Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware

Owen Harrison and John Waldron

Computer Architecture Group, Trinity College Dublin, Dublin 2, Ireland,
harrisoo@cs.tcd.ie, john.waldron@cs.tcd.ie

Abstract. Graphics processing units (GPU) are increasingly being used for general purpose computing. We present implementations of large integer modular exponentiation, the core of public-key cryptosystems such as RSA, on a DirectX 10 compliant GPU. DirectX 10 compliant graphics processors are the latest generation of GPU architecture, which provide increased programming flexibility and support for integer operations. We present high performance modular exponentiation implementations based on integers represented in both standard radix form and residue number system form. We show how a GPU implementation of a 1024-bit RSA decrypt primitive can outperform a comparable CPU implementation by up to 4 times and also improve the performance of previous GPU implementations by decreasing latency by up to 7 times and doubling throughput. We present how an adaptive approach to modular exponentiation involving implementations based on both a radix and a residue number system gives the best all-around performance on the GPU both in terms of latency and throughput. We also highlight the usage criteria necessary to allow the GPU to reach peak performance on public key cryptographic operations. [1]

Key Words: Graphics Processor, Public-Key Cryptography, RSA, Residue Number System.

1 Introduction

The graphics processing unit (GPU) has enjoyed a large increase in floating point performance compared with the CPU in the last number of years. The traditional CPU has leveled off in terms of clock frequency as power and heat concerns increasingly become dominant restrictions. The latest GPU from Nvidia's GT200 series reports a peak throughput of almost 1 TeraFlop, whereas the latest Intel CPUs reported throughput is in the order of 100 GigaFlops [1]. This competitive advantage of the GPU comes at the price of a decreased applicability to general purpose computing. The latest generation of graphics processors, which

[1] Appeared in: AfricaCrypt 2009. International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009.

are DirectX 10 [2] compliant, support integer processing and give more control over the processor's threading and memory model compared to previous GPU generations. We use this new generation of GPU to accelerate public key cryptography. In particular we use an Nvidia 8800GTX GPU with CUDA [3] to investigate the possibility of high speed 1024-bit RSA decryption. We focus on 1024-bit RSA decryption as it shows a high arithmetic intensity, ratio of arithmetic to IO operations, and also allows easy comparison with CPU implementations. We exploit the new GPU's flexibility to support a GPU sliding window [4] exponentiation implementation, based on Montgomery exponentiation [5] using both radix and residue number system (RNS) representations. We investigate both types of number representation showing how GPU occupancy and inter thread communication plays a central role to performance. Regarding the RNS implementations, we exploit the GPU's flexibility to use a more optimised base extension approach than was previously possible. We also explore various GPU implementations of single precision modular multiplication for use within the exponentiation approaches based on RNS.

2 GPU Background

The GPU that we have used in our implementations is Nvidia's 8800GTX (a 2006 release), part of the G80 series, which was the first DirectX 10 [2] compliant GPU released by Nvidia. It is Nvidia's first processor that supports the CUDA API [3] and as such all implementations using this API are forward compatible with newer CUDA compliant devices. All CUDA compatible devices support 32-bit integer processing. The 8800GTX consists of 16 SIMD processors, called Streaming Multiprocessors (SM). Each SM contains 8 ALUs which operate in lockstep controlled by a single instruction unit. The simplest instructions are executed by the SMs in 4 clock cycles, which creates an effective SIMD width of 32. Each SM contains a small amount of fast local storage which consists of a register file, a block of shared memory, a constant memory cache and a texture memory cache. The main bulk of the GPU's storage is off-chip global memory, and is thus considerably slower than the on-chip storage. A lot of the programming effort concerning a GPU is the careful usage of the different types of available storage. This is due to the scarcity of fast storage and so performance can degrade dramatically with naive implementations.

The code which runs on the GPU is referred to as a kernel. A kernel call is a single invocation of the code which runs until completion. The GPU follows a Single Process, Multiple Data (SPMD, or recently SIMT) threading model [1]. All threads must run from the same static code and must all finish before a kernel can finish. Via the CUDA API, the programmer can specify the number of threads that are required for execution during a kernel call. Threads are grouped into a programmer defined number of CUDA blocks, where each block of threads is guaranteed to run on a single SM. Threads within a block can communicate using a synchronisation barrier which ensures that all threads within the block fully commit to the same instruction before proceeding. The number of threads

per block is also programmer defined. They should be allocated in groups of 32, called a CUDA warp, to match the effective SIMD width mentioned above. If the thread execution path within a warp diverge, all paths must be executed serially on the SM. An important consideration for GPU performance is its level of occupancy. Occupancy refers to the number of threads available for execution at any one time. It is normally desirable to have a high level of occupancy as it facilitates the hiding of memory latency.

3 Related Work

The earliest attempts at accelerating cryptography using GPUs involved symmetric primitives [6–9]. Concerning public key acceleration, Moss et al. [10] presented an implementation of 1024-bit exponentiation using an RNS approach. This paper demonstrated the feasibility of a public key implementation on a GPU with 1024-bit exponentiation throughput rates of 5.7 ms/op (a theoretical rate of 1.42 ms/op (5.7/4) for 1024-bit RSA decryption). Their approach was restricted due to the use of the Nvidia 7800GTX GPU, which is not DirectX 10 compliant. Fleissner et al. [11] presented an acceleration of 192-bit exponentiation also on the 7800GTX and as such has similar restrictions as the above approach. The paper unfortunately only presents its results in terms of ratio comparisons with its own CPU implementation so objective comparisons cannot be made.

