Executing Modular Exponentiation on a Graphics Accelerator

A. Moss, D. Page, and N. Smart

Department of Computer Science,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB,
United Kingdom.
{moss|page|nigel}@cs.bris.ac.uk

Abstract. Demand in the consumer market for graphics hardware that accelerates rendering of 3D images has resulted in commodity devices capable of astonishing levels of performance. These results were achieved by specifically tailoring the hardware for the target domain. As graphics accelerators become increasingly programmable this performance makes them an attractive target for other domains. Specifically, they have motivated the transformation of costly algorithms from a general purpose computational model into a form that executes on said graphics hardware. We investigate the implementation and performance of modular exponentiation using a graphics accelerator, with the view of using it to execute operations required in the RSA public key cryptosystem.

1 Introduction

Efficient arithmetic operations modulo a large prime (or composite) number are core to the performance of public key cryptosystems. RSA [19] is based on arithmetic in the ring $\mathbb{Z}_N$, where $N = pq$ for large prime p and q, while Elliptic Curve Cryptography (ECC) [11] can be parameterised over the finite field $\mathbb{F}_p$ for large prime p. On processors with a w-bit word size, one commonly represents $0 \leq x < N$ using a vector of n radix-2^w digits. Unless specialist co-processor hardware is used, modular operations on such numbers are performed in software using well known techniques [13, 2] that operate using native integer machine operations. Given the significant computational load, it is desirable to accelerate said operations using instruction sets that harness Single Instruction Multiple Data (SIMD) parallelism; in the context of ECC, a good overview is given by Hankerson et al. [11, Chapter 5]. Although dedicated vector processors have been proposed for cryptography [9] these are far from being commodity items.

In an alternative approach, researchers have investigated cryptosystems based on arithmetic in fields modulo a small prime q or extension thereof. Since ideally we have $q < 2^w$, the representation of $0 \leq x < q$ is simply one word; the use of low-weight primes [7] offers an efficient method for modular reduction. Example systems that use such arithmetic include Optimal Extension Fields (OEF) [1] which can provide an efficient underpinning for ECC; torus based constructions such as T_{30} [8]; and the use of Residue Number Systems (RNS) [14, Chapter 3] to implement RSA. Issues of security aside, the use of such systems is attractive as operations modulo q may be more efficiently realised in integer based machine operations. This fact is reinforced by the aforementioned potential for parallelism; for example, addition operations in an OEF can be computed in a component-wise manner which directly maps onto SIMD instruction sets [11, Chapter 5].

However, the focus on use of integer operations in implementation of operations modulo large and small numbers ignores the capability for efficient floating point computation within commodity desktop class processors. This feature is often ignored and the related resources are left idle: from the perspective of efficiency we would like to utilise the potential for floating point arithmetic to accelerate our implementations. Examples of this approach are provided in ground breaking work by Bernstein who has produced high-performance floating point based implementations of primitives such as Poly1305 [3] and Curve25519 [4]. Beyond algorithmic optimisation, use of floating point hardware in general purpose processors such as the Intel Pentium 4 offered Bernstein some significant advantages. Specifically, floating point operations can often be executed in parallel with integer operations; there is often a larger and more orthogonally accessible floating point register file available; good scheduling of floating point operations can often yield a throughput close to one operation per-cycle.

Further motivation for use of this type of approach is provided by the recent availability of programmable, highly SIMD-parallel floating point co-processors in the form of Graphics Processing Units (GPU). Driven by market forces these devices have developed at a rate that has outpaced Moore's Law: for example, the Nvidia 7800-GTX uses 300 million transistors to deliver roughly 185 Gflop/s in contrast with the 55 million transistor Intel Pentium 4 which delivers roughly 7 Gflop/s. Although general purpose use of the GPU is an emerging research area [10], the only published prior usage for cryptography is by Cook et al. [5] who implemented block and stream ciphers using the OpenGL command-set; we are aware of no previous work accelerating computationally expensive public key primitives. Further, quoted performance results in previous work are somewhat underwhelming, with the GPU executing AES at only 75% the speed of a general purpose processor.

This paper seeks to gather together all three strands of work described above. Our overall aim is arithmetic modulo a large number so we can execute operations required in the RSA public key cryptosystem; we implement this arithmetic with an RNS based approach which performs arithmetic modulo small floating point values. The end result is an implementation of RSA which firstly fits the GPU programming model, and secondly makes effective use of SIMD-parallel floating point operations on which GPU performance relies. We demonstrate that with some caveats, this implementation makes it possible to improve RSA performance using the GPU versus that achieved using a general purpose processor. An alternative approach is recent work implementing RSA on the IBM Cell [6], another media-biased vector processor. However, the radically different special purpose architecture of the GPU makes the task much more difficult than on the IBM Cell (a general purpose processor), hence our differing approach.

