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

June 16, 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. Several tests are presented, evaluated and compared.

1 Introduction

The fast multiplication of two large prime numbers is necessary in some procedures to hack 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 $O(nlogn)$ complexity [Ban14]. Within this paper, General Purpose Computing On GPUs (GPGPU) is used to multiply bignum prime numbers 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 Hardware and environment

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 which is based on the Karatsuba recursive multiplication algorithm [You98]. The Karatsuba algorithm has $O(n^{1.585})$ complexity [Die12]. The `BN_mul()` function comes with the openssl library.

3 Measurement conditions

The times for the multiplications of the two numbers are measured up to the size of 28000 bits for each factor. The step size for the multiplications is 24 bits. That means ~~that we did co~~mputations for numbers of the size 24 bits, 48 bits, 72 bits and so on. The numbers used for computations are created randomly. To prevent fluctuations because of random created numbers that are easy to multiply, one hundred multiplications, each with random numbers, are done for each step of 24 bits.

4 Evaluation of the measured times

For the time measurement the C library function `clock(void)` is used. The legend of the graphics is explained in the following:

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()`.

Table 1: 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}	$9,21 \times 10^{-4}$	$3,97 \times 10^{-4}$

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.

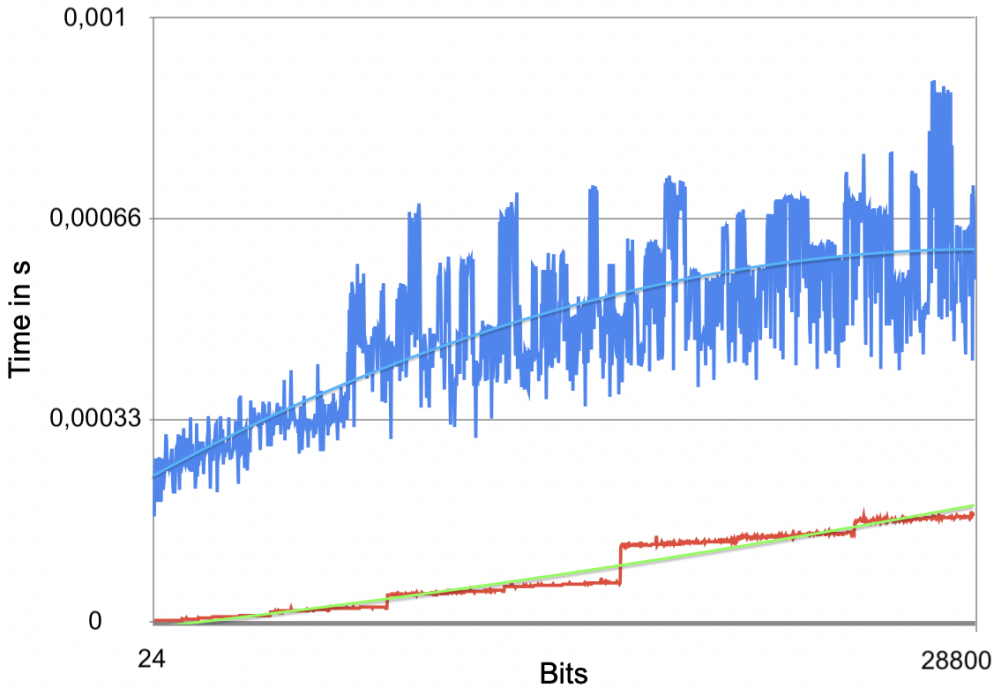
GPU_Clean
: The sum of the times needed to copy the data back to the host by using `cudaMemcpy()` and to free the graphics card memory by using `cufftDestroy()` and `cudaFree()`.

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.

In table 1 the minimum, maximum and average times for the calculations from 24 bits to 28800 bits are seen. The minimum is for all seven categories located between 24 and 120 bits, the maximum was always close to 28000 bits. ~~We~~ derive from the table, 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 1: GPU_Calc (blue) and CPU (red) with computed polynomial.



With Numbers by Apple, the second grade polynomial of GPU_Calc and CPU were computed and visualized in figure 1 with the original graphs. The slope of the blue polynomial ist getting smaller, while the slope of the green polynomial is increasing.

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

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

From the graph it can be derived, that the GPU is getting more effective in comparison to the CPU, the bigger the numbers get. This 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.

5 Issues and Improvements

The big issues are that the times needed for preparation and allocation of the GPU are totally out of range. Even if ~~we o~~nly compare the calculation process on the GPU with the CPU, 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()`.

6 Conclusion

The time measurements lead to the conclusion, that multiplying one pair of numbers on a GPU with cuFFT is way too inefficient, computing them on the CPU is the recommended way. Although more performance improvement experiments would be needed to confirm the assertion.

References

[Ban14] Bantikyan, Hovhannes: Big integer multiplication with CUDA FFT (cuFFT) library. In: *world* 2 (2014), Nr. 11

[Die12] Dietzfelbinger, Martin: Effiziente Algorithmen. (2012). `https://www.tu-ilmenau.de/fileadmin/public/iti/Lehre/Eff.Alg/SS12/EA-SS12-Kapitel1.pdf`

[You98] Young, Eric: bn_mul.c. (1995-1998). `https://opensource.apple.com/source/OpenSSL/OpenSSL-22/openssl/crypto/bn/bn_mul.c.auto.html`