Most recently, Szerwinski et al. [15] presented implementations of 1024 and 2048-bit modular exponentiations based on both radix and RNS approaches using an Nvidia 8800GTS (slightly slower GPU than the one used in this paper). The maximum throughput achieved was via a radix based approach, resulting in 1.2 ms/op for 1024 bit modular exponentiation, though with a latency of 6.9 seconds (˜0.3 ms/op with 1.7 seconds latency for a 1024-bit RSA decryption). Note, we multiply by 4 the throughput and correspondingly divide by 4 the latency to give an approximation of the RSA-1024 performance, i.e. 2 512-bit modular exponentiations [4]; this approximation approach is used in the results sections later in this paper. Due to this high latency the paper concludes that the GPU's maximum throughput can be achieved only in contexts where latency is irrelevant and thus its usefulness is quite restricted. The RNS approach presented displayed better latency characteristics, though with a throughput rate of 2.3 ms/op (0.57 ms/op for RSA-1024), which is only marginally better than the Crypto++ CPU implementation [12]. We show that with careful memory management, Section 4.1, one can reduce the high latency of a radix approach by over 7 times. We also show that through a detailed analysis of the underlying modular operations used, Section 5.2, and the optimisation of the employed RNS algorithm, Section 5.3, we can double the throughput of the RNS approach. The paper, [15], also restricts the use of the GPU to a single exponent per batch of processing, we outline how different keys can be effectively used in both radix and RNS approaches.

4 Standard Montgomery Exponentiation on the GPU

We present two different GPU implementations with varying degrees of parallelism incorporating the Montgomery reduction method in radix representation and pencil-and-paper multiplication. One observation that applies to all implementations of exponentiation on a CUDA compatible device is that it is only suitable to use a single exponent per CUDA warp, and in some scenarios per CUDA block. The reason for this is that the exponent largely determines the flow of control through the code. These conditional code paths dependant on the exponent cause thread divergence. When threads within a CUDA warp diverge on a single processor, all code paths are executed serially, thus a large performance overhead is incurred for threads that diverge for large portions of code. If inter thread communication is required, a synchronisation barrier must be used to prevent race conditions occurring. All threads within a CUDA block that perform a synchronisation barrier must not be divergent at the point of synchronisation. Thus all threads within a single CUDA block are required to execute the same path at points of synchronisation and so it follows that for exponentiation that uses inter thread communication, only one exponent can be used per CUDA block.

4.1 Serial Approach

Each thread within this implementation performs a full exponentiation without any inter thread communication or cooperation. This is a standard optimised implementation of an exponentiation using the Quisquater and Couvreur CRT approach [21], operating on two independent pairs of 16 limb numbers. The approach also uses the sliding window technique to reduce the number of Montgomery multiplies and squares required. As a single thread computes an exponentiation independently, a single exponent must be used across groups of 32 threads. In terms of RSA, assuming peak performance, this implementation is restricted to using a maximum of 1 key per 32 primitives (or messages). As we are using the CRT based approach to split the input messages in two, we also use two different exponents for a single message. Thus a message must be split into different groups of 32 threads to avoid guaranteed thread divergence. We have adopted a simple strategy to avoid divergence, whereby CUDA blocks are used in pairs. The first block handles all 16 limb numbers relating to the modulus p and the second block handles all numbers relating to the modulus q, where $n = pq$ and n is the original modulus. Note that there is an underlying assumption that the input data ensures the maximum number of keys per primitives is upheld (as mentioned above). The threading model employed is illustrated in Figure 1. This separation of p and q related data is also used in the implementations in Section 4.2 and 5.

The added support for integers, bitwise operations and increased memory flexibility such as scatter operations, in the 8800GTX, allows this implementation to execute largely in a single kernel call. The byte and bit manipulation operations required for the efficient implementation of sliding window are now

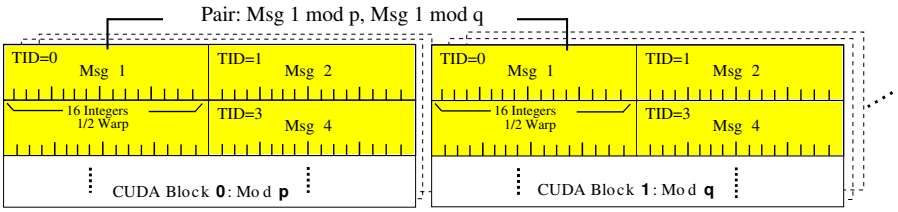


Fig. 1. Serial Thread Model.

straightforward. The macro level details of this algorithm are largely standard, consisting of: CRT and sliding window; Montgomery multiplication across normal radix representation; input data is first converted to Montgomery representation; the data is then multiplied or squared according to the exponent; and finally a Montgomery multiplication by 1 to undo the initial Montgomery representation. As such, we do not list the high level steps of the algorithm, however we draw attention to the following optimisations that were applied within the implementation: all $N \times N$ limb multiplies used cumulative addition to reduce memory operations [13]; all squaring requirements were optimised to reduce the number of required multiplies [4]; $N \times N$ limb multiplies $\bmod R$ were truncated, again to remove redundant multiplies; and the final two steps within Montgomery multiplication were combined into a single $N \times N$ multiply and accumulate. These optimisations are listed here as they are relevant to the implementation in Section 4.2.

Memory usage: The concept of a uniform, hierarchical memory structure such as a CPU's L1/L2 cache etc does not exist on the GPU and performance cliffs can be encountered without careful memory planning. The following are the highlights of the various memory interactions of this implementation. Note that the implementations in Section 4.2 and Section 5 use similar adaptive memory approaches as described below.

Adaptive memory approaches: The sliding window technique requires the pre-calculation of various powers of the input data. This data is used during the exponentiation process to act as one of the N limb inputs into an $N \times N$ multi-precision multiplication. There are two options on how to handle the storage and retrieval of this pre-calculated data. **1.** The pre-calculation is done on the GPU and is written to global memory. The data is stored in a single array with a stride width equal to the number messages being processed in a single kernel call multiplied by the message size. Reads are then made subsequently from this array direct from global memory. In this scenario only a single kernel call is required for the exponentiation. **2.** Unfortunately the data reads cannot be coalesced as each thread reads a single limb which is separated by 16 integers from the next message. Coalesced global reads require the data to start at a 128-bit boundary for a warp and require each thread of the warp to read consecutively from memory with a stride of up to 4 32-bit integers wide. Non-coalesced reads generate separate memory transactions thus significantly reducing load/store

throughput. To ameliorate this the sliding window pre-calculation data is first generated in an initialisation kernel writing its results to global memory. A texture can then be bound to this memory and the subsequent exponentiation kernel can use the pre-calculation data via texture references. Note that texture access uses the texture cache, which is a local on chip cache, however textures cannot be written to directly hence the need for a separate initialisation kernel. The first approach described above is better for smaller amounts of data. The second approach is beneficial for larger amounts of data when the advantage of texture use outweighs the fixed overhead of the extra kernel call.

