Toward Acceleration of RSA Using 3D Graphics Hardware ⋆ ⋆⋆

A. Moss, D. Page and N.P. 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 however, this performance has made 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 [22] 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. With a general modulus m taking the value N or p respectively, on processors with a w-bit word size, one commonly represents $0 \leq x < m$ using a vector of $n = \lceil m/2^w \rceil$ radix-2^w digits. Unless specialist co-processor hardware is used, modular operations on such numbers are performed in software using well known techniques [15, 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

⋆ The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT. The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

⋆⋆ The work described in this paper has been supported in part by EPSRC grant EP/C522869/1.

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 not commodity items.

In an alternative approach, researchers have investigated cryptosystems based on arithmetic in fields modulo a small prime m or extension thereof. Since ideally we have $m < 2^w$, the representation of $0 \leq x < m$ is simply one word; low-weight primes [7] offer an efficient method for modular reduction. Examples 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) [16, Chapter 3] to implement RSA. Issues of security aside, the use of such systems is attractive as operations modulo m may be more efficiently realised by 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 work by Bernstein which outline 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], until recently the only published prior usage for cryptography was 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 was recently improved, using modern GPU hardware, by Harrison and Waldron [12] who also highlight the problems of overhead in communication with the card and miss reporting of host processor utilisation while performing GPU computation.

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 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 performance using the GPU versus that achieved using a general purpose processor (or CPU). An alternative approach is recent work implementing a similar primitive 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 general purpose IBM Cell, 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 4.3 compare the GPU implementation with one on a standard CPU, with conclusions in Section 5.

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

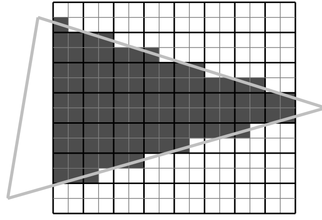


Fig. 1. Rasterisation of a graphics primitive into pixels.

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 1, 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 output 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. Read and write operations cannot be mixed on the same texture within a 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 implementation of modular exponentiation using RNS 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 texture. Increasing the number of ping-ponging steps increases the size of data-set required to break-even.

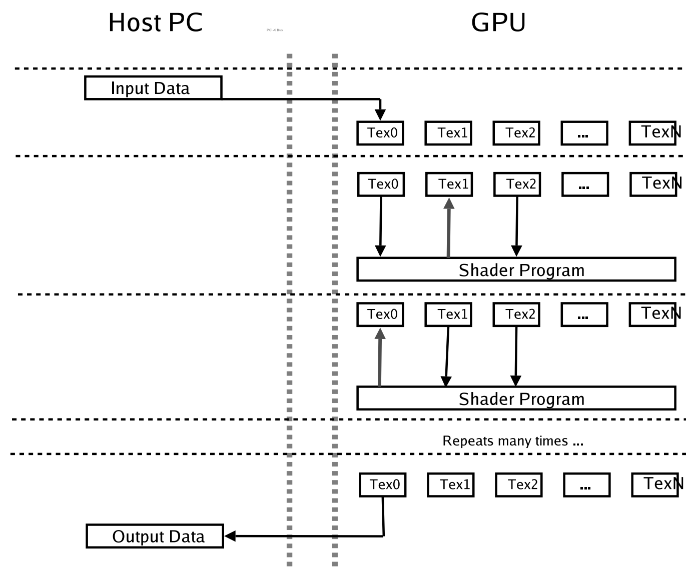


Fig. 2. Ping-pong operation in the GPU to produce intermediate results

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 Realising Modular Exponentiation with RNS

The RSA [22] 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, taking N as 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. [14, Chapter 14] for an overview.

In the specific case of RSA, the performance of a single modular exponentiation can be improved by employing small encryption exponent variants and the Chinese Remainder Theorem (CRT) [20] to accelerate the public and private key operations respectively. Since these cases are essentially specialisations of general modular exponentiation, we do not consider them. Instead we focus on using the GPU to implement the general case; one can expect performance improvements by specialising our techniques to suit, but this is a somewhat trivial step.

More relevant performance improvements can 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.

