Efficient High-Precision Integer Multiplication on the GPU

Journal Title
XX(X):1–11

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Abstract

The multiplication of large integers, which has many applications in computer science, is an operation that can be expressed as a polynomial multiplication followed by a carry normalization. This work develops two approaches for an efficient polynomial multiplication. **One approach is based on tiling the classical convolution algorithm, but taking advantage of new CUDA architectures, a novelty approach to compute the multiplication using integers without lossless of accuracy. The other one is based on the Strassen's algorithm, an algorithm that multiplies large polynomials using the FFT operation, but adapting the fastest FFT libraries for current GPUs and working on the complex field.** Previous studies reported that the Strassen's algorithm is an effective implementation for "large enough" integers on GPUs. Additionally, most of previous studies do not deepen in the implementation of the carry normalization, but this work describes a parallel implementation for this operation. Our results show the efficiency of our FFT approach for large polynomial sizes, whereas our tiling approach is very efficient for shorter sizes. Furthermore, our FFT approach turns out to be up to $2.74x$ for a single batch with respect to others implementations that use the CuFFT library.

Keywords

large integers, multiplication, FFT, GPU, CUDA

Introduction

Multiplying large integers is a common operation in many applications. Especially, many cryptography algorithms require operating on very large subsets of the integer numbers, such as the public-key cryptography, which employ arithmetic with hundreds of digits Rivest et al. (1978). Additionally, it is also frequently used to render fractal images at high magnification, such as those found in Mandelbrot set Brooks and Matelski (1978).

The classical vector multiplication has $O(N^2)$ complexity, where N is the number of digits. By using the Strassen FFT multiplication algorithm Schönhage and Strassen (1971), which has $O(NlogN(loglogN))$ complexity, the time is significantly reduced. This algorithm is derived from the fact that any integer multiplication can be expressed as a polynomial product, called vector convolution, followed by a carry normalization.

Graphics processing units (GPUs) can accelerate this computation using their massive parallelism capabilities. The GPU architectures provide considerable arithmetic processing performance, and they have become a natural choice for low-cost vector processing, since their development cost is amortized over a larger market compared with other dedicated parallel scientific architectures.

The motivation of this work comes from the fact that previous studies about high-precision integer multiplication were designed for outdated GPU architectures, using GPU libraries which are not the most efficient anymore. **Also, they are mostly focused on the Strassen FFT algorithm on the finite field. However, it is crucial to know the polynomial size where the Strassen FFT algorithm starts to run faster than other algorithms, and this size threshold varies from one architecture to others. Furthermore, it is also interesting to find out alternatives for different sizes as well as analyzing the numerical stability of implementation based on float and double numbers.** Additionally, most of previous works do not deepen in the parallel implementation of the carry normalization.

Specifically, this works presents two efficient approaches for the polynomial multiplication of large integers on GPU. On the one hand, one approach based on the Strassen FFT algorithm on current GPUs. In order to provide an efficient FFT operation, this proposal uses the approach presented in Diéguez et al. (2018), an efficient FFT proposal for large-size problems called the *ID-FFT* library, the fastest one to the best of our knowledge. On the other hand, a divide-and-conquer approach of the classical convolution operation on a GPU, which is based on tiling, is also developed. This new approach takes advantage of the recent improvements in the GPU memory bandwidth and the performance of the new atomic operations **in order to provide a fast and safe high precision integer multiplication with no numerical issues**. Finally, this work also provides a parallel approach of the carry normalization applied after the polynomial multiplication, which is based on a two-level hierarchical adder.

Related Work

There are some serial libraries and frameworks which perform large integer multiplications. For example, Microsoft introduced the BigInteger type in .NET 4.0 to compute large integers big (2010), which has no upper or lower bounds. Additionally, IntX int (2015) is a large precision integer library with fast multiplication based on the Hartley Transform Boussakta and Holt (1988). The GNU MP Library gnu (2016) also includes fast calculation algorithms for arbitrary precision arithmetic. There are also several software packages to compute symbolically with polynomials and matrices, such as Linbox lin (2017), MAGMA Bosma et al. (1997) and NTL ntl (2015), although most of them are dedicated to serial implementation in the case of polynomials. In Emerencia (2007), the FFT multiplication is implemented and compared with the normal multiplication. A mathematical review about computing the product of two large integers is analyzed in Bernstein (2001) and Zuras (1994); while a more detailed description of integer and floating-point arithmetic is presented in Brent and Zimmermann (2010). In (Toom 1963), the complexity of a scheme of functional elements, which computes the binary digits of the product of two N-digit integers, is studied. Additionally, an implementation of Schönhage-Strassen's algorithm within the GMP library is introduced in Gaudry et al. (2007) for old x86 architectures, which uses different techniques such as fast arithmetic modulo and Mersenne transforms. A more recent study is analyzed in Harvey and der Hoeven (2016), in which authors achieved an optimized variant of a FFT-based algorithm (Cooley-Tukey method followed by Bluestein's chirp transformation and the Kronecker substitution).

In the case of GPUs, to which this paper is devoted, there are several CUDA implementations of large integer multiplication. All of them are only focused on the Strassen FFT approach, ignoring any other approximation to work around the problem. Additionally, both the architectures and libraries employed in these works are completely outdated. The work presented in Emmart and Weems (2010) worked with the CUDA Fermi architecture, whereas the implementation of Zhao (2010) is even previous to Fermi. In these works, many decisions were taken based on the latency and efficiency of some operations, which have been completely improved in current architectures. Other GPU implementations can be found in Kitano and Fujimoto (2014) and Maza and Pan (2010), although they were also tested on outdated CUDA SKDs and architectures. In Bantikyan (2014), a fast integer multiplication based on the cuFFT library CUF (2012) is implemented, surpassing any other previous framework implementations.

The Strassen FFT Multiplication Algorithm

The Strassen FFT multiplication algorithm Schönhage and Strassen (1971) is based on the polynomial multiplication. Any number in base x can be decomposed into a polynomial coefficient vector. A polynomial p is described by its coefficient vector $a = [a_0, a_1, \cdots, a_{N-1}]$ as follows:

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

where x is the base of the polynomial, $p(x)$ is the evaluation of the polynomial for base x and N the number of digits of the number; i.e., the size of the polynomial. For example, considering the integer 54321, its polynomial form using $x = 10$ would be $a = [1, 2, 3, 4, 5]$ or $p(x) = 1 + 2x + 3x^2 + 4x^3 + 5x^4$. Multiplying two polynomials results in a third polynomial of size $2 \cdot N$, and this process is called vector convolution.

