb.

To balance out the shift right from previous step, I then have to call *mpz_mul_2exp* on *res_msb* to shift the whole operand back 16 bits left. After that, both msb and lsb parts can be added together using *mpz_add* implemented also in cuGMP. The process of extracting IFFT result to msb and lsb parts is illustrated in Figure 4.1 for better understanding.

4.6 Bit shifting operation

Bit shifting is done in two GMP functions that correspond to the standard C bit shifts, but are called **_mul_** and **_div_** as bit shifts can be also seen as multiplication and division. They both end with *2exp* suffix to indicate it is multiplication/division by 2^k where $k \in N$.

First there is *mpz_mul_2exp*, which is in fact shift left by *exponent* bits. First I calculate the number of whole limbs to shift - simply by dividing $exponent/GMP\_LIMB\_BITS$. Then the number of bits to shift inside each limb - this time it is not division, but modulo - $exponent\%GMP\_LIMB\_BITS$.

Then there is only one simple kernel called *shiftLeftKernel*. Inside, new mapping is composed - first *shift_limbs* least significant limbs are filled with zeroes, others are generally filled partly from the lower bits from the corresponding limb (shifted left) + upper bits from the less significant limb (shifted right). I have illustrated this process in Figure 4.2. In the illustration, limbs are only 16 bits wide and we are shifting 24 bits - so it's 1 limb + 8 bits shift.

For shifting right (division by the power of two) I have implemented *mpz_tdiv_q_2exp*. This function works in similar fashion as the previous one,

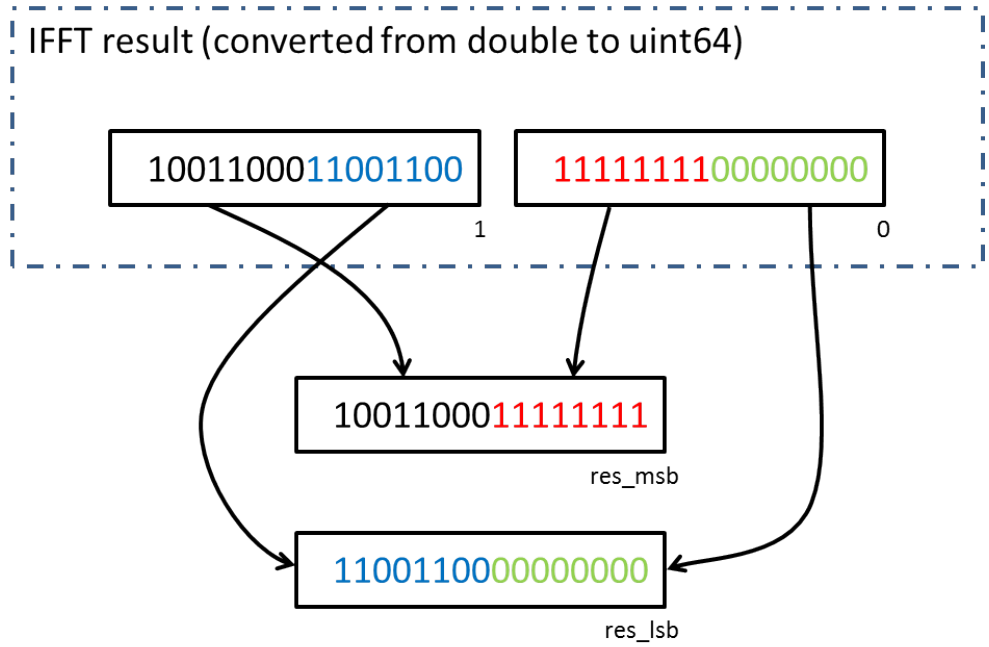


Figure 4.1: Extract IFFT result to msb and lsb operands

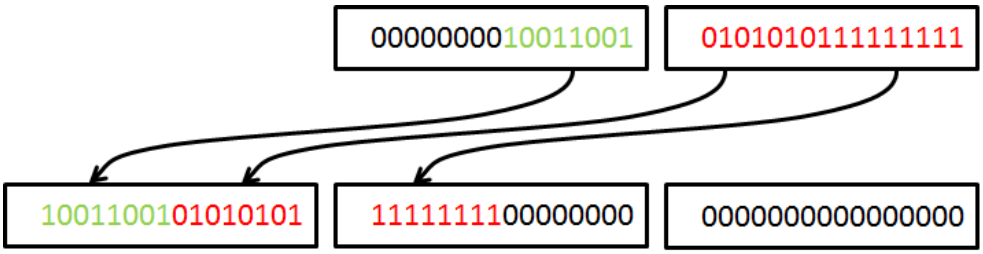


Figure 4.2: Shift left by 24 bits example

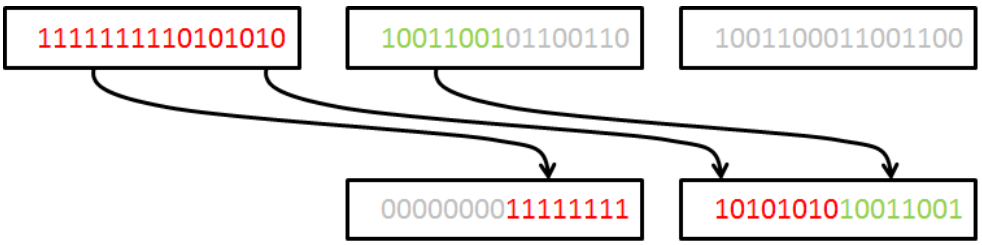


Figure 4.3: Shift right by 24 bits example

only in the opposite direction. Instead of padding bits with zeroes, this time we are truncating *exponent* least significant bits. 24 bit shift right with 16 bit limbs is illustrated in Figure 4.3.

4.7 Logical operations

Implementation of AND, OR and XOR operations is very straightforward. It means only to do the logical operation in question between corresponding limbs. That sounds very much like addition, but in this case, there is no carry propagation.

AND is implemented in *mpz_and* function. As ANDing with zero is still zero, andKernel is executed only on limbs that exist in both operands. If one operand is larger, the exceeding part is ignored completely. Size of the result is the minimum of the sizes of input operands.

That said, ANDing two operands of 1000 limbs creates 1000 threads that all have to do one logical AND between two uint64 numbers and nothing more really.

OR is implemented in *mpz_ior*. This time, as ORing with zero is not always zero, there is one more if-statement inside the orKernel. If one operand is larger than the other, exceeding limbs are simply copied from the larger operand.

If-statement does not introduce much divergence either as all thread-blocks will be executing the same branch of code and at most one thread-block will diverge.

