On the Performance of GPU Public-Key Cryptography

Samuel Neves
CISUC, Dept. of Informatics Engineering
University of Coimbra
Portugal
Email: sneves@dei.uc.pt

Filipe Araujo
CISUC, Dept. of Informatics Engineering
University of Coimbra
Portugal
Email: filipius@dei.uc.pt

***Abstract*—Graphics processing units (GPUs) have become increasingly popular over the last years as a cost-effective means of accelerating various computationally intensive tasks. We study the particular case of modular exponentiation, the crucial operation behind most modern public-key cryptography algorithms. We focus our attention on the NVIDIA GT200 architecture, currently one of the most popular for general purpose GPU computation.**

We report our efforts to run modular exponentiation faster than any other method we were aware of for GPUs. Part of our performance advantage results from a different interleaving of the Montgomery multiplication, which was neglected in previous literature. The other part comes from carefully exploring general techniques, like loop unrolling and inline PTX assembly. Our throughput results, at over 20000 RSA-1024 decryptions per second or 41426 512-bit modular exponentiations per second, present a significant speedup over previous GPU implementations, without any significant latency penalty.

Lastly, we evaluate our results in light of several popular metrics, namely performance/price and performance/watt ratios. We find that, while current GPUs generally perform better than CPUs, they show worse performance/watt ratios.

***Keywords*-CUDA, GPGPU, Montgomery multiplication, Modular exponentiation, RSA**

I. Introduction

Public-key cryptography is at the heart of modern Internet security. Widely used secure communication protocols, such as SSL and IPsec, rely on secure key exchange and digital signature algorithms such as the Diffie-Hellman [1], RSA [2] and DSA [3] algorithms. Unfortunately, public-key algorithms are not nearly as computationally cheap as symmetric encryption algorithms. A detailed study [4] of an SSL session shows that over 90% of the time spent in cryptographic operations was in the RSA key exchange, which entails a high computational cost for high-traffic websites, where the rate of new connections per second can easily reach the thousands. To offload the costs, we can resort to cryptographic accelerator cards.

Graphics processing units (GPUs) are excellent candidates to perform this acceleration, due to their flexibility and moderate cost. Additionally, modern GPUs are very powerful: their transistor count has been growing exponentially over the last few years, exceeding even CPU transistor counts [5]. It is not uncommon today for high-end GPUs to exceed 1 trillion floating point operations per second (FLOPs). They are also easily programmable, using tools like CUDA [6] and OpenCL [7].

In this paper, we harvest the power of GPUs to speed up the most common public key operation used in Web servers — RSA decryption[1]. We focus on a specific key length, 1024-bit RSA, for two main reasons: it is the most common key length in use [8], and it allows easier comparisons with other existing CPU and GPU implementations.

To improve on previous state of the art results, we focused on two aspects: fully exploiting both the tools and the underlying hardware and selecting algorithms better suited to the architecture. The former was done by performing full manual unrolling, using GPU (PTX) assembly directly and maximizing register use at the expense of less threads per block. The latter consisted of careful selection and benchmarking of various hitherto untried interleaving approaches for Montgomery reduction, which showed the method most commonly used in the literature is not the best for NVIDIA GPUs. More specifically, we found that the *Coarsely Integrated Operand Scanning* (CIOS) method of Koç et al. [9] is not the best method in the GPU; methods that "finely" integrate both multiplication and reduction in the same loop, namely *Finely Integrated Operand Scanning* (FIOS) and *Finely Integrated Product Scanning* (FIPS), seem to be better suited for the GPU, and our implementation greatly benefits from the switch to these methods.

Our resulting RSA implementation, directed at the NVIDIA GT200 architecture, outperforms all previous GPU implementations by a large margin, and remains competitive with recent implementations on newer (read: better) hardware. An NVIDIA GTX260 graphics card is able to perform over 20000 1024-bit RSA decryptions per second, outperforming the previous best previous implementation by a factor of over 2.5 [10].

Many different GPU architectures exist today, namely G80, GT200, G100 and G110 from NVIDIA, and the R700, Evergreen, and Northern Islands from AMD. We chose the GT200 architecture, however, as it has a very attractive

[1]In a typical SSL key exchange, the user encrypts his key with the server's public RSA key, and a decryption is performed server-side.