Another adaptive memory approach concerns the exponent. As mentioned, the exponent must be the same across a warp number of threads, thus all threads within a warp, when reading the exponent, access the same memory location at any one time. Constant memory has by far the best performance under this scenario [9], however is limited to 64KB on the G80. As each exponent requires 32 integers worth of storage, in an RSA 1024-bit context we can use constant memory for up to 512 different keys. If the amount of exponents exceed this threshold (in practice lower than 512 different keys as a small amount of constant memory is used for other purposes and a new key is used for at least each new block whether needed or not for lookup efficiency) then texture memory is used.

Other memory considerations: In an aim to increase the $N \times N$ multiplication performance we have allocated all of the on chip fast shared memory for storing and retrieving the most commonly used N limb multiplicand of the $N \times N$ operation. The less frequently accessed multiplier is retrieved from textures when possible. The input and output data is non exceptional in this implementation save that it cannot be coalesced due to the message stride within memory. A convolution of multiple messages could be an option to offset the lack of coalescing though this has not been explored here and would seem to be just adding extra steps to the CPU processing side. The other per key variables, $-n^{-1} (\text{mod } R)$ and $R^2 (\text{mod } n)$ (for use in generating the Montgomery representation of the input) for both moduli p and q, where $n = pq$, are stored and loaded via texture references. In the context of RSA decryption these variables are assumed to be pre-calculated and it should be noted that performance will degrade slightly if these have to be calculated with a high frequency. The results for this implementation are presented in Section 4.3 in conjunction with the parallel approach described below. Note that two parts of the exponentiation are not included in these implementations, the initial $x (\text{mod } p)$, $x (\text{mod } q)$ and the final CRT to recombine, these are done on the CPU. This is also the case for all implementations reported in this paper. These steps contribute little to the overall exponentiation runtime and so the performance impact is expected to minor.

4.2 Parallel Approach

This approach uses the same macro structure as the algorithm used above, however it executes the various stages within the algorithm in parallel. Each thread

is responsible for loading a single limb of the input data, with 16 threads combining to calculate the exponentiation. Each thread undergoes the same high level code flow, following the sliding window main loop, however the Montgomery multiplication stages are implemented in parallel. This approach relies heavily on inter thread communication, which has a performance overhead as well as an implication that only one exponent is supported per CUDA block. As the number of threads per block in this implementation is limited to 256, due to shared resource constraints, the number of 1024-bit RSA primitives per key in effect is limited to a minimum of 16. This is a hard limit in that the code is defined to have undefined behaviour if threads are divergent at points of synchronisation. The threading model uses the same separation of message pairs, for p and q, as in Figure 1, however, a single thread reads only a single integer.

The intensive $N \times N$ multiplies within the Montgomery multiplication are parallelised by their straight forward separation into individual $1 \times N$ limb multiplications. Each thread is independently responsible for a single $1 \times N$ limb multiply. This is followed by a co-operative reduction across all threads to calculate the partial product additions. This parallel reduction carries with it an overhead where more and more threads are left idle. Figure 2 shows the distribution of the $N \times N$ operation across the 16 threads and its subsequent additive reduction. It also shows the use of shared memory to store the entire operations output and input of each stage. As previously noted, the number of threads per block is limited to 256, so each RSA primitive can use up to 1KB (16KB / (256/16)) worth of shared memory. This allows for the entire $N \times N$ calculation to fit within shared memory. Also shown in the Figure 2 are the synchronisation points used to ensure all shared memory writes are committed before subsequent reads are performed. As this code is the most intensive part of the exponentiation, these synchronisation calls add a significant burden to the performance.

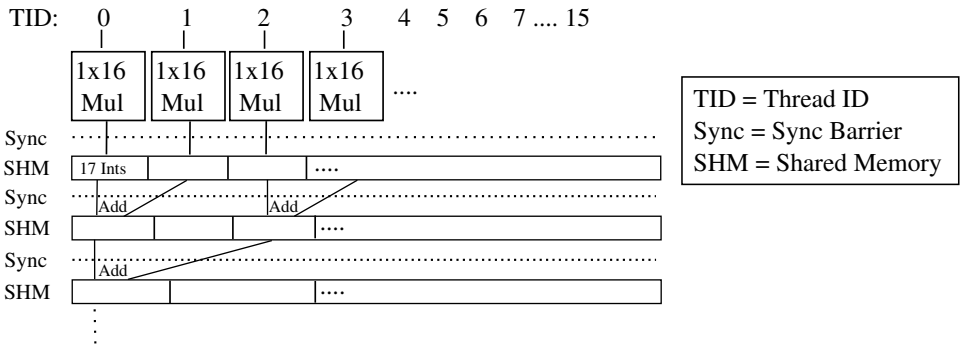


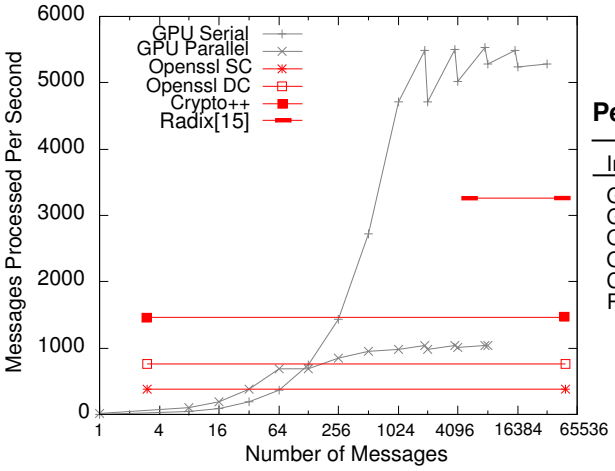
Fig. 2. $N \times N$ limb multiplication in parallel on a CUDA device.

