Speed up Large Integer Multiplication Using Fourier Transforms and CUDA Technology

Hovhannes Bantikyan

State Engineering University of Armenia (Polytechnic), 105 Teryan Str., Yerevan, Armenia

bantikyan@gmail.com

Abstract. *Multiplying large integers is an operation that has many applications in Computational Science. Many cryptographic algorithms require operations on very large subsets of the integer numbers. Using Fast Fourier Transforms (FFT) and Graphics Processing Unit (GPU), we can speed up integer multiplication and make an effective multiplication algorithm. CUDA technology used to perform FFT on GPU.*

Keywords

Fast Fourier Transformation, Large Integer Multiplication, GPGPU, CUDA programming.

1 Introduction

Large integers multiplication in traditional approaches normally required $O(n^2)$ multiplication operations, where n is the number of digits. When n is large number, it is preferred to use more efficient algorithms. By using FFT we can decrease required time to $O(nlog(n))$. This entire work is aimed to develop a strategy to compute big integer multiplication using Fast Fourier Transform more efficiently and to reduce the time it takes for calculation. For such large data volume based applications the Graphics Processing Unit (GPU) based algorithm can be the cost effective solution. GPU can process large volume data in parallel when working in single instruction multiple data (SIMD) mode. In November 2006, the Compute Unified Device Architecture (CUDA) which is specialized for compute intensive highly parallel computation is unveiled by NVIDIA.

2 Theoretical Part

For using FFT first we have to represent an integer as a polynomial. The default notation for a polynomial is its coefficient form. A polynomial p represented in coefficient form is described by its coefficient vector $a = [a_0, a_1, ..., a_{n-1}]$ as follows:

$$p(x) = \sum_{x=0}^{n-1} a_i x^i \tag{1}$$

We call x the base of the polynomial, and p(x) is the evaluation of the polynomial, defined by its coefficient vector a, for base x. Multiplying two polynomials results in a third polynomial, and this process is called vector convolution. As with multiplying integers, vector convolution takes $O(n^2)$ time.

When we represent an integer as a polynomial, we have a choice in what base B to use. Any positive integer can be used as a base, but for the sake of simplicity we restrict ourselves to choosing a base that is a power of 10. With a base that is a power of 10, converting an integer given in decimal form to its coefficient vector is trivial. Consider the integer 123456, whose polynomial form using B = 10 is a = [6, 5, 4, 3, 2, 1].

2.1 Discrete Fourier Transform (DFT)

The one dimension of discrete fourier transform (DFT of an N-point discrete-time signal f(x)) is given by the equation

$$F(u) = \sum_{x=0}^{N-1} f(x) \mathrm{e}^{-\frac{j2\pi ux}{N}} \tag{2}$$

for u = 0,1,2, ..., N-1

Similarly, for given F(u) we can obtain the original discrete function f(x) by inverse DFT

$$f(x) = \frac{1}{N} \sum_{u=0}^{N-1} F(u) \mathrm{e}^{\frac{j2\pi ux}{N}} \tag{3}$$

for x = 0,1,2, ..., N-1

The Discrete Fourier Transform is frequently evaluated for each data sample, and can be regarded as extracting particular frequency components from a signal.

2.2 Fast Fourier Transform

By deploying a divide-and-conquer-strategy, we can construct an algorithm to compute the Fourier transform of a function f(x) of length $N = 2^p$ in $O(nlog(n))$ steps. We first 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. To investigate this, we need to split p(x) into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4) \tag{4}$$

Notice that the terms in parentheses are polynomials in x^2. More generally,

$$p(x) = p_{even}(x^2) + xp_{odd}(x^2) \tag{5}$$

where p_{even}, with the even-numbered coefficients, and p_{odd}, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$ (assume for convenience that n is even). Given paired points $\pm x_i$, the calculations needed for $p(x_i)$ can be recycled toward computing $p(-x_i)$

$$p(x_i) = p_{even}(x_i^2) + xp_{odd}(x_i^2) \tag{6}$$

$$p(-x_i) = p_{even}(x_i^2) - xp_{odd}(x_i^2) \tag{7}$$

The original problem of size n is in this way recast as two subproblems of size n/2, followed by some linear-time arithmetic. If we could recurse, we would get a divide-and-conquer procedure with running time

$$T(n) = 2T(n/2) + O(n) \tag{8}$$

which is $O(nlog(n))$, exactly what we want.

