Implementing modular arithmetic using OpenCL

Fredrik Gundersen

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2010

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Implementing modular arithmetic using OpenCL

Fredrik Gundersen

2010/07/01

Abstract

Problem description: Most public key algorithms are based on modular arithmetic. The simplest, and original, implementation of the protocol uses the multiplicative group of integers modulo p, where p is prime and g is primitive root mod p. This is the way Diffie-Hellman is implemented. RSA is implemented in a similar way $c = m^e \bmod p*q$. For this reason public key crypto RSA is much slower than symmetric key algorithms, like DES and AES. Recently the field of using Graphics Processing Units (GPUs) for general purpose computing has become more widespread. Many computational problems have gained a significant performance increase by using the highly parallel properties of the GPU.

Motivation: Implementing public key algorithms using OpenCL allows the implementation to query the system for OpenCL enabled devices(GPU,CPU and other parallel processors) to select the best device in order to run the encrypting/decrypting of data. The same implementation can be run on a variety of different system with different GPUs, CPU as long as at least one device is able to run OpenCL programs/code.

Planned contribution: The planned outcome of this project is a fast implementation of public key algorithms able to run in parallel on a variety of parallel devices(GPU,CPU and other parallel processors) that is capable to run OpenCL code/programs.

Preface

This is a master thesis written by Fredrik Gundersen at the department of computer science and media technology at Gjøvik University College. The project was written in the spring semester of 2010. Patrick Bours at the Norwegian Information Security Lab (NISLab) was both supervisor and responsible professor for the project.

First I want to thank my wife Anne Karin for enabling me to have the time to write this thesis. Also thanks to my kids for understanding that daddy has to work on the weekends.

Second, thanks to Patrick for helping me with all my questions and always be available for helping me out. Also thanks to Robert Szerwinski and Owen Harrison for helping me with questions on how they did there solutions using CUDA.

Fredrik Gundersen, 2010/07/01

Contents

List of Figures

List of Tables

1 Introduction

1.1 Topic covered by the project

Most public key algorithms are based on modular arithmetic, e.g. RSA and Diffie-Hellman. Public key encryption and decryption are computationally heavy because a lot of modular multiplications with very large numbers are needed to perform these tasks. The security of the RSA crypto system is based on two mathematical problems: the problem of factoring large numbers and the RSA problem. In cryptography, the RSA problem summarizes the task of performing an RSA private-key operation given only the public key. Full decryption of an RSA cipher text is thought to be infeasible on the assumption that both of these problems are hard, i.e., no efficient algorithm exists for solving them[1]. Providing security against partial decryption may require the addition of a secure padding scheme. Diffie-Hellman key exchange (D-H) is a cryptographic protocol that allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher. The simplest, and original, implementation of the protocol uses the multiplicative group of integers modulo p, where p is prime and g is primitive root mod p.

1.2 Problem description

The RSA problem is also the main reason public key crypto is much slower than symmetric key algorithms, like DES and AES. Recently the field of using Graphics Processing Units (GPUs) for general purpose computing has become more widespread[2,3]. Many computational problems have gained a significant performance increase by using the highly parallel properties of the GPU.

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL was initially developed by Apple Inc., which holds trademark rights, and refined into an initial proposal in collaboration with technical teams at AMD, Intel, and Nvidia. Apple submitted this initial proposal to the Khronos Group[4]. On June 16, 2008, the Khronos Compute Working Group was formed[5] with representatives from CPU, GPU, embedded-processor, and software companies. This group worked for five months to finish the technical details of the specification for OpenCL 1.0 by November 18, 2008. This technical specification was reviewed by the Khronos members and approved for public release on December 8, 2008[6]. OpenCL 1.0 has been released with Mac OS X v10.6 ("Snow Leopard"). According to an Apple press release[7]:

> Snow Leopard further extends support for modern hardware with Open Computing Language

[1] http://en.wikipedia.org/wiki/RSA_Factoring_Challenge
[2] http://gpgpu.org
[3] http://www.nvidia.com/object/cuda_home.html#
[4] http://www.khronos.org/opencl/
[5] http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/
[6] http://www.khronos.org/news/press/releases/the_khronos_group_releases_opencl_1.0_specification/
[7] http://www.apple.com/pr/library/2008/06/09snowleopard.html

> (OpenCL), which lets any application tap into the vast gigaflops of GPU computing power previously available only to graphics applications. OpenCL is based on the C programming language and has been proposed as an open standard.