According to the convolution theorem, if c is the convolution of two input vectors a and b, $c = a \cdot b$, then the Discrete Fourier Transform (DFT) of c is equal to the pairwise multiplication of the DFT transform of each input vector, $DFT(c) = DFT(a)DFT(b)$, where *pairwise multiplication* means multiplying the vectors in pairs, element by element. Thus, the c vector can be also obtained as the Inverse Discrete Fourier Transform (IDFT) of this pairwise multiplication:

$$c = IDFT(DFT(a)DFT(b))$$

Given two input values, a and b, each one with N and M digits respectively, the Strassen FFT algorithm performs as follows. Firstly, the integers are represented as polynomials in their coefficient-form representation, $a = a_0, a_1, \cdots, a_{N-1}$ and $b = b_0, b_1, \cdots, b_{M-1}$. If the input vectors do not have the same length, $M < N$, the shortest one is filled with zeros until $M = N$. Once the integers are represented as polynomials, the convolution theorem is applied. In order to easily compute the DFT of each vector, the Fast Fourier Transform is performed for each vector. After that, the pairwise multiplication is applied as well as the inverse FFT. Finally, the coefficients have to be normalized to the same base as the one in which the integer is represented (looping to propagate the carry).

The ID-FFT library for GPUs

On the one hand, Graphics Processing Units (GPUs) are massively parallel processors with thousands of parallel active threads. Considering CUDA GPU architectures, logical threadblocks are distributed by the hardware among the available Streaming Multiprocessors (SM) and, depending on the amount of required resources, each SM may be able to simultaneously execute several threadblocks. Each threadblock executes threads which are grouped into SIMD units of width 32, called warps, and each warp executes instructions in lockstep. The memory hierarchy is organized into a large global memory accessible by all threads, smaller and faster shared memories for each threadblock, and local registers for each thread. The shuffle instructions are a warp-wide exchange mechanism that communicates threads within a warp across registers. If the threads do not belong to the same warp, but they belong to the shame threadblock, the fastest communication mechanism is the shared memory; otherwise, the communication is

performed accross global memory.

The Pascal architecture improves previous architectures' features in many aspects: more double precision units, larger L2 cache, larger shared memory in each SM, HBM2 memory instead of GDDR5, native support for 32-bit shared-memory atomics (already introduced in Maxwell) and 64-bit global-memory atomics. The most recent CUDA architecture is called Volta and delivers the highest GPU performance so far. This architecture is focused on deep learning, and improves Pascal with a larger L2 cache and larger shared memory per SM, more NVLINK bandwidth and introduces an independent thread scheduling, where each thread has its own program counter, breaking the SIMD warp philosophy of previous architectures.

On the other hand, the *ID-FFT* * library Diéguez et al. (2018) Lobeiras et al. (2016) is, to the best of our knowledge, the fastest library that performs the FFT for large-problem sizes on a GPU. This recent library has surpassed the performance of any other GPU library when solving several large-size FFT problems simultaneously.

The performance of this library comes from using a coalescing-friendly global memory pattern, which efficiently uses the memory bandwidth, and is built with a set of parameterized kernels. A parameterized kernel is a template skeleton that receives the optimal performance values for each architecture as template arguments. These arguments specify the optimal performance values which maximize the SM parallelism for each data size and target architecture (number of threads per block, amount of registers per thread, shared memory per block and number of kernels employed). Thus, each supported architecture has a static table in the code, where each entry represents one problem size with the corresponding parameter values that maximize performance for the given size. As kernels were written as templates, all instances of the table are generated at compile-time with the specified parameter values, and then, during the execution, the corresponding instance is loaded depending on the given problem size.

The CUDA FFT-based Multiplication Approach

In this approach, we have employed the *ID-FFT* library to compute the FFT operation of the Strassen algorithm for multiplying large integers, in a similar way as the authors of Bantikyan (2014) did with the cuFFT library.

When using the Strassen algorithm, most of the existing implementations use the finite field $\mathbb{Z}/p\mathbb{Z}$, with prime p, instead of the complex field $\mathbb{C}$, since the error analysis is easier. In that case, it is desirable to use a p number which minimizes the modulo operation latency, and Fermat prime numbers are chosen in most cases. Nevertheless, there are several important restrictions with the finite field. Being x the base and n the size of the FFT in the finite field:

- The field $\mathbb{Z}/p\mathbb{Z}$ requires a n^{th} root of unity.
- The maximum value must fit in the field, i.e., $n/2(x-1)^2 < p$.
- Multiplying in $\mathbb{Z}/p\mathbb{Z}$ must be modulo p, thus the existence of a fast modulo p operator is desirable (like Montgomery reduction algorithm).

Traditionally, the implementations on the finite field have been less efficient than the implementation on the complex field, as different modulo operations are required. Taking previous restrictions into account, our FFT approach has been designed to work with the complex field $\mathbb{C}$. Additionally, this work uses the ID-FFT implementation, which only supports the $\mathbb{C}$ complex numbers in base $x = 10$, but it may be extended to other base. Two different proposals were developed following this approach: the *Complex-ID* proposal and the *Real-ID* proposal.

The floating-point design forces taking care of the numerical accuracy. In order to achieve accurate results, the N-digit operands are transformed into polynomial coefficient vectors with base $x = 10$, as explained before. Specifically, each polynomial coefficient is a number in $[0, 9]$; thus, before the convolution, each N-digit operand is transformed into N subwords, each composed of 1 digit. Decreasing the number of computing operations is possible by increasing the number of digits per subword to k digits, i.e., reducing the number of subwords (N/k). However, this choice would also hinder the accuracy of the result owing to the use of floating-point data in the FFT, as the result section presents. Thanks to GPU computing, which allows us achieving massive data parallelism, it is possible to work with N subwords of 1-digit at high-performance; i.e., using $k = 1$. After the multiplication, each subword must contain a number from 0 to $(N/k) \cdot (x^k - 1) \cdot (x^k - 1)$, with $x = 10$ and $k = 1$ in our work. However, as the FFT convolution is employed for this purpose, this number is a floating-point value with decimal digits, which is rounded to be an integer. This number is later normalized to a 1-digit coefficient in the carry propagation operation.

The Complex-ID Proposal

This proposal is tagged as *Complex-ID* in the result section. The steps explained previously in *"The Strassen FFT Multiplication Algorithm"* section are performed here for the vectors a and b of size N, using the ID-FFT implementation for complex numbers (*Complex-ID* function). The resulting polynomial of multiplying a and b, c, will have a degree two times greater than the highest degree of a and b; thus the size of c is $2 \cdot N$. Before performing the forward FFT, a and b are extended up to $2 \cdot N$, padding with zeros. The imaginary part of each element is set to zero, and the coefficients are assigned to the real part. Once the FFT is applied for each input vector, both signals are pairwise multiplied, where x represents the real part of the number and y is its imaginary part:

$$c.x = a.x * b.x - a.y * b.y; \quad (2)$$

$$c.y = a.x * b.y + a.y * b.x; \quad (3)$$

*Available at http://bplg.des.udc.es

At this point, the inverse FFT is performed for the resulting vector from the pairwise multiplication. It should be noted that a pairwise multiplication kernel has been additionally developed to multiply the elements, which are already in the GPU memory.