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

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

$$B = \langle\, B[1], B[2], \ldots, B[n] \,\rangle \text{ with } \gcd(B[i], B[j]) = 1 \text{ whenever } \; i \neq j$$

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 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 modular exponentiation depends on modular multiplication. In a positional number system this operation is commonly performed using Montgomery representation [15]. 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

Algorithm 1: Cox-Rower Algorithm [21] 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
$s_A \leftarrow x_A \cdot y_A$		$s_B \leftarrow x_B \cdot y_B$
		$t_B \leftarrow s_B \cdot \left(-N^{-1} \bmod \lvert B\rvert\right)$
	base extend $t_A \leftarrow t_B$	
$u_A \leftarrow t_A \cdot N_A$		
$v_A \leftarrow s_A + u_A$		
$w_A \leftarrow v_A \cdot \left(\lvert B\rvert^{-1} \bmod A\right)$		
	base extend $w_A \rightarrow w_B$	
return w_A, w_B		

that $x_M \equiv xR \pmod N$. 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 [18, 21]. As the suitable input range of the Posch algorithm [18] 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. [21]. 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 [24, 19, 21]. 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 [24] 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. [19] and Kawamura et al. [21] 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 [23] 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 mod r cannot be "carried through" from one operation to another. 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. To make this approach work for systems using modulo arithmetic, each operation requires an expensive reduction into base r before the redundant channel can be used to speed up the reduction for the other bases in the system.

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**
$m[i] \leftarrow t_B[i]$;
$t_B[i{+}1\,..\,n] \leftarrow t_B[i{+}1\,..\,n] - t_B[i]$;
$t_B[i{+}1\,..\,n] \leftarrow t_B[i{+}1\,..\,n] \cdot B^{-1}[i, n]$

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**
$m[i] \leftarrow w_A[i]$;
$w_A[i{+}1\,..\,n] \leftarrow w_A[i{+}1\,..\,n] - w_A[i]$;
$w_A[i{+}1\,..\,n] \leftarrow w_A[i{+}1\,..\,n] \cdot A^{-1}[i, n]$

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

4 Mapping RNS Arithmetic to the 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 an indexed variable refers to a single component. The algorithm describes operations on a vector architecture without specifying an explicit representation. On the GPU each stage will be encoded as a separate shader program written in the OpenGL Shading Language (GLSL [17]). One instance of the shader will execute for each vector component (each vector is the same length). Operations performed over vectors refer to a component-wise application; vector operations do not require communication between shaders.

The horizontal bars that separate each stage represent parallel barriers in the algorithm; either a communication is required between independent shaders, 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 shader 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. When communication is required between individual shaders in the loop then a new rendering step will be required. For example the loop in Stage 2 performs an update on the elements of t_B beyond the i-th position. Each iteration $i + 1$ requires the updated value from the i-th

position. Hence each iteration of the loop requires a new rendering step, both as a barrier and to allow the computed value to be retrieve from the ping-pong.

Stages one and four constitute the Cox-Rower algorithm [21] 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 A to produce $t_A = t_B \bmod A$. 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.

The most important issue in implementing Algorithm 1 is how the vector components are mapped into the GPU. Memory on the GPU is organised as two dimensional arrays called textures. Each element (pixel) is a tuple of four floating-point values. The floating-point format uses a 24-bit mantissa that allows the representation of integers up to $2^{24} - 1$. Within this range arithmetic operations produce the correct values. When this range is exceeded precision is lost as the floating-point format stores the most significant bits. This poses a significant problem for cryptographic application where the least significant bits are required.

There are two aspects to the issue of choosing a mapping from the vectors in Algorithm 1 that both impact performance. The issue of how to represent words within each floating point value is covered in Section 4.1. How the arrangement of the tuples of floating point values affects the execution of shaders is covered in Section 4.2. Neither topic can be explained completely independently of the other and unfortunately some material is split across both sections. In Section 4.3 we describe the performance results of the different choices of representation.

4.1 Floating-point Arithmetic

The most problematic operation is multiplication of two words, and then reducing the result by a modulus in the system. The simplest method of avoiding overflow is to use half-word values; bounding the moduli by 2^{12}. This ensures that products are correctly represented in a single word and the GLSL mod operation can be used for reduction. Twice as many moduli are required when they are restricted to half-words, this causes two major problems: the necessary memory bandwidth is doubled, the complexity of the base extension is $O(n^2)$ so doubling the number of digits will quadruple the amount of computation required. The main advantages are the simplicity of the implementation and the fast speed of word-sized modulo multiplications. The `mod` compiles into 3 instructions that are executed in 1.5 cycles producing a modular multiplication of 2 cycles. In the results section we refer to the code using this representaton as Implementation-A.