XOR is implemented in *mpz_xor*. Only difference between OR and XOR kernel is the bit-wise operation used inside.

That said I do not see much room for performance improvements.

4.8 Implemented operations

GMP library divides operations in two main parts - MPZ are high-level operations intended to be used outside the library, they do take care of memory

management etc. Then there are low-level MPN functions, used to implement the high-level GMP functions, but also intended for time-critical user code.

I have primarily focused on MPZ functions to provide the whole arithmetic operation. To sum up all GMP operations implemented:

- mpz_add - addition
- mpz_sub - subtraction
- mpz_mul - multiplication
- mpz_mul_2exp - shift left
- mpz_tdiv_q_2exp - shift right
- mpz_and - logical AND
- mpz_ior - logical OR
- mpz_xor - logical XOR
- mpz_init - operand initialization
- mpz_init2 - operand initialization and limbs allocation
- mpz_clear - operand memory cleanup
- mpz_get_str - convert operand to its string representation
- mpz_set_str - initialize operand from its string representation

4.9 Porting to linux

To be able to measure performance on our university GPU cluster called STAR, I had to adapt the project to be able to work on Linux OS. Performance measurement on Linux is discussed in section 5.1.1.

Apart from that, on several places, I had to include cstdlib and cstring headers to support various functions, that on Linux worked differently or had different name and definition. I had to rename and, or and xor operations in my Operation enum, because in g++ compiler, these apparently are some keywords, that cannot be used in various contexts.

I also had to re-implement _ui64toa function that, on Windows C++ library, can convert integer to its string representation in any given base. I have not been able to find any suitable substitution in the Linux libc.

4.9.1 Makefile

After that all I had to do was to build the library and TestApp. To do that I have implemented simple Makefile, that uses regular expression to find all the necessary source files (only in valid paths), builds each one into an object file (using nvcc for .cu files and g++ for .cpp files). In the end it links all of them together.

It also supports the standard clean operation (removes all built files and object files) and it can also build only cuGMP library without TestApp.

Sample usage:

make - builds cuGMP and TestApp in testApp binary
make cuGMP.a - builds only cuGMP using nvcc
make clean - removes all object files and built files

Chapter 5

Results

In this chapter I will present testing methodology, describe hardware configuration used for performance measurement and also provide measured data and graphs of performance speedup against GMP library.

5.1 Performance measurement

To accurately measure computational time I have implemented a simple, but platform dependent, set of functions. It uses thoughts described in detail in article [22]. It requires to include windows.h header file for QueryPerformanceCounter and QueryPerformanceFrequency API calls.

The performance measurement functionality exposes simple functions to start/stop measurement and to compute elapsed micro-seconds, mili-seconds and seconds. It uses a simple struct (called __measurement) to store results. It then allows me to measure computational time in micro-second precision, which wouldn't be possible without OS API calls.

Example usage of this functionality is shown in Source code 6. To be more accurate in the results I usually do repeat the same calculation several times, using average computational time as a result. Example of this is shown in Source code 7. Measurement functions actually accept number of repeats as a parameter.

Source code 6 Example measurement

```
__measurement measurement;
start_measurement(&measurement);
<DO THE EXPENSIVE WORK HERE>
stop_measurement(&measurement);
return get_measurement_microseconds(&measurement);
```

Source code 7 Example measurement - repeats

```
__measurement measurement;
start_measurement(&measurement);
for (size_t i = 0; i < REPEATS; i++)
{
    <DO THE EXPENSIVE WORK HERE>
}
stop_measurement(&measurement);
return get_measurement_microseconds(&measurement, REPEATS);
```

5.1.1 Performance measurement on Linux

In later part of the project I have decided to add support for Linux OS. I have therefore followed suggestions from article [22] and added Linux support to performance measurement.

If the compiler does not define _WIN32 (that means, we are not on Windows), our start and stop measurement functions use gettimeofday function instead of QueryPerformanceCounter. It also includes sys/time.h instead of windows.h header file. To store the time values, there is also different timeval struct instead of LARGE_INTEGER struct on Windows.

In the end however, on both platforms the same API is exposed and on both systems, the measurement is therefore done in the same way. The API returns number of micro seconds as uint64_t which is universal type for 64bit unsigned integer.

5.2 Testing methodology

In order to rightfully compare performance with GMP library I've devised a standardized methodology to call arithmetic functions in each library. The main problem that I had to address was that both libraries do number initialization and memory allocation differently. Reason is that some features of GMP library are out of scope of this project - all those limitations are discussed in detail in section 4.1.

5.2.1 TestApp

In order to make performance measurement I implemented console application called TestApp. This application uses two libraries - GMP and cuGMP.

It consist of a few files. TestApp.cpp is responsible for argument parsing or batch file loading and parsing and then executing appropriate tests. Tests are defined in Tests.cpp - those test consist of calling both GMP test and cuGMP test. Here, result evaluation and output is taken care of. Structures for passing

test results are defined in Tests.h header file. Then there are TestGMP.cpp and TestCuGMP.cpp files with all library tests.

5.2.2 Library test

Each combined test (explained in section 5.2.3) consist of two separate library tests that are structurally very similar but one use GMP library and the other use cuGMP library.

5.2.2.1 Random number generation

This part of library test is specific to GMP tests because cuGMP currently does not support random number generation. Random numbers used in GMP calculation are then transferred to the second test that use cuGMP library. Operand transfers are discussed later in section 5.2.3.1.

This is done this way to ensure that if performance results would depend on argument properties (which it can as described in section 3.1), both libraries use the same numbers and therefore differences in this matter will appear in test results.

5.2.2.2 Variables

Variables (usually of type mpz_t - explained in section 3.6.2) are named a, b and c meaning:

c = a (operation) b

This means that c is used to store the result and a and b are computation operands.

All operands have to be initialized before usage (initialization explained in section 3.6.2.1). For this I sometimes use mpz_init2 function for result variable, because it can allocate appropriate amount of memory in advance - that way I can eliminate influence of result re-allocation inside arithmetic functions. That occurs only if result operand is initialized with not enough memory allocated.

5.2.2.3 Repeated calculation

To have more accurate performance results even for operations taking not much time to complete I do often repeat the same calculation several times, dividing the measured time after. Immediate results show that there is not much more precision in the results using this method, so I have not used this functionality in the end.

5.2.2.4 Memory cleanup

All test have to clean allocated memory after themselves to allow stable measurement in batches. GMP and cuGMP operands both use standard function mpz_clear(mpz_ptr x) that frees all allocated memory. CuGMP implementation ensures correct cleanup in both host and device memory via standard free() and cudaFree() functions.