The Real-ID Proposal

The previous proposal wastes half of the memory bandwidth carrying zeros in the imaginary part of each number. In this proposal, tagged as *Real-ID* in the result section, the FFT operation of the ID-FFT library is extended with real-number support. **There are many approaches to perform the real FFT efficiently Sorensen (1987), but we use the one that packs the signal in a vector with half of the size, reading each two consecutive real values as a single complex number, as CUF (2012) does**.

Given a signal of real data x and its transform y the following symmetry property is met:

$$y_k = \overline{y_{N-k}},\ 1 \leq k \leq N/2 \tag{4}$$

Where $\overline{y_{N-k}}$ is the complex conjugate, such that $\overline{a+bj} = a-bj$. The values in $[y_1 \cdots y_{N/2-1}]$ have an imaginary component but, when N is even, both y_0 and $y_{N/2}$ are pure real numbers. Observe that, in consequence, half of the information in y is redundant. Hence, x can be processed as a complex signal of half the length where:

$$x'_k = x_k + x_{2k+1}j \tag{5}$$

Then $[y_0 \cdots y_{N/2-1}]$ is obtained as:

$$y_k = \frac{1}{2}(z_k + \overline{z_{N/2-k}}) - \frac{j}{2}e^{-2\pi k/N}(z_k - \overline{z_{N/2-k}}) \tag{6}$$

with z as the complex transform of the signal x'_k. Furthermore, $y_N/2 = Re(z_0) + jIm(z_0)$. The remaining values are easily obtained using the enunciated symmetry property.

To achieve this end, two new functions are developed, a Real-to-Complex (R2C) function for the Forward FFT, and a Complex-to-Real (C2R) function for the Inverse FFT. To do this, the real signal is packed in a vector with half of the size (reading each two consecutive real values as a single complex number), and then the *Complex-ID* function performs the transform of this half-size signal. After this, a post-processing stage is used to combine the output and unpack the data, consuming half of the memory bandwidth with respect to the previous proposal. This can be achieved thanks to the complex conjugate property, where half of the information in the transformed signal is redundant. It should be observed that the computation of the post-processing stage is performed in an additional kernel after (before) the forward (inverse) FFT; thus, in addition to the kernels given by the ID-FFT library, a kernel which computes the post-processing stage has had to be developed.

```
1 for( k from 0 to 2*N-1)
2   c[k]=0
3 for (i from 0 to N-1)
4   for (j from 0 to N-1)
5     c[i+j]+= a[i]*b[j]
```

Figure 1. Pseudocode of the vector convolution operation

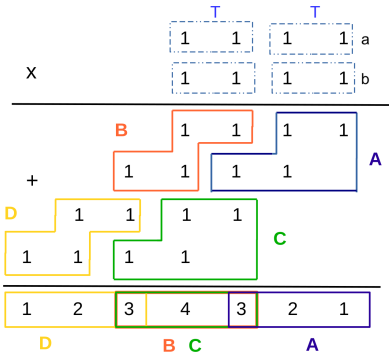


Figure 2. Classical multiplication operation in tiles of size T=2 on the input vectors a and b, where A,B,C and D represent the data processed by each computing unit

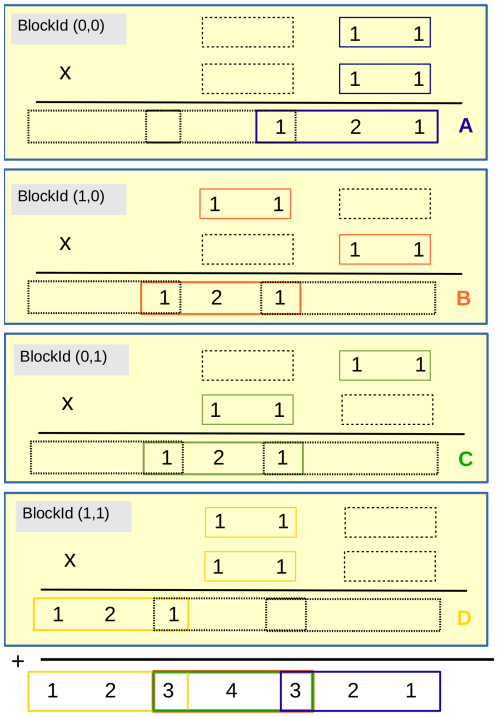


Figure 3. GPU implementation of the tiled multiplication where $N = 4$ and $T = 2$.

The CUDA Tiling Multiplication Approach

In this section, a new approach for computing an efficient multiplication of two large integers on a GPU is proposed. This new approach is based on the classical vector convolution algorithm and avoids working with the Discrete Fourier Transform.

The vector convolution algorithm

Although the classical algorithm of the polynomial multiplication seems sequential, as Figure 1 shows, it is possible to apply a divide-and-conquer strategy to compute the multiplication in parallel. This approach, tagged as *Tiling-based* in the result section, divides the computation of the c reduction (line 5 in pseudo-code) through several data blocks, where each data block works with tiles of size T. Specifically, each data block computes $2 \times T - 1$ elements of c, taking T elements from vector a and T elements from vector b as inputs. Then, each data block has to integrate its partial result with the others, in a sequential reduction, in order to obtain the overall result; whereas the number of data blocks is given by $(N/T) \times (N/T)$. From a computer architecture perspective, each data block is computed by one computing unit of the target architecture and the optimal value of the tile size, T, also depends on the given architecture.

Figure 2 depicts an example of this approach with $N = 4$ and $T = 2$. There are $2 \times 2 = 4$ computing units, where each computing unit computes 3 elements of the solution, reading 2 elements from a, and 2 from b **(see convolution of each computing unit as *A,B,C* and *D* in figure)**.

CUDA implementation

When this approach is implemented on a GPU, each computing unit corresponds with a threadblock. Each threadblock works with T elements from a and T from b. The whole computation is performed in a single kernel invocation and the overall result is calculated by integrating the partial results with atomic instructions in global memory.

Figure 3 depicts the work performed by each threadblock. Each pair of T elements from input vectors is assigned to one threadblock. Then, each threadblock divides the computation of the multiplication among its threads. To do this computation, this approach does not use shared memory, since all exchanges are performed by shuffle instructions. The partial result of each threadblock is stored in private registers of its threads, and is carried to its positions in the result array, performing the corresponding reduction with other threadblocks in global memory.