The optimisations applied to the different $N \times N$ multiplies, listed in the serial approach, are not possible in the parallel approach. The squaring optimisation,

and also the modulo multiplication step, in general only execute half the limb multiplies that are required compared to a full $N \times N$ multiply. However, the longest limb within the $N \times N$ multiply dictates its overall execution time as all threads within a warp must execute in lock step. Thus, although one thread only executes a single multiply, it must wait until the largest $1 \times N$ multiply finishes. Also, as each thread executes its own $1 \times N$ multiply separately, the cumulative approach to addition must also be separated from the multiplication process. The results for this approach are presented below.

4.3 Results

Figure 3 illustrates the performance of both the parallel and serial approaches presented above. All measurements presented represent the number of 1024-bit RSA decrypt primitives executed per second. The GPU implementations show their dependence on an increasing number of messages per batch of work to approach their peak performance. This is due to having more threads available to hide memory read/write latency and also an increased ratio of kernel work compared to the fixed overheads associated with data transfer and kernel calls. We can see the advantage of the parallel approach over the serial approach at lower primitives per kernel call due to an higher level of occupancy. However the performance bottlenecks of excessive synchronisations and lack of optimisations limit the parallel approach.



Peak Performance

Impl.	Msgs p/s	ms/op
GPU Serial	5536.75	0.18
GPU Parallel	1041.55	0.96
Openssl SC	379.7	2.63
Openssl DC	762.5	1.31
Crypto++	1470.58	0.68
Radix [15]	3252.0	0.30

Fig. 3. GPU Radix based Montgomery Exponentiation: 1024-bit RSA Decryption.

Also included in Figure 3, is the fastest implementation reported on the Crypto++ [12] website for a 1024-bit RSA decrypt, which is running on an AMD Opteron 2.4 GHz processor (note that this figure does not include memory bandwidth so will likely be slower). Both the CPU and the GPU used in

these tests are slightly old, however their results are expected to scale in a straightforward manner. Also included are the performance measurements for Openssl's [14] speed test for 1024-bit RSA decryption running in both single (SC) and dual core (DC) modes on an AMD Athlon 64 X2 Dual Core 3800+. As can be seen at peak performance, the serial approach on the GPU is almost 4 times the speed of the fastest CPU implementation at 5536.75 primitives per second. We can see that the serial approach becomes competitive with the fastest CPU implementation at batches of 256 primitives. We also include the results for the radix approach presented in Szerwinski et al. [15], where the maximum throughput was 3252 primitives per second, though with a minimum latency of 1.7 seconds for all throughput rates. As a point of comparison we achieve a throughput rate of 4707 primitives per second at a latency of 218ms or 1024 primitives per batch of processing, i.e. a latency reduction of over 7 times. The increase in performance is largely due to the lack of reliance on global memory and the increased use of the faster on chip memory stores. We have only plotted the graph for [15] from the lowest number of message per batch corresponding to the reported minimum latency; the time taken to process smaller batches of messages remains the same.

5 Montgomery Exponentiation in RNS on the GPU

The motivation for using RNS is to improve on the parallel approach in Section 4, with an aim to reducing the number of primitives required before the GPU becomes competitive with the CPU. In RNS representation a number x, is denoted as $< x >_a$, where $< x >_a = (|x|_{a1}, |x|_{a2}...|x|_{an})$, $|x|_{ai} = x(\text{mod } a_i)$ and $a = \{a_1, a_2...a_n\}$, called the RNS base, is a set whose members are co-prime. $< x >_a$ can be converted into radix form, x, by the use of the Chinese Remainder Theorem (CRT). The CRT conversion of $< x >_a$ into radix form produces $x(\text{ mod } A)$, where $A = \prod_{i=1}^{n} a_i$, called the RNS's dynamic range. Numbers in RNS form have an advantage whereby multiplication, addition and subtraction can be performed as: $< x >_a \ op \ < y >_a = (|(|x|_{a1} \ op \ |y|_{a1})|_{a1}, ...|(|x|_{an} \ op \ |y|_{an})|_{an})$, where op is $+$, $-$ or $*$. As each operation is executed independently, this form of modular arithmetic is suited to parallel processing. However, divide within an RNS is difficult to perform, and as such presents an issue with regard to exponentiation as we will see.

5.1 Montgomery in RNS

As Montgomery multiplication consists primarily of multiplication and addition, there is a temptation to perform this via RNS, thus simply parallelising the entire exponentiation process. However, two parts of the algorithm cause problems within RNS - the mod R operation and the final divide by R, referring to notation used in Section 4. Unfortunately both of these operations cannot be performed in a single RNS base. As has been presented in papers [16] and [17], a

way around this issue is to use 2 bases, one for the execution of the mod R operation, and the other for execution of the divide. Thus one of the bases in effect acts as Montgomery's R, the other acts as a facilitator to be able to represent R^{-1} and perform a division by inverse multiplication. Table 1 contains an outline of Montgomery reduction in RNS as presented by Kawamura et al. [17], which we largely base our implementations upon. Note the RNS bases are denoted as a and b, and B ($\prod_{i=1}^{n} b_i$) is equivalent to R in the standard algorithm.

Input: $< x >_{a\cup b}, < y >_{a\cup b}$, (where $x,\, y < 2N$)	
Output: $< w >_{a\cup b}$ (where $w \equiv xyB^{-1} (\text{mod } N), w < 2N$)	
Base a Operation	Base b Operation
1: $< s >_a \leftarrow < x >_a . < y >_a$	$< s >_b \leftarrow < x >_b . < y >_b$
2a: —	$< t >_b \leftarrow < s >_b . < -N^{-1} >_b$
2b: $< t >_{a\cup b} \Longleftarrow < t >_b$	
3: $< u >_a \leftarrow < t >_a . < N >_a$	—
4: $< v >_a \leftarrow < s >_a + < u >_a$	—
5a: $< w >_a \leftarrow < v >_a . < B^{-1} >_a$	—
5b: $< w >_a \Longrightarrow < w >_{a\cup b}$	

Table 1. Kawamura et al. [17] Montgomery RNS.