We organise the paper as follows. In Section 2 we give an overview of GPU architecture and capabilities. We use Section 3 to describe the algorithms used to implement modular exponentiation in RNS before describing the GPU implementation in Section 4. The experimental results in Section 5 compare the GPU implementation with one on a standard processor, with conclusions in Section 6.

2 An Overview of GPU Architecture

Original graphics accelerator cards were special purpose hardware accelerators for the OpenGL and DirectX Application Programming Interfaces (APIs). Programs used the API to describe a 3D scene using polygons. The polygons have surfaces filled with a 2D pattern called a texture. The API produced an image for display to the user. Images are arrays of Picture Elements, or *pixels*, formed by the perspective-correct projection of the primitives onto a 2D plane. Each pixel describes the colour and intensity of a point on the display.

Graphics cards developed to allow the fixed functionality to be reprogrammed. Vector shaders are programs that transform 3D vectors within the graphics pipeline by custom projections and calculations. Pixel shaders allow the value of pixels to be specified by the programmer, as part of this specification a set of textures can be indexed by the program. Results can be directed into textures held in memory rather than to the display. We ignore 3D functionality and render a single scaled 2D rectangle parallel to the display plane, enforcing a 1:1 relation between input and output pixels, thereby turning the GPU into a vector processor. Each pixel is a 4-vector of single-precision floating-point values.

In the GPU programming model a single pixel shader is executed over a 2D rectangle of pixels. Each output pixel is computed by a separate instance of the shader, with no communication between program instances. Control-flow proceeds in lockstep between the instances to implement a SIMD vector processor. The program instance can use its 2D position within the array to parameterise computations, furthermore we can provide *uniform* variables which are constant over a single rendering step; each program instance has read-only access to such variables which are used to communicate parameters from the host to the shader programs. The output of a value parametrised only by the coordinates of the pixel characterises the GPU programming model as *stream-processing*.

In this paper we specifically consider the Nvidia 7800-GTX as an archetype of the GPU. From here on, references to the GPU can be read as meaning the GPU within the Nvidia 7800-GTX accelerator. The pixel shading step within the GPU happens when the 3D geometry is *rasterised* onto the 2D image array. This process is shown in Figure 3, each rendered group of 2×2 pixels is termed a *quad*. The GPU contains 24 pipelines, arranged as six groups of quad-processors; each quad-processor operates on 4 pixels containing 16 values. Each pixel pipeline contains two execution units that can dispatch two independent 4-vector operations per clock cycle. If there are enough independent operations within the pixel-shader then each pipeline can dispatch a total of four vector operations per clock cycle. This gives a theoretical peak performance of $24 \times 4 = 96$ vector operations, or 384 single-precision floating point operations, per clock cycle.

The GPU contains a number of ports that are connected to textures stored in local memory. Each port is uni-directional and allows either pixels to be read from a texture, or results to be written to a texture. The location of the pixels within output textures is fixed by the rendering operation and cannot be altered within a pixel shader instance. In stream processing terminology *gather* operations, e.g. accumulation, are possible but *scatter* operations are not. The same texture cannot be attached to multiple ports and so read and write operations cannot be mixed on a texture within a single rendering step.

The lack of concurrent read and write operations, and communication between execution units, limits the class of programs that can be executed in a shader program, in particular modifying intermediate results is not directly possible. A solution to this problem called *ping-ponging* has been developed by the general purpose community [10]. The technique shown in Figure 2 uses multiple shader programs executing in sequence, with the textures holding intermediate results being written in one pass, and then read in a subsequent pass. We have identified the RNS version of RSA encryption as possible using this technique. There is a constant overhead associated with the setup of each pixel shader program, split between the OpenGL API, the graphics card driver and the latency filling the GPU pipelines. To achieve high-performance, this constant cost must amortised over a large enough texture. Increasing the number of ping-ponging steps increases the size of data-set required to break-even on the overhead.

Texture lookup within a pixel shader allows any pixel in a texture bound to an input port to be read. Texture lookups are not free operations; at a core frequency of 430 MHz, latency is in the order of 100 clock cycles. Exposing this latency is not feasible so it is hidden by multi-threading the quads processed on each quad-processor. Quads are issued in large batches, and all reads from instances within those batches are issued to a cache. This produces the two dimensional cache locality that is observed in GPU execution. Once a pixel has passed through the pipeline it is retired into the texture buffer. Recombination of the output stream is performed by 16 Raster Operators (ROP) which introduce a bottleneck of 25.6GB/s bandwidth shared between all shader instances in every pass. This hard limit creates a lower bound of six cycles on the execution of each shader instance; lacking the operations to fill these “free” cycles is a bottleneck.