To ensure that there is no performance dependency on previous tests I also call cudaDeviceReset() after each test. That CUDA API call ensures that CUDA device clears all state, so it should be the same as restarting the program on freshly booted machine.

5.2.2.5 Return values

All tests return number of micro-seconds that one call to the tested operation took. This does not include operand initialization, random number generation, string manipulations etc. Main focus is to compare the main computation algorithms.

5.2.3 Combined test

All complete tests are placed in Tests.cpp file. In related header file, there is also a struct to contain test result and another for test setup to support batch processing and to be able to manage test output from one place.

Combined test usually ensures that GMP test is run first. Inside GMP test there are random numbers generated for operands. Operation is then executed with those operands. Both operands and the result is then extracted in the string representation for following evaluation.

5.2.3.1 Random number generation

Random numbers for the tests are generated using GMP library. The random number state is seeded with current time + test iterator. The iteratir is added for the case that the test is faster then current system time changes value.

To provide error checking and fair comparison, cuGMP uses the same numbers that GMP generated. To do so, operands used in GMP are transfered in string representation in base 16 utilizing mpz_get_str in GMP and mpz_set_str in cuGMP library.

5.2.3.2 Compare results

As both libraries use the same operands in TestApp, I am able to validate results of both libraries against each other. To do so, results from both libraries are extracted using mpz_get_str function. Results are then compared character by character to check if they match exactly - this way it is ensured that cuGMP behaves exactly the same as GMP.

In case of multiplication, however, there is some rounding error (explained in section 3.4.6) that means the result strings often do not match. To measure the size of the rounding error, I use GMP again and divide the cuGMP result by GMP result. Ideally, when no rounding error occurs, the result is 1.

All arithmetic operations implemented in cuGMP library (except multiplication) return absolutely the same results as GMP library. Measurement would fail if they would not. Size of the rounding error in the result of multiplication will be discussed in section 5.5.

5.3 Hardware configurations

As a test setup I have used a computer running Windows 10 x64 with the following HW configuration:

- CPU: Intel Core i3-4330 @ 3.50 GHz
- GPU: NVIDIA GeForce GTX 750 Ti
  - GPU Frequency: 1059 MHz
  - Memory frequency: 1350 MHz
  - Memory: 2048 MB (GDDR5)
  - Bus width: 128 bit
  - Theoretical memory bandwidth: 86.4 GB/s
  - CUDA cores: 640
- RAM: 8 GB (DDR3, 1600 MHz)

On top of that I have measured performance on university cluster called STAR. I have made measurements on two nodes, gpu-01 and gpu-02, both running x64 Linux, with the following configuration for **gpu-01**:

- CPU: 2x 6core Xeon 2620 v2 @ 2.10 GHz
- GPU: NVIDIA Tesla K40c
  - GPU frequency: 875 MHz
  - Memory frequency: 3 GHz
  - Bus width: 384 bit
  - Theoretical memory bandwidth: 288 GB/s
  - CUDA cores: 2880
- RAM: 32 GB

And for **gpu-02**:

- CPU: 2x 6core Xeon 2620 v2 @ 2.10 GHz
- GPU: NVIDIA GeForce GTX 780 Ti

  GPU frequency: 928 MHz

  Memory frequency: 3,5 GHz

  Bus width: 384 bit

  Theoretical memory bandwidth: 336 GB/s

  CUDA cores: 2880
- RAM: 32 GB

5.3.1 Compiler settings

Compilers on both Windows and Linux systems use mostly the default settings. Nvcc utilizes *–machine 64* switch to force the code to be 64bit. No debug switches are activated. For compute capability, *sm_30,compute_30* is specified to support grids larger than 65536 blocks and to support atomicMax.

Compiler settings can be changed in Makefile and in Visual Studio project settings.

5.4 Performance comparison

In the following section I will compare performance in terms of speedup versus operand size. All operand sizes will be registered as a number of bits. To get a number of limbs, simply divide by 64. To compare with GMP I will take measurements from gpu-01 node as a base. It uses Linux GMP library installed on the system.

Most measurements will consider operand sizes in range from 2^{20} to 2^{32} (or 1mbit to 4gbit). Smaller operands are not very interesting as they usually take less than 1ms to compute on both CPU and GPU. Larger operands on the other hand do not fit in available memory. Multiplication, with its FFT transforms, consumes even more memory, therefore only operands of size 2^{27} and smaller were compared. Memory consumption of multiplication operation is discussed in detail in section 4.5.

5.4.1 Addition operation

As we can see in Figure 5.1, real speedup comes when using operands larger than 2^{25}bits. This might be due to the memory allocation needed for my implementation of carry buffer discussed in section 4.3.1.

To determine why small operands take more time to compute (compared to GMP library) I have profiled addition operation with 2^{20} bit large operands using NSIGHT profiler. All three kernels together (add, carry propagate and

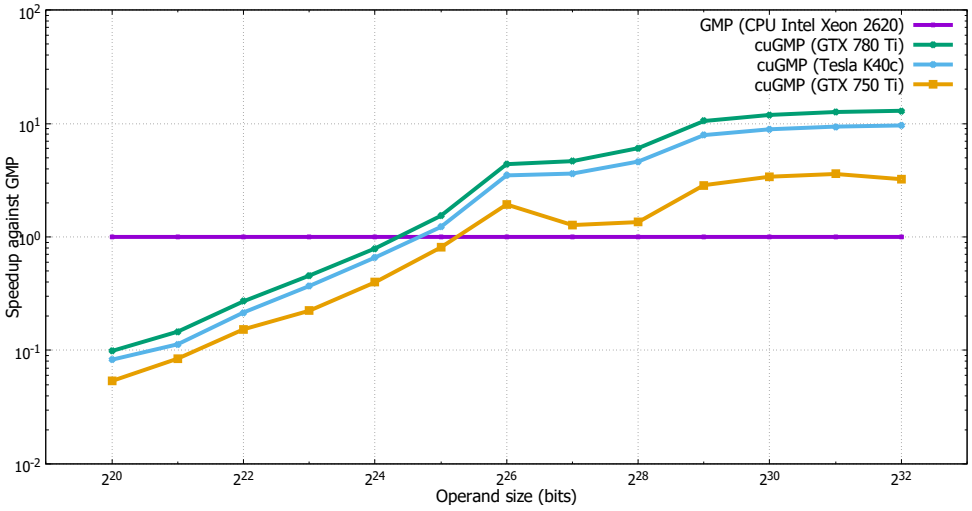


Figure 5.1: Addition speedup

Table 5.1: Computation time - addition (micro seconds)