But we have a problem: The plus-minus trick only works at the top level of the recursion. To recurse at the next level, we need the n/2 evaluation points $x_0^2, x_1^2, ..., x_{n/2-1}^2$ to be themselves plus-minus pairs. But how can a square be negative? The task seems impossible! Unless, of course, we use complex numbers.

At the very bottom of the recursion, we have a single point. This point might as well be 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$.

The next level up then has $\pm\sqrt{+1} = \pm 1$ as well as the complex numbers $\pm\sqrt{-1} = \pm i$, where i is the imaginary unit. By continuing in this manner, we eventually reach the initial set of n points. Perhaps you have already guessed what they are: the complex nth roots of unity, that is, the n complex solutions to the equation $z^n = 1$.

Figure 1 introduces the nth roots of unity: the complex numbers $1, \omega, \omega^2, ..., \omega^{n-1}$, where $\omega = \mathrm{e}^{2\pi i/n}$. If n is even,

- The nth roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$,
- Squaring them produces the (n/2)nth roots of unity.

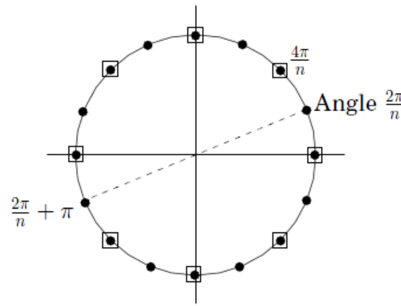

Figure 1. The nth complex roots of unity.

Therefore, if we start with these numbers for some n that is a power of 2, then at successive levels of recursion we will have the $(n/2^k)$th roots of unity, for k = 0, 1, 2, 3, ... All these sets of numbers are plus-minus paired, and so our divide-and-conquer, as shown in the last panel, works perfectly. The resulting algorithm is the fast Fourier transform (Figure 2).

function FFT(p, ω)

Input: Coefficient representation of a polynomial p(x) of degree $\leq n - 1$, where n is a power of 2,

ω, an nth root of unity

Output: Value representation $p(\omega^0), ..., p(\omega^{n-1})$

if ω = 1: return p(1)

express p(x) in the form $p_{even}(x^2) + xp_{odd}(x^2)$

call FFT(p_{even}, ω^2) to evaluate p_{even} at even powers of ω

call FFT(p_{odd}, ω^2) to evaluate p_{odd} at odd powers of ω

for j = 0 to n - 1:

compute $p(\omega^j) = p_{even}(\omega^{2j}) + \omega^j p_{odd}(\omega^{2j})$

return $p(\omega^0), ..., p(\omega^{n-1})$

Figure 2. The fast Fourier transform (polynomial formulation).

Polynomial product of two n - 1 degree polynomials p and q, has degree 2n - 1, so it requires evaluations in at least 2n distinct points to be uniquely identified. In order to get 2n evaluations, we use the 2nth primitive roots of unity, and so for our algorithm we need a polynomial with a coefficient vector of at least length 2n.

To be more precise, for our divide-and-conquer algorithm it is required that n is a power of 2, so instead of padding the coefficient vectors to length 2n, we pad them to length 2^k, where k is the lowest integer such that $2^k \geq 2n$. In the following we will assume without loss of generality that $2^k = 2n$.

The next step is computing the FFT's y=FFT(a) and z=FFT(b), with our FFT algorithm.