3 RNS Implementation of RSA

The RSA [19] public key cryptosystem uses a public key pair (N, e) where N is chosen to be the product of two large primes p and q that are kept secret. A private key d is selected such that $e \cdot d \equiv 1 \pmod{\phi(N)}$. To encrypt a plaintext message m, the ciphertext c is computed as $c = m^e \pmod{N}$ while to reverse the operation and decrypt the ciphertext one computes $m = c^d \pmod{N}$. As such, the core computational requirement is efficient arithmetic modulo N, in particular modular exponentiation. Efficient realisation of this primitive relies heavily on efficient modular multiplication as a building block.

We focus on implementing modular multiplication using a Residue Number System (RNS) with standard values of N; that is N is a 1024-bit number. The advantage of using RNS is that operations such as addition and multiplication can be computed independently for the digits that represent a given value. As such, RNS operations can be highly efficient on a vector processor, such as the GPU, since one can operate component-wise in parallel across a vector of digits. Beyond this, selection of the exponentiation algorithm itself requires some analysis of the trade-off between time and space; see Menezes et al. [12, Chapter 14] for an overview.

In the specific case of RSA, the cost of a single modular exponentiation can be reduced by employing small encryption exponent variants and the Chinese Remainder Theorem (CRT) [17] to accelerate the public and private key operations respectively. We do not consider these techniques, instead concentrating on implementation of vanilla modular exponentiation on the GPU. Even so, performance improvements can still be achieved over multiple invocations of the modular exponentiation primitive. Consider an application where one is required to decrypt k ciphertexts c_i with the same decryption exponent, i.e.

$$m_i = c_i^d \pmod{N_i}, \quad i \in \{1, 2, \dots k\}. \tag{1}$$

These separate computations can be viewed as a single SIMD-parallel program; control flow is uniform over the k operations, while the data values differ. As justification consider a server communicating with k clients: each client encrypts and communicates information using a single public key pair; batches of ciphertexts are decrypted with a common exponent. Considering this form of operation is important since it enables us to capitalised on both fine-grained parallelism at the RNS arithmetic level, and course-grained parallelism in the exponentiation level.

3.1 Standard Arithmetic in an RNS

Numbers are stored in a conventional radix number representation by selecting coefficients of the various powers of the radix that sum to the number. On processors with a w-bit word size, one typically selects the radix $b = 2^w$ so that each coefficient can be stored in a separate word. An RNS operates by selecting a *basis*, a set of co-prime integers that are fixed for all of the numbers being operated upon. An example basis B might be defined as

$$B = \langle\, B[1], B[2], \dots, B[n] \,\rangle \text{ s.t. } i \neq j \iff \gcd(B[i], B[j]) = 1$$

For efficiency, each of the moduli $B[i]$ should fit within a word. The size of a basis is defined as the product of the moduli, that is

$$|B| = \prod_{i=1}^{n} B[i]$$

such that the largest uniquely representable number is less than $|B|$. Arithmetic operations performed in B implicitly produce results modulo $|B|$.

When the integer x is encoded into an RNS representation in the basis B, it is stored as a vector of words; there are n words, one for each $B[i] \in B$. Each component of the vector holds the residue of x modulo the respective $B[i]$. Thus, using standard notation, the encoding of x under basis B is the vector

$$x_B =< \ x \bmod B[1], \ x \bmod B[2], \ \ldots, \ x \bmod B[n] \ >$$

such that $x_B[i]$ denotes the i-th component of the vector. The Chinese Remainder Theorem (CRT) defines a bijection between the integers modulo $|B|$ and the set of representations stored in the basis B. To decode x_B, the CRT is applied

$$x = \sum_{i=1}^{n} \left(\hat{B[i]} \cdot \frac{x_B[i]}{\hat{B[i]}} \bmod B[i] \right) \bmod |B|$$

where $\hat{B[i]} = \frac{|B|}{B[i]}$. It is useful to rewrite the CRT as shown in Equation 2. In this form the reduction factor k is directly expressed, which is used by the base extension algorithms in Section 3.3. Note that unless the RSA values are initially stored or transmitted in RNS representation, the conversion process represents an overhead.

$$x = \sum_{i=1}^{n} \hat{B[i]} \cdot \frac{x_B[i]}{\hat{B[i]}} \ - k|B| \tag{2}$$

Multiplication and addition of two integers x_B and y_B, encoded using an RNS representation under the basis B, is performed component-wise on the vectors. For example, multiplication is given by the vector

$$x_B \cdot y_B = \ \langle \ x[1] \cdot y[1] \bmod B[1], \ \ x[2] \cdot y[2] \bmod B[2], \ \ \ldots, \ \ x[n] \cdot y[n] \bmod B[n] \ \rangle$$

Each component is independent, and so on a vector architecture individual terms in the computation can be evaluated in parallel. Assuming m execution units on the GPU, and n moduli in the RNS basis, then a single operation will only take $\lceil \frac{n}{m} \rceil$ clock cycles. Eliminating the communication, and hence synchronisation, between the execution units makes this speed possible. It is imperative to emphasise this advantage of RNS; propagation of carries between words within the GPU is very difficult to achieve and thus SIMD-parallel methods for realising arithmetic modulo N on general purpose processors are not viable.