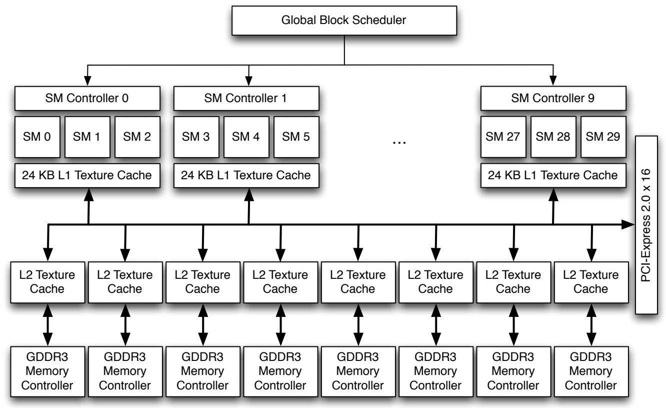

Figure 1. The GT200 architecture — a scalable array of SMs

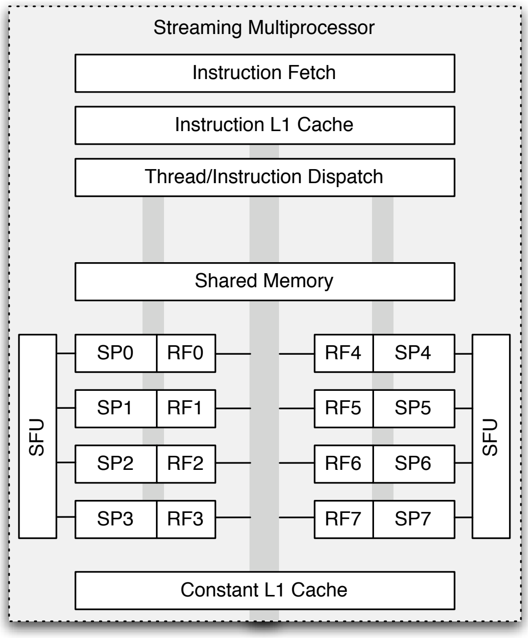

Figure 2. The streaming multiprocessor, building block of the GT200 architecture.

price/performance ratio as measured in flops per dollar, is quite widespread, and is easily programmable using the CUDA toolkit. At the time of this writing, the Geforce GTX260 card costs about \$100 and delivers 715 Gflops, giving a ratio of $715/100 = 7.15$ Gflops per dollar. The GTX 480 has a ratio of $1344/400 = 3.36$ Gflops/dollar; the newer GTX 580 decreases this ratio to roughly $1581/500 = 3.16$ Gflops/dollar.

In the remainder of this article, we will describe how we were able to achieve our performances. Section II describes the target architecture, NVIDIA GT200, and exposes its strengths and weaknesses. Section III exposes the crucial algorithms in our implementation, i.e. the multi-precision multiplication, modular reduction and exponentiation techniques employed and how they are mapped into actual CUDA code effectively. Section V summarizes our results. In Section IV, we compare our results with related work in GPU public-key cryptography. Finally, Section VI concludes the paper.

II. The GT200 architecture

NVIDIA's GT200 architecture is a natural evolution of the previous G80 architecture. [11] gives a thorough coverage of the G80 architecture. The hardware architecture of the G80 matches quite well the CUDA programming model described in [6]: the computing portion of the card is seen as an array of *streaming multiprocessors* (SM). Early G80 GPUs were composed of 16 SMs; GT200 models have up to 60.

A. The streaming multiprocessor

In Figure 2 we show a diagram of the streaming multiprocessor (SM). Each SM contains its own shared memory and register file and also its own constant and texture memory cache. Besides these specialized fast memories, the GPU has access to local and global memory, which reside outside the chip and are not cached. Additionally, each SM contains a single instruction cache, 8 ALUs and 2 Special Function Units (SFU). To maximize the ALU area on the chip, each of these ALUs operates in a SIMD fashion, in groups of 32 threads, called *warps*, controlled by a single instruction sequencer. At each cycle, the SM thread scheduler chooses a warp to be executed. Since each of the 8 ALUs supports up to 128 concurrent thread contexts, i.e., each ALU can be aware of up to 128 concurrent threads operating in it, it is possible to have 1024 concurrent threads executing on a single SM — in a 60-SM GPU, this amounts to up to 61440 simultaneous threads being executed at any given time.

1) Fast memory: As mentioned in the previous section, each SM contains its own register file and shared memory. The amount of registers available to a single kernel is dictated by the number of threads currently executing in a given SM, and has a maximum limit of 128 registers per thread. For example, a kernel running 192 threads per block will use at most 84 registers per thread[2]. Once the register count per thread rises over 64 registers, some restrictions exist, however: the second source of an assembly instruction cannot be in a register larger than 63 (the instruction set only reserves 6 bits for this operand) and the register 127 (`$r7f`) cannot be accessed at all [13].

Shared memory provides variable array indexing and allows for communication between threads within an SM. The shared memory bank in the GT200 is split into 16 banks, each capable of dispatching a memory read/write every 2 cycles. Thus, if all threads within a half-warp access a value in a different bank (or all access the same value), there is no performance penalty. Otherwise, there will be *bank conflicts*, and memory accesses to values sharing the same bank will be serialized [6]. Microbenchmarking results show the latency for shared memory accesses is about 38 cycles [14].

[2]Since the number of registers of a kernel is always rounded to the next multiple of 4 [12], this limit is the largest multiple of 4 below or equal to $16384/192$

Each SM also sports constant and texture caches. Constant caches enable fast memory accesses, as long as all threads try to access the same memory location; accesses are serialized otherwise. In the GT200, each SM contains 2KB of L1 constant cache, with a latency of 8 cycles [14]. Unlike the other fast memories, texture caches are outside the SM and have much higher latencies, placed around 260 cycles [14].

2) Arithmetic units: Each of the 8 ALUs can compute simple arithmetic instructions, be it integer, logical or single precision floating point, per cycle. Moreover, each ALU can compute an MAD (multiply-and-add) operation per cycle. Each SFU unit can compute transcendental functions (e.g., $\sin$, $\cos$), and contains 4 floating point multipliers. The GT200 also introduces a double-precision arithmetic unit per SM, able to dispatch 1 double precision multiply-add per cycle. Thus, an SM can compute $8 \cdot 2 + 4 \cdot 2 = 24$ floating-point operations per cycle — a 60-SM GPU at 1242 MHz can (theoretically) compute $1,788,480,000,000$ floating-point operations per second, a little over 1.7 Tflop/s, or around $149,040,000,000$ (149 TFlops) double precision floating-point operations per second.