Now we have the evaluations of the polynomials p and q at the same 2n distinct inputs (the 2nth roots of unity), so if we multiply the 2n evaluations of p with the respective 2n evaluations of q, we calculate 2n products in total, that together uniquely represent the polynomial product of p and q:

$$m = y \times z = [y_0 z_0, y_1 z_1, ..., y_{2n-1} z_{2n-1}] \quad (9)$$

The remaining step is to compute the inverse FFT of m, to transform the vector of polynomial evaluations (FFT form), to the vector of its coefficients (coefficient form).

2.3 Overview of the GPU Architecture and CUDA

GPUs are massively multithreaded manycore chips. NVIDIA Tesla products have up to 128 scalar processors, over 12,000 concurrent threads in flight, over 470 GFLOPS sustained performance. NVidia graphics card architecture consists of a number of so-called streaming multiprocessors (SM). Each one includes 8 shader processor (SP) cores, a local memory shared by all SP, 16384 registers, and fast ALU units for hardware acceleration of trancendental functions. A global memory is shared by all SMs and provides capacity up to 4 GB and memory bandwidth up to 144 GB/s. FERMI architecture introduces new SMs equipped with 32 SPs and 32768 registers, improved ALU units for fast double precission floating point performance, and L1 cache.

CUDA is a scalable parallel programming model and a software environment for parallel computing. It is very easy to use for programmer introduces a small number of extensions to C language, in order to provide parallel execution. Another important features are flexibility of data structures, explicit access on the different physical memory levels of the GPU, and a good framework for programmers including a compiler, CUDA Software Development Kit (CUDA SDK), a debugger, a profiler, and CUFFT and CUBLAS scientific libraries.

The GPU executes instructions in a SIMT – single-instruction, multiple-thread – fashion. The execution of a typical CUDA program is illustrated in Figure 3

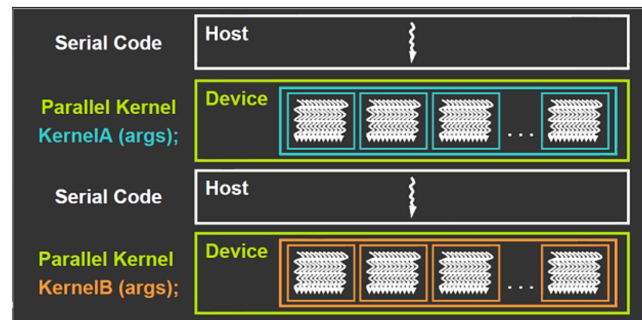

Figure 3. Execution of a CUDA program. Serial program with parallel kernels.

3 Conclusion

For small ffts, CUDA FFT performs much slower than CPU FFT, even in serial. Likely due to there not being much work, as the transform fits in CPU cache. Batching (multiple transform plan) results in much better speedup on the GPU. For larger ffts, CUDA FFT results in up to a 5x speedup over threaded CPU FFT.

4 Acknowledgements

I would like to thank Hakob Sarukhanyan for useful and pragmatic suggestions.

References

[1] NVIDIA CUDA C Programming Guide. NVIDIA Corp. 2012.

[2] CUFFT Library. Programming Guide. NVIDIA Corp. 2012.

[3] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani: Algorithms. 2006.

[4] ANDO EMERENCIA: MULTIPLYING HUGE INTEGERS USING FOURIER TRANSFORMS. 2007.

[5] Big Integer Arithmetic based on Fast Fourier Transforms. rattle, .aware Computer Security Research. 2007

[6] Mohammad Nazmul Haque, Mohammad Shorif Uddin: Accelerating Fast Fourier Transformation for Image Processing using Graphics Processing Unit. *Journal of Emerging Trends in Computing and Information Sciences*, 367-375, 2011.

[7] CUFFT Library. Programming Guide. NVIDIA Corp. 2012

[8] J. J. Dongarra: Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment. *Technology and Science of Informatics*, 3(5): 317-321, 1984.