Public Key Cryptography on Modern 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 (GPUs) are increasingly being used for general purpose processing. 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 (RNS) form. We show how a GPU implementation of a 1024-bit RSA decrypt primitive can outperform for the first time a comparable CPU implementation by up to 4 times. 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. We also highlight the criteria necessary to allow the GPU to improve the performance of public key cryptographic operations. [1]

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 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 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

[1] Published in booklet format and appeared as poster: EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009.

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.

2.1 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 [6], 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. 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 2.2 and 3.

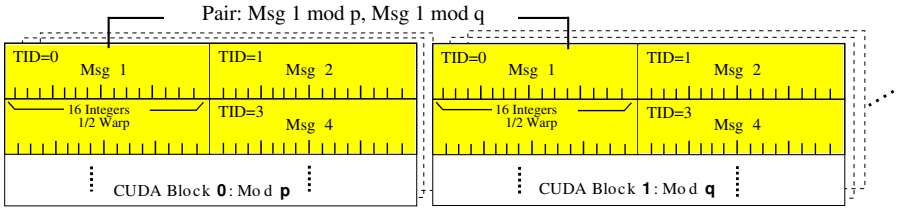


Fig. 1. Serial Thread Model

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 straightforward. The macro level details of this algorithm are largely standard and 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 nxn limb multiplies used cumulative addition to reduce memory operations [7]; all squaring requirements were optimised to reduce the number of required multiplies [4]; nxn limb multiplies *mod R* were truncated, again to remove redundant multiplies; and the final two steps within Montgomery multiplication were combined into a single nxn multiply and accumulate.

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 2.2 and Section 3 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 nxn 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 the best performance under this scenario [8], 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 nxn 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 nxn 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, $-n^{-1}(mod\ R)$ and $R^2(mod\ n)$ (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 2.3 in conjunction with the parallel approach described below. Note that two parts of the exponentiation are not included in these implementations, the initial $x(mod\ p)$, $x(mod\ q)$ 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.

2.2 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. The nxn multiplies within the Montgomery multiplication are parallelised by their separation into individual 1xn limb multiplications. Each thread is independently responsible for a single 1xn 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 nxn 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. Also shown in the Figure 2 are the synchronisation points used to ensure all shared memory writes are committed before subsequent reads are performed, which add a signification performance burden.

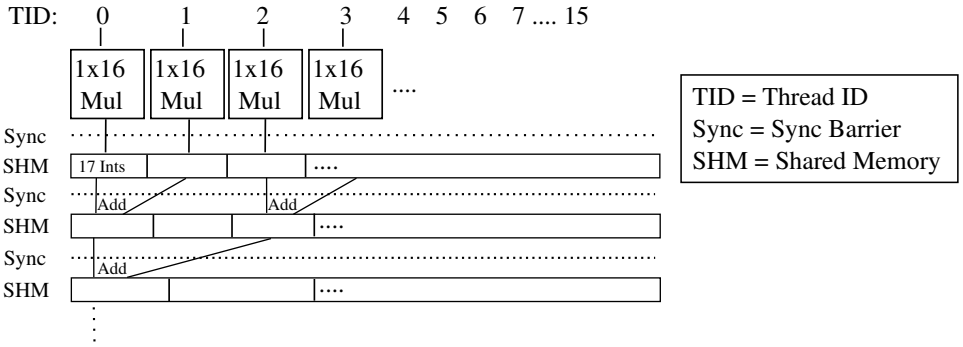


Fig. 2. nxn limb multiplication in parallel on a CUDA device

The optimisations applied to the different nxn 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 nxn multiply. However, the longest limb within the nxn 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 1xn multiply finishes. Also, as each thread executes its own 1xn multiply separately, the cumulative approach to addition must also be separated from the multiplication process.

2.3 Results

Figure 3 illustrates the performance of both the parallel and serial approaches. All measurements represent the number of 1024-bit RSA decrypt primitives executed per second. The GPU implementations show their dependence on an increasing number of messages 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.

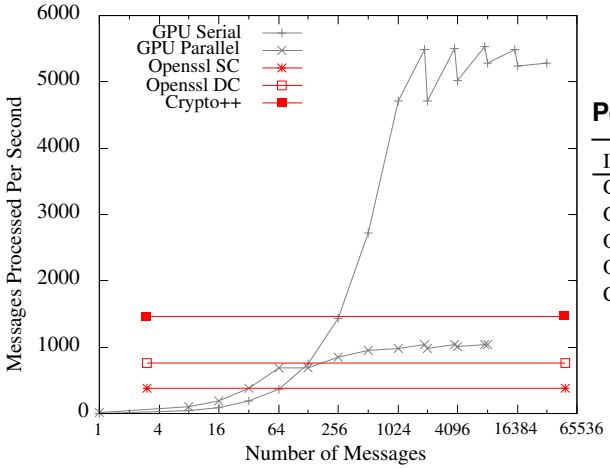
Also included in Figure 3, is the fastest implementation reported on the Crypto++ [9] website for a 1024-bit RSA decrypt, which is running on an AMD Opteron 2.4 GHz processor. Also included are the performance measurements for Openssl's [10] speed test for 1024-bit RSA decryption running in both single (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 256 primitives per second. Unfortunately the parallel approach at no point is faster than both the serial GPU approach and the CPU implementations.

3 Montgomery Exponentiation in RNS on the GPU

3.1 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.

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 [7] to generate the remainder.