Operand size	GMP (gpu-01)	Tesla K40c	GTX 780 Ti	GTX 750 Ti
2^{20}	45,4	547,7	459,3	843,9
2^{21}	67,1	592,6	458,9	793,5
2^{22}	130,6	604,7	482,2	851,5
2^{23}	253,0	685,1	554,8	1131,0
2^{24}	503,5	764,4	636,2	1264,3
2^{25}	1121,2	916,0	731,0	1387,4
2^{26}	4374,0	1254,8	999,0	2264,5
2^{27}	6190,7	1710,9	1330,3	4863,9
2^{28}	12086,7	2620,0	1995,7	8943,7
2^{29}	34711,4	4396,8	3304,5	12139,4
2^{30}	70539,6	7939,6	5931,1	20780,9
2^{31}	141066,4	15027,8	11177,9	39324,1
2^{32}	281121,0	29266,1	21728,2	87123,3

calculate result size) took around 75 micro seconds. The whole addition operation, on the other hand, was measured 850 micro seconds. There is also a single cudaMalloc call (to allocate carry buffer) that took 400 micro seconds.

This affects only small operands, because the cudaMalloc has almost the same duration for small and large operands. For example for 10 times larger operand it takes only 430 micro seconds (less than 10% increase).

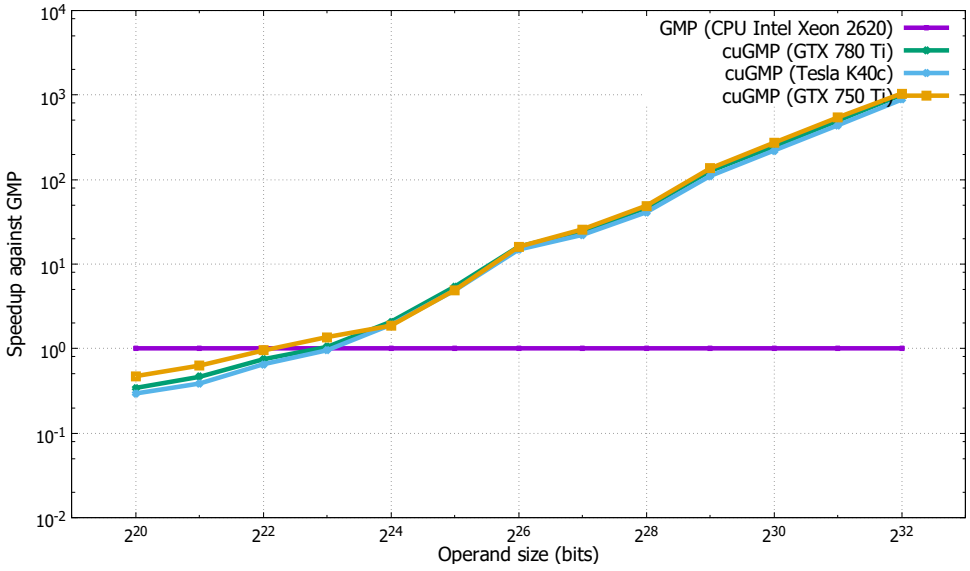


Figure 5.2: Logical AND speedup

5.4.2 Subtraction operation

Subtraction has almost the same performance as addition. There is a small slowdown due to the operand swapping - there is one extra kernel to detect which operand is larger, followed by cudaMemcpyFromSymbol. Other than that, the performance is very similar.

5.4.3 Logical operations

To keep this chapter short, I will present graph and results table only for AND operation - OR and XOR results are almost identical. All three logical operations implemented, AND, OR and XOR, perform very well on CUDA. This is due to the simplicity of the operations - there is no overflow propagation etc. There is only one kernel, that loads two limbs and stores one result limb. I have accomplished almost 1000x speedup compared to GMP for the larges operands as illustrated in Figure 5.2. In the result table 5.2 we can see that computation time is almost constant to the operand size.

5.4.4 Bitwise shift operation

To illustrate performance of bit shifting, I present results for bit shift left, implemented in mpz_mul_2exp function. As seen in Figure 5.3, speedup comes around 2^{24}bit large operand. Maximum speedup is around 50x for GTX 780 Ti device, which has the fastest memory access of up to 336 GB/s.

Table 5.2: Computation time - logical AND (micro seconds)

Operand size	GMP (gpu-01)	Tesla K40c	GTX 780 Ti	GTX 750 Ti
2^{20}	45,7	155,0	133,3	96,7
2^{21}	66,1	171,0	142,5	104,9
2^{22}	127,5	195,0	170,0	133,9
2^{23}	237,6	248,2	226,5	173,2
2^{24}	488,9	254,8	234,7	261,2
2^{25}	1305,1	266,2	240,6	264,0
2^{26}	4387,4	295,0	272,1	273,3
2^{27}	6744,7	303,9	277,1	261,3
2^{28}	12861,9	313,1	276,7	262,1
2^{29}	35021,0	317,2	277,3	258,3
2^{30}	70423,5	319,2	285,0	257,9
2^{31}	141349,3	322,2	284,1	260,3
2^{32}	281662,2	319,0	279,7	270,0

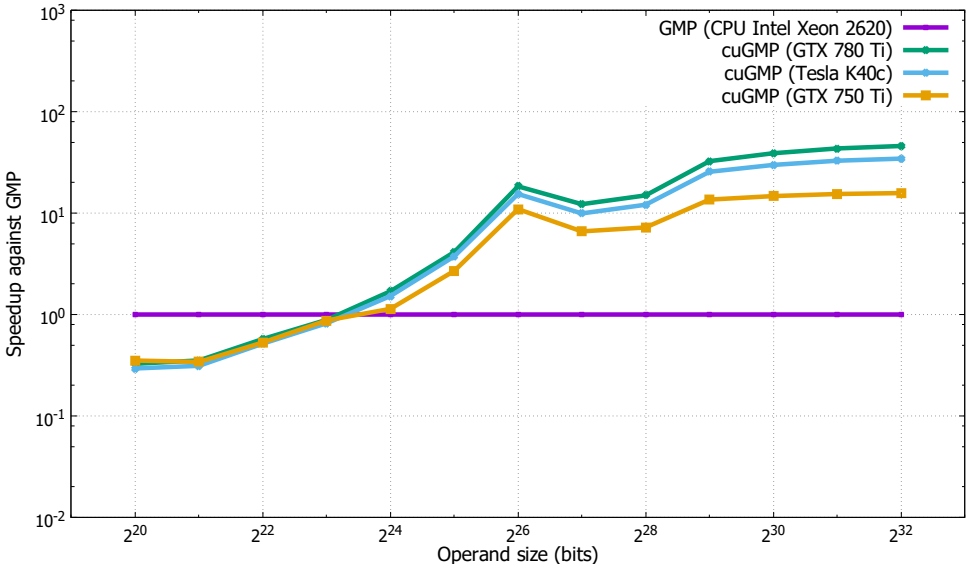


Figure 5.3: Bit shift left speedup

Table 5.3: Computation time - bit shift left (micro seconds)