Integer multiplication in the GT200, as in the G80, is limited to native 24-bit multiplication provided by the `mul24` and `mad24` instructions. Full 32-bit multiplication is performed by a combination of 24-bit multiplications, shifts and additions, which render its throughput around 4 times lower, from approx. 8 `mul24` per cycle to approx. 1.7 32-bit multiplications per cycle [14].

III. Fast modular exponentiation

The most important operation in RSA is modular exponentiation. In the case of RSA-1024 decryption, it is fairly standard to replace one large 1024-bit exponentiation by 2 smaller 512-bit independent ones — this technique, discovered by Quisquater, is often called the "CRT trick" [15]. This improvement matches quite well the GPU programming paradigm and, as such, in the remainder of this section we study the performance of 512-bit modular exponentiation in the GT200.

A. Integer representation

One of the first and foremost concerns when implementing multi-precision arithmetic on a new architecture is how to represent the numbers so as to better take advantage of the features of said architecture. In the case of the GT200, the biggest concerns are twofold: integer multiplication and carry propagation.

One can certainly use the usual positional number system with base 2^{32}, equal to the native word size. This allows one to use the native carry handling instructions for addition and subtraction[3], `addc` and `subr`. This representation has the drawback of "wasting" cycles by ignoring that the native integer multiplication of the GT200 is 24-bit.

[3]The NVCC compiler does not have intrinsic functions that enable one to make use of such instruction effectively. We workaround this issue by using NVCC's undocumented inline PTX assembly.

Using a base 2^{24} representation would do away with this waste, but would introduce 8 unused bits per word. Carry propagation would also need 2 instructions (a shift and add), as opposed to just one. We found that the storage waste represents a higher (throuhgput) penalty than the computation waste, and chose the base 2^{32} representation.

There are other, alternative, representations. Residue number systems are quite popular in the literature [10], [16], [17]; they fail to beat positional systems in throughput in seemingly all cases [10], [17]. Some implementations use floating point numbers to represent integers, e.g., [18]. Once again, representing an integer in $[-2^9, 2^9]$ reliably using floating point wastes some bits per word, i.e., the exponent bits.

B. Modular multiplication

Fast modular multiplication is at the heart of any fast modular exponentiation algorithm. We perform multiplication using the classic quadratic "schoolbook" algorithm. Asymptotically superior algorithms, such as Karatsuba [19] or Toom-Cook [20], [21], are not yet suitable at 512 bits — they require more temporary space [22] and more elementary operations than schoolbook multiplication.

Algorithm 1: Montgomery multiplication.

```
Input: Integers $A, B, N, R$. $N < R$ must be odd.
Output: Product $ABR^{-1} \bmod N$.
$\mu \leftarrow -1/N \bmod R$;
$C \leftarrow AB$;
$Q \leftarrow C\mu$;
$C \leftarrow (C + QN)/R$;
if $C \geq N$ then
    return $C - N$;
else
    return $C$;
end
```

Modular reduction is performed using Montgomery's algorithm [23]. Montgomery's algorithm essentially replaces one division by 2 multiplications, which is often much faster than schoolbook division, as common hardware (including GPUs) typically has poor integer division support. While Barrett's [24] reduction has the same complexity as Montgomery, Montgomery is often faster in software [25]. Algorithm 1 describes the original Montgomery algorithm for modular multiplication. As we can see, this algorithm requires enough storage for a full 512-bit multiplication, i.e., at least $2n$ words of temporary storage for n-word numbers. This is far from optimal.

Koç et al. [9] show how to implement Montgomery multiplication using $n+3$ words of temporary storage. By interleaving multiplication with reduction, Koç et al. were able to both reduce the storage requirements and increase the speed of modular multiplication. We implemented, tuned and benchmarked 3 of the most successful algorithms derived by this approach: CIOS, FIOS and FIPS.

In the following section, the 3 methods are described and their mapping into hardware explained. The notation used in this section is as follows: A and B are s-digits integers, and a_i (resp. b_i) denotes the ith digit of A (resp. B) for some digit length w; in our case, $w = 32$. N is the modulus, and is also s-digit long.

Algorithm 2: Montgomery multiplication (CIOS method).

```
Input: s-word integers $A, B, 2^{n-1} \leq N \leq 2^n$.
       $n'_0 = -n_0^{-1} \bmod 2^w$. $w$ is word bit length.
Output: Product $P = AB2^{-n} \bmod N$.
$P \leftarrow 0$;
for $i$ from $0$ to $s-1$ do
    $u \leftarrow 0$;
    for $j$ from $0$ to $s-1$ do
        $(u, v) \leftarrow a_j b_i + p_j + u$;
        $p_j \leftarrow v$;
    end
    $(u, v) \leftarrow p_s + u$;
    $p_s \leftarrow v$;
    $p_{s+1} \leftarrow u$;
    $q \leftarrow p_0 n'_0 \bmod 2^w$;
    $(u, v) \leftarrow p_0 + n_0 q$;
    for $j$ from $1$ to $s-1$ do
        $(u, v) \leftarrow n_j q + p_j + u$;
        $p_{j-1} \leftarrow v$;
    end
    $(u, v) \leftarrow p_s + u$;
    $p_{s-1} \leftarrow v$;
    $p_s \leftarrow p_{s+1} + u$;
end
if $P \geq N$ then return $P - N$;
else return $P$;
```

1) Montgomery by CIOS: The CIOS approach, described in Algorithm 2, is the most used approach both in software and in GPUs [10], [17]. CIOS stands for *Coarsely Integrated Operand Scanning*, which describes accurately how this algorithm works. For each word of the modulus, CIOS performs two separate loops: one for the multiplication step, another one for the reduction step. In each of these steps, the product is multiplied by one digit of the multiplicand, and subsequently reduced; this reduction is quite fast, since the the partial product is at most 32-bit larger than the modulus.