3.2 Modular Arithmetic in an RNS

Although we have described standard multiplication using RNS, implementation of RSA requires modular multiplication. In a positional number system this operation is commonly performed using Montgomery representation [13]. To define the Montgomery representation of x, denoted x_M, one selects an $R = b^t > N$ for some integer t; the representation then specifies that $x_M \equiv xR \pmod{N}$.

Algorithm 1: Cox-Rower Algorithm [18] for Montgomery multiplication in RNS.

```
Input  : $x$ and $y$, both encoded into $A$ and $B$.
Output: $x \cdot y \cdot A^{-1}$, encoded into both $A$ and $B$.
    In Base A                                         In Base B
    $s_A \leftarrow x_A \cdot y_A$                    $s_B \leftarrow x_B \cdot y_B$
                                                      $t_B \leftarrow s_B \cdot \left(-N^{-1} \bmod |B|\right)$
                          base extend $t_A \leftarrow t_B$
    $u_A \leftarrow t_A \cdot N_A$
    $v_A \leftarrow s_A + u_A$
    $w_A \leftarrow v_A \cdot \left(|B|^{-1} \bmod A\right)$
                          base extend $w_A \rightarrow w_B$
    return $w_A, w_B$
```

To compute the product of x_M and y_M, termed Montgomery multiplication, one interleaves a standard integer multiplication with an efficient reduction by R

$$x_M \star y_M = x_M y_M R^{-1}$$

The same basic approach can be applied to integers represented in an RNS [15, 18]. As the suitable input range of the Posch algorithm [15] is tighter than the output produced, a conditional subtraction may be required. To avoid this conditional control flow we have used the Cox-Rower algorithm of Kawamura et al. [18]. Roughly, a basis $|A|$ is chosen as the R by which intermediate reductions are performed. Operations performed within the basis A are implicitly reduced by $|A|$ for free. Montgomery multiplication requires the use of integers up to RN, or $|A|N$ in size. In order to represent numbers larger than $|A|$ a second basis B is chosen.

The combination of the values from A and B allow representation of integers less than $|A| \cdot |B|$. Consideration of the residues of the integer stored in either basis allows a free reduction by $|A|$ or by $|B|$. To take advantage of this free reduction, we require a base extension operation. When a number represented in RNS basis A, say x_A, is extended to basis B to form $x_B = x_A \bmod B$, the residue of $x \bmod |A|$ is computed modulo each of the moduli in B.

Algorithm 1 details the Cox-Rower algorithm for Montgomery multiplication in an RNS. The algorithm computes the product in both bases, using the product modulo the first basis to compute the reduction. Each of the basic arithmetic operations on encoded numbers is highly efficient on a vector architecture as are computed component-wise in parallel. Efficient implementation of the Montgomery Multiplication requires both the reduction of x_A modulo $|A|$, and the base extension to be inexpensive. Note that in the RNS representation A, reduction by $|A|$ is free, as it is a side-effect of representation in that basis.

3.3 Base Extension

There are several well-known algorithms for RNS base extension in the literature [21, 16, 18]. In each case the convention is to measure time complexity (or circuit depth) by the number of operations to produce a residue modulo a single prime. For use in Montgomery multiplication as described above, each algorithm must be executed n times to produce residues for each target moduli.

The earliest result is the Szabo-Tanaka algorithm [21] with $O(n)$ time complexity. The RNS number is first converted into a Mixed Residue System (MRS). The MRS is a positional format where the coefficients of the digits are products of the primes in the RNS system, rather than powers of a radix. So the RNS value x_A would be represented in MRS as an n-vector $m(x_A)$ such that

$$x_A = \sum_{i=1}^{n} \left(m(x_A)[i] \cdot \prod_{j=0}^{i-1} A[j] \right) \quad \text{where } A[0] = 1$$

This conversion is achieved in an n-step process. In each step the residue of the smallest remaining prime is used as the next MRS digit. This MRS digit is subtracted from the remaining residues, and the resultant number is an exact product of the prime; the number can be divided efficiently by the prime through multiplication by the inverse. Each step consume a single digit of the RNS representation and produces a single digit in the MRS representation. The Szabo-Tanaka algorithm then proceeds to accumulate the product of each MRS digit and its digit coefficient modulo each of the target moduli to construct the RNS representation. To allow single word operations the residues of the coefficients can be precomputed. The algorithm is highly suitable for a vector architecture as it uses uniform control flow over each vector component.

