Efficient Multiplication of Polynomials on Graphics Hardware

Pavel Emeliyanenko

Max-Planck-Institut für Informatik, Saarbrücken, Germany
asm@mpi-inf.mpg.de

Abstract. We present the algorithm to multiply univariate polynomials with integer coefficients efficiently using the Number Theoretic transform (NTT) on Graphics Processing Units (GPU). The same approach can be used to multiply large integers encoded as polynomials. Our algorithm exploits fused multiply-add capabilities of the graphics hardware. NTT multiplications are executed in parallel for a set of distinct primes followed by reconstruction using the Chinese Remainder theorem (CRT) on the GPU. Our benchmarking experiences show the NTT multiplication performance up to 77 GMul/s[1]. We compared our approach with CPU-based implementations of polynomial and large integer multiplication provided by NTL and GMP[2] libraries.

Keywords: large integer arithmetic, parallel computations, graphics hardware, GPU, CUDA.

1 Introduction

Large integer and polynomial arithmetic constitutes the core of many scientific computations. For instance, algorithms in algebraic geometry involve a substantial amount of symbolic computations performed over integer polynomials in one or more variables (e.g., polynomial subresultants and derived quantities [5]). The performance of public key cryptosystems also relies on the efficiency of large integer arithmetic.

Schönhage and Strassen [21] have shown that the Number Theoretic transform (NTT), as generalization of discrete Fourier transform to finite fields, is asymptotically the fastest known way to multiply two large integers. Moreover, the inherent parallel structure of the NTT and the absence of round-off errors, as opposed to floating-point Fourier transforms, makes it very tempting candidate for realization on parallel architectures. Unfortunately, the graphics hardware, driven by the needs of the game industry, was originally designed for efficient low-precision floating-point arithmetic.

[1] GMul/s stands for "10^9 modular multiplications per second", not to confuse with GFlop/s, see Section 6 for explanations.

[2] NTL: `http://www.shoup.net/ntl`, GMP: `http://gmplib.org`

Y. Dou, R. Gruber, and J. Joller (Eds.): APPT 2009, LNCS 5737, pp. 134–149, 2009.

Although floating-point Fourier transforms are also applicable to integer convolutions[3], the number of bits to be stored in a floating-point number to guarantee the provably correct rounding is substantially limited (see [20] for precise estimates). As a result, single-precision floating-point is practically not applicable for error-free integer convolutions, while the double-precision arithmetic is still relatively slow on modern GPUs. The NVIDIA's CUDA API [2] makes it possible to utilize graphics processors for integer computations.

Main contribution. we present the algorithm to compute integer polynomial products using the NTT on graphics processors. We use efficient 24-bit modular multiplication which reflects the native multiplication capabilities of the GPU. Our algorithm operates on partially reduced 24-bit residues represented by 32-bit integers, deferring the final reduction as long as possible. This enables us to avoid a great deal of expensive operations. Optimized FFT-kernels utilize fused multiply-add capabilities of the graphics hardware. The reconstruction of convolution digits is performed on the GPU using the Chinese Remainder theorem (CRT) allowing us to multiply polynomials with moderate coefficient bit-length entirely on the GPU.

The remaining part of the paper is structured as follows. In Section 2 we survey existing algorithms for modular techniques and large integer multiplication on parallel architectures. Section 3 gives an overview of 3D graphics hardware and CUDA programming model. Some background theory underlying the Number Theoretic transforms and the CRT reconstruction is presented in Section 4. In Section 5 we discuss the algorithm and its mapping to the GPU in detail. Then, in Section 6 we compare our algorithm with existing CPU-based implementations and draw conclusions in Section 7.

2 Related Work

Over the past years there was a lot of research carried out to implement efficient FFT algorithms on graphics processors ([10], [1], [17]). Unfortunately, all of them operate in a single-precision floating-point arithmetic and, hence, are not suitable for integer convolutions. There were attempts to emulate extended precision using a pair or a quad of low-precision floating-point numbers ([12], [11]). However, this leads to rather complicated arithmetic operations thereby annihilating all the advantages of the floating-point and, moreover, it doubles the memory bandwidth between the host and the graphics card which is a major performance killer for GPU algorithms.

There are two recent papers employing modular techniques on the GPU ([18], [24]). Despite the fact that they are concentrated on the acceleration of modular exponentiation, it is interesting within our context how they deal with the modular reduction after multiplication.

The authors of [18] used a traditional shader approach to program the GPU. As a result, they could only handle integers that fit the floating-point mantissa

[3] By convolution we mean here the integer polynomial product (acyclic convolution) which is a cyclic convolution of zero-padded sequences, see Section 4.

(24 bits). They suggested to use composite moduli consisting of 2 primes whose product fits in 24 bits. Hence, unfolding the CRT over these two primes, the modular multiplication can proceed without intermediate values that exceed 24 bits. We find that this method involves too many arithmetic operations and does not take any advantage of the floating-point nature of the arithmetic.

The second paper [24] used CUDA framework and all computations were carried out in integer arithmetic. The authors reported that, while the graphics hardware supports fast 24-bit integer multiplication, CUDA does *not* expose an intrinsic to obtain the most-significant 16 bits of the product. Therefore, they were constrained to use full 32-bit moduli and slow 32-bit multiplication. Luckily, we have been able to deal with this limitation (see Section 5.3). Unfortunately, their paper does not explain in a concrete way how the 32-bit modular reduction is realized.

An interesting approach to large integer multiplication on parallel architectures appears in [8]. It uses multi-dimensional Fermat Number transform (FNT)[4]. Although, the FNT has clear advantages credited to Schönhage and Strassen, we believe that the modular approach with CRT is more suitable for GPU implementation because of its relative simplicity (as opposed to multi-dimensional transform) and flexibility since it allows us to convolve variable length sequences using the same transform length. On the contrary, dimensionality of the FNT depends on the length of input sequences. Moreover, according to [8], 1024-point FNT requires 2^{13} processors arranged in a 4-dimensional hypercube to work cooperatively. This number exceeds by far the maximal number of threads allowed per one GPU's thread block while threads of different blocks cannot communicate with each other directly (see Section 3).