The operations are performed in operand scanning form, which translates into regular memory accesses across both operands and modulus. However, CIOS generates long carry chains across the inner loops that make instruction-level parallelism hard to accomplish.

2) Montgomery by FIOS: The *Finely Integrated Operand Scanning* method goes one step further and integrates both multiplication *and* reduction in the same inner loop, which is executed $s^2 - s$ times. This fine integration is this method's greatest advantage, as it reduces greatly the overhead and code size inside the outer loop. This method is particularly attractive for RISC processors [26].

However, the FIOS method requires the addition of 2-word quantities, which entails the propagation of a possible carry over to a third word. Instead, we follow the approach of [26], who alleviate the need for fast add-with-carry instructions with a redundant representation, where two w-bit words represent a $(w+1)$-bit quantity. Algorithm 3 describes this latter approach to FIOS.

Algorithm 3: Montgomery multiplication (FIOS method).

```
Input: s-word integers $A, B, 2^{n-1} \leq N \leq 2^n$.
       $n'_0 = -n_0^{-1} \bmod 2^w$. $w$ is word bit length.
Output: Product $P = AB2^{-n} \bmod N$.
$P \leftarrow 0$;
for $i$ from $0$ to $s-1$ do
    $(u, v) \leftarrow a_0 b_i + p_0$;
    $t \leftarrow u$;
    $q \leftarrow v n'_0 \bmod 2^w$;
    $(u, v) \leftarrow n_0 q + v$;
    for $j$ from $1$ to $s-1$ do
        $(u, v) \leftarrow a_j b_i + t + u$;
        $t \leftarrow u$;
        $(u, v) \leftarrow n_j q + p_j + v$;
        $p_{j-1} \leftarrow v$;
    end
    $(u, v) \leftarrow p_s + t + u$;
    $p_{s-1} \leftarrow v$;
    $p_s \leftarrow u$;
end
if $P \geq N$ then return $P - N$;
else return $P$;
```

3) Montgomery by FIPS: The *Finely Integrated Product Scanning Method* follows the alternative approach to multiplication popularized by Comba [27]. Instead of going through the multiplicands in the usual "schoolbook" fashion, FIPS' outer loop goes through the words of the final product itself. This results in a method with more potential parallelism (each word of the final product can be calculated individually until the very end), but it has the drawback of requiring more add-with-carry instructions — consecutive sums of products do not fit in just two words, and require a third one to house the resulting carries.

Although FIPS is "finely integrated", it does not result in a tight single inner loop like the FIOS method. In fact, as shown in Algorithm 4, FIPS has 2 outer loops, each with an inner loop that cannot be fully unrolled, as it depends on the current outer loop iteration. FIPS also has somewhat irregular memory access patterns, which does not help its performance. However, outer-loop iterations have more implicit parallelism than operand scanning methods, which is possible to exploit in fully unrolled implementations. This FIPS method is particularly successful in DSP and ASIC chip implementations [28], [29].

C. Exponentiation

Algorithm 4: Montgomery multiplication (FIPS method).

```
Input: $s$-word integers $A, B, 2^{n-1} \leq N \leq 2^n$.
       $n'_0 = -n_0^{-1} \bmod 2^w$. $w$ is word bit length.
Output: Product $P = AB2^{-n} \bmod N$.
$(t, u, v) \leftarrow 0$;
for $i$ from $0$ to $s-1$ do
    for $j$ from $1$ to $i-1$ do
        $(t, u, v) \leftarrow (t, u, v) + a_j b_{i-j}$;
        $(t, u, v) \leftarrow (t, u, v) + p_j n_{i-j}$;
    end
    $(t, u, v) \leftarrow (t, u, v) + a_i b_0$;
    $p_i \leftarrow v n'_0 \bmod 2^w$;
    $(t, u, v) \leftarrow (t, u, v) + p_i n_0$;
    $v \leftarrow u, u \leftarrow t, t \leftarrow 0$;
end
for $i$ from $s$ to $2s-1$ do
    for $j$ from $i-s+1$ to $s-1$ do
        $(t, u, v) \leftarrow (t, u, v) + a_j b_{i-j}$;
        $(t, u, v) \leftarrow (t, u, v) + p_j n_{i-j}$;
    end
    $p_{i-s} \leftarrow v$;
    $v \leftarrow u, u \leftarrow t, t \leftarrow 0$;
end
$p_s \leftarrow v$;
if $P \geq N$ then return $P - N$;
else return $P$;
```

Once fast modular multiplication is achieved, it is still necessary to minimize the number of multiplications through an efficient exponentiation algorithm. Among the choices available, we have the classic *binary exponentiation*, *k-ary exponentiation* and *sliding window* exponentiation [30, Section 2.6]. The sliding window method results in the lowest amount of modular multiplications.