Operand size	GMP (gpu-01)	Tesla K40c	GTX 780 Ti	GTX 750 Ti
2^{20}	50,5	171,6	154,4	143,4
2^{21}	57,7	184,6	163,5	169,1
2^{22}	111,1	215,2	193,7	209,6
2^{23}	224,0	273,0	252,9	256,2
2^{24}	443,6	291,4	261,7	389,9
2^{25}	1239,4	331,0	300,3	460,2
2^{26}	6396,7	414,3	348,1	584,1
2^{27}	5315,0	535,3	433,6	805,0
2^{28}	9071,4	749,3	606,8	1254,9
2^{29}	29782,2	1164,0	915,2	2190,4
2^{30}	59849,4	2004,1	1537,1	4063,5
2^{31}	120737,6	3672,3	2780,0	7817,1
2^{32}	241371,5	7001,2	5259,8	15317,1

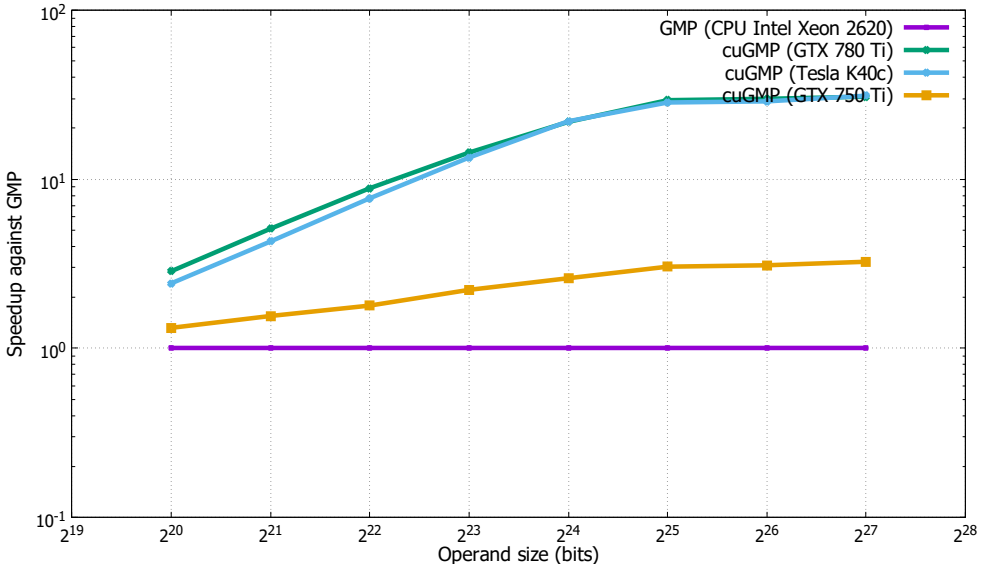


Figure 5.4: Multiplication speedup

5.4.5 Multiplication operation

Multiplication operation using cuFFT library provides very stable speedup even for smaller (around 1mbit) operands. For operands larger than 2^{25}bits, speedup becomes almost constant - around 30x for the most powerful devices.

As discussed in section 1.2.2, GMP uses different multiplication algorithms based on operand size. Results I have measured indicated, that the threshold

Table 5.4: Computation time - multiplication (micro seconds)

Operand size	GMP (gpu-01)	Tesla K40c	GTX 780 Ti	GTX 750 Ti
2^{20}	9813,3	4052,1	3435,6	7428,6
2^{21}	21646,8	5050,7	4246,2	13945,7
2^{22}	49540,9	6418,0	5598,8	27695,1
2^{23}	109000,6	8112,6	7578,7	49202,3
2^{24}	242472,2	11000,4	11101,6	93401,4
2^{25}	542244,9	19065,8	18474,8	178437,9
2^{26}	1123717,9	38893,2	37579,2	362871,7
2^{27}	2321044,9	74382,2	75360,1	713915,5

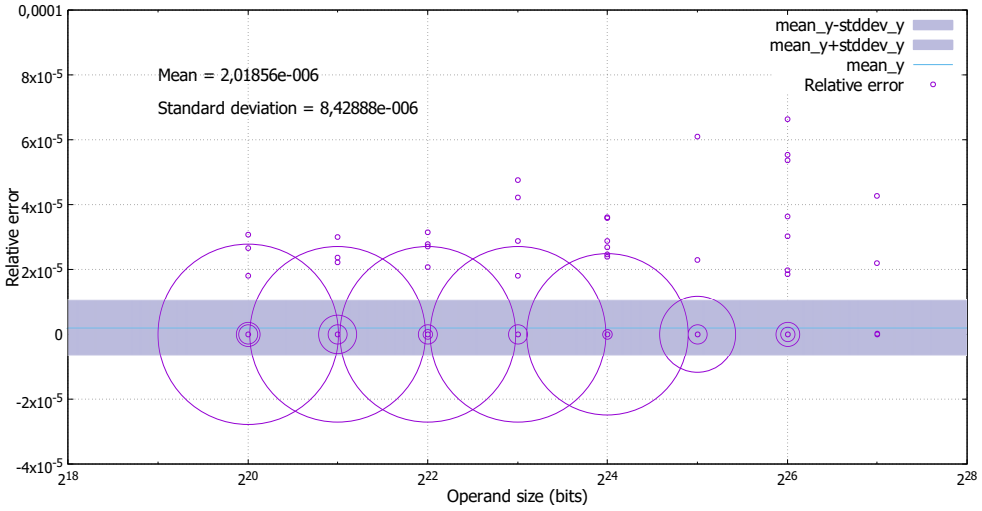


Figure 5.5: Relative error - multiplication

might be around 2^{25} bits, because from there on the GMP scales similarly to cuGMP. I have looked into this and it seems the thresholds differ from CPU to CPU, but they are usually set between 5000 and 10000 limbs as stated in manual [23]. That means, GMP uses FFT multiplication for operands larger than 0.5-0.7mbit. All operands I have tested are larger than 1mbit. That means, GMP is also using FFT multiplication - only difference is, that GMP does not utilize parallel algorithm.