Nevertheless, these operations may be a big bottleneck for large-size inputs. It should be noted that each memory location is atomically accessed as many times as the number of tiles. Thus, large problem sizes will suffer memory contention due to atomics, despite of the new improvements on these operations in new architectures. Since higher number of tiles implies higher contention, and the number of tiles is equal to the number of threadblocks employed, each block must be executed with the greatest number of threads possible and each thread must be on charge of the maximum number of elements possible in order to reduce the number of threadblocks.

In order to find the suitable T value, an exhaustive search is empirically computed for each supported architecture, finding out its optimal value. The optimal T value is affected by the GPU global memory bandwidth, its SM parallelism, the performance of the global memory atomics in that GPU and the size N of the input vectors. A similar tuning methodology as the *FFT-ID* one has been employed for the Tiling approach, see previous *"The ID-FFT library for GPUs"* section, where kernels are written as templates and receive the optimal performance values which maximize the SM parallelism at compile-time.

```
1  CarryPropagation(srcVector, dstVector)
2
3  carry:=0
4  for ( i from 0 to srcVector.length)
5    sum:= carry+srcVector[i]
6    mod:= sum \% 10
7    dstVector.Add( mod)
8    carry:= sum/10
9  while(carry>0)
10   if(carry.lenght >1)
11     index:=carry.lenght-1
12   else
13     index:= 0
14   dstVector.Add(carry[index])
15   carry:= carry /10
```

Figure 4. Pseudocode of the carry-propagation operation for large integer multiplication.

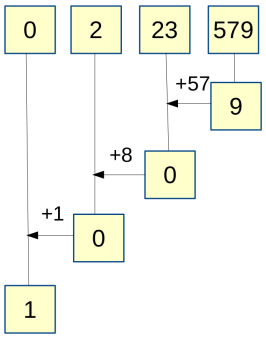


Figure 5. Carry propagation: Serial implementation

The Carry Normalization

After multiplying the vector inputs, each element in the output vector is the result of the corresponding product, and might be composed of several digits. In order to obtain the final result, each element should perform the modulo operation and propagate the carry to the next more significant elements. This implies that each element receives a carry-in from the less significant elements, performs the modulo operation and propagates its carry-out to the more significant elements. This process is called *Carry Normalization* or *Carry Propagation* and its pseudo-code is illustrated in Figure 4 for the base $x = 10$.

In contrast with traditional adders, where the carry flag is a single digit used to indicate when a carry-out has been generated and is propagated to the immediately adjacent more significant position, the carry accumulator here (line 8) may be composed of several digits; i.e., the carry must be propagated to several more-significant elements. This fact limits the parallelization of the algorithm using a

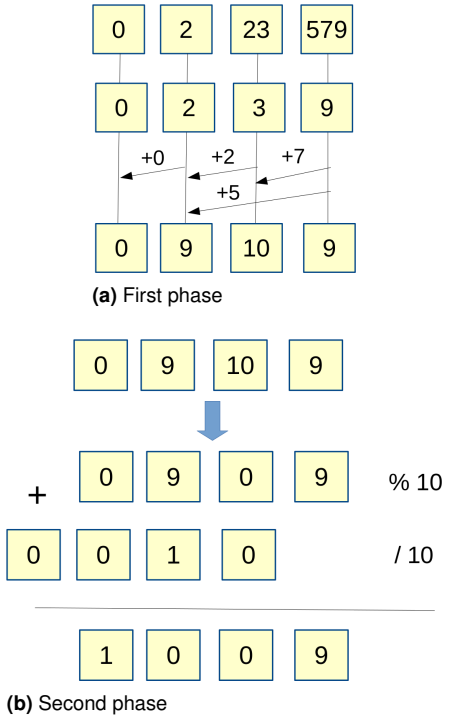


Figure 6. Parallel carry propagation design

carry look-ahead scheme, since this scheme is designed to propagate a single-digit carry, not a multiple-digit carry, as Figure 5 shows for an example, which is the result of multiplying two polynomials, and whose polynomial form is $a = [579, 23, 2, 0]$. **This hierarchical carry look-ahead idea is implemented in different hardware adders in VLSI designs. In Emmart and Weems (2010), the authors transform this hardware behavior into a GPU equivalent software operation, which is, to the best of our knowledge, the only GPU implementation of this operation. We have improved this GPU implementation by introducing shuffle instructions and two different phases (two kernel calls).**

In the first phase, each element will be normalized to a number of two digits in base 10. To do this, considering integer type codification, each element may be composed of up to 10 digits; keeping the first digit as the element's value and the remaining 9 digits are assigned as the element's carry-out. Therefore, each digit of the element's carry-out has to be propagated to the corresponding adjacent more-significant elements, nine at most. In other words, each element receives a single-digit carry-in (a number in $[0, 9]$) from 9 elements at most. After adding the single-digit carry-in from its adjacent less-significant elements to itself, each element will be composed of two digits at most: considering the extreme scenario where the element's value is 9 and the nine carry-ins received are also 9, the total addition would be 90, two digits. The implementation of this idea is depicted in Figure 6 (a). Firstly, every element performs the modulo operation at a time, obtaining each element's value, $[9, 3, 2, 0]$ in the example. After this, the generated carry-outs, $[57, 2, 0, 0]$, have to be propagated to the next elements. Thus, each element propagates its generated carry-out to its adjacent elements, using shared memory, where each digit of the carry-out is sent to the corresponding adjacent element. In the example, the first element sends 7 to the second element and 5 to the third element; whereas the second element sends 2 to the third one and the third element sends 0 to the forth one.

	Pascal Platform	*Volta Platform*
CPU	Xeon E5-2630	Xeon E5-2698
Memory	256 GB	512 GB
GPU	NVIDIA Pascal P100	NVIDIA Volta V100
Driver	375.51, SDK 8.0	384.81, SDK 9.0

Table 1. Description of the computing platforms employed

In the second phase, the vector is decomposed into two vectors: the one with the result of applying the modulo operation, and another one with the corresponding single-digit carry-out generated. The final result is built with their addition. As these two vectors are composed of single-digit elements, a carry look-ahead scheme can be applied now for their addition. Figure 6 (b) shows the second phase of the implementation. Specifically, in order to compute the carry look-ahead schema of the second phase, let us define $critical[i]$ as a boolean array where the i^{th} bit is set if the i^{th} element is critical; i.e., sensitive to produce a carry-out if and only if there is a carry-in (i.e., i^{th} element is the digit 9). Also, let us define $c[i]$ as a boolean array where the i^{th} bit is set if the i^{th} element generates a carry-out. Then, the carry look-ahead function is as follows:

$$carry[i] = (c[i]) \ \textbf{or} \ (critical[i] \ \ and \ \ c[i-1]) \ \textbf{or} \ (critical[i] \ \ and \ \ critical[i-1] \ \ and \ \ c[i-2]) \ \textbf{or} \ \cdots$$

Although this expression seems very slow to evaluate, it can be replaced by integers instead of boolean arrays, getting the following expression that can be evaluated in a single step:

$$carry = ((c << 1) + carry - in + critical) \ \textbf{xor} \ critical$$

where carry-in is a single carry bit from the previous block of elements. Previous numerical expression can be evaluated at different levels: thread registers, warp and threadblock until reaching the final result, as explained in Emmart and Weems (2010).