Given the low fast memory space available in the GPU, we chose the sliding window method with a window size smaller than recommended: the optimal size in our implementation for 512 bits was 4 (instead of the theoretical best 5), which requires the storage of only 8 extra numbers. The extra logic required for the sliding window implementation is negligible compared to the savings in modular multiplications.

D. CUDA implementation

In our implementation, we perform one entire 512-bit exponentiation per thread, to maximize throughput. To maximize the amount of registers available per thread, an initial thread block size of 128 was considered. This, however, translated into poor GPU occupancy. Since a thread block size of 192 is the minimum recommended size [6], we selected this as our thread block size, leaving us with 84 registers per thread to work with. Later, we decreased the sliding-window size to 4 and increased the block size to 224 threads (resp. 64 registers per thread), with slightly superior results.

```
__device__ void __umul32mad1(u32 a, u32 b, u32 c,
    u32 &p0, u32 &p1)
{
    asm("{ .reg .u64 %prod; \n\t"
        ".reg .u64 %sum; \n\t"
        "cvt.u64.u32 %sum, %4; \n\t"
        "mad.wide.u32 %prod, %2, %3, %sum; \n\t"
        "cvt.u32.u64 %0, %prod;      \n\t"
        "shr.u64 %prod, %prod, 32;  \n\t"
        "cvt.u32.u64 %1, %prod;      \n\t"
        "}                          \n\t"
        : "=r"(p0), "=r"(p1)
        : "r"(a), "r"(b), "r"(c));
}
```

Figure 3. CUDA device function, using PTX assembly, for the $a \times b + c$ operation.

To make the above algorithms fast, we had to perform heavy loop unrolling. Loop unrolling serves two purposes: remove the necessity of indexed accesses to words, storing them in registers, and remove the overhead associated with loop control flow. We performed unrolling manually, and used inline PTX assembly whenever appropriate, i.e., to perform 32×32-bit wide multiplications (and respective multiply-add instructions) and to use add-with-carry instructions not available in the CUDA C dialect. Figure 3 depicts one of the most important uses of inline PTX in our implementation, the $a \times b + c$ operation, with 32 bit inputs and 64 bit output. Some unrolled implementations were cleaner than others: FIOS was particularly unrolling friendly, due to its simple single inner loop. FIPS was the opposite, and required unrolling both the inner and outer loops, which resulted in a performance penalty.

Whenever possible, we specialized the modular multiplication algorithms to the squaring case, thus saving a significant amount of multiplications due to the symmetry of this operation. This was one of the single biggest performance boosts, following inline PTX assembly use. Squaring adaptations of all the methods, however, were not unrolling-friendly, requiring full manual unrolling.

To avoid divergent threads, and following results by Walter et al. [31], [32], we added one extra word to all algorithms. This removes the necessity of the final conditional subtraction in all variants of Montgomery multiplication, thus avoiding diverging threads and possible side-channel attacks [33]. This had no negative performance impact in our implementation; on the contrary, it was actually slightly faster.

IV. Related work

The first GPU implementation of a public key primitive was performed by Moss et al. [16] on an NVIDIA 7800 GTX GPU. Moss et al. used residue number systems (RNS) to represent large integers and performed modular exponentiation in this representation with moderate success: a speedup factor of 3 relative to the reference CPU was obtained when computing batches of 100000 modular exponentiations. Also on the NVIDIA 7800 GTX, Fleissner [34] implemented 192-bit modular exponentiation. Numbers of this length, however, are not suitable for (non-elliptic curve) public-key cryptographic keys.

More recently, Szerwinski et al. employed the newer G80 architecture from NVIDIA and the CUDA framework to develop efficient modular exponentiation and elliptic curve scalar multiplication [17]. Their work, which included implementations of both Montgomery and RNS arithmetic, yielded a throughput of up to 813 modular exponentiations per second on an NVIDIA 8800GTS; the minimum latency for this throughput, however, was of over 6 seconds. Harrison and Waldron improved this figure for the special case of RSA-1024 decryption, where they obtained a peak throughput of 5536.75 decryptions per second [10].

Elliptic curve cryptography GPU implementations have also been studied in the literature [17], [18], [35]–[38]. The current champion is the implementation of [36], which reports 481 million 210-bit modular multiplications per second on an NVIDIA GTX 295.

In parallel to our work, Jang et al. [39] have announced preliminary results in the NVIDIA Fermi architecture. They achieved 74732 RSA-1024 decryptions per second on a GTX 580 card. They achieve this by parallel arithmetic across threads using lazy carry propagation, a well-known strategy in vector processors [40]–[45].

V. Results and discussion

Our performance figures were measured on an NVIDIA GTX260 GPU. Figure 4 shows the throughput obtained when running our exponentiation kernel on batches of increasing amounts of 512-bit exponentiations, using each of the interleaving methods described in Section III. Evidently the performance of modular exponentiation on the GPU is highly dependent on the amount of parallel work it receives: a single exponentiation performed will be quite slow in comparison with common CPUs. The FIOS method is also clearly superior in seemingly all cases, with the CIOS method trailing behind both FIOS and FIPS.

Figure 5 shows the observed latency of our implementation, and compares it to a reference CPU implementation using GMP [46] and an Intel Xeon W3565 processor. When performing a single exponentiation, the GPU has an enormous latency when compared to the CPU. The GPU quickly recovers, and when performing about 1000 simultaneous exponentiations, it already beats the CPU.