5.5 Multiplication precision

As discussed in section 3.4.6, multiplication algorithm using cuFFT library suffers from rounding errors. To determine how big those errors are, I have measured not only computation time, but also relative error. This error is

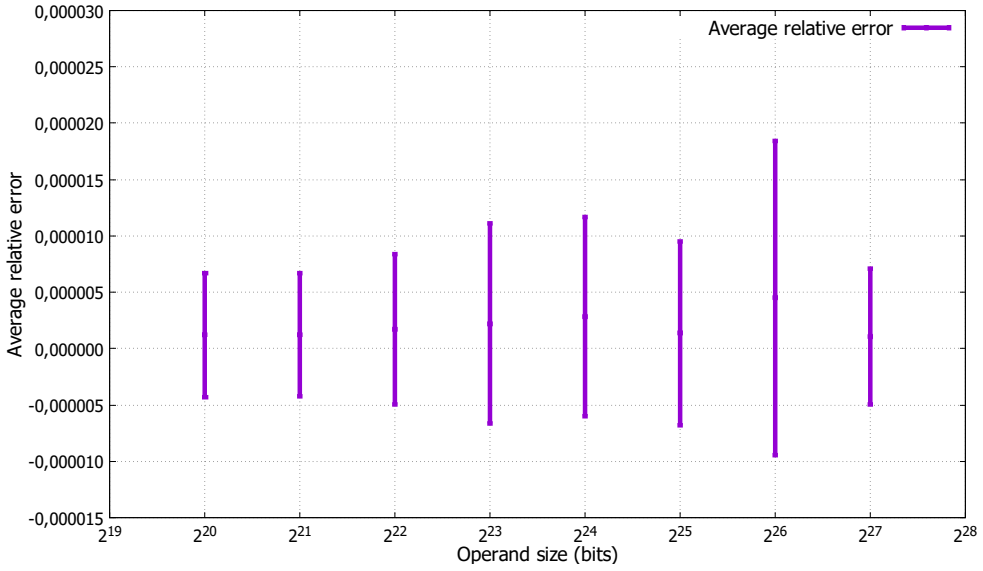


Figure 5.6: Relative error based on operand size - multiplication

Table 5.5: Relative error and standard deviation based on operand size.

Operand size	Average Error	Standard deviation
2^{20}	1,21591E-06	5,51864E-06
2^{21}	1,22158E-06	5,46757E-06
2^{22}	1,72845E-06	6,65414E-06
2^{23}	2,20125E-06	8,88246E-06
2^{24}	2,84342E-06	8,81661E-06
2^{25}	1,35525E-06	8,16525E-06
2^{26}	4,52421E-06	1,39462E-05
2^{27}	1,05841E-06	6,01884E-06

obtained by dividing cuGMP result by GMP result and then subtracting 1.

In Figure 5.5 we can see the errors measured for different operand sizes. Size of the circle illustrates the number of results with the same value. Biggest circles are those with the same value - usually 0, that means no rounding error.

Average error for operands from 2^{20} to 2^{27}bits (around 160 different data points) is $2,02 \cdot 10^{-6}$.

To illustrate that there is no obvious relation between operand sizes and the size of the relative error, I have made Figure 5.6. The middle of each vertical line represent the average error for corresponding operand size. Upper and lower ends of each line represent the size of standard deviation. Error and deviation is also illustrated in table 5.5

5.6 Results - conclusion

All implemented operations provide significant speedup, mainly for large operands. To achieve this, all arithmetic operations had to be implemented entirely on the GPU to avoid unnecessary memory transfers.

The largest speedup has been recorded for logical operations. They perform up to 1000x faster than GMP library on CPU. Other operations provide around 10x - 50x speedup for largest operands.

One drawback is multiplication precision that, unfortunately, isn't 100%, because of the use of floating point FFT transform. This effectively means, that current multiplication implementation cannot be used in practice. Possible solution is suggested in future work, section 6.3.

All other implemented operations provide both speedup and accuracy and can therefore be used for practical computation.

On the graphs illustrating speedup against GMP library, we can see that the implemented algorithms scale nicely in terms of parallelism - the larger the operands, the greater the speedup. We can also compare mid-range CUDA device (GTX 750 Ti) against hi-end cards (Tesla K40c and GTX780 Ti). I have also verified, that the library works consistently on both Windows and Linux platform.

Chapter 6

Future work

In this chapter I will summarize topics, that I think can be optimized further. I will also discuss operations, that were not implemented yet in cuGMP library.

6.1 Addition operation

Carry propagation can be very slow in specific conditions - for example when adding 1 to a large integer that has every bit set to one. In that scenario, carry has to travel from least significant bit to the most significant and in current implementation, this takes $O(n)$ steps for n limbs.

That would not be a problem for multiplication algorithm that can have complexity $O(n^2)$, but for addition that is $O(n)$ as the whole operation that means a lot - especially as the rest of the algorithm is done in parallel, but carry propagation is not.

This scenario is not very probable or often in real life, but even so, it could be optimized a bit further. This can be done via parallel prefix sum algorithm as proposed in article [24]. The only difference from normal PPS algorithm is in the operation - instead of sum (addition) we would decide if carry should or should not propagate.

6.1.1 Memory allocation

As proposed in section 4.3.1, small performance could be achieved (mainly for smaller operands) by avoiding the use of carry buffer. That would probably mean more uint64 computations, which tend to be slower on CUDA. On the other hand, for small operands, allocating carry buffer itself takes more time than the rest of the addition process.

6.1.2 CUDA PTX Carry detection

As explained in article [25], CUDA supports overflow detection inside its instruction set architecture. This is, however, not usable in the scope of this thesis. Carry-out flag, that is set using *uadd* assembly function can be consumed only using another assembly function like *uaddc*.

What I would need is extract the carry-out flag and save it to the carry buffer for later processing. This way I could save some, but probably not large, amount of computational work.

I think one way this could be exploited to gain performance advantage it through loop unrolling - addition could be unrolled inside the kernel, doing several limb additions per one thread, utilizing *uaddc* assembly function. This way some memory bandwidth may be saved as carry information would not have to be stored inside the carry buffer for all the limbs. This however adds complexity as number of limbs would have to be divisible by the number of limbs processed in each thread.