Posch et al. [16] and Kawamura et al. [18] both achieve $O(\log n)$. The approaches are similar and use the CRT as formulated in Equation 2, computing a partial approximation of k. The computation is iterated until the result has converged to within a suitable error bound. The iterative process requires complex control-flow creating inefficiency in the GPU programming model.

Shenoy and Kumaresan [20] claim that an extra redundant residue can be carried through arithmetic operations, and used to speed up the conversion process. Unfortunately their approach does not appear to work for modulo arithmetic. Assuming that our system operates in basis B, all arithmetic operations are implicitly modulo $|B|$. The redundant channel r is co-prime to $|B|$ and thus results $\bmod r$ cannot be "carried through". Whenever the result of an operation is greater than $|B|$, the result in the redundant channel would need to be reduced by $|B|$ before the reduction by r. This approach is not suited to applications involving modulo arithmetic unless r is precomputed before each operation.

4 RSA Implementation on GPU

Algorithm 2 is an overview of our GPU implementation of Montgomery multiplication. In the array-language syntax each variable refers to a vector, and

Algorithm 2: Cox-Rower Montgomery multiplier with Szabo-Tanaka Extension.

```
Input  : $x_A, x_B$ and $y_A, y_B$.
Output: $x \cdot y \cdot A^{-1}$, encoded into $w_A, w_B$.

Stage1   $s_A \leftarrow x_A \cdot y_B$ ;   $s_B \leftarrow x_B \cdot y_B$ ;   $t_B \leftarrow s_B \cdot -N^{-1} \pmod{|B|}$

Stage2   for $i = 1$ upto $n$ do
             $m[i] \leftarrow t_B[i]$ ;   $t_B \leftarrow t_B - t_B[i]$ ;   $t_B \leftarrow t_B \cdot B^{-1}[i, n]$

Stage3   for $i = 1$ upto $n$ do
             $t_A[i] \leftarrow \sum_j^n m[j] \cdot C_A[i, j] \pmod{|A|}$

Stage4   $u_A = t_A \cdot N$ ;   $v_A = s_A + u_A$ ;   $w_A = v_A \cdot |B|^{-1} \pmod{|A|}$

Stage5   for $i = 0$ upto $n$ do
             $m[i] \leftarrow w_A[i]$ ;   $w_A \leftarrow w_A - w_A[i]$ ;   $w_A \leftarrow w_A \cdot A^{-1}[i, n]$

Stage6   for $i = 1$ upto $n$ do
             $w_B[i] \leftarrow \sum_j^n m[j] \cdot C_B[i, j] \pmod{|B|}$
return $w_A, w_B$
```

an indexed variable refers to a single component. This representation shows the problem clearly by abstracting away important implementation details. Where operations are performed over vectors they refer to the component-wise application of the operation. A mapping is required between these arbitrary vectors and 4-vectors organised in two dimensional textures. Each shader instance on the GPU outputs a specific 4-vector within the texture. The coordinates of the target pixel must be used to parameterise the execution of each shader instance.

The horizontal bars that separate each stage represent parallel barriers in the algorithm; either a communication between independent processors or a change in the *shape* of the control-flow that executes. As the control-flow of each shader is lock-stepped this requires a new rendering operation. Executing single instances of the program over the different components of a vector provides a single implicit loop; the explicit loops must either be unfolded within the shader or implemented in multiple rendering steps.

Stages one and four constitute the Cox-Rower algorithm [18] for Montgomery multiplication in RNS. Variable names are retained from Algorithm 1 for clarity. Stages two and three together form a single base extension; converting the RNS value in t_B into MRS form in the vector m, then reducing these digits by the moduli of basis B. The matrix B^{-1} stores the inverse of each prime in B under every other prime. The MRS coefficients reduced by each prime in A are stored in the matrix C_A. The operation is repeated in stages five and six, extending w_A into w_B to provide the result in both bases.

Stage two requires a single output into $m[i]$ and modification of the t_B vector within a single rendering step. Each loop iteration is a single shader execution, the iteration counter i is passed as a uniform variable to all shader instances.