3 Overview of the GPU Architecture and CUDA Framework

In this overview we only consider the GPUs with NVIDIA Tesla architecture [16]. However, the new standard for heterogeneous programming OpenCL [19] will provide a unified API (which is very similar to that of CUDA) and will be supported by many other vendors. The NVIDIA Tesla architecture unifies vertex and fragment processors in streaming multiprocessors (SMs) that can execute any shader programs as well as general-purpose parallel programs. For instance, GeForce GTX 280 GPU contains 30 SMs.

The GPU executes instructions in a SIMT – *single-instruction, multiple-thread* – fashion. In other words, the SM's instruction issue unit (MT issue, see Figure 1) applies a single instruction to a group of 32 threads called *warps*. As a result, threads of a single warp are always executed synchronously. When the threads follow different execution paths (diverge on a branch instruction), the warp has to serially execute all taken branch paths. The full efficiency is attained when

[4] A well-known restriction of using Fermat ring to compute convolutions relates to the fact that the maximal transform length is proportional to the modulus bit-length. Multi-dimensional techniques are supposed to overcome this difficulty.

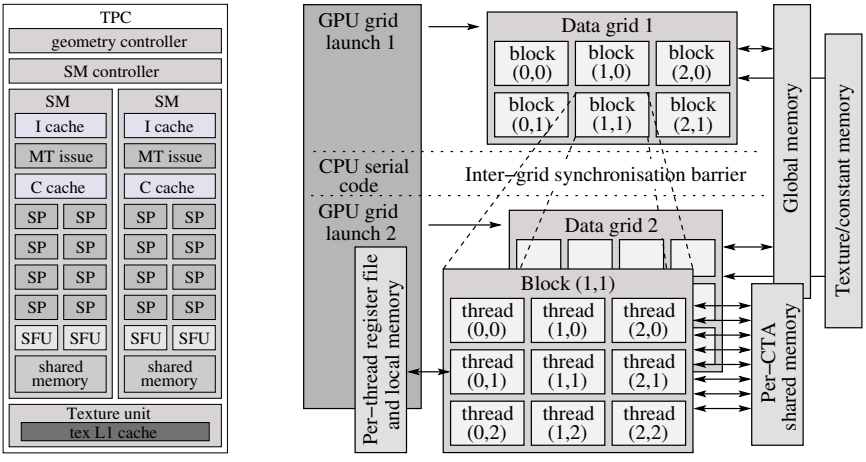

Fig. 1. Texture/processor cluster (TPC) comprising two SMs (left); CUDA execution model, thread and memory hierarchy (right)

a branch condition is *warp-aligned*. Different warps are independent from each other and can execute disjoint paths without penalties.

Each SM contains two special function units (SFU) and eight streaming processors cores (SP), see Figure 1. The SM processes simple arithmetic instructions in *four* clock cycles for the entire warp. These instructions also include single-precision floating-point multiply/multiply-add and 24-bit integer multiply/multiply-add. Integer division and modulo are particularly costly and should be avoided, we use floating-point arithmetic instead.

CUDA is a heterogeneous *serial-parallel* programming model, i.e., a parallel GPU code is interleaved with a serial code executed on the host (see Figure 1). On the top level, threads are grouped into *cooperative thread arrays* (CTAs) or *thread blocks*. Each block consists of up to 512 concurrent threads which execute the same CUDA code, can share the results of computations and synchronize their execution with barriers. In its turn, blocks are organized in a *grid* of thread blocks which is launched on a single CUDA program. Threads of different blocks cannot communicate with each other explicitly but can share the results by means of global memory[5]. *Inter-grid synchronization* can be achieved by serialized grid launches. The CTA model implements a coarse-grained parallelism as opposed to fine-grained parallelism achieved by warps.

Memory system of the GPU is organized as follows: each thread has its own *register file*. The SM has a fixed number of registers split evenly between threads of a block, by exceeding this amount registers get spilled into slow *local memory*

[5] Block independence naturally comes from the scalability requirements allowing a binary program to run unchanged on any number of SMs. However, it imposes additional difficulties in algorithms' realization. In this respect, we find the Intel's Larrabee architecture more advantageous, see Section 7.

residing in external DRAM. All threads within a single block can access the fast on-chip *shared memory* (see Figure 1). It is organized in 16 *banks* in such a way that consecutive addresses are mapped to different banks. If all 16 threads of a half-warp access memory from different banks, no delays occur. Memory accesses with a stride s where $GCD(s, 16) \neq 1$ lead to *bank conflicts* and are serialized[6]. The remaining three memory spaces – read-write *global memory* and read-only *constant* and *texture memory* – are visible to all threads of the entire grid. Global memory is not cached and has much higher latency than shared memory, it is important to access it in a way that separate memory accesses of a half-warp can be coalesced in a single wide memory access. A good programming practise is to preload data from global memory at once, and then use shared memory for subsequent computations. Constant memory has on-chip cache, amortized access to it is fast provided that all threads of a warp read the same address. Texture memory is also cached and optimized for 2D spatial locality.

High memory access latencies can be hidden as long as the code has high arithmetic intensity and the SM has enough warps to switch between in order to interleave memory access with ALU operations.

4 Mathematical Preliminaries

In this section we overview some basic facts from the number theory underlying fast multiplication algorithms in finite fields and recall the *Chinese remainder theorem* (CRT) to recover multidigit result after modular multiplication.