The missing steps are mostly due to the observation that as B represents the Montgomery R, v is a multiple of B and thus all its residues will be zero. As can be seen, stage 2b and 5b consist of the conversion between bases, this is called base extension. The most computationally expensive part of this algorithm are these two base extensions. A base extension in its essence is the conversion from $< x >_a$ into its weighted radix form x and then into $< x >_b$. However, the conversion from $< x >_a$ into x via a full CRT is inefficient and is impractical to form part of the inner loop of an exponentiation. Szabo and Tanaka [18] detailed an early base extension algorithm which is based on preliminary conversion to a mixed radix representation (MRS) before extension into the new base. Although more efficient than a full CRT round trip, the conversion into MRS is inherently serial and thus difficult to accelerate on a parallel device. Our approach below is based on a more efficient base conversion technique using CRT as presented by Posch et al. [19] and Kawamura et al. [17]. This base extension technique does not undergo a full CRT round trip and allows for a greater degree of parallelism than the MRS technique. The details of this base extension are presented in detail below in Section 5.3.

5.2 Single Precision Modular Multiplication on the GPU

The most executed primitive operation within Montgomery RNS is a single precision modular multiplication. On the Nvidia CUDA hardware series the integer operations are limited to 32-bit input and output. Integer multiplies are reported to take 16 cycles, where divides are not quoted in cycles but rather a

recommendation to avoid if possible [1]. Here we present an investigation into 6 different techniques for achieving single precision modular multiplication suitable for RNS based exponentiation implementations. The techniques are limited to those applicable within the context of RNS and 1024-bit RSA.

1. 32-bit Simple Long Division: given two 32-bit unsigned integers we use the native multiply operation and the __umulhi(x,y) CUDA intrinsic to generate the low and high 32-bit parts of the product. We then use the product as a 4 16-bit limb dividend and divide by the 2 16-bit limb divisor using standard multi-precision division [13] to generate the remainder.

2. 32-bit Division by Invariant Integers using Multiplication: we make the observation that the divisors within an RNS Montgomery implementation, i.e. the base's moduli, are static. Also, as we select the moduli, they can be chosen to be close to the word size of the GPU. Thus we can assume that all invariant divisors, within the context of our implementation, are normalised (i.e. they have their most significant bit set). These two observations allow us to use an optimised variant of Granlund and Montgomery's approach for division by invariants using multiplication [20]. The basic concept used by [20] to calculate n/d is to find a sufficiently accurate approximation of $1/d$ in the form $m/2^x$. Thus the division can be performed by the multiplication of $n * m$ and cheap byte manipulation for division. We pre-calculate m for each of the base residues used and transfer them to the GPU for use via texture lookups. The algorithm in Table 2 removes all normalisation requirements from the original algorithm. It also rearranges some of the calculations to suit the efficient predication available on the GPU. Inputs: N is the word bit length on the GPU; single word multiplier and multiplicand x and y; m is a pre-calculated value dependent on d alone; d is the divisor. Output: r, the remainder. Some of these operations require 2 word precision and thus require extra instructions on the GPU. hiword() indicates the most significant word of a two word integer, where loword() indicates the least significant word. For a thorough explanation of the concepts involved, refer to [20].

$n = x * y, n1 = \text{hiword}(n), n0 = \text{loword}(n)$
$ns = n0 >> (N-1)$
$if(ns > 0)\ n0+ = d$
$t = \text{hiword}((m * (n1 + ns)) + n0)$
$q1 = n1 + t$
$dr = (n - (d << N)) + ((2^N - 1 - q1) * d)$
$r = \text{loword}(dr) + (d\ \&\ \text{hiword}(dr))$

Table 2. Granlund and Montgomery's division by invariants optimised for GPU and RNS.

3. 32-bit Reduction by Residue Multiplication: in this approach we use the observation that the moduli comprising the RNS bases can be selected close the GPU's maximum single word value. For 1024-bit RSA we can deter-

mine that for all moduli, d, the following holds $|2^N|_d < 2^{11}$, where N is the word bit length of the GPU, i.e. 32. As such, given a single precision multiplication $n = xy$, and using the convention that $n1$ is the most significant word of n, and $n0$ the least significant word, we can rewrite n as $|n1 * 2^N + n0|_d$. By repeatedly applying this representation to the most significant part of the equation, and using the pre-calculated value $r = |2^N|_d$, we can derive an algorithm for executing modular multiplication with multiplies and additions only. This observation is more formally stated in Table 3 (left), and the resultant pseudocode is also listed (right). Note that this approach benefits from being able to use CUDA's __umul24 limited precision fast multiply operation in the calculation of $n1r1r$.

Observation:	**Pseudocode:**
$\|x * y\|_d = \|n\|_d$	$n = x * y$
$= \|\|n1\|_d * \|2^N\|_d + \|n0\|_d\|_d$	$n0 = \text{loword}(n), n1 = \text{hiword}(n)$
Let $r = \|2^N\|_d$ /* $r < 2^{11}$*/,	$n1r = n1 * r$
$\|n\|_d = \|\|n1\|_d * r + \|n0\|_d\|_d$	$n1r0 = \text{loword}(n1r)$
$= \|\|n1r\|_d + \|n0\|_d\|_d$ /* $n1r < 2^{43}$*/	$n1r1 = \text{hiword}(n1r)$
$\|n1r\|_d = \|\|n1r1\|_d * r + \|n1r0\|_d\|_d$	$n1r1r = \text{loword}(n1r1 * r)$
$= \|\|n1r1r\|_d + \|n1r0\|_d\|_d$ /* $n1r1r < 2^{22}$ */	$r = n1r1r + n1r0 + n0$
Thus:	if$(r < d)r- = d$
$\|n\|_d \equiv \|n1r1r\|_d + \|n1r0\|_d + \|n0\|_d$, which is $< 3d$	if$(r < d)r- = d$.

Table 3. Algorithm for 32-bit Reduction by Residue Multiplication.