An alternative approach is to use primes less than 2^{24} as moduli, and perform multi-precision arithmetic to handle the double-word product. Techniques [13] for multi-precision arithmetic in floating point are well known. Dekker-style splits can be used to compute the low-order bits by subtracting partial products from

the high-order bits in a floating-point product. The 15 operations required are executed in 7.5 cycles but the technique requires guard bits in the floating-point unit to guarantee correctness. Experiments to determine the validity of this approach on the floating-point implementation in the 7800-GTX were unsuccesful.

The reductions in complexity and memory traffic from full-word moduli are desirable, and so we have investigated alternatives to the multi-precision approach. One possibility is to pack two half-words in every word. This creates the reduction in memory traffic, and once a word is retrieved it can be unpacked into two separate values. The modular multiplication is then the 2 cycle operation over half-words. The number of digits in the system is not reduced, but there is still a performance advantage. Unfortunately the Nvidia compiler necessary to use GLSL on the GPU is still immature, and has difficulty performing register colouring correctly when this much data is active in a shader. Examination of the output of the compiler shows that twice as many registers as required are allocated, and that the actual performance of the shader suffers. As there is no way to load binary code directly onto the device it is not possible to check the performance of correctly scheduled code with proper register allocation. Experiments with similar code fragments suggest that when the compiler matures enough to compile this code correctly there will be a significant increase in performance for this method. Similar experiments [25] have shown dramatic decreases in performance for each power of two that the number of registers exceeds, this suggests that scheduling shader invocations with a shared register bank causes the architectural problem, and that the compiler is unable to avoid this case. This representation is used in Implementation-C along with the specialisation techniques described below.

Another representation is provided by the condition that RNS moduli are not required to be prime, merely coprime to each other. Consider a modulus m chosen with two prime factors p and q, such that $m < 2^{24}$. For $a, b \in (0 \mathinner{..} m-1)$ the product $a \cdot b \bmod m$ can be computed without intermediate values that exceed 2^{24}. To do so, compute:

$$a_p \leftarrow a \bmod p,\ a_q \leftarrow a \bmod q,$$
$$b_p \leftarrow b \bmod p,\ b_q \leftarrow b \bmod q,$$
$$t_1 \leftarrow a_p \cdot b_p \bmod p,\ t_2 \leftarrow t_1 \cdot q^{-1} \bmod p,\ t_3 \leftarrow t_2 \cdot q,$$
$$t_4 \leftarrow a_q \cdot b_q \bmod q,\ t_5 \leftarrow t_4 \cdot p^{-1} \bmod q,\ t_6 \leftarrow t_5 \cdot p,$$
$$(t_3 \cdot s(t_3) \cdot m) + t_6$$

The modulo operations in each term are necessary to keep the intermediate values within the word range and prevent overflow. The final term is the product reduced by m. The function $s(x)$ denotes the sign of x and is a primitive function in GLSL. This unusual reduction trades multiplications for inexpensive modulo operations. On the GPU the cost of a word-sized `mod` is only 1.5 cycles, whereas the cost of a multiplication that overflows the word is large[1]. By avoiding

[1] Observed cost per operation increased with the number of operations due to register spilling in the Nvidia compiler

overflow in the floating-point registers with `mod` operations we are treating them as mantissa sized integer registers. The entire operation takes 16 cycles. Although this is much higher than the half-word approach the total cost of the shaders is comparable because register spilling is reduced and the GPU appears to dispatch the shader faster. In our results this representation is used in Implementation-D.

The reduction described can be seen as an unfolding of the CRT over two moduli. This leads to our final approach to modular multiplication in the GPU. The independent results under p and q do not have to be recombined on each operation. Each digit in the RNS that we are using is represented by a smaller scale RNS comprised of two primes. This digit can be operated on mod$p$ and mod$q$ for fast operation, or the entire digit can be used as part of the larger RNS. This choice of representation with a fast map between the two allows us to compute the chain of operations in a shader independently in each base, but to execute the base conversion over a lower number of digits and so reduce the complexity. When this reduction is combined with the specialisation technique described below, many of the mappings from $x \bmod m \mapsto \langle x_p, x_q \rangle$ are paired with mappings from $\langle x_p, x_q \rangle \mapsto x \bmod m$ and so can be removed from the computation. Each of the individual modular multiplications are primitive GLSL `mod` operations. This results in a substantial increase in overall performance. The code using this technique is refered to as Implementation-E.