6.2 Comparison operation

There is a set of functions in GMP that could also be implemented - Comparison Functions. Those are discussed in manual [26]. These functions are mpz_cmp, mpz_cmpabs and mpz_sgn. Mpz_cmp returns positive integer, if first operand is larger, zero if they are the same and negative integer, if the second operand is larger. Mpz_abs does the same, but only with the absolute values.

I believe CUDA can provide significant speedup, but mainly for numbers that are mostly the same, or the same value completely. Otherwise, GMP will probably return the result almost instantly.

6.3 Multiplication - Number Theoretic Transform

There is a problem of rounding errors in the result of multiplication as discussed in section 3.4.6. This is caused by floating point nature of cuFFT library. GMP actually does not use such FFT and the reason is explained in GMP manual [27]:

> "Floating point FFTs use complex numbers approximating Nth roots of unity. Some processors have special support for such FFTs. But these are not used in GMP since it's very difficult to guarantee an exact result (to some number of bits). An occasional difference of 1 in the last bit might not matter to a typical signal processing algorithm, but is of course of vital importance to GMP."

As discussed article in [13], there is a way to avoid rounding error in the process of multiplication by the use of NTT. This is also a way that GMP uses as stated in manual [28]. GMP manual specifically mentions Fermat style FFT.

Results of article [13] suggests that this method does provide reasonable performance when implemented right.

I believe that implementing NTT capable library on CUDA would be very beneficial work.

6.4 Division operation

As proposed for example in thesis [29], large integer division can be computed with the same asymptotic complexity as multiplication. To be able to do that, there are so called inversion algorithms, that do quite a lot of precomputation, but end up with large integer multiplication. Most common algorithm of this type is called Newton Division. This method is probably suitable for really large integers as the precomputation phase can be quite large.

6.5 Measure performance on different hardware

Future readers are welcome to perform measurements on their CUDA hardware. Anything with compute capability 2.0 should work out of the box.

The Visual Studio project is currently configured with compute capability 3.0. This is done only to support larger x-dimension of a grid of thread blocks, therefore supporting larger operands. Compute capability less than 3.0 supports only 65535. That effectively limits current multiplication implementation to operands of around 16 milion limbs. This could of course be easily fixed via multiple kernel executions for larger operands, but since I have compute capability 5.0 GPU, I did not have to deal with that.

Linux is now also supported. When CUDA and GMP is installed on the system, installation of cuGMP with TestApp is a question of cloning the GIT repository and issuing make command.

As for testing methodology, it is largely determined in the implementation of TestApp, so it should not be difficult to execute similar tests as I have executed. Output is basically a valid CSV file, that could be easily submitted on github for comparison.

Conclusion

I have successfully implemented large integer arithmetic using parallel algorithms running on CUDA device. Implemented arithmetic operations are addition, subtraction, bitwise shift left and right, logical AND, OR and XOR and multiplication. I have compared my solution against GMP library running on CPU.

Main focus of this thesis was performance - all implemented operations provide significant speedup for large enough integers - from 10x speedup in case of addition and subtraction to up to 1000x speedup in case of logical operations (AND, OR, XOR).

To provide reasonable comparison, I have ported the library to Linux and tested it not only on Windows OS on a consumer PC, but also on our university GPU server cluster called STAR.

The main disappointment is implementation of multiplication algorithm using floating point FFT transform. It provides significant speedup against GMP library, but also suffers from rounding errors, which makes it practically unusable for most uses.

Bibliography

[1] Free Software Foundation. *What is GMP?* Available from: https://gmplib.org/

[2] Hao Jun Liu, C. T. GMP implementation on CUDA - A Backward Compatible Design With Performance Tuning. 2010. Available from: http://individual.utoronto.ca/haojunliu/courses/ECE1724_Report.pdf

[3] Bantikyan, H. Big Integer Multiplication with CUDA FFT (cuFFT) Library. 2014. Available from: http://www.rroij.com/open-access/big-integer-multiplication-with-cuda-fftcufft-library.pdf

[4] Free Software Foundation. *GNU MP - Nomenclature and Types.* Available from: https://gmplib.org/manual/Nomenclature-and-Types.html

[5] Tohoku University. *Hardware algorithms for arithmetic modules [online].* Available from: http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html

[6] Free Software Foundation. *GMP - Multiplication Algorithms.* Available from: https://gmplib.org/manual/Multiplication-Algorithms.html

[7] Weisstein, E. W. Fast Fourier Transform. *MathWorld - A Wolfram Web Resource.* Available from: http://mathworld.wolfram.com/FastFourierTransform.html

[8] NVIDIA Corporation. *cuFFT - CUDA Toolkit Documentation - accuracy and performance.* 2015. Available from: http://docs.nvidia.com/cuda/cufft/#accuracy-and-performance

[9] Wikipedia-contributors. Cooley–Tukey FFT algorithm. 2016. Available from: https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm

[10] Emerencia, A. MULTIPLYING HUGE INTEGERS USING FOURIER TRANSFORMS. 2007. Available from: http://www.cs.rug.nl/~ando/pdfs/Ando_Emerencia_multiplying_huge_integers_using_fourier_transforms_paper.pdf

[11] Tembhurne, J. V. High Throughput Long Integer Multiplication using Fast Fourier Transform on Parallel Workstation. 2014. Available from: https://www.researchgate.net/publication/272122279

[12] Šnajdr, B. K. *Počítání s rozšířenou přesností na GPU*. Masters thesis, Fakulta informačních technologií ČVUT, 2014.

[13] Emeliyanenko, P. Efficient Multiplication of Polynomials on Graphics Hardware. 2009. Available from: http://domino.mpi-inf.mpg.de/intranet/ag1/ag1publ.nsf/AuthorEditorIndividualView/ca00677497561c7ec125763c0044a41a/$FILE/gpgpu_mul.pdf

[14] NVIDIA Corporation. *CUDA - SIMT Architecture*. 2015. Available from: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture

[15] NVIDIA Corporation. *cuFFT - CUDA Toolkit Documentation*. 2015. Available from: http://docs.nvidia.com/cuda/cufft/

[16] Goldberg, D. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 1991.