Algorithm 3: Pseudo-code implementation of stage two of Algorithm 2.

```
Input  : $i, x, y$
Output: $o$

$v \leftarrow T[x, y]$
$v' \leftarrow T[i, y]$
$B^{-1} \leftarrow P[x, i]$
$A \leftarrow P[n + x, 0]$
$t \leftarrow (v - v') \cdot B^{-1}$
$t_2 \leftarrow t \bmod A$
$c \leftarrow (x \leq i)$
$o \leftarrow v \cdot c + t_2 \cdot (1 - c)$
return $o$
```

When the current pixel column $x \leq i$ the input value is output, otherwise the operations on t_B are performed. To execute this logic as a uniform control-flow the switch is encoded into floating-point arithmetic. The encoding of stage two into a shader with uniform control-flow is shown in Algorithm 3. T and P are 2D textures that are read from GPU memory during execution. The 4-vector v is the current state of the digit this instance executes upon. v' holds component $m[i]$ that each digit is being reduced by. The P texture holds precomputed values such as B^{-1} and the primes in basis A. The computed value t_2 is the potential output if $x > i$. The logical condition is computed as a 0 or 1 in c, and then the product is used to decide the output in the final step.

The direct implementation in Algorithm 3 is quite inefficient as many unnecessary computations are performed to preserve uniform control-flow. Program specialisation offers a solution to this problem; by restricting the (i, x, y) context that the shader executes in we can generate specialised versions of the program that only contain the necessary operations. When these contexts correspond to rectangular 2D regions in the target texture this leads to an efficient implementation. The specialisation system that we have developed is beyond the scope of this paper, and will be covered in future work. Our description of the implementation is still valid; the generated programs preserve the semantics the original and only the execution speed is affected.

The implementation of Algorithm 3 is given in Appendix 6 to demonstrate the GLSL syntax and show the additional complexities in reality. Mainly, the differences are concerned with the formatting of texture coordinates and hence the two dimensional layout of the data in the texture. As a rough measure of complexity, the entire implementation uses 10 separate pixel shaders, which are specialised at runtime into 52 separate programs.

The representation of each RNS value requires several values to be precomputed and stored in a texture; two vectors of moduli to describe the bases, two matrices holding the inverse of each prime in a basis under every prime in the other basis, and two matrices holding the coefficients of each MRS modulo the primes in the other basis. At runtime the working textures hold vectors of

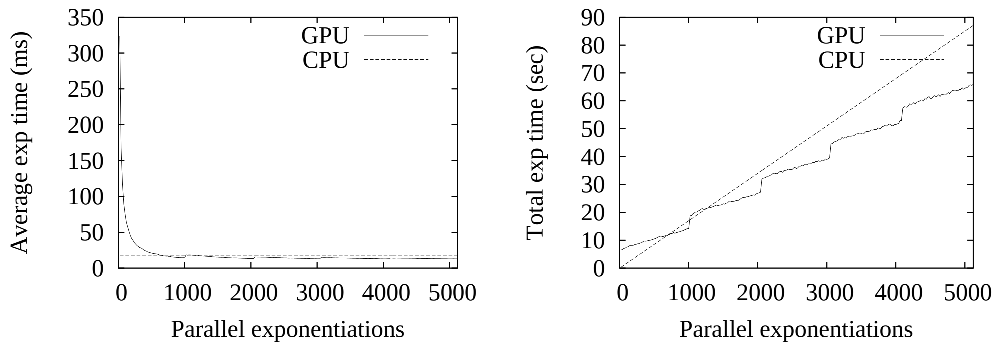

Fig. 1. Comparison of latency and throughput on the GPU vs an AMD-64.

residues to describe each of the variables in the algorithm. Each component of these vectors will be stored in a 24-bit mantissa of a floating-point value.

5 Experimental Results

Our goal in evaluating the performance of our GPU based implementation is comparison with a roughly equivalent implementation on a general purpose processor (CPU), namely an 2.2 GHz AMD-64 3200+. This processor was mounted in a computer that also acted as the host for the graphics accelerator; all experiments were conducted on this common platform.

The implementations described in Section 4 were developed for the Nvidia 7800-GTX graphics accelerator. The *direct* version executes the algorithm exactly as described. The *optimised* version features both low-level and high-level optimisations; although detailed description of these optimisations are beyond the scope of this paper, computation is functionally as previously described. Briefly, the low-level optimisations included changes to the internal texture format used by OpenGL, while the high-level optimisations included code transformations and specialisation of the pixel-shader programs. In both cases, the GPU based Montgomery multiplication was used within a binary exponentiation method [12, Chapter 14] in order to compute 1024-bit modular exponentiations. We additionally wrote a standard C implementation of Montgomery multiplication for our CPU; this included small assembly language inserts to expose critical features in the processor. The CPU based Montgomery multiplication was used within a 4-bit sliding window method [12, Chapter 14] to compute the same 1024-bit modular exponentiations.