4.1 Number Theoretic Transforms and Fast Convolutions

The forward and backward Number Theoretic transforms are defined respectively as follows:

$$X_k \equiv_m \sum_{j=0}^{N-1} x_j \alpha^{jk} \quad \text{and} \quad x_j \equiv_m N^{-1} \sum_{k=0}^{N-1} X_k \alpha^{-jk},$$

where $j, k = 0, \ldots, N-1$, all arithmetic is performed over Z/mZ and α is an N-th primitive root of unity (an element of order N). The necessary and sufficient conditions for existence of such transforms are [7]:

- $N \mid GCD\{(p_i - 1), i = 1, \ldots, l\}$, where $m = \prod_{i=1}^{l} p_i^{r_i}$;
- $GCD(N, m) = 1$ (existence of modular inverse);
- $\alpha^s \neq 1 \pmod m : \forall s = [1, 2, \ldots, N-1]$.

A *cyclic convolution* of two length-n sequences $a = [a_0 \ldots a_{n-1}]$ and $b = [b_0 \ldots b_{n-1}]$ is a length-n sequence $h = a * b$ with $h_j = \sum_{i=0}^{n-1} a_i b_{(j-i) \bmod n}$. Once the conditions above are satisfied, the transform possesses the so-called

[6] Bank conflicts only occur within a half-warp – a group of 16 threads.

cyclic convolution property (CCP) allowing for fast convolutions in Z/mZ. The CCP states that if

$$X_k \equiv_m \sum_{j=0}^{N-1} x_j \alpha^{jk} \quad \text{and} \quad Y_k \equiv_m \sum_{j=0}^{N-1} y_j \alpha^{jk}, \text{ then for } h = x * y,$$

$$h_j \equiv_m N^{-1} \sum_{k=0}^{N-1} H_k \alpha^{-jk} \quad \text{where} \quad H_k \equiv_m X_k \cdot Y_k.$$

Accordingly, the usual polynomial product of a and b, defined as $r_j = \sum_{i=0}^{n-1} a_i b_{j-i}$, is a cyclic convolution of zero-padded sequences, i.e., $[a_{n/2} \dots a_{n-1}] = 0$ and $[b_{n/2} \dots b_{n-1}] = 0$. To multiply two K-bit integers using this technique, they are first partitioned into $N/2$ chunks of $P = 2K/N$ bits each, where N is the size of the transform. Then, the resulting sequences a and b are zero-padded and convolved, i.e., $r \equiv_m a * b$. The modulus m is chosen to be large enough so that the "convolution digits" are recovered exactly (see estimates in Section 5.1). Finally, one obtains the resulting product by evaluating: $z = \sum_{i=0}^{N-1} r_i \cdot 2^{Pi}$.

In our approach we use 24-bit prime moduli of the form $m = 2^n \cdot k + 1$ (for transforms of length 2^n). The reasons for that are: first, the number of 24-bit primes of this form is considerably large, which is suitable for the CRT reconstruction. Second, the modular reduction with 24-bit primes can be performed efficiently in floating-point arithmetic.

4.2 Chinese Remainder Theorem

Let $(m_1, m_2, \dots, m_k)$ be pairwise coprime moduli and $M = \prod_{i=1}^{k} m_i$ (M is called dynamic range). Then, for the set of residues $(x_1, x_2, \dots, x_k)$ with $0 \le x_i < m_i$ $(1 \le i \le k)$ there exists a unique X $(0 \le X < M)$, such that: $x_i = X \bmod m_i$.

A classical approach for incremental Chinese remaindering is the one of Szabo and Tanaka [23] based on Mixed Radix System (MRS). Here X is defined by the associated mixed-radix digits $(\alpha_1, \alpha_2, \dots, \alpha_k)$ in the following way:

$$X = \alpha_1 M_1 + \alpha_2 M_2 + \ldots + \alpha_k M_k$$

where $M_1 = 1$, $M_j = m_1 m_2 \dots m_{j-1}$ $(2 \le j \le k)$. We omit precise formulae for α_i for brevity. There exist efficient MRS conversion algorithms based on look-up tables (see [14], [3]), however the size of the tables they require is proportional to the modulus bit-length which draw them impractical for the GPUs[7]. We have decided in favour a simple algorithm from [25] which rearranges Szabo and Tanaka formulae in a more structured way, thereby exposing some parallelism. The α_i are computed as below $(1 \le i \le k)$:

$$\alpha_1 = x_1,\ \alpha_2 = (x_2 - \alpha_1) c_2 \bmod m_2$$

[7] Using large look-up tables residing in external DRAM turn to be inefficient on the GPU due to the high memory latencies and the lack of *gather* operation.

$$\alpha_3 = ((x_3 - \alpha_1)c_3 - (\alpha_2 M_2 c_3 \bmod m_3)) \bmod m_3$$
$$\alpha_i = ((x_i - \alpha_1)c_i - (\alpha_2 M_2 c_i \bmod m_i) - \ldots$$
$$-(\alpha_{i-1} M_{i-1} c_i \bmod m_i)) \bmod m_i$$

where $c_i = (m_1 m_2 \ldots m_{i-1})^{-1} \bmod m_i$. Here c_i and $M_j c_i \bmod m_i$ can be precomputed in advance.

5 Mapping Multiplication Algorithm to Graphics Processor

In this section we consider the multiplication algorithm step-by-step. First, we present our approach at a high-level to give the reader an intuitive feeling about the algorithm. Then, we describe how the FFT algorithm is mapped to the graphics hardware to achieve even work distribution between threads. The next sections cover the efficient modular reduction and optimizations aimed to utilize fused multiply-add capabilities of the GPU and reduce the amount of reductions using redundant residue representation. At the end, we discuss how the CRT reconstruction is realized on the graphics processor.

5.1 Algorithm Overview

The multiplication on the GPU proceeds as follows: we are given a set of N integer polynomials of degree at most 2^{n-1}, where 2^n is the size of the transform[8]. Polynomials of higher degree can be processed by encoding them in fixed degree polynomials using the binary segmentation [9]. Large integers are handled by partitioning them into respective number of pieces.