[17] Weisstein, E. W. Complex Multiplication. *MathWorld - A Wolfram Web Resource*. Available from: http://mathworld.wolfram.com/ComplexMultiplication.html

[18] Microsoft Corporation. *Windows Data Types*. 2012. Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/aa383751(v=vs.85).aspx

[19] Standard C++ Foundation. *C++11 Language Extensions — Other Types*. Available from: https://isocpp.org/wiki/faq/cpp11-language-types#long-long

[20] Free Software Foundation. *GNU MP - Initialization Functions*. Available from: https://gmplib.org/manual/Initializing-Integers.html

[21] NVIDIA Corporation. *cuFFT - CUDA Toolkit Documentation - cuFFT API Reference*. 2015. Available from: http://docs.nvidia.com/cuda/cufft/#cufft-api-reference

[22] Ahn, S. H. High Resolution Timer [online]. 2005. Available from: http://www.songho.ca/misc/timer/timer.html

[23] Free Software Foundation. *MUL_FFT_THRESHOLD*. Available from: https://gmplib.org/devel/thres/MUL_FFT_THRESHOLD.html

[24] O'Donovan, A. Parallel Prefix Sum on the GPU (Scan). Available from: http://www.umiacs.umd.edu/~ramani/cmsc828e_gpusci/ScanTalk.pdf

[25] NVIDIA Corporation. *Parallel Thread Execution ISA Version 4.3 - Extended-Precision Integer Arithmetic Instructions*. 2015. Available from: http://docs.nvidia.com/cuda/parallel-thread-execution/index.html#extended-precision-integer-arithmetic-instructions

[26] Free Software Foundation. *GMP - Comparison Functions*. Available from: https://gmplib.org/manual/Integer-Comparisons.html

[27] Free Software Foundation. *GNU MP - Other Multiplication*. Available from: https://gmplib.org/manual/Other-Multiplication.html

[28] Free Software Foundation. *GMP - FFT Multiplication*. Available from: https://gmplib.org/manual/FFT-Multiplication.html

[29] Hasselström, K. *Fast Division of Large Integers*. Masters thesis, Royal Institute of Technology, Stockholm, SWEDEN, 2003.

Appendix A

Links and contacts

As I would like future readers to be able to contact the author if anything is unclear and to be able to browse the source code, possibly even use it or extend it in any way I provide my e-mail address and URL of my github repository. Main reason is that my university e-mail address will cease to exist short after I will finish studying.

- E-mail: petr@petrous.cz
- Repository: https://github.com/trubus/cuGMP

Appendix B

Installation

To start using cuGMP library, there are several prerequisites. To build the library code, CUDA toolkit with nvcc compiler has to be installed. To build TestApp that does the performance measurements, GMP library has to be installed or built. In this chapter I will go through the installation process for two supported platforms - Windows OS with Visual Studio IDE and Linux OS with g++ compiler.

B.1 Windows x64

Start by preparing folder for cuGMP project. I will use `c:\cuGMP` for reference, but you can use whatever path you like.

Download the source files inside `c:\cuGMP` either by git clone command from cuGMP repository (URL in chapter A) or by copying them from attached CD.

B.1.1 Visual Studio

Then install Visual Studio IDE. For users without MSDN subscription, there is a free *Community* edition that is also supported by CUDA. At the time, latest stable CUDA framework for Windows supports only Visual Studio 2013. Support for 2015 version is in beta stage. For *Community* edition, download installer package from `https://www.visualstudio.com/en-us/news/vs2013-community-vs.aspx`.

B.1.2 CUDA

To install nvcc compiler, cuFFT library, NSIGHT profiler, Visual Studio support and CUDA enabled graphics driver, download installer package from `https://developer.nvidia.com/cuda-downloads`. Nvidia GeForce Experience software (if installed) will complain, that you do not have the latest

graphics drivers installed - ignore that, you have the latest CUDA drivers installed.

B.1.3 GMP

To be able to build GMP library using the same toolset (Visual Studio with MSBuild), there is a fork of GMP called MPIR. It exposes the same API, but it comes with Visual Studio projects (for many recent versions of Visual Studio) that can compile almost instantly without the need to install further software or tweaking the configuration.

- Download MPIR from `http://www.mpir.org/mpir-2.7.2.zip`
- Extract obtained package into `c:\cuGMP\mpir-2.7.2`
- Open `mpir-2.7.2\build.vc12\mpir.sln` (VS Solution for VS2013)
- To assure correct build order, add build dependency of lib_mpir_cxx to lib_mpir_gc (Right click lib_mpir_cxx ->Build dependencies ->Project build order)
- Setup batch build of all configurations of projects lib_mpir_gc and lib_mpir_cxx (Build ->Batch build)
- Select *Build* in the *Batch build* dialog

This way, all configurations of GMP (32 and 64 bit, debug and release) are built on one click. If newer version of MPIR is used, make sure to update include paths in TestApp project.

B.1.4 cuGMP and TestApp

Now everything is ready to start using cuGMP. Open `c:\cuGMP\cuGMP.sln` (VS solution). Now there are two projects - cuGMP (the library using CUDA compiler) and TestApp (measurement and testing application using Visual C++ compiler).

B.2 Linux

Start by preparing folder for cuGMP project. I will use `~/cuGMP` for reference. Inside you can download sources from repository by issuing *git clone* command or you can copy source files from attached CD.

B.2.1 Prerequisites

I have actually used Linux servers that already had GMP library and CUDA toolkit installed. GMP library can be obtained from package repository or by downloading source files from `http://gmplib.org` and compiling using Makefile. For CUDA toolkit, packages can be obtained from `https://developer.nvidia.com/cuda-downloads`. Installation may differ slightly according to Linux distribution used.

B.2.2 cuGMP and TestApp

In order to build cuGMP and TestApp, go to the directory with source files. There is a Makefile that takes care of the build process.

In order to build cuGMP library only, issue *make cuGMP.a* command.

In order to make the library and TestApp, issue *make all* command.

On different environment, there might be a need to some variables inside the Makefile, mainly CCCUDAINCL (path to CUDA installation).

Appendix C

Acronyms

CUDA Compute Unified Device Architecture

GPU Graphics Processing Unit

GPGPU General-purpose computing on graphics processing units

GMP The GNU Multiple Precision Arithmetic Library

MPIR Multiple Precision Integers and Rationals (GMP fork)

cuFFT NVIDIA® CUDA™ Fast Fourier Transform

nvcc NVIDIA® CUDA™ Compiler

API Application Program Interface

OS Operating System

NSIGHT profiler CUDA toolkit profiler

DFT Discrete Fourier Transform

FFT Fast Fourier Transform

NTT Number Theoretic Transform

ALU Arithmetic Logic Unit

PPS Parallel Prefix Sum

IDE Integrated Development Environment

MS Microsoft

Appendix D

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
  - `win64` ..... Windows x64 executables
  - `linux64` ..... Linux x64 executables
- `results` ..... Raw measured data in excel format
- `src` ..... the directory of source codes
  - `cuGMP` ..... cuGMP and TestApp implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format