4. 32-bit Native Reduction using CRT: for the RNS bases we can use moduli that are the product of two co-prime factors, and also co-prime to each other. Using a modulus with two co-prime factors p and q, we can represent the modular multiplication input values, x and y, as $|x|_p$, $|x|_q$, $|y|_p$, $|y|_q$. Thus we have a mini RNS representation and as such can multiply these independently. We use CRT to recombine to give the final product. As p and q can be 16-bit, we are able to use the GPU's native integer modulus operator while maintaining 32-bit operands for our modular multiplication. This approach is described in more detail in the Moss et al. paper [10].

5. 16-bit Native Reduction: we can use 16-bit integers as the basic operand size of our modular multiplication, both input and output. We can then simply use the GPU's native multiply and modulus operators without any concern of overflow. However, we need to maintain the original dynamic range of the RNS bases when using 32-bit moduli. We can achieve this by doubling the number of moduli used in each base (note there is plenty of extra dynamic range when using 17 32-bit integers to accommodate this simple doubling).

6. 12-bit Native Reduction: this is the same concept as the 16-bit native approach above, however using 12-bit input and outputs we can use the much faster floating point multiplies and modulus operators without overflow concerns. Again we need to maintain the dynamic range by approximately tripling the original 32-bit moduli. Also there is an issue where the Kawamura approxi-

mations require the base moduli to be within a certain range of the next power of 2. This is not discussed further here, though note that a full 12-bit implementation would require the use of a different base extension method than the one described below.

Results: All tests of the above approaches processed the same amount of data, 2^{32} bytes, executing modular multiplication operations, reading and accumulating from and to shared memory. Thus the 16-bit implementation performed twice as many operations as the 32-bit approaches, with similar logic applied to the 12-bit implementation. The results can be seen in Table 4. We can see that the 12-bit and 16-bit approaches show the best performance, however a correction step is required for these figures. As we will see, the base extension executes in $O(n)$ time across n processors, where n is the number of moduli in the RNS base. In the context of 1024-bit RSA, the 12-bit approach requires a minimum of 43 moduli (512 bits / 12 bits) compared to 17 32-bit moduli for each RNS base. Also, the base extension step in Montgomery RNS is the most intensive part of our implementations consuming 80% of the execution time. A minimum approximation correction for the 12-bit result presented here is a division of 2, and for 16-bit 1.5. In effect the most efficient approach for use in Montgomery RNS is **Reduction by Residue Multiplication**.

	Modular Multiplication Approach	Modular multiplications per second
1.	32-bit LongDiv	2.89 * 10^9
2.	32-bit Inverse Mul	3.63 * 10^9
3.	32-bit Residue Mul	4.64 * 10^9
4.	32-bit Native+CRT	1.12 * 10^9
5.	16-bit Native	4.71 * 10^9
6.	12-bit Native	7.99 * 10^9

Table 4. GPU Modular Multiplication throughput using a variety of techniques.

5.3 Exponentation using Kawamura on the GPU

Our Montgomery RNS implementation is based on Kawamura et al.'s [17] approach. Its base extension algorithm relies on the following representation of CRT.

$x = \sum_{i=1}^{n}(|x|_{mi}|M_i^{-1}|_{mi} \mod m_i)M_i - kM$, where m is a set of moduli, $M = \prod_{i=1}^{n} m_i$, $M_i = M/m_i$ and $|M_i^{-1}|_{mi}$ is the multiplicative inverse of $M_i (\mod m_i)$. To base extend from $< x >_m$ to a single moduli m_1' and letting $E_i = |x|_{mi}|M_i^{-1}|_{mi} \mod m_i$ we can write $|x|_{m'1} = |\sum_{i=1}^{n}(|E_iM_i|_{m'1}) - k|M|_{m'1}|_{m'1}$.

Here E_i, for each i, can be calculated independently and thus in parallel. $|M_i|_{m'1}$ and $|M|_{m'1}$ are based on invariant moduli and as such can be precalculated. To calculate the base extension into multiple new moduli, m', each residue can be calculated independently. Also, if k can be calculated in parallel, we can use a parallel reduction which executes a base extension in $O(\log\ n)$

time, as reported in papers based on this technique [19]. However, in practice this requires a growth of processing units in the order of $O(n^2)$, where n is the number of moduli in the new base. On an SPMD machine, and with regards to throughput and not latency, it is more efficient to apply the effect of k serially for each new modulus.

The generation of k above can be written as $\lfloor \sum_{i=1}^{n} E_i/m_i \rfloor$. Kawamura et al. calculate this divide using an approximation based on the observation that m_i can be close to a power of 2. Also E_i is approximated, using a certain number of its most significant bits (the emphasis for Kawamura's approach is on VLSI design). In Table 5 we present a modified version of Kawamura's base extension algorithm which does not use the approximation of E_i and is suitable for a 32-bit processor. Inputs: $< x >_m$, m, m', α, Output: $< z >_{m \cup m'} = (< x >_m, < x >'_m)$, where m is the source base, m' is the new destination base and α is used to compensate for the approximations introduced, typically set to $2^{N/2}$.

$E_i = \|\|x\|_{mi} M_i^{-1}\|_{mi} (\forall i)$
for $j = 1$ to n
$\quad \alpha 0 = \alpha$, $\alpha += E_i$ /* note α wraps at 2^{32} on GPU*/
$\quad if(\alpha < \alpha 0)$ $r_i = \|r_i + (\| - M|_{m'i})|_{m'i} (\forall i)$
$\quad r_i = \|r_i + E_j \|M_j|_{m'i}|_{m'i} (\forall i)$

Table 5. Optimised Kawamura base extension for 32-bit processor.

α must be higher than the maximum error caused by the approximations above, however lower than 2^N [17], where N is the word bit length of the GPU. As we have removed the error due to approximation of E_i the only determinant of the size of the error is the distance between the moduli used and their next power of 2. This puts a restriction on the number of moduli that can be used with this technique. This base extension can be used in the context of 1024-bit RSA with 32 and 16-bit moduli, while 12-bit moduli require the use of a different method.