4.2 Memory Layout and Shader Specialisation

Each instance of a shader executes on a single pixel in a texture. Four floating-point values are computed by the execution of every shader, and are stored in a fixed location within the texture. When a shader is rendered over a rectangle in a texture the output location is fixed for each instance. The execution of each shader then proceeds in lockstep, with the same instructions being dispatched in each processor pipeline.

Initially the only distinction between the execution of the programs is the coordinate of the pixel, available to the program as a 2D vector `gl_TexCoord[0]`. One major difficulty in writing GPU shaders is distinguishing the computation in that particular instance, from that performed in every other instance. Although shaders can make differing texture lookups from other instances, sufficient information to decide which data to lookup must be encoded in the pixel coordinates.

Our strategy for memory layout is to encode which instance, and which vector components within that instance a particular pixel holds by the spatial location of the pixel. Each row of the texture holds c instances. The pixels containing the first instance are located in columns 0, c, $2c$, $\ldots$, nc where $4n$ is the number of moduli in the RSA system. The moduli associated with the 4 residues in a pixel are consistent across c columns, and the entire height of the texture. The advantage of this striding pattern is that contiguous rectangles describe pixels associated with particular moduli.

Of the four textures available for input and output, we use two to perform a ping-pong operation. Essentially these textures are used to hold the *new* and *previous* states of mutable variables. For the case of 1024-bit instances, 88 moduli

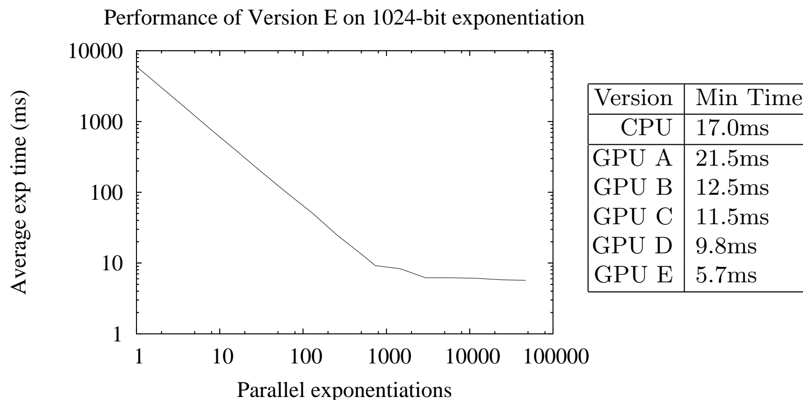
less than 2^{12} describe the system, or 44 moduli less than 2^{24}. Because values are required in bases A and B we split the textures in two vertically and have two independent sets of columns to hold values. We use the largest textures possible [2] on the drivers and hardware which are 1024×1024 pixels in size. For the half-word case this allows 23 instances per row for a total of 23552 parallel instances. In the full-word case a texture stores 47104 parallel instances. For the binary operations in the Cox-Rower Algorithm we use a third texture as a data source. The texture is updated between modular multiplication algorithms as part of the exponentiation. The fourth texture contains auxiliary read-only data such as the values of inverses, primes and other data.

Our initial experimental results used this layout scheme, but profiling of the performance suggested that the memory traffic reading the auxiliary texture dominated the runtime. Current work in the GPGPU community suggests that this approach of treating textures as arrays is common. One reason for this is that the 'arrays' offered in GLSL are not true arrays because of hardware limitations. All array indices must be compile-time constants, which limits the potential for storing auxiliary data in local lookup tables and indexing it by the current pixel coordinate.

To circumvent this restriction we have employed a program transformation technique called specialisation. Some of the inputs to a specialised program are fixed to constants known at compile-time. This allows constant propagation to remove operations in the compiled program. Constants are created by fissioning program templates into separate shaders that run over c columns in the texture. For each pixel rendered by a particular shader the set of moduli is constant. This propagates through the program and makes the other auxiliary data constant by virtue of the fact that the lookup table indices become constant. After specialisation the program template generates either 11 or 22 separate shaders, depending on if half-word or full-word arithmetic is used. Within each shader the constant auxiliary data is coded as a local array of constants. In GLSL for Nvidia GPUs constant data is encoded within the instruction stream, removing all memory transfers to fetch the data. As a side-effect this frees the fourth texture which may allow performance improvements in the future. An example of this technique is shown in Listing 1.1. The code is a program template for the MRS reduction step in Implementation-A. The <1> syntax refers to terms that are compile-time constants, our implementation generates multiple shaders from each template replacing the parameters with constant values in each instance — these generated programs can then be compiled as normal GLSL code. The implementations that use half-word values store a 1024-bit number in 22 pixels, using 4 floating-point numbers in each. Each template generates 22 separate shaders in this case. The full-word versions store the same value as 11 pixels, and generate 11 separate shader programs.