Figure 1 shows a comparison of the optimised implementation against the general purpose processor in terms of latency and throughput. The disappointing feature is that the CPU evaluates a single exponentiation an order or magnitude faster than the GPU. The reason for this is obvious: there are significant overheads in initialising the OpenGL API, the (many) shader programs and transfer of data to and from the accelerator. More positively, if one considers evaluation of parallel exponentiations, as rationalised in Equation 1, the GPU is eventually able to amortise this overhead as the total workload grows; the break-even point is around 700 parallel exponentiations. The graph of throughput shows some unusual behaviour with respect to GPU performance; the vertical steps are an increase in program cost that could either be due to the memory cache or the thread scheduling on the accelerator. Further work will determine which is the case and how to mitigate the cost.

6 Conclusion

We have presented an investigation into the implementation and performance of modular exponentiation, the core computational operation in cryptosystems such as RSA, on a commodity GPU graphics accelerator. In a sense, this work represents an interesting aside from implementation on devices normally constrained by a lack of computational or memory resources: here the constraint is architecture targeted at a different application domain.

In an attempt to mitigate this difference and fit the algorithm into the GPU programming model, we employed vector arithmetic in an RNS which allowed us to capitalise on fine-grained SIMD-parallel floating point computation. Our experimental results show that there is a significant latency associated with invoking operations on the GPU, due to overhead imposed by OpenGL and transfer of data to and from the accelerator. Even so, if a large number of similar modular exponentiations are required, the GPU can capitalise on course-grained parallelism at the exponentiation level and out-perform a CPU. Although this comparison is unfair in a number of respects (the CPU uses windowed exponentiation while the GPU does not, the number of parallel exponentiations is unreasonably large) it is crucial to see that even small improvements in performance are important: the use of the GPU essentially enables utilisation of a "free" co-processor that would otherwise sit idle.

The advantages and disadvantages highlighted by our approach detail a number of issues worthy of further research; we expect that this will significantly improve GPU performance beyond the presented proof of concept. Two clear examples are the use of windowed exponentiation on the GPU to improve basic performance, and improvements in driver software to reduce fixed overheads. Future work should also highlight the role of program transformation techniques in generating code for these exotic and quickly evolving architectures. Finally, a newer GPU architecture, the 8800-GTX, is already available. The use of a higher core clock speed, faster memory and features such as a more general programming model, a lockable cache, and true scatter operations will all provide massive improvements in performance over the 7800-GTX used in this paper.

References

1. D.V. Bailey and C. Paar. Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography. In *Journal of Cryptology* **14** (3), 153–176, 2001.
2. P.D. Barrett. Implementing the Rivest, Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 263, 311–323, 1986.
3. D.J. Bernstein. The Poly1305-AES Message-Authentication Code. In *Fast Software Encryption (FSE)*, Springer-Verlag LNCS 3557, 32–49, 2005.
4. D.J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography (PKC)*, Springer-Verlag LNCS 3958, 207–228, 2006.
5. D.L. Cook, A.D. Keromytis, J. Ioannidis and J. Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards In *RSA Conference, Cryptographer's Track (CT-RSA)*, Springer-Verlag LNCS 3376, 334–350, 2005.
6. N. Costigan and M. Scott. Accelerating SSL using the Vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3. In *Cryptology ePrint Archive*, Report 2007/061, 2007.
7. R.E. Crandall. *Method and Apparatus for Public Key Exchange in a Cryptographic System.* U.S. Patent Number $5,159,632$, 1992.
8. M. van Dijk, R. Granger, D. Page, K. Rubin, A. Silverberg, M. Stam and D. Woodruff. Practical Cryptography in High Dimensional Tori. In *Advances in Cryptology (EUROCRYPT)*, Springer-Verlag LNCS 3494, 234–250, 2005.
9. J. Fournier and S. Moore. A Vectorial Approach to Cryptographic Implementation. In *International Conference on Digital Rights Management*, 2005.
10. GPGPU: General-Purpose Computation Using Graphics Hardware. Available at: `http://www.gpgpu.org/`
11. D. Hankerson, A. Menezes and S. Vanstone. *Guide to Elliptic Curve Cryptography*, Springer-Verlag, 2004.
12. A.J. Menezes, P.C. van Oorschot and S.A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1997.
13. P.L. Montgomery. Modular Multiplication Without Trial Division. In *Mathematics of Computation*, **44**, 519–521, 1985.
14. B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford University Press, 2000.
15. K.C. Posch and R. Posch. Modulo Reduction in Residue Number Systems. In *IEEE Transactions on Parallel and Distributed Systems*, **6** (5), 449–454, 1995.
16. K.C. Posch and R. Posch. Base Extension Using a Convolution Sum in Residue Number Systems. In *Computing 50*, 93–104, 1993.
17. J-J. Quisquater and C. Couvreur. Fast Decipherment Algorithm for RSA Public-key Cryptosystem. In *IEE Electronics Letters*, **18** (21), 905–907, 1982.
18. S. Kawamura, M. Koike, F. Sano and A. Shimbo Cox-Rower Architecture for Fast Parallel Montgomery Multiplication In *Advances in Cryptology (EUROCRYPT)*, Springer-Verlag LNCS 1807, 523–538, 2000.