Despite considerably improving throughput our implementation does not suffer from added latency relatively to previous implementations. The latency for a single exponentiation in our implementations is 70ms; the implementation of [17] has a minimum of 6930ms. [10] reports a latency of 218ms when executing batches of 1024 exponentiations; when executing batches of 1024 exponentiations, we achieve a latency of 109ms, which is maintained for batches of up to 4608 exponentiations.

Table I summarizes the results of the literature and our own

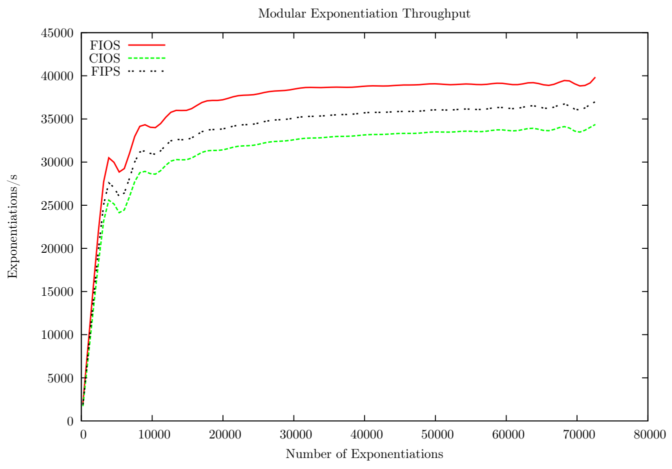

Figure 4. 512-bit modular exponentiation throughputs on the GPU with different Montgomery interleaving methods.

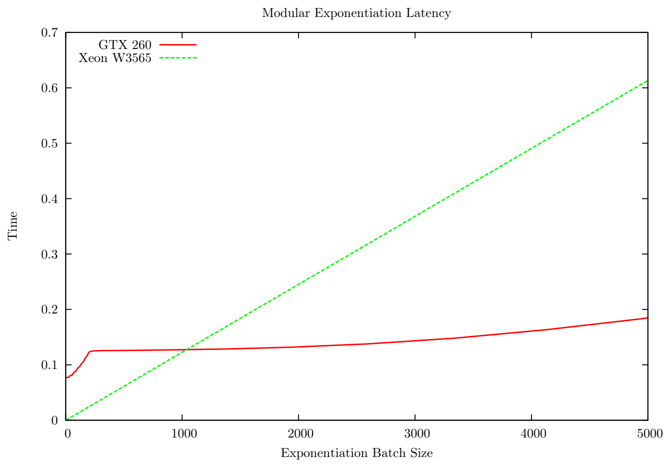

Figure 5. 512-bit modular exponentiation latency on the GPU and CPU.

on 512-bit modular exponentiation speed[4]. We scaled some of the results as required; we assumed a perfect speedup of 8 from the 813 modular exponentiations per second of [17], and we multiplied the 1024-bit decryption results of [10] and [39] by 2, to have a common operation for all data points — 512-bit modular exponentiation. The "Modexp/s (scaled)" line of Table I scales each of the GPU results into our hardware, the NVIDIA GTX 260, taking into account number of cores and frequency.

Our implementation is faster than the previous ones, i.e., [17] and [10], even when taking hardware differences into account. Our implementation fails to outperform the recent work of [39] by a small margin, which is easily explained by the support of full 32×32 integer multiplications of the Fermi architecture. When compared to high-end processors,

[4]The price information was obtained at the time of this writing from the lowest price for each model found on the shopping search website http://www.pricegrabber.com/.

	GTS8800 [17]	GTX8800 [10]	GTX260 (This paper)	GTX580 [39]	Intel W3565 [46]	AMD Phenom II 1090T [46]
Cores	112	128	192	512	4	6
Frequency (MHz)	1188	1350	1294	1544	3200	3200
Price (USD)	250	173	100	500	300	200
TDP (W)	150	155	202	244	130	125
GFLOPS	399	518	715	1581	102	153
Modexp/s	6504	11074	41426	149464	32608	77002
Modexp/s (scaled)	13052	15282	41426	46973	N/A	N/A
Modexp/s/W	43	71	205	612	250	616
Modexp/s/USD	26	64	414	298	131	385

Table I
COMPARISONS OF MODULAR EXPONENTIATION PERFORMANCES ON VARIOUS CPU AND GPU IMPLEMENTATIONS.

our implementation outperforms Intel, but not AMD CPUs.

Jang et al [39] claim their implementation of RSA is comparable to 3 hexa-core processors. They focused only, however, on Intel CPUs. As Table I shows, AMD processors have quite a performance advantage over Intel processors when it comes to fast modular arithmetic. One GTX580 GPU using the implementation in [39] is, instead, equivalent to a little over 12 AMD K10 cores at 3.2GHz. Such 12-core chips are already available as the AMD Opteron 6100 series [47].

Performance per watt changes the landscape considerably. Out implementation is able to perform 205 exponentiations per second per watt. The recent GTX 580 card packs many more cores in the same die, while consuming only slightly more energy; this raises its performance to nearly 3 times our implementation, at 612 exponentiations per second per watt. The GTX 580 is only beat in performance per watt by the AMD processor, which has the best performance/watt ratio of 616. When it comes to performance per dollar, our numbers surpass every other, at 414 exponentiations per second per dollar.

