Multiplication of Two Bignums on a Nvidia Graphic Card

Johannes Häbe
jh-191149@hs-weingarten.de

Maximilian Nestle
mn-192181@hs-weingarten.de

Ravensburg-Weingarten University of Applied Sciences

July 7, 2020

Abstract

This paper is about testing the performance of the NVIDIA CUDA Fast Fourier Transform library (cuFFT) by multiplying large numbers (bignums) on the graphics card. The performance is measured up to the size of 28000 bits for each factor. The factors are also multiplied on the CPU using the Karatsuba algorithm. For the multiplication of two bignums, performance of the CPU turned out to be in average about one hundred times faster than multiplication on the GPU. Even taking only the measured times for the pure multiplication, the GPU is still slower than the CPU. By increasing size of the factors from 24 bit up to 28000 bits, the measured times between CPU and GPU are gradually approaching.

1 Introduction

The fast multiplication of two large prime numbers is necessary in some procedures to attack asymmetrical encryption algorithms like the RSA encryption. These computations are normally made on the CPU. Classical approaches have $O(n^2)$ complexity, but polynomial multiplication with FFT has $O(n \cdot logn)$ complexity [1]. Within this paper, General Purpose Computing On GPUs (GPGPU) is used to multiply bignums with the help of the

NVIDIA CUDA Fast Fourier Transform library (cuFFT). For this purpose, the computing time of bignum multiplication on CPU and GPU is compared and evaluated in this paper.

2 Related Work

Hovhannes Bantikyan from the University of Armenia did time measurements up to the size of 200000 digits for each factor and compared the results with other bignum multiplication libraries [1]. In contrast to that, the results of cuFFT are compared with the `BN_mul()` function from the openssl library in this paper. The size of the factors is also increased and measured in smaller steps of 24 bits. Ando Emerencia also included time measurements in his paper "Multiplying huge Integers using Fourier Transforms" [2]. He compared time measurements of mulitplications with different bases, furthermore he did Maximum Square Error tests.

3 Technical Background

3.1 Fast Fourier Transformation

The Fast Fourier Transformation (FFT) is based on the Discrete Fourier Transformation (DFT). The DFT of a vector $(x_0, \ldots, x_{2n-1})$ of the size $2n$ is [3]:

$$f_m = \sum_{k=0}^{2n-1} x_k \cdot e^{-\frac{2\pi \cdot i}{2n} mk}$$

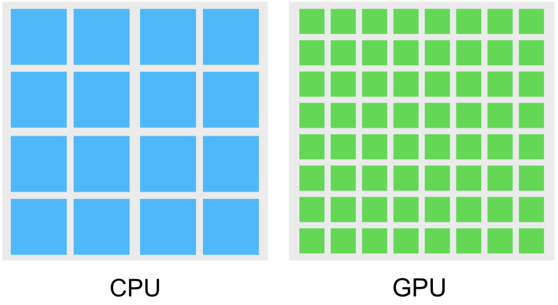
$$\text{with} \quad m = 0, \ldots, 2n-1.$$

Fast Fourier Transformation can be used to perform fast integer multiplication via fast polynomial multiplication. CUFFT Library employs the Cooley-Tukey algorithm to reduce the number of required operations resulting in $O(n \cdot logn)$ complexity [4]. The resulting complexity is reached by an divide and conquer approach [5]. Radix-2 decimation-in-time (DIT), which is the simplest and most common form of the Cooley–Tukey algorithm, first computes the DFTs of the even-indexed inputs and of the odd-indexed inputs and then combines those two results to produce the DFT of the whole sequence [6]. By performing this recursively, the overall runtime can be reduced to $O(n \cdot logn)$.

3.2 Features GPU and CPU

CPUs have less cores than GPUs, but the cores are more powerful, that is visualized in figure 1 .GPUs that are used in modern computers do have hundreds of cores and many more threads, thus they are very effective in parallel computing [7]. Parallel Computing speeds up the process when multiplying two numbers, because multiplication is always a process of many tiny multiplications that are summed up. Computations on the CPU are preferred when multiplying tiny numbers, because of the low memory access times [8].