Snow Leopard was released to the public on friday 28.08.2009.

1.3 Keywords

- Data encryption
- Public key cryptosystems
- Performance evaluation
- Parallel processing

1.4 Justification, motivation and benefits

The objective of this project is to make a fast implementation of public key algorithms on a GPU using the OpenCL specification as implemented in OS X 10.6. The operation that needs to be executed in parallel is modular multiplication, as this is the basis of modular exponentiation. Furthermore a performance comparison between the GPU and a normal CPU implementation should be made. Implementing public key algorithms using OpenCL allows the implementation to query the system for OpenCL enabled devices(GPU,CPU and other parallel processors) to select the best device in order to run the encrypting/decrypting of data. The benefits is that the same implementation can be run on a variety of different systems with different GPUs, CPUs as long as at least one device is able to run OpenCL programs/code.

1.5 Research questions

- How to make modular multiplication execute in parallel on a GPU as efficiently as possible. This is the essence of the project as this is the basis of modular exponentiation. This operation is used both for RSA and for Diffie-Hellman.
- How to take full advantage of the parallel execution of the GPU when implementing public key algorithms using OpenCL.
- How to best utilize the memory bandwidth of the graphics card, this is a key component in making a fast implementation of public key algorithms.

1.6 Planned contributions

The planned outcome of this project is a fast implementation of public key algorithms able to run in parallel on a variety of parallel devices(GPU,CPU and other parallel processors) that is able to run OpenCL code/programs.

2 Related work

There have been several others that have done work in the field of making modular multiplication execute in parallel on a GPU, from the early work using specialized API's to the more recent that use Nvidia's CUDA[1] , so far there don't seem to exist any papers where the use of ATI's Stream SDK[2] have been tried. This is one of the things to be tried in this thesis, to implement a fast modular multiplication algorithm that run on both Nvidia and ATI using the OpenCL capabilities of OS X[3]. This implementation can be used on any system as long as it is able to run OpenCL programs/software. Both ATI[4] and Nvidia[5] are working on developing OpenCL support for their products, so it looks like there will be wide support of OpenCL on different operation systems soon. In most papers from the later years on related work, we see that CUDA is the method that is used the most.

2.1 GPU computing power

Adrin Boeing did in 2008 a survey of the current state of cryptographic function implementations on GPGPUs(General-Purpose computation on Graphics Processing Units) [1]. He did not limit his finding to just Nvidia but included hardware from other vendors as well, ATI/AMD, Intel and IBM/Sony/Toshiba. He created a table that showed the performance at the time of the survey. Table 1 on Page 4 shows the details of his findings.

GPU's have improved significantly since the survey was conducted. ATI, Nvidia and Intel have done much work on further developing their GPU platforms. ATI has released their next generation GPU the ATI Radeon™ HD 5800 Series[6] with a promise of processing power (single precision): 2.72 TeraFLOPS and processing power (double precision): 544 GigaFLOPS for the 5870 model. This is a huge leap forward from the 1.2 TeraFLOPS of the old 4870 model as shown in Boeing's survey in Table 1 on Page 4. Nvidia is working on its next generation CUDA Architecture called Fermi[7] they promise up to 512 CUDA cores in one GPU. Maybe the most interesting news in this GPU architecture is the support for concurrent kernel execution, where different kernels of the same application context can execute on the GPU at the same time. Concurrent kernel execution allows programs that execute a number of small kernels to utilize the whole GPU[8]. So the news from the two largest GPU manufactures tells of an exciting time to come for GPU

[1] http://www.nvidia.com/object/cuda_home.html#
[2] http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx
[3] http://www.apple.com/macosx/
[4] http://ati.amd.com/technology/streamcomputing/opencl.html
[5] http://www.nvidia.com/object/cuda_opencl.html
[6] http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx
[7] http://www.nvidia.com/object/fermi_architecture.html
[8] http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