Each piece (or polynomial coefficient) is reduced modulo a set of distinct 24-bit primes, the number of primes K is chosen such that the resulting products can be recovered exactly[9]. The GPU executes $N \times K$ NTT modular multiplications in parallel. Once all products are ready, another kernel groups every K modular products and recovers multiprecision digits using the CRT (see Section 5.5). K is chosen to be small enough (typically $K < 10$), so that the CRT reconstruction can proceed entirely on the GPU. However, this is not a restriction – the GPU can run modular convolutions for large values of K and recover multiprecision digits only partially, leaving the final reconstruction for the CPU.

The number of primes required to recover the product of two large integers is estimated as follows: each "digit" after 2^n-point convolution is bounded by $2^{2M} \cdot 2^{n-1}$, where M is a bit-length of an input sequence digit. Hence, a "convolution digit" has at most $2M + n - 1$ bits. For the CRT reconstruction with c primes, it holds that: $2M+n-1 = 23 \cdot c$ or $M_c = (23 \cdot c - n + 1)/2$, here we assumed that each prime is 23-bits long on the average. Thus, c convolutions with different moduli

[8] Recall that, the input sequences must be initially zero-padded, hence these numbers.
[9] For small values of K, the initial modular reduction can be done directly on the graphics processor.

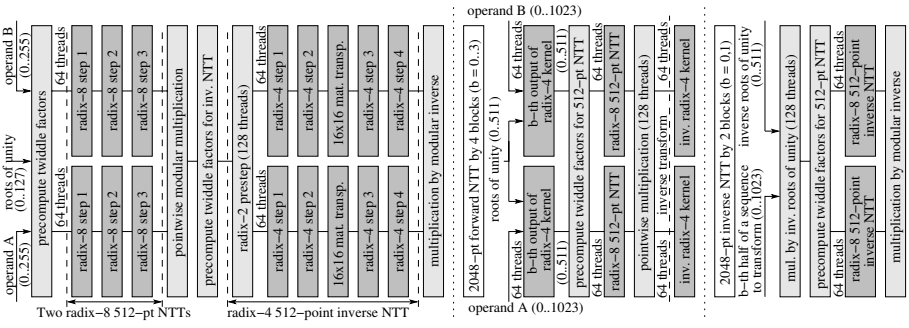

Fig. 2. Schematic view of 512-point (left) and 2048-point (right) NTT multiplication on the GPU

are enough to multiply numbers of $2^{n-1}M_c$ bit-length. For example, with $c = 4$, 2048-point transform can be used to multiply integers having at most $1024 \cdot 41$ bits each (1312 32-bit machine words).

5.2 The FFT Algorithm

Parallel FFT algorithms are commonly based on the Stockham out-of-place FFT. We use Bailey's variation of this algorithm [4]. This is a self-sorting algorithm, such that an expensive index permutation phase (as opposed to the classical Cooley-Tukey FFT [6]) can be skipped. Moreover, all data fetches and stores are performed solely with unit strides, hence, no bank conflicts occur. Roots of unity are still accessed with power-of-two strides but this can be alleviated by storing the roots in contiguous arrays for each FFT step. In contrast to floating-point transforms, the roots of unity in Z/mZ cannot be computed on-the-fly, but must be precomputed in advance and loaded to the GPU. We have implemented Bailey's FFT for transform sizes 512, 1024 and 2048.

Figure 2 depicts the mapping of 512- and 2048-point NTTs to the graphics hardware, 1024-point transform is realized by analogy. The core of the algorithm constitute radix-2, -4 and -8 kernels (or "butterfly" operations). The radix-n FFT-kernel is defined as: $[y_0, \ldots, y_{n-1}] = F_n diag(1, \alpha^k, \ldots, \alpha^{(n-1)k})[x_0, \ldots, x_{n-1}]$, where α^k is a twiddle factor and F_n is an $n \times n$ Fourier matrix, i.e., $F_n = [w_n^{j \cdot k}]_{j,k=0,\ldots,n-1}$ (w_n is an n-th root of unity). In the following subsections we consider optimized FFT-kernels in detail.

The 512-point NTT multiplication is done by a single block of 128 threads, after each radix-4/-8 step the data is reordered in shared memory. The forward 2048-point transform is run by 4 blocks collectively, they first evaluate a radix-4 kernel for both multiplicands. Then, the outputs are split evenly between the blocks, each single block processes its parts, multiplies them elementwise and runs the first radix-4 step of the inverse transform. By multiplying the operands early in the forward kernel, we effectively reduce the memory bandwidth because only one (resulting) sequence is written out to global memory. The inverse

Algorithm 1. 24-bit modular multiplication: computes $a \cdot b \bmod m$

```
1: procedure MUL_MOD(a, b, m, invm)        ▷ invm = 2^16/m (in floating-point)
2:     hi = __umul24hi(a, b)               ▷ compute upper 32 bits of the product
3:     prodf = __fmul_rn(hi, invm)         ▷ multiplication in floating-point
4:     l = __float2uint_rz(prodf)          ▷ integer truncation: l = ⌊hi · 2^16/m⌋
5:     return (__umul24(a, b) − __umul24(l, m))   ▷ in [−m + ε; m + ε] with 0 ≤ ε < m
6: end procedure
```

2048-point NTT is run by 2 blocks, each block transforms its 1024-element part separately.

5.3 Multiplication and Modular Reduction

The reason for choosing 24-bit primes was that the graphics hardware does not support a full 32-bit integer multiplication natively. It provides only 24-bit multiplication realized in mul24.lo/hi instructions[10]. mul24.lo computes multiplies 24 least significant bits (LSB) of the operands and returns 32 LSB of 48-bit product, it is available via __umul24 intrinsic. mul24.hi returns 32 most significant bits of the product respectively. Strangely enough, it is not accessible from a high-level CUDA code. Fortunately, we have been able to rebuild the *nvopencc* (which is based on *open64*) from sources to insert the "missing intrinsic", in what follows we will refer to it as __umul24hi[11].