Our most performance competitive implementation is based on the Reduction by Residue Multiplication approach for modular multiplication as described above, and as such uses 32-bit moduli. For 1024-bit RSA, we require two sets of 17 32-bit moduli for use as the two bases, assuming an implementation based on Quisquater et al.'s CRT approach [21]. Groups of 17 consecutive threads within a CUDA block operate co-operatively to calculate an exponentiation. Each thread reads in two residues, $|x|_{ai}$ and $|x|_{bi}$, thus a single thread executes both the left and right sides of the Montgomery RNS algorithm, see Table 1, for a pair of residues, ensuring each thread is continuously busy.

Memory Usage: As the residues are in groups of 17, we employ a padding scheme for the ciphertext/plaintext data whereby the start address used by the first thread of a CUDA block is aligned to a 128-bit boundary. We also pad the number of threads per block to match this padding of input data, which allows a simple address mapping scheme while allowing for fully coalesced reads. The

CUDA thread allocation scheme for ensuring even distribution across available SMs and also correct padding is show in Table 6, where RNS_SIZE is the number of moduli per base, MAX_THREADS_PER_BLOCK is a predefined constant dependant on shared resource pressure of the kernel and BLOCK_GROUP is the number of SMs on the GPU.

$total\_threads = noMsgs * RNS\_SIZE * 2$
$blocks\_required = \lceil total\_threads / MAX\_THREADS\_PER\_BLOCK \rceil$
$blocks\_required = \lceil blocks\_required \rceil^{BLOCK\_GROUP}$
$threads\_per\_block = \lceil total\_threads / blocks\_required \rceil$
$threads\_per\_block = \lceil \lceil threads\_per\_block \rceil^{RNS\_SIZE} \rceil^{WARP\_SIZE}$

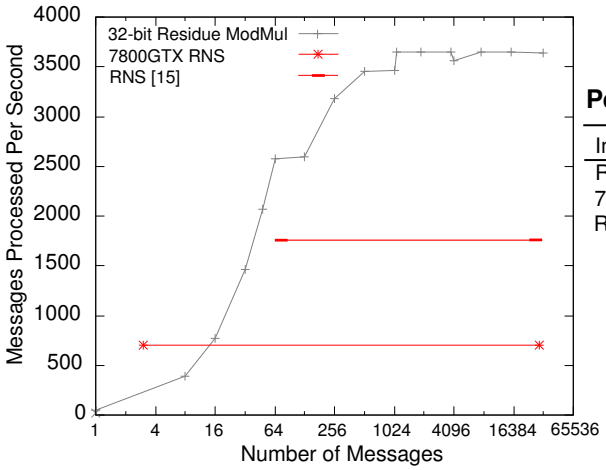
Table 6. CUDA thread allocation scheme for RNS based modular exponentiation.

Each thread is responsible for calculating a single E_i, after which point a synchronisation barrier is used to ensure all new E_i values can be safely used by all threads. As discussed in Section 4, this synchronisation barrier, along with the general performance issues with thread divergence dictates that only a single exponent can be used for each CUDA block of threads. In practice, for peak performance with 1024-bit RSA, a single exponent should be used a maximum of once for every 15 primitives (256 threads per block / 17 threads per primitive). With regards to the shared memory use, we use two different locations for storing the values of E_i. The storage locations are alternated for each subsequent base extension. This permits a single synchronisation point to be used, rather than two - one before and one after the generation of E_i, which is necessary when only one location is used for storing the values of E_i. We also use shared memory to accelerate the most intensive table lookup corresponding to $|M_j|_{m'i}$ in the base extension. At the start of each kernel call, all threads within a block cooperate in loading into shared memory the entirety of the two arrays, $|A_j|_{bi}$ and $|B_j|_{ai}$ (one for each base), via texture lookups.

5.4 Results

Figure 4 shows the throughput of our RNS implementation using CRT, sliding window and 32-bit modular reduction using the Reduction by Residue Multiplication as described previously. Also included is the peak RNS performance from Szerwinski et al. [15] at 1756 primitives per second, we can see that our peak performance has 2 times higher throughput. The gains in performance is deemed mainly to come from the improved single precision modular multiplication and optimised base extension algorithm. Like in the radix results, we have only plotted the graph for [15] from the point of the reported minimum latency. We have also included as a historical reference, the previous RNS implementation on an Nvidia 7800GTX by Moss et al. [10] to show the improvements possible due to the advances in hardware and software libraries available. [10] reports a performance of 175.5 true 1024-bit exponentiation operations per second, which we

have multiplied by 4 [4] and use as a best case scenario estimate of the runtime for a 1024-bit RSA decrypt, i.e. split into 2 512-bit exponentiations. Comparing the peak performance of the Crypto++ CPU implementations listed in Figure 3 we can see that our peak performance for our RNS implementation has up to 2.5 times higher throughput.



Peak Performance

Impl.	Msgs p/s	ms/op
Residue Mul	3653.54	0.27
7800 GTX	704.22	1.42
RNS [15]	1756.0	0.57

Fig. 4. GPU RNS based Montgomery Exponentiation: 1024-bit RSA Decryption.

5.5 Radix vs RNS on the GPU

We use Figure 5 to illustrate the effectiveness of our RNS implementation at accelerating RSA in comparison to our radix based implementations. As can be seen the RNS implementation gives superior throughput with much smaller number of messages per kernel call. The point at which the serial radix approach becomes faster than the CPU is at 256 messages, where the RNS approach has better performance at 32 messages per kernel. The greater performance at smaller message numbers is due to a higher GPU occupancy for the RNS approach over the serial radix approach. The RNS approach also does not suffer from the extreme levels of synchronisation during a Montgomery multiplication as the parallel radix approach. Using RNS can greatly improve the GPU's ability to provide feasible acceleration for RSA decryption, or any public key cryptographic scheme where the modulus and exponent change with a low frequency. It is clear from Figure 5 that an adaptive approach to exponentiation would provide the best overall performance, switching from an RNS based implementation at low message requirements, to a radix based approach at high message requirements.