Technology	Specifications	Peak billion operations per second
Nvidia GTX 280	30 (cores) * 8 (SIMD units) * 3 (operations per unit) * 1300 MHz (clock speed)	936
ATI/AMD Radeon 4870	10 (cores) * 16 (SIMD units) * 10 (operations per unit) * 750 MHz (clock speed)	1200
Sony, Toshiba, IBM CELL	8 (SPE cores) * 4 (SIMD units) * 2 (operations per unit) * 4 GHz (clock speed)	256
Intel Larrabee (predicted)	32 (cpu cores) * 16 (SIMD units) * 2 (operations per unit) * 2 GHz (clock speed)	2000
Intel, Core 2 E7200	2 (CPU cores) * 4 (SIMD units) * 2 (operations per unit) * 2.53 GHz (clock speed)	40

Table 1: Peak performance characteristics of computing technology

based performance and the use of GPU calculations in end user applications, and not just game performance.

2.2 Modular multiplication and RSA

From [2] we can find that the work of modular arithmetic is very old.

> Original works on modular arithmetic are very old. The Chinese Remainder Theorem was first proposed around the fifth century by Sun Tsu [3] But the use of this arithmetic to represent numbers was introduced only in 1959 by H.L. Garner [4].

Peter Lawrence Montgomery is an American mathematician who is widely published in the mathematical end of the field of cryptography.

> Montgomery's Modular Multiplication Algorithm [5]:
> $\{Pre-condition: 0 \leq A < r^n\}$
> R := 0 ;
> For i := 0 to n-1 do
> Begin
> $R := R + a_i * B$;
> $q_i := (-r_0 m_0^{-1}) \bmod r$; $R := (R + q_i * M)$ div r;
> $Invariant: 0 \leq R < M + B$
> End
> { Post-condition: $Rr^n = A * B + Q * M$ and, consequently, $R \equiv (A * B * r^{-n}) \bmod M$ }
> Define an N-residue to be a residue class modulo N. Select a radix R coprime to N such that R > N and such that computations modulo R are inexpensive. Let iR^{-1} and N′ be integers satisfying $0 < R^{-1} < N$ and $0 < N' < R$ and RR^{-1} - NN' = 1.

If we define $A_i = \sum_{j=i}^{n-1} a_j r^{j-i}$ so that $A_i = rA_{i+1} + a_i$ and use similar notation for Q_i then it is easy to prove by induction that at the end of the *i*th iteration $R = A_i xB - Q_i xM$. Hence the

post-condition holds.

Many others have built on this algorithm to produce a smaller and faster way of doing modular multiplication.

A Residue Number System (RNS) represents a large integer using a set of smaller integers, so that computation may be performed more efficiently. It relies on the Chinese remainder theorem of modular arithmetic for its operation, a mathematical idea from Sun Tsu Suan-Ching (Master Sun's Arithmetic Manual) in the 4th century AD.

A residue number system is defined by a set of N integer constants, $\{m_1, m_2, m_3, \ldots, m_n\}$, referred to as the moduli. Let M be the least common multiple of all the m_i. Any arbitrary integer X smaller than M can be represented in the defined residue number system as a set of N smaller integers $\{x_1, x_2, x_3, \ldots, x_N\}$ with x_i = X modulo m_i representing the residue class of X to that modulus. Note that for maximum representational efficiency it is imperative that all the moduli are coprime, that is, no modulus may have a common factor with any other. M is then the product of all the m_i. The main interest of the Residue Number Systems is to distribute integer operations on evaluations with the residues values. Thus an operation with large integers is made on the residues which are small numbers and where computations can be executed independently for each modulo allowing a complete parallelization of the calculus.