Figure 1: CPU cores (blue) and GPU cores (green)



4 Methodology

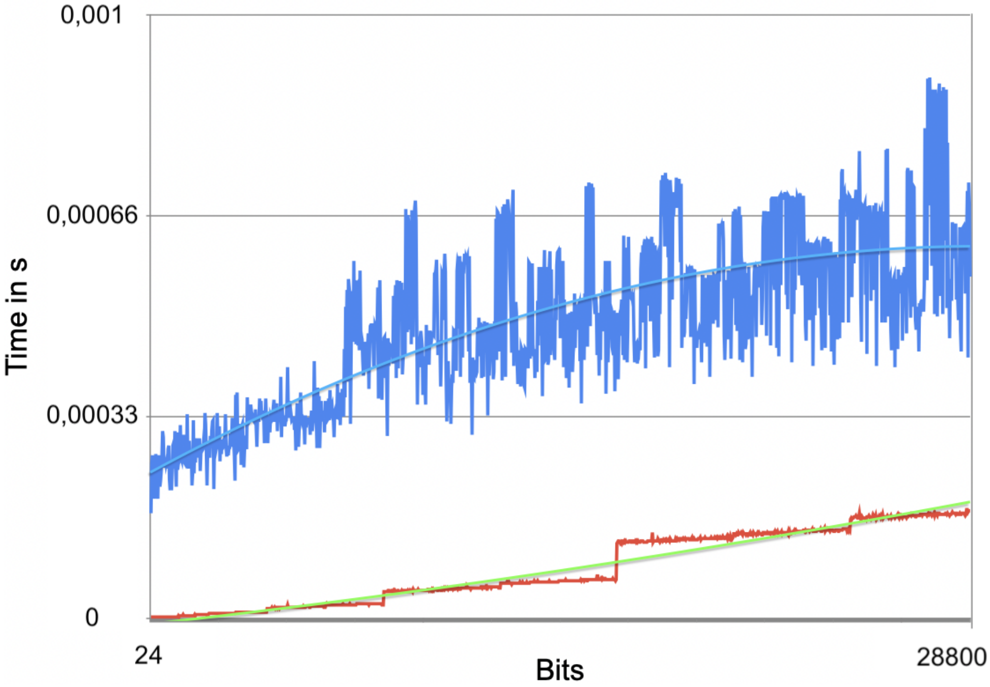
For the multiplications an Acer Aspire V3-772G is used. The Acer has got a NVIDIA GTX 850M graphics card and an Intel Core i5 4200M CPU. The graphics card disposes of 2004 MIB storage. The Acer is running the Linux distribution Ubuntu. The code that is used to multiply two bignums on the GPU is using the cuFFT library from NVIDIA. The multiplications on the CPU are done with the `BN_mul()` function from the openssl library which is based on the Karatsuba recursive multiplication algorithm [9]. The Karatsuba algorithm has $O(n^{1.585})$ complexity [10].

The computation time for the multiplications of the two numbers is measured up to the size of 28000 bits for each factor. The step size for the multiplications is 24 bits. That means that computations are done for numbers of the size 24 bits, 48 bits, 72 bits and so on. The numbers used for computations are created randomly. To prevent variations caused by random created numbers that are easy to multiply, one hundred multiplications, each with random numbers, are done for each step of 24 bits.

5 Results

For the time measurement the C library function `clock()` is used. The legend of the measured times is explained in table 1. In table 2 the minimum, maximum and average times for the calculations from 24 bits to 28800 bits are listed. The minimum for all seven categories is located between 24 and 120 bits, the maximum was always close to 28000 bits. Table 2 shows that, that GPU_Alloc and CUDA_Pre cost by far the most time. They take each in average about one hundred times more time than the CPU calculation.

Figure 2: GPU_Calc (blue) and CPU (red) with computed polynomial.



The second grade polynomial of GPU_Calc and CPU were computed and visualized in figure 2 with the original graphs.

$$GPU\_Calc_{poly}(x) = -1.269 \cdot 10^{-10}x^2 + 3.086 \cdot 10^{-7}x + 0.0001$$

$$CPU_{poly}(x) = 1.77 \cdot 10^{-11}x^2 + 6.295 \cdot 10^{-8}x - 4.799 \cdot 10^{-6}$$

The slope of the blue polynomial is decreasing, while the slope of the green polynomial is increasing. It can be derived, that the GPU is getting more effective in comparison to the CPU, the bigger the numbers get. This

Table 1: Descriptions of the abbreviations of the measured times.