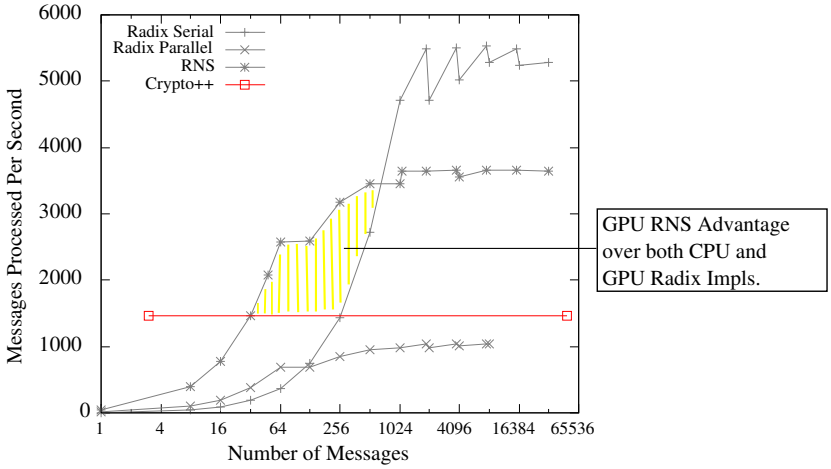


Fig. 5. RNS vs Radix: 1024-bit RSA Decryption.

6 Conclusions

In this paper we have presented implementations of modular exponentiation suitable for public key cryptography. We have focused on 1024-bit RSA decryption running on an Nvidia 8800GTX and demonstrated a peak throughput of 0.18 ms/op giving a 4 times improvement over a comparable CPU implementation. We have also shown that the significant problems with latency in previous GPU implementations can be overcome while also improving on throughput rates. We have shown that an adaptive approach to modular exponentiation on the GPU provides the best performance across a range of usage scenarios. A standard serial implementation of Montgomery exponentiation gives the best performance in the context of a high number of parallel messages, while an RNS based Montgomery exponentiation gives better performance with fewer messages. We show that an optimised RNS approach is more performant than a CPU implementation at 32 messages per kernel call and that the pencil-and-paper approach proves better than the RNS approach at 256 messages.

Also covered in the paper is the applicability of the GPU to general public key cryptography, where the observation is made that peak performance is only achievable in the context of substantial key reuse. In the case of 1024-bit RSA using RNS, peak performance requires the key to change at a maximum rate of once per 15 messages, and once per 32 messages when using a serial pencil-and-paper approach. Thus the GPU can be effectively used in an RSA decryption capacity where it is common for a server to use a limited number of keys.

RNS based approaches are highly dependent on efficient support of single precision modular multiplication, which the paper illustrates is non trivial on the GPU. In this context we have explored a variety of techniques for achieving efficient modular multiplication and show that a novel approach suited to

RNS and the GPU gives the best performance. The GPU could offer improved performance with RNS based approaches if future architectures provide efficient native modular operations. Future work consists of extending the implementations presented to cater for larger public key exponents and messages such those used in 2048 and 4096-bit RSA. This may lead to the feasibility of other forms of big integer multiplication such as Karatsuba or Toom-Cook techniques.

References

1. Nvidia CUDA Programming Guide, Version 2.0, 2008.
2. Microsoft, "Direct X Technology", `http://msdn.microsoft.com/directx/`.
3. Nvidia Corporation, "CUDA", `http://developer.nvidia.com/object/cuda.html`.
4. A. Menezes, P. van Oorschot, and S. Vanstone, "Handbook of Applied Cryptography". CRC Press, October 1996. ISBN 0-8493-8523-7.
5. P.L. Montgomery. "Modular Multiplication Without Trial Division". Mathematics of Computation, 44, 519-521, 1985.
6. D. Cook and J. Ioannidis and A. Keromytis and J. Luck, "CryptoGraphics: Secret Key Cryptography Using Graphics Cards". RSA Conference, Cryptographer's Track (CT-RSA), February 2005.
7. O. Harrison and J. Waldron, "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units". CHES 2007. Vienna, Austria. September 10-13, 2007.
8. J. Yang and J. Goodman, "Symmetric Key Cryptography on Modern Graphics Hardware". ASIACRYPT 2007. Kuching, Malaysia. December 2-6, 2007.
9. O. Harrison and J. Waldron, "Practical Symmetric Key Cryptography on Modern Graphics Hardware". 17th USENIX Security Symposium. San Jose, CA. July 28 - August 1, 2008.
10. A. Moss, D. Page and N.P. Smart, "Toward Acceleration of RSA Using 3D Graphics Hardware". 11th IMA International Conference on Cryptography and Coding. Cirencester, UK. December 18-20, 2007.
11. S. Fleissner, "GPU-Accelerated Montgomery Exponentiation". Computational Science ICCS 2007, 7th International Conference. Beijing, China. May 27-30, 2007.
12. AMD 64 RSA Benchmarks, `http://www.cryptopp.com/benchmarks-amd64.html`.
13. D.E. Knuth. "The Art of Computer Programming. Vol 2". Addison-Wesley, 3rd ed., 1997.
14. OpenSSL Open Source Project, `http://www.openssl.org/`.
15. R. Szerwinski and T. Güneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography". 10th International Workshop on Cryptographic Hardware and Embedded Systems. Washington DC, USA. August 10-13, 2008.
16. K.C. Posch and R. Posch. "Modulo Reduction in Residues Numbers Systems" IEEE Trans. on Parallel and Distributed Systems, Vol.6, No.5, 449-454, May 1995.
17. S. Kawamura, M. Koike, F. Sano and A. Shimbo, "Cox-Rower Architecture for Fast Parallel Montgomery Multiplication In Advances in Cryptology". Springer-Verlag LNCS 1807, 523538, 2000.
18. N.S. Szabo and R.I. Tanaka. "Residue Arithmetic and its Applications to Computer Technology". McGraw-Hill, 1967.
19. K.C. Posch and R. Posch. "Base Extension Using a Convolution Sum in Residue Number Systems". In Computing 50, Pages 93104, 1993.

20. T. Granlund and P. Montgomery. “Division by Invariant Integers using Multiplication”. SIGPLAN ’94 Conference on Programming Language Design and Implementation. Orlando, Florida. June 1994.
21. J-J. Quisquater and C. Couvreur. “Fast Decipherment Algorithm for RSA Public-Key Cryptosystem”. Electronics Letters, Vol. 18, No. 21, Pages 905-907, 1982.