Mixed radix numeral systems are non-standard positional numeral systems in which the numerical base varies from position to position. Such numerical representation is advantageous when representing units that are equivalent to each other, but not by the same ratio. For example, 32 weeks, 5 days, 7 hours, 45 minutes, 15 seconds, and 500 milliseconds might be rendered relative to minutes in mixed-radix notation as:
$32_{52}5_7 7_{24}45_{60}.15_{60}500_{1000}$
There are a number of ways to convert a number in one base (radix) to the equivalent number in another base. The standard techniques are all variations on three basic methods. The most straightforward technique is perhaps the expansion method. Suppose we wish to convert the binary number 10101.1 to decimal. We may do so merely by using the definition of a number representation as an abbreviated polynomial, Thus, we may write
$10101.1_2 = 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1}$
$= 16 + 0 + 4 + 0 + 1 + 0.5$
$= 21.5_{10}$

But suppose we wish to go the other way. How would we convert 21.5_{10} to binary? Writing $21.5_{10} = 2x10^1 + 1x10^0 + 5x10^{-1}$ does not seem to be of much help. But look at what we get when we write this polynomial in binary notation ($10_{10} = 1010_2$ and $5_{10} = 101_2$, of course):
$21.5_{10} = (2x10^1 + 1x10^0 + 5x10^{-1})_{10}$
$= (10x1010^1 + 1x1010^0 + 101x1010^{-1})_2$
$= (10100 + 1 + 0.1)_2$
$= 10101.1_2$
The above examples illustrate an important fact about the conversion techniques that we will examine—namely, that they may be used to convert from any base to any other base. This is

important to remember, particularly because many texts show conversion from radix-a to radix-b being done one way, and conversion from radix-b to radix-a being done another, the implication being that the methods of conversion are fundamentally asymmetric.

The Chinese remainder theorem is a result about congruences in number theory and it's generalizations in abstract algebra. The original form of the theorem, contained in a third-century AD book Sun Zi suanjing (The Mathematical Classic by Sun Zi) by Chinese mathematician Sun Tzu and later republished in a 1247 book by Qin Jiushao, the Shushu Jiuzhang (Mathematical Treatise in Nine Sections) is a statement about simultaneous congruences (see modular arithmetic). Suppose $n_1, n_2, \ldots, n_k$ are positive integers which are pairwise coprime. Then, for any given integers $a_1, a_2, \ldots, a_k$, there exists an integer x solving the system of simultaneous congruences
$X \equiv a_1 \pmod{n_1}$
$X \equiv a_2 \pmod{n_2}$
$\vdots$
$X \equiv a_k \pmod{n_k}$
Furthermore, all solutions x to this system are congruent modulo the product $N = n_1 n_2 \ldots n_k$.
Hence $X \equiv y \pmod{n_i}$ for all $1 \leq i \leq k$, if and only if $X \equiv y \pmod{N}$.
Sometimes, the simultaneous congruences can be solved even if the n_i's are not pairwise coprime. A solution x exists if and only if:
$a_i \equiv a_j \pmod{\gcd(n_i, n_j)}$ for all i and j.
All solutions x are then congruent modulo the least common multiple of the n_i.

In mathematics, a Mersenne number is a positive integer that is one less than a power of two:
$M_p = 2^p - 1$
Some definitions of Mersenne numbers require that the exponent p be prime. A Mersenne prime is a Mersenne number that is prime.

In [6] the authors outlines 5 different ways to implement a fast modular multiplication algorithm:

- Modular Multiplication Using Montgomery's Technique
- Modular Multiplication in Residue Number Systems (RNS)
- Base Extension Using a Mixed Radix System (MRS)
- Base Extension Using the Chinese Remainder Theorem (CRT)
- Multiplication Modulo Generalized Mersenne Primes

There has been extensive work done in the area of modular multiplication Bajard et al.[2, 7, 8, 9, 10, 11, 12, 13] has done extensive work using different approaches to speed up the modular multiplication. In [14] Bajard et al. implemented RSA using RNS this was the first implementation of RSA in RNS which do not require any conversion, either from radix to RNS beforehand or RNS to radix afterward. The proposed algorithms have a high parallelization possibility when implemented in either software or hardware. Wu et al. [15] are focusing on how

to use the Chinese Remainder Theorem and Montgomery's Modular Multiplication Algorithm in the design of a circuits that are able of modular multiplication in parallel. Colin D. Walter [16] focuses on designing a chip that is able to perform modular multiplication in parallel.