Having all prerequisites at hand, we now discuss how the modular arithmetic is realized on the GPU. We consider only modular multiplication in detail (see Algorithm 1) as the remaining operations (addition and subtraction) are rather trivial. Algorithm 1 splits the product in two parts, i.e., $a \cdot b = 2^{16}hi + lo$ (32 and 16 bits), and the following holds ($0 \leq r < m$):

$$2^{16}hi + lo = (m \cdot l + r) + lo \equiv_m r + lo = 2^{16}hi + lo - m \cdot l = a \cdot b - l \cdot m$$

Observe that, $l = \lfloor 2^{16}hi/m \rfloor$ is at most 24-bits long, thus it is exactly representable with 24-bit mantissa. Let $\gamma = a \cdot b - l \cdot m = lo + r$, hence $\gamma \in [0; m + \varepsilon]$ ($0 \leq \varepsilon < m$)[12]. As a result, γ fits into 32 bits and is computed as a difference of 32 LSB of both products using __umul24 intrinsic. The final reduction needs two additional steps to map the range $[-m + \varepsilon; m + \varepsilon]$ to $[0; m - 1]$. Owing to the redundant representation of residues, these steps can be deferred until the next modular multiplication takes place. We discuss this and other optimizations in the following section.

[10] The 32-bit integer multiplication gets demoted to a more primitive operations and is 4 times slower than its 24-bit counterpart.

[11] The compiler built for Linux platform with a set of new intrinsics is available at `http://www.mpi-inf.mpg.de/~emeliyan/cuda-compiler`

[12] According to our tests, $\gamma \in [-m+\varepsilon; m+\varepsilon]$ due to the loss of accuracy when converting hi to floating-point but this is not critical for us.

Algorithm 2. Realization of radix-2 kernel (FMA_BFY2) and modular reduction of 32-bit operand (REDUCE_MOD)

```
 1: procedure FMA_BFY2(x0, x1, w, m, invm)      ▷ invm = 2^16/m (in floating-point)
 2:     hi = __umul24hi(x1, w)                  ▷ compute upper 32 bits of the product
 3:     prodf = hi * invm + 2.0f                ▷ floating-point multiply-add
 4:     l = __float2uint_rz(prodf)              ▷ integer truncation: l = ⌊hi · 2^16/m⌋ + 2
 5:     y0 = x0 + __umul24(x1, w) − __umul24(l, m)   ▷ a pair of 24-bit multiply-adds
 6:     return [y0, sad(x0, y0, x0)]            ▷ y1 = |x0 − y0| + x0 = 2x0 − y0
 7: end procedure
 8: procedure REDUCE_MOD(a, m, invm)            ▷ invm = 1/m (in floating-point)
 9:     ai = a + __umul24(100, m)               ▷ make sure a is positive
10:     af = __fmul_rn(__uint2float_rn(ai), invm)   ▷ multiply in floating-point
11:     l = __float2uint_rz(af)                 ▷ integer truncation: l = ⌊a/m⌋ + 100
12:     r = ai − __umul24(l, m)                 ▷ r ∈ [−m + ε; ε] with 0 ≤ ε < m
13:     if r < 0 then r = r + m                 ▷ adjust the result in case of negative sign
14:     return r
15: end procedure
```

5.4 FMA-Optimized FFT Kernels and Exploiting Redundancy in Residue Representation

The graphics hardware has fused multiply-add (FMA) capabilities. Namely, it supports floating-point FMA as well as 24-bit integer FMA instructions. To achieve the full efficiency, it is therefore important to respect these hardware features. In our implementation we use both of them. Our radix-4 and -8 FFT kernels are based on the FMA-optimized factorization of a matrix product given in [15]. In its core it has a primitive radix-2 "butterfly" defined as $([y_0, y_1] = \mathsf{fma\_bfy2}([x_0, x_1], w))$: $y_0 \leftarrow x_0 + x_1 \cdot w$ and $y_1 \leftarrow 2 \cdot x_0 - y_0$. Its realization is given by procedure FMA_BFY2 of Algorithm 2. Remark that, y_1 cannot be computed with 24-bit FMA because x_0 can exceed 24 bits (when redundant representation is used). Remarkably, the GPU has a native $\mathsf{sad}(x, y, z)$ instruction which computes $|x - y| + z$. Thus, if we ensure that $x_0 - y_0 > 0$, we can use sad to compute y_1. We guarantee this by adding 2 to *prodf* in line 3 of the algorithm. Indeed, $x_0 - y_0 = l \cdot m - x_1 \cdot w = \gamma$, and, according to the estimates above, $\gamma \in [-m + \varepsilon; m + \varepsilon]$. Altogether, the fma_bfy2 is compiled in *6 flops* on the GPU[13].

Remark that, y_0 and y_1 in general are not valid residues, while __umul24 can only handle 24-bit operands. To this end, the argument x_1 of the next fma_bfy2 must be reduced prior to multiplication, this is achieved by procedure REDUCE_MOD of Algorithm 2. By adding $100 \cdot m$ to a we ensure that $l = \lfloor a/m \rfloor + 100$ is positive and, hence, $\mathsf{\_umul24}(l, m)$ delivers the correct result[14]. We will refer to

[13] Generated low-level GPU assembly code can be inspected using the *decuda* tool: `http://www.cs.rug.nl/~wladimir/decuda`

[14] It can be estimated that a never deviates from 0 by more than $100 \cdot m$, thus, $a + 100 \cdot m$ is guaranteed to be positive and fits within 32 bits.