Despite the increased costs of running shaders over smaller regions, and flushing the program cache on the GPU because of the large number of programs

[2] The 7800-GTX supports 4096×4096 textures but we were unable to get stable results with the current drivers



Version	Min Time
CPU	17.0ms
GPU A	21.5ms
GPU B	12.5ms
GPU C	11.5ms
GPU D	9.8ms
GPU E	5.7ms

Fig. 3. Performance of 1024-bit Modular Exponentiation

being executed — the technique still produces a significant increase in performance.

4.3 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.

To explore the optimisation space for the algorithm on the 7800-GTX and gauge the importance of the trade-offs described in Section 4 we implemented several versions. In all cases the modular multiplication was used within a binary exponentiation method [14, Chapter 14] to compute 1024-bit modular exponentiations. The exponents were all chosen to have hamming weights of 512. A standard C implementation of Montgomery multiplication was written for our CPU; this included small assembly language inserts to expose critical features in the processor (i.e. retrieving double word products). The CPU based Montgomery multiplication used a 4-bit sliding window method [14, Chapter 14] to compute the same 1024-bit modular exponentiations. The CPU implementation achieved 17ms on our hardware, using sliding windows which we have not implemented on the GPU due to difficulties in memory management. The performance of each implementation is shown in Figure 3 along with the details of how the increase in parallelism is reflected in the average operation times (throughput). The latency for the data in this graph is the number of operations $\times$ the average time.

Implementation-A This is an unoptimised version that uses the half-word representation and no specialisation. This was a first attempt to naively treat the GPU as a set of fast memory arrays and ALUs. The result was

somewhat disappointing given the 35 : 1 performance ratio between the two platforms.

Implementation-B This version is specialised over 22 shaders for each template; expanding the templates produces $2 \cdot 22 + 8 = 52$ shaders in total. The half-word representation is still used but some low level optimisations including changes to the internal texture format used by OpenGL have been applied. The increase in performance is dramatic, although somewhat hampered by the switching cost of 52 separate shaders being executed.

Implementation-C This version alters Implementation-B by adapting the full-word data representation; there are still 52 shaders but with an overall increase in performance because of the lower memory bandwidth requirements.

Implementation-D This version alters Implementation-B by adapting smooth word-size representation. Although the reduction is expensive compared to the previous methods, the combination of reduction and only 11 shaders increases the performance again. The increase was not uniform, the shaders for Stage2 benefited from increased performance, while the shaders for Stage3 decreased in performance by 50%. The relative expense of the reduction (compared to GLSL `mod` methods) means that performance is dependent on the memory bandwidth to computation ratio in the particular shader.

Implementation-E Finally, this version alters Implementation-D, still using the smooth word-size representation, by substituting the reductions for GLSL `mod` operations working in each of the paired bases. The improvement in performance is dramatic, resulting in a 3 fold increase over the CPU implementation. This result is despite the lack of windowing in the GPU implementation.

Each of these figures was produced over many parallel instances of exponentiations which results in a large latency. This is necessary because the drivers are optimised for graphics workloads, and the rapid switching of shader programs is causing problems. Some of the latency issues could be rectified if Nvidia optimised their drivers for non-graphics workloads, but this would require an uptake of GPGPU in the marketplace.

Our performance figures for exponentiation on the CPU should be seen only as a broad guideline. The implementation uses a common choice of algorithms but has not had extensive low-level optimisation. Reports from other researchers and the figures in the *Crypto++* benchmarks suggest that aggressive low-level optimisation can reduce this figure on our reference platform to about 5ms. The GPU implementation cannot be aggressively optimised in the same way, but gives results comparable to the fastest CPU implementation. Our experiments suggest that given low-level access to the GPU (bypassing the GLSL compiler) there are similar potential performance increases in the GPU platform.

5 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. Previous results in GPGPU have shown that such a limited architecture can be successfully targeted at problems outside of the intended domain. We believe that integer modular exponentiation is the most radically different problem domain so far reported.