2.3 Modular multiplication on a GPU

In [17] the authors were focusing on using Residue Number System(RNS) with standard values of N, that is taking N as a 1024-bit number. They also touch on the Chinese Remainder Theorem(CRT) with the statement that this theorem can accelerate the public and private key operations of RSA. They stop at this statement and do not provide any further proof or references. Instead they focus on using RNS, continuing to implement modular exponentiation with 5 different attempts to create a fast method of doing modular exponentiation. There was no CUDA on the market when they did their research so the used OpenGL to implement the programs to run their simulations. Moss et al. [17] came to a conclusion based on their experiments that there is a:

> overhead imposed by OpenGL and transfer of data to and from the accelerator(GPU).

This finding is supported by the recommendation B1 from Szerwinski et al.[6] that is listed below.

In [6] the authors provide a valuable list that shows some of the key areas in how to create a fast implementation. This list shows key points in making code run fast on a GPU:

Maximize use of available processing power

A1 Maximize independent parallelism in the algorithm to enable easy partitioning in threads and blocks.

A2 Keep resource usage low to allow concurrent execution of as many threads as possible, i.e., use only a small number of registers per thread and shared memory per block.

A3 Maximize arithmetic intensity, i.e., match the arithmetic to band- width ratio to the GPU design philosophy: GPUs spend their transistors on ALUs, not caches. Bearing this in mind allows to hide memory access latency by the use of independent computations (latency hiding). Examples include using arithmetic instructions with high throughput as well as re-computing values instead of saving them for later use.

A4 Avoid divergent threads in the same warp.

Maximize use of available memory bandwidth

B1 Avoid memory transfers between host and device by shifting more computations from the host to the GPU.

B2 Use shared memory instead of global memory for variables.

B3 Use constant or texture memory instead of global memory for constants.

B4 Coalesce global memory accesses, i.e., choose access patterns that allow to combine several accesses in the same warp to one, wider access.

B5 Avoid bank conflicts when utilizing shared memory, i.e., choose patterns that result in the access of different banks per warp.

B6 Match access patterns for constant and texture memory to the cache design.