Experimental Results

In this section, an analysis of the results is presented. This analysis is split in two studies: a numerical study for the FFT-based approach, which uses floating point precision, and a performance study for the two approaches.

Numerical analysis

While the classical algorithm and the finite-field FFT-based approaches work with exact computations, the FFT-based implementations on the Complex field can show some numerical inaccuracy owing to the use of floating-point operations. Most of this numerical inexactness can be solved

executing a round function after the calculation, as see below. It should be observed that the *Tiling-based* approach already works with integers, thus there is no numerical inaccuracy in its execution.

If the FFT-based multiplication is employed for multiplying large integers, then two N-digit integers are multiplied in base x, and each operand is transformed into N/k elements of k digits, defining the FFT input vectors for a given k. The resulting vector after the FFT multiplication is composed of $2 \cdot N/k$ elements, and each element (before normalizing each resulting element to k digits) contains a value in the range $[0, (N/k) * (x^k - 1) * (x^k - 1)]$. This value is represented in either 32 or 64 bits (FP32 or FP64 precision). If the resulting value after the FFT multiplication is large, or it has a large decimal part, certain accuracy can be lost since this value is represented in only 32 or 64 bits.

Considering that each value after the inverse FFT must be an integer, we take the nearest integer value, the numerical error α on the inverse FFT process can be proved to be

$$\alpha \leq 6N^2x^2log(N)\epsilon \tag{7}$$

where N is the digit size, x is the chosen base, and ϵ is the *machine epsilon*, the upper bound due to rounding in floating point, which is around $\exp -16$ for double precision ($\exp -8$ when single floating point) as explained at Gourdon and Sebah (2001). But this bounds the worst case. In the practice, the numerical stability of the FFT is good and the bound $\alpha = O(Nx^2\epsilon)$ is fulfilled. To guarantee the numerical accuracy, α must be less than 0.5, thus choosing a base x with a small k value usually suffices to have the desired accuracy. In this case $x = 10$ and $k = 1$, which allows to maximize the value of N for a given α.

Therefore, as each resulting element is a float in the FFT multiplication, we need that the absolute individual error on each element to be less than 0.5, as coefficients (elements) are rounded to the nearest integer before normalization. Table 2 shows the MAE (Mean Absolute Error) error for different operand sizes, as well as the number of elements for whose absolute individual error is greater than 0.5 (counter metric). **The accuracy formula of the previous paragraph bounds the worst case scenario, but this table represents the results for a regular computation which sets to '9' all digits of the two input vectors. Many other random generated inputs have been tested, with similar accuracy. In practice and as table shows, FP32 approaches are not accurate for extremely large integers only. In contrast, FP64 approaches show really high precision, even at extremely large sizes. However, it should be pointed out that our Tiling-approach accurately computes the result for such cases.**

Another analysis is given in Table 3 for different k values and N-size operands, with base $x = 10$, using FP64 and also setting all input values to '9'. While $k = 1$ has no accuracy issues, the analysis is different for $k = 2$. The

N	*FP32 Error*		*FP64 Error*	
	Err Counter	*MAE*	*Err Counter*	*MAE*
4096	0	0.000173	0	0.000001
8192	0	0.000573	0	0.000001
16384	0	0.001279	0	0.000001
32768	0	0.002766	0	0.000001
65536	0	0.005246	0	0.000001
131072	0	0.009930	0	0.000001
262144	0	0.034507	0	0.000001
524288	0	0.058532	0	0.000001
1048576	0	0.143468	0	0.000001
2097152	12268	0.264665	0	0.000001
4194304	2377658	0.563110	0	0.000002
8388608	6157308	1.057919	0	0.000003

Table 2. MAE absolute error and individual error counter for our FFT proposals when setting all operand digits to 9.

N	*MAE k=1*	*MAE k=2*	*MAE k=4*
32768	0.000001	0.00013	1.32986
65536	0.000001	0.00013	1.33007
131072	0.000001	0.00013	1.33023
262144	0.000001	0.00013	1.3305

Table 3. MAE and individual error counter for different k values in FP64.

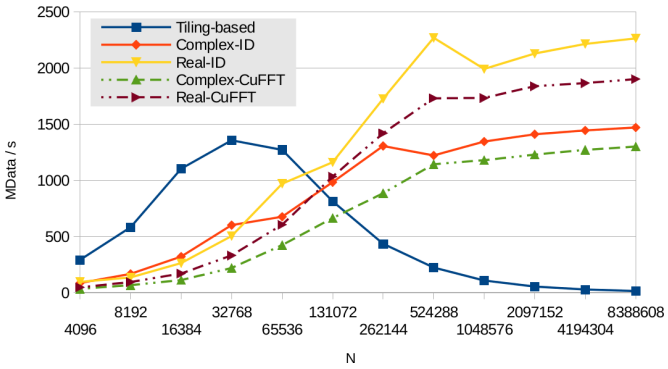
MAE for $k = 2$ increases, although there is no element with an individual error greater than 0.5 yet. The problem starts with $k = 4$, since there are many elements with an individual error > 0.5, invalidating this configuration as an option to our FFT proposal. As our proposal should produce the most accurate result, we use $k = 1$ for all problem sizes.

Performance analysis

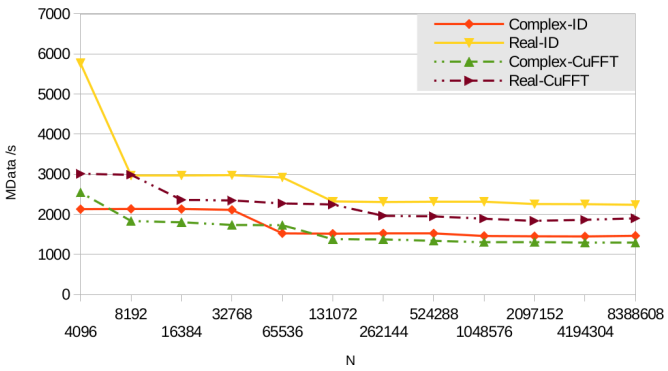
The following results were taken in the computing platforms shown in Table 1. All data elements were in the GPU memory prior to the GPU execution, thus CPU-GPU memory transfers are not included in the metrics. Specifically, the $MData/s$ metric gives the number of digits (in millions) calculated by second for the resulting polynomial.