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 uneven in a number of respects (the CPU uses windowed exponentiation while the GPU can not, the GPU uses an unreasonably large number of parallel exponentiations) it is crucial to see that exploiting a commodity resource for cryptographic primitives offers interesting possibilities for the future. Although current drivers miss-report 100% utilisation of the CPU during GPU operation, this is not a requirement since synchronisation between CPU and GPU is essentially implemented as a spin-lock. Most of the GPU computation time does not require CPU intervention and manually scheduling code before calling the `glFinish()` synchronisation allows use of the CPU for other tasks. This raises the possibility of utilising an otherwise idle resource for cryptographic processing.

5.1 Further Work

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.

- Due to the difficulty of managing multiple textures (that might represent pre-computed input, for example) on the GPU, in this work we opted to use a very basic exponentiation algorithm. Clearly huge advantages can be made if some form of pre-computation and windowing can be used [14, Chapter 14]; we aim to investigate this issue as a means of optimisation in further work. In particular the specialisation technique frees up a texture that can be used as a second output in *multi-texturing*.
- A major limiting factor in deploying these techniques in real systems is the large number of parallel exponentiations that need to be executed in order to break-even on various overheads imposed by the GPU. In particular, the driver software for the GPU is optimised to deal with repeated execution of a limited set of pixel shaders on large data-sets; this contrasts with our demands for larger numbers of pixel-shaders and comparatively smaller data-sets.

– Program transformation and specialisation techniques to generating code for these exotic and quickly evolving architectures is paramount to success. Our target architecture forced us to use somewhat complex instruction flows (consisting of 52 generated shaders); we expect that by specialising, for a specific RSA exponent for example, the associated overhead can be significantly reduced. An automated system for achieving this task efficiently is work in progress.

5.2 Evolving Architectures

The CPU market is rapidly evolving. A new class of architecture is released every year, with roughly twice the performance of the previous generation. This rapid progress makes the GPU a desirable target for cryptographic code, however it also creates a problem that results may not stay relevant for very long. While this paper was being written a new generation of Nvidia GPU, the 8800-GTX, has been released. The strict separation between programmer and hardware that is imposed by the driver allows GPUs to be completely redesigned in each generation.

The new 8800-GTX uses a higher core clock speed and faster memory to change the memory / computation trade-offs in GPU programming. The vector units are finer-grained, with 128 scalar ALUs available to the programmer. The cache is lockable through software and primitive are available to control the mapping of software instances onto the vector array. Finally, the memory architecture allows scatter operations which enlarges the set of programs that can be written.

All of these changes would make interesting future research for implementing cryptographic code. However the results in this paper are still relevant as the *cost* in terms of transistor count, or power consumption for these more fully featured ALUs is higher than the simple ALUs used in the 7800-GTX. Hence these results are applicable to future resource constrained systems with a similar architecture.

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. O. Harrison and J. Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 4727, 209–226, 2007.
13. D.E. Knuth. *The Art of Computer Programming.* Vol. 1-3. Addison-Wesley, 3d ed., 1997. Additions to v.2 can be found in http://www-cs-faculty.stanford.edu/ knuth/err2-2e.ps.gz http://www-cs-faculty.stanford.edu/~knuth/err2-2e.ps.gz.
14. A.J. Menezes, P.C. van Oorschot and S.A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1997.
15. P.L. Montgomery. Modular Multiplication Without Trial Division. In *Mathematics of Computation*, **44**, 519–521, 1985.
16. B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.
17. Randi J. Rost *OpenGL Shading Language*. Addison Wesley, 2004.
18. 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.
19. K.C. Posch and R. Posch. Base Extension Using a Convolution Sum in Residue Number Systems. In *Computing 50*, 93–104, 1993.
20. J-J. Quisquater and C. Couvreur. Fast Decipherment Algorithm for RSA Public-key Cryptosystem. In *IEE Electronics Letters*, **18** (21), 905–907, 1982.
21. 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.
22. 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.
23. 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.
24. N.S. Szabo and R.I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology*, McGraw-Hill, 1967.
25. Ian Bucks. Invited Talk at *Eurographics/SIGGRAPH Workshop on Graphics Hardware 2003*, Slides available at `http://graphics.stanford.edu/~ianbuck/GH03-Brook.ppt`

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. An example program template.

```
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);
}
```