$\mathsf{reduce\_mod}(\mathsf{x}_1)$ followed by $\mathsf{fma\_bfy2}([\mathsf{x}_0, \mathsf{x}_1, \mathsf{w}])$ as $\mathsf{fma\_red\_bfy2}$. FMA-optimized radix-4 kernel is defined below ($[y_0, \ldots, y_3] = \mathsf{fma\_bfy4}([x_0, \ldots x_3], u)$):

$$[d_0, d_1] = \mathsf{fma\_red\_bfy2}([x_0, x_2], u^2) \quad [d_2, d_3] = \mathsf{fma\_red\_bfy2}([x_1, x_3], u^2)$$
$$[y_0, y_2] = \mathsf{fma\_red\_bfy2}([d_0, d_2], u) \quad [y_1, y_3] = \mathsf{fma\_red\_bfy2}([d_1, d_3], u \cdot w_4),$$

here $u = \alpha^k$ denotes a twiddle factor and w_4 is 4-th root of unity. Radix-8 kernel is realized by analogy. Note that, the first step of the FFT algorithm does not need any twiddle factors and FFT-kernels are simplified. Moreover, the input sequences are initially zero-padded, hence the first stage of the forward transform can be simplified even further.

The redundancy in residue representation is exploited as follows: *modular reductions after addition/subtraction as well as correction steps after multiplication are performed on demand only.* In other words, they are deferred until either the next multiplication takes place or until the very last stage of the NTT algorithm.

5.5 CRT Reconstruction on the GPU

Owing to the fact that each "convolution digit" is recovered independently, it is advantageous to run the CRT reconstruction directly on the GPU provided that the number of moduli k is small (typically $k \leq 10$). We compute the MRS digits α_i defined in Section 4.2 in a straightforward way. Threads are split logically into groups of $P = (k-2)/2$ threads each. We require P to be a *power-of-two*, so that the groups of P threads are always warp-aligned and access to shared memory proceeds without synchronization. We have chosen the block size of 64 threads[15].

We assume that the moduli are sorted, i.e., $m_1 < m_2 < \ldots < m_k$. Thus, for respective residues $x_1, x_2, \ldots, x_k$, it holds that $x_i < m_j$ for $1 \leq i < j \leq k$. This enables us to save on reductions when the quantities of the form $(x_j - x_i) \bmod m_j$ are computed. Each thread computes two values of x_i in one step (we refer to them by $\delta = \{1, 2\}$). Let $j = 1 \ldots k/2$, the algorithm takes $k-1$ steps:

step 1: for threads $i = 1 \ldots P$: $x_{2i+\delta} \leftarrow (x_{2i+\delta} - x_1)c_{2i+\delta} \bmod m_{2i+\delta}$. For the 1st thread additionally: $x_2 \leftarrow (x_2 - x_1)c_2 \bmod m_2$;
step $(2j)$**:** for threads $i = j \ldots P$: $x_{2i+\delta} \leftarrow (x_{2i+\delta} - x_{2j}M_{2j}c_{2i+\delta}) \bmod m_{2i+\delta}$;
step $(2j+1)$**:** for threads $i = j - 1 \ldots P$: $x_{2i+\delta} \leftarrow (x_{2i+\delta} - x_{2j-1}M_{2j-1}c_{2j+\delta}) \bmod m_{2j+\delta}$, a thread $i = j-1$ computes only x_{2i+2}.

The number of threads involved decrements every 2 steps, this way we achieve sufficiently even work distribution. We use precomputed values for c_i and $s_i^l = M_l c_i \bmod m_i$ defined in Section 4.2. Once MRS digits are computed, the resulting "convolution digit" is recovered as: $X = \alpha_1 M_1 + \alpha_2 M_2 + \ldots + \alpha_k M_k$. We extract some parallelism by evaluating this expression in a "tree-like" fashion. To realize multiprecision additions required here, we use addition-with-carry intrinsics provided by our *nvopencc* compiler.

[15] As we only need P threads to work cooperatively, a small block size is reasonable since the GPU has more freedom in scheduling light-weight blocks to hide memory access latencies.

6 Experimental Results and Comparison

We have tested our algorithm on the *GeForce GTX 280* graphics processor and compared it with GMP 4.2.1 (`http://gmplib.org`) for large-integer multiplication and with NTL 5.5 (`http://www.shoup.net/ntl`) for polynomial multiplication. As a target CPU we have used *Quad-Core Intel Xeon E5420* clocked at 2.5Ghz with 12MB L2 cache and 8Gb RAM. Both libraries were built under native 64-bit Linux platform (Debian Etch), such that they were able to benefit from AMD64 instruction set.

For benchmarks we have implemented two versions of the CRT reconstruction: a completely inlined one using 4 moduli where each digit is processed separately by a single thread, and the 6-moduli CRT which realizes the algorithm from Section 5.5. The *initial modular reduction* of input digits was performed directly on the GPU prior to modular multiplications because the digits' bit-length is small. The bit-length of numbers to be multiplied depending on the CRT size and the transform length was estimated using the formula from Section 5.1. We use these estimates to compare our multiplication with that of provided by GMP and NTL. For instance, 1024-point NTT with 6-moduli CRT is enough to multiply 512·64-bit numbers exactly. Hence, GMP was used to multiply numbers of 512·64 bit-length, while NTL – to multiply 512-degree polynomials with 64-bit coefficients.

Remark that, our algorithm does not perform the digit adjustment after multiplying two integers encoded as polynomials. In other words, we do not compute the sum of "convolution digits", i.e., $z = \sum_{i=0}^{N-1} r_i \cdot 2^{Pi}$ (see Section 4.1). Nevertheless, we suppose this would not make our comparison with GMP unfair