For each kernel launched in each proposal and architecture, the optimal performance parameters that maximize the GPU parallelism have been found empirically following the tuning methodology mentioned in previous *"The ID-FFT library for GPUs"* section and explained at Diéguez et al. (2018) in greater detail. It should be noted that the most important performance factor is to get the maximum memory bandwidth in the case of the FFT proposals, and to minimize the number of atomic operations in the case of the Tiling approach. In order to compare our approaches with other implementation, we have also implemented the FFT-based approach using the CuFFT library to perform the FFTs, since authors of Bantikyan (2014) claimed that it surpasses any other state-of-the-art implementation.

The Pascal Platform Figure 7 (a) shows the performance of our approaches in the Pascal Platform, using FP32 for the FFT-based proposals, when executing a single-batch



(a) Single-batch execution



(b) Multi-batch execution

Figure 7. Performance comparison for our FP32 approaches in the Pascal Architecture

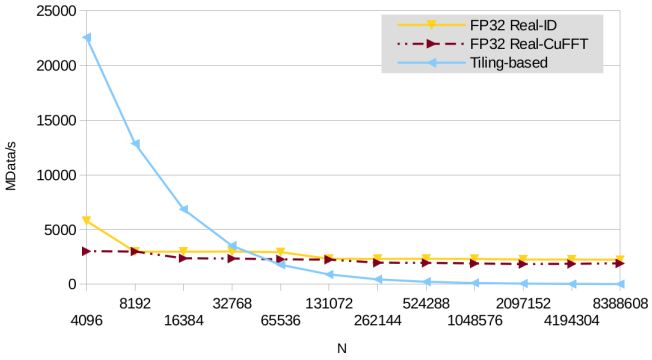
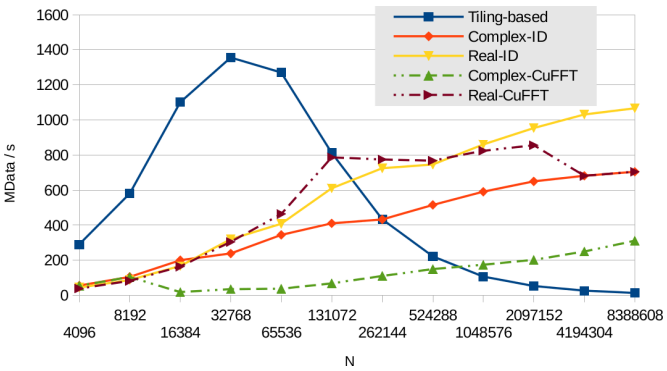


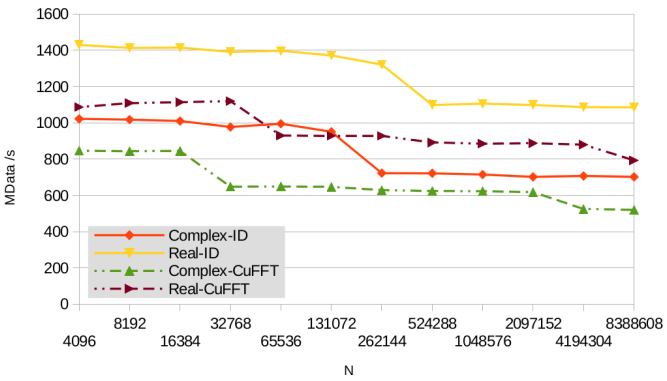
Figure 8. Performance comparison for the multi-batch GPU-tiling proposal on the Pascal Architecture

problem. Our *Real-ID* proposal is up to 2.08x compared with the real implementation of the *Real-CuFFT* proposal; whereas the *Complex-ID* proposal is up to 2.74x faster than the *Complex-CuFFT* proposal. The *Tiling-based* proposal outperforms the *Real-ID* proposal while $N \leq 65536$. In the case of $N = 8192$, the *Tiling-based* is 4.34x faster than the *Real-ID* proposal and 6.23x with respect to the *Real-CuFFT* proposal.

However, the ID-FFT library has been designed to solve several batch problems simultaneously. In order to perform a multi-batch execution, 2^{24} digits are allocated for the resulting polynomials where the number of batches, G, is equal to $G = \frac{2^{24}}{2 \cdot N}$, being N the size of each input vector. This multi-batch case is shown in Figure 7 (b), where the Tiling-approach is not shown for the sake of appearance



(a) Single-batch execution



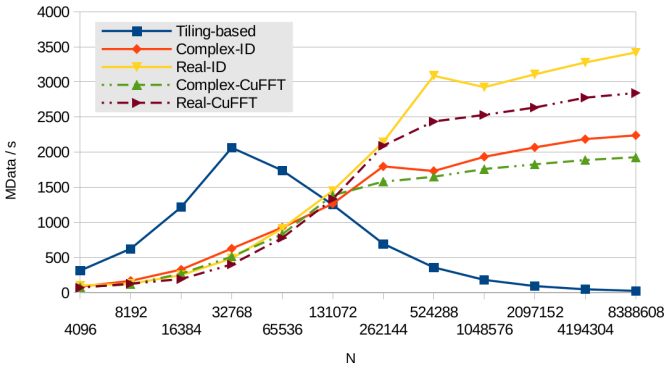
(b) Multi-batch execution

Figure 9. Performance comparison for our FP64 approaches in the Pascal Architecture

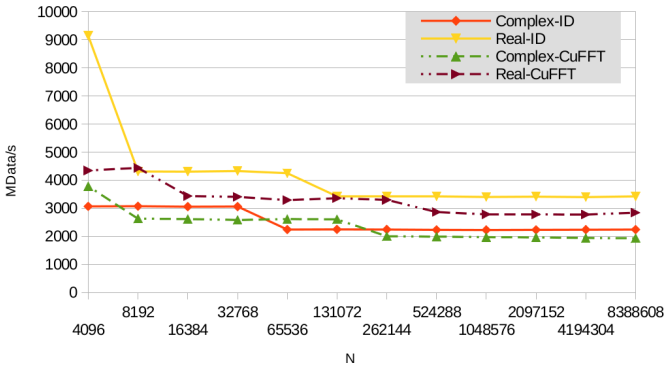
due to the range of the axis and the magnitude of the approach values. Our *Real-ID* proposal is up to 1.91x faster compared with the real implementation of the *Real-CuFFT* proposal; whereas the *Complex-ID* proposal is up to 1.21x faster than the *Complex-CuFFT* proposal. The performance of the multi-batch Tiling approach is shown in Figure 8 and compared against the single-precision FFT-based approaches for this architecture. It outperforms the *Real-ID* proposal when $N \leq 32768$. In the case of $N = 4096$, the *Tiling-based* is 3.92x faster than the *Real-ID* proposal and 7.49x faster with respect to the *Real-CuFFT* proposal.