CPU	The time needed for the multiplications of the two bignums on the CPU using the `BN_mul()` function.
GPU_All	The sum of the times needed for GPU_Alloc, GPU_Calc, GPU_Clean, CUDA_Pre, and CUDA_Post.
GPU_Alloc	The time needed to allocate the amount of graphics card memory needed for the two bignums using `cudaMalloc()` and copying them to the graphics card memory by using `cudaMemcpy()`.
GPU_Calc	The amount of time needed for the calculation of the two bignums on the GPU. This includes converting the bignums to frequency domain, multiplying them with `ComplexPointwiseMulAndScale()` and converting them back to time domain.
GPU_Clean	The sum of the times needed to copy the data back to the host by using `cudaMemcpy()` and to free the graphics card memory by using `cufftDestroy()` and `cudaFree()`.
CUDA_Pre	The time needed to prepare the data for the multiplication. The algorithm needs to convert the two numbers from the datatype bignum to float vectors. Float vectors are required for the `cudaMemcpy()` function. Also the size of the vectors needs to be adjusted, because cuFFT saves the solution in one of the initial bignums.
CUDA_Post	The time needed to prepare the result. This includes removing excess zeros, processing carry and turning the result from a float vector to a bignum.

Table 2: Average, minimum and maximum times measured of all bit sizes.

	Times in seconds		
	Minimum	*Maximum*	*Average*
CPU	1×10^{-6}	9×10^{-5}	$4,16 \times 10^{-5}$
GPU_All	$1,72 \times 10^{-3}$	$1,48 \times 10^{-2}$	$7,19 \times 10^{-3}$
GPU_Alloc	$1,42 \times 10^{-3}$	$6,13 \times 10^{-3}$	$3,67 \times 10^{-3}$
GPU_Calc	$8,7 \times 10^{-5}$	$4,48 \times 10^{-4}$	$2,45 \times 10^{-4}$
GPU_Clean	$1,71 \times 10^{-4}$	$8,17 \times 10^{-4}$	$4,23 \times 10^{-4}$
CUDA_Pre	8×10^{-6}	$6,88 \times 10^{-3}$	$2,44 \times 10^{-3}$
CUDA_Post	2×10^{-6}	$9,21 \times 10^{-4}$	$3,97 \times 10^{-4}$

assumption is only valid as long as we use Karatsuba for CPU and FFT for GPU. The graph reflects the complexities of the algorithms.

The measurements were also done on two more computers, to make sure, that the structure of the resulting graphs is nearly the same and not a specific result of the depending hardware of the computers. The assumption was confirmed.

6 Conclusion

The big issues are that the times needed for preparation and allocation of the GPU are totally out of range. The CPU is able to store all data in cache, GPU data transfer takes too long [11]. Even if only the calculation process on the GPU with the CPU are compared, the CPU is up to 28000 bits still at least 5 times faster than the GPU. With parallel computing of multiple pairs of numbers on the GPU, it can be assumed that the process can be accelerated, for example by allocating a bigger amount of VRAM and putting multiple pairs of numbers on the graphics card memory with just one call of `cudaMemcpy()`.

The time measurements lead to the conclusion, that multiplying only one pair of numbers on a GPU with cuFFT is way too inefficient, computing them on the CPU is the recommended way. An exception could be mulitplying numbers that are way bigger than 28000 bits, because the larger the numbers get, the more effective becomes the multiplication on the GPU. Although more performance improvement experiments would be needed to confirm the assertion.

References

[1] Hovhannes Bantikyan, “Big integer multiplication with cuda fft (cufft) library”, *world*, vol. 2, no. 11, 2014.

[2] Ando Emerencia, “Multiplying huge integers using fourier transforms”, *Online slides. URL http://www. cs. rug. nl/˜ ando/pdfs/Ando_Emerencia_ multiplying_huge_integers_using_fourier_transforms_presentation. pdf*, 2007.

[3] Wikipedia, “Schnelle fourier-transformation — wikipedia, die freie enzyklopädie”, 2020, [Online; Stand 4. Juli 2020].

[4] *CUDA Toolkit 4.2 CUFFT Library*, NVIDIA Corporation, 2012.

[5] Amente Bekele, “Cooley-tukey fft algorithms”, *Advanced algorithms*, 2016.

[6] Wikipedia contributors, “Cooley–tukey fft algorithm — Wikipedia, the free encyclopedia”, 2020, [Online; accessed 24-June-2020].

[7] Jonathan Palacios and Josh Triska, “A comparison of modern gpu and cpu architectures: And the common convergence of both”, 2011.

[8] Sebastian Albers, “Grafikkarten-architektur - parallele strukturen in der gpu”, 2009.

[9] Eric Young, “bn_mul.c”, 1995-1998.

[10] Martin Dietzfelbinger, “Effiziente algorithmen”, 2012.

[11] Christopher Cooper, “Gpu computing with cuda”, 2011.