It is harder to fairly compare our work with the elliptic curve arithmetic implementation of Bernstein et al [36]. Their article reports 481 million 210-bit modular multiplications per second on an NVIDIA GTX295, running at the same frequency as our GTX260 but with 2.5 times more SMs. Scaling down, [36] is expected to obtain 194.4 million 210-bit modular multiplications per second on a GTX260 card. Our own implementation performs roughly 26 million 512-bit modular multiplications per second. Adjusting by a quadratic factor ($(512/210)^2$) to 210-bit modular multiplications results in 155 million modular multiplications per second. However, [36] reports that their implementation slows down by a logarithmic factor of -2.46 as modulus bit length rises; using this adjusting exponent, our implementation would achieve 233 million modular multiplications.

VI. CONCLUSION

In this paper we have carefully implemented the arithmetic required for fast RSA-1024 decryption in GT200 GPUs. Our findings were twofold: previous implementations were underusing the full potential of G80 and GT200 GPUs, accessible only through direct PTX assembly. We also found that CIOS, the usual technique to interleave Montgomery multiplication and reduction, is not optimal on the GT200. Both FIPS and FIOS showed to be superior, and FIOS enabled us to reach the best recorded performances in the GT200 architecture to date.

While our GPU implementation has excellent performance, its performance/watt ratio is lower than that of a regular processor. In fact, no GPU implementation of RSA has been able to beat the best CPUs in this metric. The work of Bernstein et al [36] has been able to beat AMD processors in performance per watt; they did so, however, when dealing with integers of at most 300 bits, where there is much less register and memory pressure. Thus, at this point CPUs still offer better performance per watt for modular exponentiation. It's harder to assess performance per dollar, since prices are quite volatile. As older graphics cards become increasingly common and cheaper, however, we expect this ratio to remain stronger for GPUs than CPUs.

ACKNOWLEDGMENT

This work has been supported by the project CMU-PT/RNQ/0015/2009, TRONE — Trustworthy and Resilient Operations in a Network Environment.

REFERENCES

[1] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, pp. 644–654, 1976.

[2] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[3] National Institute of Standards and Technology, "The digital signature standard, proposal and discussion," *Communications of the ACM*, vol. 35, no. 7, pp. 36–54, July 1992.

[4] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan, "Anatomy and Performance of SSL Processing," in *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 197–206.

[5] J. Nickolls and W. J. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, pp. 56–69, 2010.

[6] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2010. [Online]. Available: {http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf}

[7] A. Munshi, *OpenCL 1.0 Specification*, Khronos Group, May 2009, uRL: http://www.khronos.org/registry/cl/.

[8] I. Ristic, "Internet SSL Survey 2010," https://community.qualys.com/docs/DOC-1421, July 2010.

[9] c. K. Koç, T. Acar, and B. S. Kaliski, Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, 1996.

[10] O. Harrison and J. Waldron, “Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware,” in *AFRICACRYPT*, ser. Lecture Notes in Computer Science, B. Preneel, Ed., vol. 5580. Springer, 2009, pp. 350–367.

[11] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[12] , “The CUDA Compiler Driver NVCC,” http://people.maths.ox.ac.uk/gilesm/cuda/doc/nvcc.pdf, August 2010.

[13] W. J. van der Laan, “decuda,” https://github.com/laanwj/decuda, July 2009.

[14] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying GPU microarchitecture through microbenchmarking,” in *ISPASS*. IEEE Computer Society, 2010, pp. 235–246.

[15] J. Quisquater and C. Couvreur, “Fast Decipherment Algorithm for RSA Public-key Cryptosystem,” *Electronics Letters*, vol. 18, no. 21, pp. 905–907, October 1982.

[16] A. Moss, D. Page, and N. Smart, “Toward Acceleration of RSA Using 3D Graphics Hardware,” in *Cryptography and Coding*. Springer-Verlag LNCS 4887, December 2007, pp. 369–388. [Online]. Available: http://www.cs.bris.ac.uk/Publications/Papers/2000772.pdf

[17] R. Szerwinski and T. Güneysu, “Exploiting the Power of GPUs for Asymmetric Cryptography,” in *Cryptographic Hardware and Embedded Systems — CHES 2008*, 2008, pp. 79–99.

[18] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang, “ECM on Graphics Cards,” in *EUROCRYPT ’09: Proceedings of the 28th Annual International Conference on Advances in Cryptology*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 483–501.

[19] A. A. Karatsuba and Y. Ofman, “Multiplication of multidigit numbers on automata,” *Soviet Physics Doklady*, vol. 7, pp. 595–596, 1963.

[20] A. L. Toom, “The complexity of a scheme of functional elements realizing the multiplication of integers,” *Soviet Mathematics Doklady*, vol. 3, pp. 714–716, 1963.

[21] S. A. Cook, “On the minimum computation time of functions,” Ph.D. dissertation, 1966.

[22] R. Maeder, “Storage Allocation for the Karatsuba Integer Multiplication Algorithm,” in *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, ser. DISCO ’93. London, UK: Springer-Verlag, 1993, pp. 59–65. [Online]. Available: http://portal.acm.org/citation.cfm?id=646137.680622

[23] P. L. Montgomery, “Modular multiplication without trial division,” *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[24] P. Barrett, “Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor,” in *Advances in Cryptology - CRYPTO ’86, Santa Barbara, California*, ser. LNCS, A. Odlyzko, Ed., vol. 263. Springer, 1987, pp. 311–323.