Figure 9 performs the same comparative for the double precision execution. As the FFT function is a memory-bound operation, the achieved performance is the half of the FP32-proposals performance. In the case of a single batch, the FP64 *Real-ID* proposal is up to 1.51x faster than the *Real-CuFFT* one; whereas up to 11x is obtained for the complex case, depending on the data point. In a multi-batch execution, the FP64 *Real-ID* proposal is up to 1.51x faster than the *Real-CuFFT* one; whereas 1.50x is obtained for the complex case.

The Volta Platform Figure 10 shows the performance analysis of our proposals in the Volta Platform, where the FFT-based proposals use FP32 datatypes. It should be observed that the Volta execution is 1.5x faster, on average, than the Pascal one for a single-batch execution. Again, this is because the memory-bound nature of the problem. The new generation of memory controllers in Volta provides



(a) Single-batch execution



(b) Multi-batch execution

Figure 10. Performance comparison for our FP32 approaches in the Volta Architecture

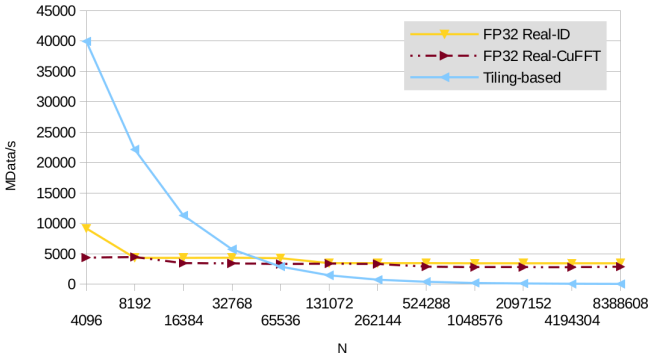


Figure 11. Performance comparison for the multi-batch GPU-tiling proposal on the Volta Architecture

1.5x delivered memory bandwidth with respect to the Pascal GP100. In this platform, the ID-FFT proposals continue surpassing the CuFFT-based ones, achieving, on average, 1.44x and 1.42x speedups for the case of real and complex numbers, respectively. In general terms, this architecture performance is the double than the Pascal one, as can be observed. Regarding the Tiling approach, the atomic operations still limit the performance for larger problem sizes in this architecture, although this approach is up to 5.2x faster than the *Real-ID* proposal and up to 6.33x with respect to the *Real-CuFFT* proposal. In the case of a multi-batch execution, the ID-FFT proposals continue surpassing the CuFFT-based ones, achieving, on average, 1.21x and 1.12x speedups for the case of real and complex numbers, respectively. Figure 11 shows the multi-batch *Tiling-based* proposal in this platform, which is is up to 3.91x faster than the *Real-ID* proposal and up to 7.48x with

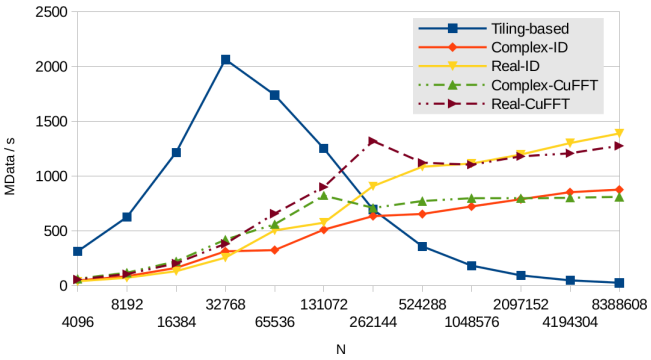


Figure 12. Performance comparison for the FP64 FFT-based proposals and the Tiling approach executing a single-batch in the Volta Architecture

respect to the *Real-CuFFT* proposal.

Figure 12 shows the results for double precision in this architecture, when executing a single-batch. In this case, the ID-FFT proposals are competitive when $N > 1048576$, whereas the CuFFT-based proposals run faster for smaller problem sizes. In the case of executing several batches, we have obtained inconsistent results for all the versions when compiling with compute capabilities 7.0 and CUDA 9.0. Specifically, extremely low $MData/s$ is achieved when several bidimensional threadblocks write double values in global memory. This issue has been reported, since the performance increases, until reaching expected values, when compiling with other compute capabilities for this architecture.

Results Discussion

On the one hand, the FFT-based approaches have some performance weaknesses, in addition to work with precision inaccuracy. Firstly, each invocation of the FFT operation launches several kernels, as well as the kernel invocation for the pairwise multiplication, with the corresponding performance lost. Additionally, the ID-FFT library consumes a huge amount of shared memory and shuffle instruction optimization is not possible Diéguez et al. (2018), affecting performance massively when working on double precision. Although new GPU architectures have higher theoretical performance and higher memory bandwidth, these two factors significantly limit the actual performance that can be achieved. Despite of these aspects, they show to be the most efficient implementation for the large problem sizes.

On the other hand, the Tiling approach performance depends too much on the atomic implementation efficiency, and although the atomic operations might be replaced by a reduce-pattern, this would imply the use of additional kernels and global memory. Specifically, the reduction of each element in a reduce-pattern would need an additional vector in memory with as many elements as the number of threadblocks the *Tiling-based* kernel is invoked. This amount of memory surpasses the memory availability of a single-GPU when solving large problem sizes. From CUDA 9 on, it is possible to perform this reduce operator in a single kernel and with no additional memory, using

the *gridSynchronize* global barrier. However, the number of invoked threadblocks must be less or equal than the number of resident active threadblocks, in the same manner as persistent threads do, but the *Tiling-based* kernel uses more than that number of threadblocks. Thus, the current implementation of the Tiling approach is the most efficient we have found.

Thus, analyzing the results obtained, we can conclude that the Tiling approach shows high performance when multiplying a small or medium number of digits, and the FFT-based approaches are the most suitable ones to compute large problem sizes. **However, the Tiling approach guarantees the numerical accuracy, whereas the FFT can only guarantee this accuracy for certain sizes**. Specifically for the single-batch execution, the Tiling approach should be used when $N \leq 65536$ for both Pascal and Volta architectures, whereas larger problem sizes should be solved with the FFT-based approaches. In the case of a multi-batch execution, the Tiling approach should be used when $N \leq 32768$. Although the FFT-based approaches suffer from some numerical inaccuracy, this work shows that it is very low and might be acceptable in most of the applications which use the large-integer multiplication. Also, this work demonstrates that the FFT-based approaches that use the ID-FFT library to compute the Fast Fourier Transform run faster than the ones which use the CuFFT library.