19. R. Rivest, A. Shamir and L. M. Adleman A Method for Obtaining Digital Signatures and Public-key Cryptosystems. In *Communications of the ACM*, **21** (2), 120–126, 1978.
20. P.P. Shenoy and R. Kumaresan. Fast Base Extension Using a Redundant Modulus in RNS. In *IEEE Transactions on Computers* **38**(2), 292–297, 1989.
21. N.S. Szabo and R.I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology*, McGraw-Hill, 1967.

Appendix: Program Listing

This source code is a program template. At run-time our implementation substitutes a constant value for the template parameters indicated by <0> and <1>. These values are determined by the batch size and control the number of exponentiation instances encoded into each row, and the size of the texture respectively.

Listing 1.1. A GLSL pixel-shader program that implements Algorithm 3.

```
uniform sampler2D T;   // The matrix with intermediate results
                       // from the previous stage of ping-pong
uniform sampler2D P;   // The matrix with auxillery precomputed data
                       // such as A,B,B^-1 etc
uniform float i;    // The loop index - shader executes onces per iteration
void main(void)
{
    vec2  where    = gl_TexCoord[0].xy;    // Retrieve the xy parameters for
                                           // this instance.
    // Which of the moduli covers this target pixel
    float base     = floor( where.x / <0> );

    // Which exponentiation instance we are within the row
    float inst     = mod( where.x, <0> );

    // The location within P of the inverse, the scaling accounts for
    // coordinate normalisation
    vec2  invWhere = vec2( i, base ) / <1>;
    vec4  bInvs = texture2D( P, invWhere );

    // The moduli in base A for this pixel (4 components of tB)
    vec4  primes   = texture2D( P, vec2(88+base,0) / <1> );

    // Retrieve the "current" values from the ping-pong texture
    vec4  v     = texture2D( T, where / <1> );
    // Retreive the values of the current subtraction digit from P (v')
    vec4  v2  = texture2D( T, vec2(i * <0> +inst,where.y) / <1> );
    vec4  t2   = mod( (v-v2) * bInvs, primes );

    // Switch between passing through v or t2. This guarded form is
    // recognised by the compiler and produces straight-line code
    float c = (where.x<=i) ? 1 : 0;
    gl_FragColor = v*vec4(c) + t2*(1-(vec4)c);
}
```

Appendix : Ping-pong operation

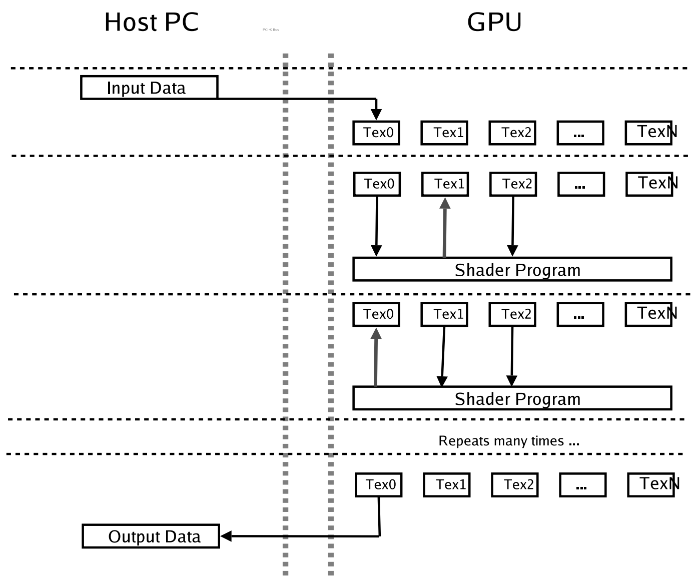

Fig. 2. Ping-pong operation in the GPU to produce intermediate results

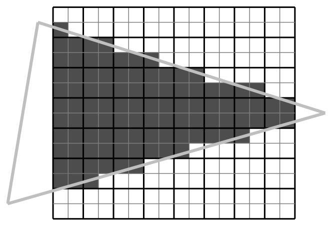

Fig. 3. Rasterisation of a graphics primitive into pixels.