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

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

2. 32-bit Division by Invariant Integers using Multiplication: we make the observation that the divisors within an RNS Montgomery implementation 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 a simplified variant of Granlund and Montgomery's approach for division by invariants using multiplication [11]. The basic concept used by [11] 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 below 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.

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

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 determine 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 signification 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 signification 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 as follows (left), and the resultant pseudocode is also listed (right).

Observation:

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

Pseudocode:

$n = x * y$
$n0 = \text{loword}(n), n1 = \text{hiword}(n)$
$n1r = n1 * r$
$n1r0 = \text{loword}(n1r)$
$n1r1 = \text{hiword}(n1r)$
$n1r1r = \text{loword}(n1r1 * r)$
$r = n1r1r + n1r0 + n0$
if$(r < d)r- = d$
if$(r < d)r- = d$.

4. 32-bit Native Reduction using CRT: 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 [12].

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 approximations 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 another base extension method than the one described below.

Results: All tests of the above approaches process 2^{32} bytes, executing modular multiplication operations, reading and accumulating from and to shared memory. The results can be seen in Table 1. 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. The most effective 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 1. GPU Modular Multiplication throughput using a variety of techniques

3.2 Exponentation using Kawamura on the GPU

Our Montgomery RNS implementation is based on Kawamura et al.'s [13] 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 in parallel. $|M_i|_{m'1}$ and $|M|_{m'1}$ are based on invariant moduli and as such can be pre-calculated. 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 [14]. 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. The generation of k 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 signification bits (the emphasis for Kawamura's approach is on VLSI design). Below we present a modified version of Kawamura's base extension 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
  $\alpha 0 = \alpha$, $\alpha += E_i$ /* note α wraps at 2^{32} on GPU*/
  $if(\alpha < \alpha 0)\ r_i = |r_i + (|-M|_{m'i})|_{m'i}(\forall i)$
  $r_i = |r_i + E_j|M_j|_{m'i}|_{m'i}(\forall i)$

α must be higher than the maximum error caused by the approximations above, however lower than 2^N [13], 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 efficient implementation is based on the Reduction by Residue Multiplication approach for modular multiplication as described previously. 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 [6]. 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 parts of the Montgomery RNS algorithm, see [13], 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 below, 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.

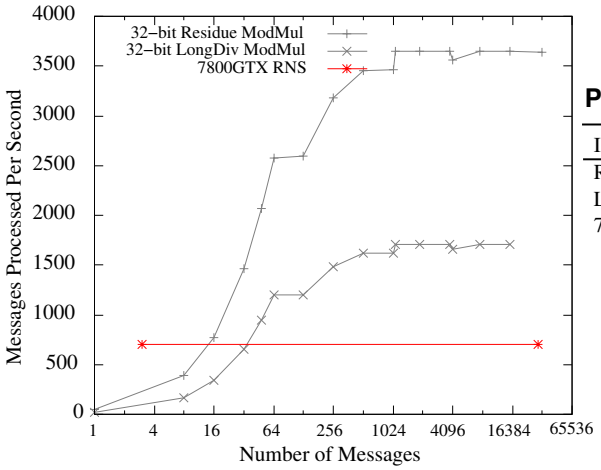
$total\_threads = noMsgs * RNS\_SIZE * 2$
$blocks\_required = \lceil total\_threads/MAX\_THREADS\_PER\_BLOCK \rceil$
$blocks\_required = \lceil blocks\_required \rceil^{BLOCK\_GROUP}$
$threads\_per\_block = \lceil total\_threads/blocks\_required \rceil$
$threads\_per\_block = \lceil \lceil threads\_per\_block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

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 2, 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.

3.3 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. Included in the results is the same implementation using 32-bit long division to illustrate the heavy reliance on single precision modular multiplication. We have also included as a historical reference, the previous RNS implementation on an Nvidia 7800GTX by Moss et al. [12] to show the improvements possible due to the advances in hardware and software libraries available. 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
LongDiv	1704.63	0.59
7800 GTX	704.22	1.42

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

4 Conclusions

We have focused on 1024-bit RSA decryption running on an Nvidia 8800 GTX and demonstrated a peak throughput of 0.18 ms/op giving a 4 times improvement over a comparable CPU implementation. We have shown that 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 proves better performance than a CPU implementation at 32 parallel ciphertext/plaintext messages per kernel call and the pencil-and-paper approach proves better than the RNS approach at 256 parallel 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. We have also presented a variety of techniques for achieving efficient GPU based modular multiplication suitable for RNS.

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. J-J. Quisquater and C. Couvreur. "Fast Decipherment Algorithm for RSA Public-Key Cryptosystem". Electronics Letters, Vol. 18, No. 21, Pages 905907, 1982.
7. D.E. Knuth. "The Art of Computer Programming. Vol 2". Addison-Wesley, 3rd ed., 1997.
8. 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.
9. AMD 64 RSA Benchmarks, http://www.cryptopp.com/benchmarks-amd64.html
10. OpenSSL Open Source Project. http://www.openssl.org/.
11. T. Granlund and P. Montgomery. "Division by Invariant Integers using Multiplication". SIGPLAN '94 Conference on Programming Language Design and Implementation. Orlando, Florida. June 1994.
12. A. Moss, D. Page and N.P. Smart, "Toward Acceleration of RSA Using 3D Graphics Hardware". Eleventh IMA International Conference on Cryptography and Coding. Cirencester, UK. December 18-20, 2007.
13. 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.
14. K.C. Posch and R. Posch. "Base Extension Using a Convolution Sum in Residue Number Systems". In Computing 50, Pages 93104, 1993.