[25] A. Bosselaers, R. Govaerts, and J. Vandewalle, “Comparison of three modular reduction functions,” in *Proceedings of the 13th annual international cryptology conference on Advances in cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1994, pp. 175–186. [Online]. Available: http://portal.acm.org/citation.cfm?id=188105.188147

[26] J. Großschädl and G.-A. Kamendje, “Optimized RISC Architecture for Multiple-Precision Modular Arithmetic,” in *Security in Pervasive Computing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 2802, pp. 105–174. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-39881-3_22

[27] P. G. Comba, “Exponentiation Cryptosystems on the IBM PC,” *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, 1990.

[28] D. Zhang, M. Gao, L. Li, Z. Cheng, and X. Wang, “An implementation method of a rsa crypto processor based on modified montgomery algorithm,” in *Asicon 2003: 2003 5th International Conference on ASIC: Proceedings*, 2003, pp. 1332–1336.

[29] P. Gastaldo, G. Parodi, and R. Zunino, “Enhanced Montgomery Multiplication on DSP Architectures for Embedded Public-Key Cryptosystems,” *EURASIP Journal on Embedded Systems*, p. 9, 2008.

[30] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[31] C. D. Walter, “Montgomery exponentiation needs no final subtractions,” *Electronics Letters*, vol. 35, no. 21, pp. 1831–1832, October 1999.

[32] G. Hachez and J.-J. Quisquater, “Montgomery exponentiation with no final subtraction: Improved results,” in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000*, C. P. Ç.K. Koç, Ed. Springer-Verlag, 2000, pp. 293–301.

[33] C. D. Walter, “Leakage from montgomery multiplication,” in *Cryptographic Engineering*, Ç. K. Koç, Ed. Springer US, 2009, pp. 431–449. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-71817-0_16

[34] S. Fleissner, “GPU-Accelerated Montgomery Exponentiation,” in *ICCS ’07: Proceedings of the 7th international conference on Computational Science, Part I*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 213–220.

[35] P. Giorgi, T. Izard, and A. Tisserand, “Comparison of modular arithmetic algorithms on gpus,” in *Proc. International Conference on Parallel Computing ParCo*, Lyon, France, September 2009.

[36] D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang, “The billion-mulmod-per-second PC,” in *Proceedings 4th Workshop on Special-purpose Hardware for Attacking Cryptograhic Systems*, September 2009, pp. 131–144.

[37] S. Antao, J.-C. Bajard, and L. Sousa, “Elliptic Curve point multiplication on GPUs,” in *ASAP*, F. Charot, F. Hannig, J. Teich, and C. Wolinski, Eds. IEEE, 2010, pp. 192–199.

[38] A. E. Cohen and K. K. Parhi, “GPU Accelerated Elliptic Curve Cryptography in $GF(2^m)$,” in *Proceedings of the 2010 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Seattle, WA, August 2010, pp. 57–60.

[39] K. Jang, S. Han, S. Han, S. Moon, and K. Park, “SSLShader: Cheap SSL Acceleration with Commodity Processors,” http://www.ndsl.kaist.edu/papers/sslshader.pdf, March 2011, to appear in NSDI 2011.

[40] D. H. Bailey, “The computation of π to 29,360,000 decimal digits using Borweins’ quartically convergent algorithm,” *Mathematics of Computation*, vol. 50, pp. 283–296, 1988.

[41] P. L. Montgomery, “An fft extension of the elliptic curve method of factorization,” Ph.D. dissertation, Los Angeles, CA, USA, 1992, uMI Order No. GAX93-10894.

[42] A. Lenstra, “Massively parallel computing and factoring,” in *LATIN ’92*, ser. Lecture Notes in Computer Science, I. Simon, Ed. Springer Berlin / Heidelberg, 1992, vol. 583, pp. 344–355, 10.1007/BFb0023840. [Online]. Available: http://dx.doi.org/10.1007/BFb0023840

[43] B. Dixon and A. K. Lenstra, “Massively parallel elliptic curve factoring,” in *Proceedings of the 11th annual international conference on Theory and application of cryptographic techniques*, ser. EUROCRYPT’92. Berlin, Heidelberg: Springer-Verlag, 1993, pp. 183–193. [Online]. Available: http://portal.acm.org/citation.cfm?id=1754948.1754970

[44] T. Jebelean, “Integer and rational arithmetic on masPar,” in *Design and Implementation of Symbolic Computation Systems*, ser. Lecture Notes in Computer Science, J. Calmet and C. Limongelli, Eds. Springer Berlin / Heidelberg, 1996, vol. 1128, pp. 162–173. [Online]. Available: http://dx.doi.org/10.1007/3-540-61697-7_15

[45] S. Neves, “Cryptography in GPUs,” Master’s thesis, Universidade de Coimbra, Coimbra, July 2009. [Online]. Available: http://eden.dei.uc.pt/~sneves/gpucrypto.pdf

[46] T. G. et al., “GNU multiple precision arithmetic library 5.0.1,” December 2010, http://gmplib.org/.

[47] “AMD Opteron 6000 Series Platform,” http://www.amd.com/us/products/server/processors/6000-series-platform/pages/6000-series-platform.aspx, March 2011.