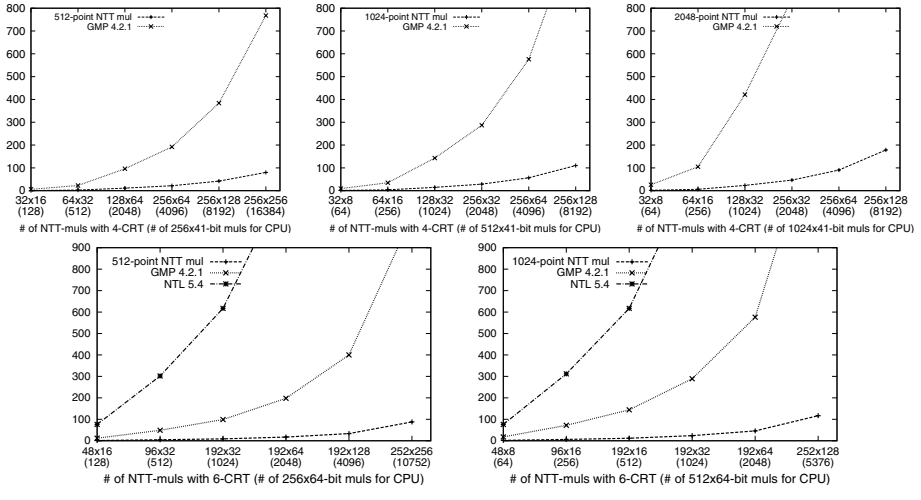

Fig. 3. Time comparison of batched large integer/polynomial multiplication with GMP/NTL implementations. **Top row:** 512-(left), 1024-(middle) and 2048-point(right) NTTs with **4**-moduli CRT. **Bottom row:** 512-(left) and 1024-point(right) NTTs with **6**-moduli CRT. All times are in milliseconds.

Table 1. Performance of the 512-point and 2048-point convolutions in "GMul/s": 10^9 modular multiplications per second

# of 512-point NTTs	32x16	64x32	128x64	256x64	256x128	256x256
time (ms)	0.26	0.98	4.04	7.78	15.22	29.13
GMul/s	68	72	73	75	77	77
# of 2048-point NTTs	32x8	64x16	128x32	256x32	256x64	256x128
time (ms)	0.74	2.49	9.83	19.34	38.39	76.55
GMul/s	58	67	72	73	73	74

because this step is rather cheap as it only involves additions[16]. Moreover, the digit adjustment is not required in case of polynomial multiplication.

Figure 3 shows the time comparisons for batched multiplications. The labels along x-axes have the following meaning: for instance, on the top-left plot 32x16(128) denotes that the CPU performs 128 multiplications of 256×41-bit numbers while the GPU runs 16 512-point convolutions for each of 32 moduli (total of 512 convolutions) since a group of every 4 moduli contributes to a single multiplication. The GPU timing includes the time taken for memory transfer to the GPU and back to the host for a more objective comparison. We use *page-locked* memory to achieve higher bandwidth. From Figure 3 one can see that the GPU is superior for batched multiplications with moderate bit-lengths. Moreover, the gap increases for larger transforms. Increasing the number of CRT moduli is also advantageous for our algorithm, although it is yet unclear whether increasing the transform length or increasing the number of moduli is overall more efficient. Note that, NTL performs worse than GMP which is expectable because GMP uses hand-optimized assembly while NTL is written in a high-level language.

Table 1 summarizes the "effective" performance of the NTT multiplication, computed as: $GMul/s = 10^{-9} \cdot batch \cdot (3 \cdot 2.5N \log_2 N + 2N)/t$, here $2.5N \log_2 N$ is the complexity of the Cooley-Tukey style NTT (N is the transform length), t is the elapsed time in seconds and *batch* is the number of parallel multiplications. Each multiplication uses 2 forward and 1 backward transform, hence, the factor 3 in front of the formula. The term $2N$ represents the complexity of the pointwise multiplication and the multiplication by modular inverse. Remark that, the Cooley-Tukey NTT bound counts the number of multiplications in Z/mZ. To evaluate the "real" performance in *flops* recall that fma_bfy2 realizing modular multiplication executes in 6 flops (see Section 5.4). Hence, 77 GMul/s is roughly equivalent to 462 GFlop/s of the real performance[17], while the GeForce GTX 280 has peak parallel performance of 933 GFlop/s.

[16] Carry propagation after mutliprecision addition can be realized efficiently, for instance, using Hillis-and-Steele-style reductions [13].

[17] It worth mentioning that the Cooley-Tukey bound tends to overestimate the number of multiplications, nevertheless it is a commonly used tool to evaluate the FFT/NTT performance.

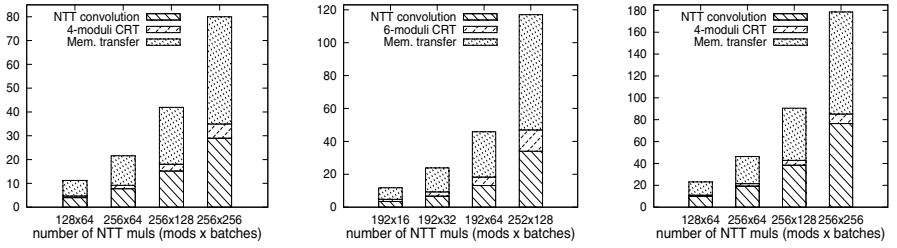

Fig. 4. Time breakdown (in milliseconds) for 512-(left), 1024-(middle) and 2048-point(right) transforms. Abbreviations along x-axis are the same as in Figure 3.

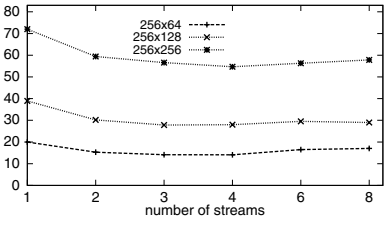

Fig. 5. The number of parallel streams influencing the overall time (in milliseconds) for 512-point NTTs including the memory transfer

Figure 4 depicts the time distribution over algorithm stages. Observe that, the CRT reconstruction is rather cheap while the time needed for memory transfer comprises the major part. This is known to be the main bottleneck for GPU algorithms. Fortunately, CUDA allows us to split a single kernel launch into several *streams* which execute asynchronously such that the memory transfer of one stream can overlap with a kernel execution of another stream[18]. Figure 5 evaluates the performance of 512-point NTTs with several streams. The figure shows that the optimal number of streams is 4.