Previous results do not include an analysis of the carry performance in detail, since this operation is always executed after the convolution, despite of the algorithm used. Please, keep in mind that the parallel carry workload has represented half of the total execution time in all platforms, when executing the FFT approach. With respect to the serial implementation, the performance of the whole process for the GPU multiplication is one order of magnitude higher if we use the GPU carry approach rather than the CPU carry approach. Actually, the performance of the whole operation (convolution and carry) keeps constant for any input size when using the CPU carry, since the CPU carry implementation is clearly bounding the global performance. In summary, as the same carry operation is always executed in the last step, we have focused on the performance analysis of the convolution, which varies depending on the chosen algorithm; but it should be mentioned that our GPU carry implementation outperforms the CPU-implementation performance in one order of magnitude.

Conclusions

This work presents two approaches for computing an efficient multiplication of large integers on different GPU architectures: the FFT-based approach, which uses the Strassen-FFT algorithm, and the Tiling approach, which is based on the classical algorithm, partitioning the data vectors in tiles. These approaches are derived from the fact that any integer multiplication can be expressed as a polynomial product followed by a carry propagation. The FFT-based approach is focused on complex numbers, instead of on a finite-field, and uses the ID-FFT to implement the Strassen algorithm. This approach outperforms other implementations which use the CuFFT library by 2.74x in Pascal and 1.44x in Volta architectures. Additionally, a numerical accuracy analysis is performed, since this FFT-based approach works with complex numbers rather than finite fields. The FFT-based approaches are very suitable when dealing with extreme-large polynomials. Additionally, the Tiling approach is much more efficient when working with medium and large polynomials **compared to FFT approaches, but also guarantees the numerical stability for any digit size**. For each architecture and kind of execution (simple-batch or multi-batch) this work also provides the optimal algorithm which should be executed for each data point. Finally, a parallel implementation for the carry propagation is also given. This work fulfills the demands of computing efficiently the multiplication of large integers in current GPU architectures, providing two different alternatives **depending on the problem size and the required accuracy**.

Acknowledgements

This work is supported by the Ministry of Economy and Competitiveness of Spain, TIN2016-75845-P (AEI/FEDER, UE), by the Galician Government and FEDER funds under the Consolidation Program of Competitive Reference Groups (GRC2013-055) as well as under the Consolidation Programme of Competitive Research Units [Ref. R2014/049 and Ref. R2016/037]; and by the FPU Program of the Ministry of Education of Spain (FPU14/02801). It is also partially supported by JST CREST [JPMJCR1303 and JPMJCR1687] and NVIDIA GPU Center of Excellence; and conducted as research activities of AIST-TokyoTech Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL).

References

(2010) *.NET BigInteger Library*. Microsoft. Available at https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx.

(2012) *CUDA CUFFT Library*. NVIDIA. V5.0.

(2015) *CodePlex IntX Library*. CodePlex Open Source Project. Available at https://intx.codeplex.com.

(2015) *NTL: A Library for doing Number Theory*. Victor Shoup. Available at http://www.shoup.net/ntl/.

(2016) *GNU Multiple Precision Arithmetic Library*. GNU Open Source Project. Available at https://gmplib.org.

Sorensen H and Jones D and Heideman M and Burrus C(1987) Real-Valued Fast Fourier Transform Algorithms. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(6):849–863.

(2017) *LinBox - C++ library for exact, high-performance linear algebra*. Bastien Vialla, Jean-Guillaume Dumas. Available at https://linalg.org.

Bantikyan H (2014) Big Integer Multiplication with CUDA FFT (cuFFT) Library. 11.

Bernstein D (2001) Multidigit multiplication for mathematicians. URL https://cr.yp.to/papers/m3.pdf.

Bosma W, Cannon J and Playoust C (1997) The Magma algebra system. I. The user language. *J. Symbolic Comput.* 24(3-4): 235–265.

Boussakta S and Holt AGJ (1988) Fast multidimensional discrete Hartley transform using fermat number transform. *IEEE Proceedings of Electronic Circuits and Systems* 135(6): 253–257.

Brent R and Zimmermann P (2010) *Modern Computer Arithmetic*. Cambridge University Press.

Brooks R and Matelski J (1978) The dynamics of 2-generator subgroups of psl(2,c). pp. 65–71.

Diéguez AP, Amor M, Lobeiras J and Doallo R (2018) Solving Large Problem Sizes of Index-Digit Algorithms on GPU: FFT and Tridiagonal System Solvers. *IEEE Transactions on Computers* 67(1): 86–101.

Emerencia A (2007) Multiplying Huge Integers Using Fourier Transforms. ICS Project. University of Groningen. URL http://www.cs.rug.nl/~ando/pdfs/Ando_Emerencia_multiplying_huge_integers_using_fourier_transforms_paper.pdf.

Emmart N and Weems C (2010) High precision integer addition, subtraction and multiplication with a graphics processing unit. *Parallel Processing Letters* 20(04): 293–306.

Gaudry P, Kruppa A and Zimmermann P (2007) A gmp-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In: *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation (ISSAC'07)*. ACM, pp. 167–174.

Gourdon X and Sebah P (2001) FFT based multiplication of large numbers. *Numbers, constants and computation* 2(17): 1–6.

Harvey L and der Hoeven V (2016) Even faster integer multiplication. *J. Complex.* 36(C): 1–30.

Kitano K and Fujimoto N (2014) Multiple precision integer multiplication on GPUs. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* .

Lobeiras J, Amor M and Doallo R (2016) Designing Efficient Index-Digit Algorithms for CUDA GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 27(5): 1331–1343.

Maza MM and Pan W (2010) Fast polynomial multiplication on a GPU. *Journal of Physics: Conference Series* 256(1): 009–012. URL http://stacks.iop.org/1742-6596/256/i=1/a=012009.

Rivest RL, Shamir A and Adleman L (1978) A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21(2): 120–126.

Schönhage A and Strassen V (1971) Schnelle multiplikation großer zahlen 7: 281–292.

Toom A (1963) The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady* 3: 714–716.

Zhao K (2010) Implementation of Multiple-precision Modular Multiplication on GPU. *Tech. Report* .

Zuras D (1994) More on squaring and multiplying large integers. *IEEE Transactions on Computers* 43(8): 899–908.