To sum-up, our algorithm outperforms GMP and NTL for batched multiplications with moderate bit-length. We agree that this is not an objective picture because, for instance, GMP is particularly fast when the numbers of million bits are multiplied. We were not able to benchmark our algorithm on such instances due to the lack of implementation of larger transform lengths which is an object of ongoing research. Still, we believe that the this gives a good estimate of what the GPUs are practically capable of, because this area of GPU application is yet not well-explored.

[18] At the time of writing the new CUDA 2.2 has been released. It supports allocation of *pinned memory* mapped to the device's address space (cudaHostAllocMapped). Due to the time limitations we have not been able to test its performance.

7 Summary and Outlook

We have presented the algorithm to multiply polynomials on the GPU using the NTT modular convolution with the CRT reconstruction. Our approach shows a good performance for batched multiplication of polynomials with moderate coefficient bit-length. Clearly, the approach presented here is only the first step in realization of a robust large integer and polynomial arithmetic on the GPU.

Note that, this application domain is pretty novel for the graphics hardware and we see many promising perspectives for future work. First, we would like to increase the NTT transform length and make it adaptable to the bit-lengths of numbers to be multiplied. Second, we would like to realize multiprecision addition on the GPU using parallel reductions in order to be able to reconstruct multiprecision numbers by means of binary segmentation. It is also worthwhile to try out the technique called *GPU virtualization* given in [10] to handle inputs that do not fit in a single grid launch due to the hardware limitations. Finally, we would like to realize other algorithms requiring multiprecision arithmetic on the GPU using the modular approach, for example, evaluation of matrix determinants with large integer coefficients which is a fundamental operation in many scientific fields.

We also find very promising the oncoming Intel's Larrabee architecture [22] and would like to test our algorithm with it. It has a number of salient features lacked on the current GPUs. First, Larrabee's 16-wide Vector Processing Unit (VPU – somewhat similar to SM) supports double-precision arithmetic at full speed[19], which allows us to increase the moduli bit-length (up to 54 bits) or employ floating-point transforms for integer convolutions. Second, Larrabee has a coherent L2 cache, such that the data is transparently shared between all processor cores (in contrast, GPU thread blocks can share data only through a high-latency GDDR memory). This considerably simplifies the realization of large FFT transforms which are realized by a hierarchy of grid launches on the GPU. Moreover, Larrabee supports scatter/gather operations, i.e., VPU lanes can access data at non-contiguous addresses while uncoalesced global memory access by a half-warp is considerably slow and should be avoided.

Acknowledgements. We would like to thank Michael Kerber for reviewing the paper and for useful and pragmatic suggestions.

References

1. CUDA CUFFT library. NVIDIA Corp. (2007)
2. NVIDIA CUDA: Compute Unified Device Architecture. NVIDIA Corp. (2007)
3. Akkal, M., Siy, P.: A new Mixed Radix Conversion algorithm MRC-II. J. Syst. Archit. 53, 577–586 (2007)
4. Bailey, D.H.: A High-Performance FFT Algorithm for Vector Supercomputers. International Journal of Supercomputer Applications 2, 82–87 (1988)

[19] The GPU has only one double-precision FPU per SM, therefore the double-precision arithmetic is 8 times slower than the single-precision.

5. Basu, S., Pollack, R., Roy, M.-F.: Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics). Springer, New York (2006)
6. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation 19, 297–301 (1965)
7. Elliott, D.F., Rao, K.R.: Fast Transforms: Algorithms, Analyses, Applications. Academic Press, Inc., Orlando (1983)
8. Fagin, B.S.: Large integer multiplication on hypercubes. J. Parallel Distrib. Comput. 14, 426–430 (1992)
9. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra. Cambridge University Press, Cambridge (1999)
10. Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High performance discrete Fourier transforms on graphics processors. In: SC 2008, pp. 1–12. IEEE Press, Los Alamitos (2008)
11. Graça, G.D., Defour, D.: Implementation of float-float operators on graphics hardware. CoRR **abs/cs/0603115** (2006)
12. Hida, Y., Li, X.S., Bailey, D.H.: Algorithms for Quad-Double Precision Floating Point Arithmetic. In: Proceedings of the 15th Symposium on Computer Arithmetic, pp. 155–162. IEEE Computer Society Press, Los Alamitos (2001)
13. Hillis, W.D., Steele Jr., G.L.: Data parallel algorithms. ACM Commun. 29, 1170–1183 (1986)
14. Huang, C.H.: A Fully Parallel Mixed-Radix Conversion Algorithm for Residue Number Applications. IEEE Trans. Computers 32, 398–402 (1983)
15. Karner, H., Auer, M., Ueberhuber, C.W.: Accelerating FFTW by Multiply-Add Optimization. Tech. rep., Institute for Applied and Numerical Mathematics, Vienna University of Technology, TR1999-13 (1999)
16. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro. 28, 39–55 (2008)
17. Moreland, K., Angel, E.: The FFT on a GPU. In: HWWS 2003. Eurographics Association, pp. 112–119. ACM Press, New York (2003)
18. Moss, A., Page, D., Smart, N.: Toward Acceleration of RSA Using 3D Graphics Hardware. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 364–383. Springer, Heidelberg (2007)
19. Munshi, A.: OpenCL: Parallel Computing on the GPU and CPU. SIGGRAPH 2008 (2008) (presentation)
20. Percival, C.: Rapid multiplication modulo the sum and difference of highly composite numbers. Mathematics of Computation 72, 241, 387–395 (2003)
21. Schönhage, A., Strassen, V.: Schnelle Multiplikation grosser Zahlen. Computing 7, 281–292 (1971)
22. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph. 27, 1–15 (2008)
23. Szabo, N., Tanaka, R.: Residue arithmetic and its applications to computer technology. SIAM 11, 103–104 (1969)
24. Szerwinski, R., Güneysu, T.: Exploiting the Power of GPUs for Asymmetric Cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)
25. Yassine, M.: Matrix Mixed-Radix Conversion For RNS Arithmetic Architectures (1991)