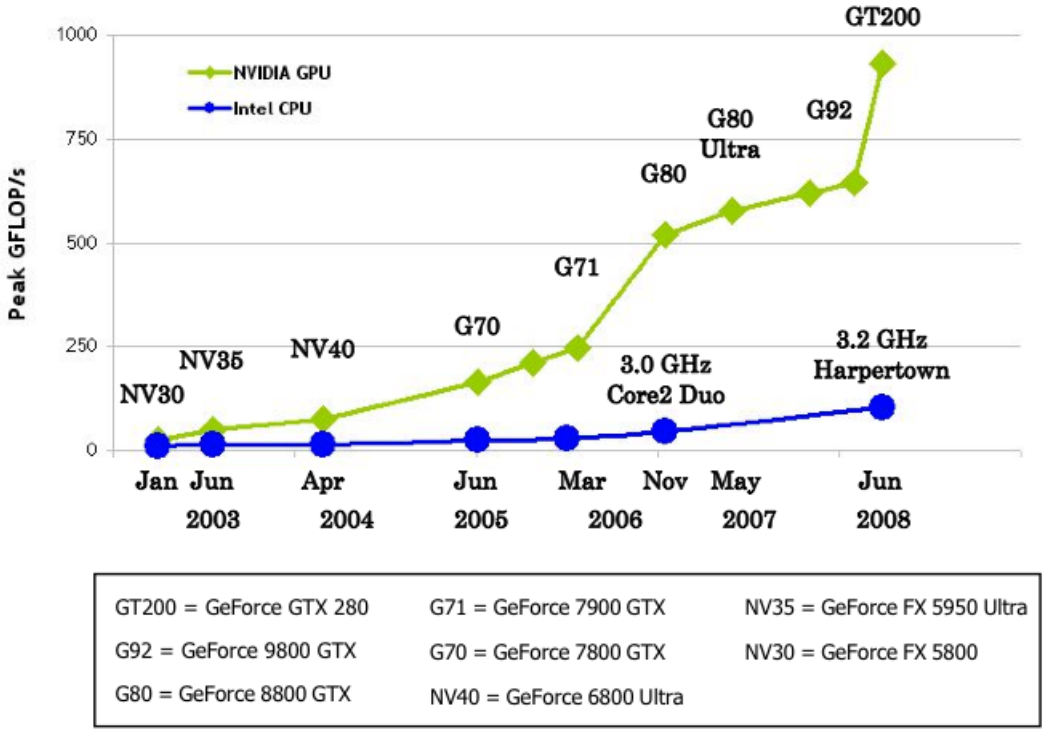

Figure 1: Performance of Nvidia GPU vs Intel CPU

Szerwinski et al.[6] selected to use a combination of CRT-RNS as their choice to produce a implementation using CUDA. The HW was a Nvidia 8800GTS GPU. The paper concludes with performance numbers showing that their work resulted in a fast implementation that out performed earlier work done by others. But they do not include any source code so we can test their findings. They do not say if they used single precision or double precision. Current Nvidia GPUs have double precision support, but it is 8-10 times slower than single precision. This slowdown in speed is documented in the Nvidia Tesla C1060 documentation[9], there is a similar slowdown for the other Nvidia products.

Harrison & Waldron [18] have implemented RSA using CUDA on a Nvidia 8800GTX GPU. They also touch different ways of doing modular multiplications, but they clearly state that they use single precision in their implementation. They choose to try RNS-CRT like Szerwinski and Guneysu[6] to show performance numbers that support their claim that the GPU implementation outperform a CPU implementation. Both these papers use the Nvidia G80 series of GPU. The G200 series of Nvidia GPU's has more cores than the G80 series and it would be interesting to test if the conclusions from their papers will be different on a GPU with more cores. More specifically, the GPU is especially well-suited to address problems that can be expressed as data-

[9] http://www.siliconmechanics.com/files/TeslaPSCDatasheet.pdf

parallel computations. So if the program is written in a way that can fully utilize the GPU and all its cores, then the more cores the GPU has the faster the program will run. This will not scale linearly but if the program is well written and the problem is well-suited to run in parallel, there will be an increase in performance with more cores.

From the papers published in this field there is an improvement of performance for every new paper, the main focus of the papers in the most recent years is to try to implement the modular multiplication in RSA using RNS with the combination of CRT-RNS. The chosen method of implementation does not vary much. The one thing that does change in the papers is the hardware used to implement the modular exponentiation. This can lead to some of the performance gain shown in the papers. As you can see from the Figure 1 on Page 8 there has been a huge development of GPU performance [10].

[10] http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

3 Theoretical Analysis

This chapter includes some of the possibilities of how to solve modular arthritic. The algorithms need to be studied to see if any of them is suited for use on a GPU, the factors to look out for is parallelism and if the algorithm is suited for SIMD(Single instruction, multiple data).

In the following sections we will give different ways to realize modular arithmetic efficiently, i.e. ways to compute addition, subtraction and multiplication modulo *m*, where *m* is called the modulus. RSA uses modulo N, where N is the product of two large primes p and q: $N = p * q$.

3.1 Multiplication Using Montgomery's Technique

In [19] C Kaya Koc et al. has described five different algorithms for doing multiplication using Montgomery. The algorithms showed in the paper is:

- Separated Operand Scanning(SOS).
- Coarsely Integrated Operand Scanning(CIOS).
- Finely Integrated Operand Scanning(FIOS).
- Finely Integrated Product Scanning(FIPS).
- Coarsely Integrated Hybrid Scanning(CIHS).

The Montgomery multiplication algorithm speeds up the modular multiplications and squaring required for exponentiation. It computes the Montgomery product

MonPro(a,b)$=abr^{-1}$ mod n

given a,b $<$ n and r such that the greatest common denominator (n,r) $=1$.

To describe the Montgomery reduction algorithm, we need an additional quantity, n', the integer with property rr^{-1} nn'$=1$. We can compute both integers r^{-1} and n' with the extended Euclidean algorithm. We compute MonPro(a,b) as follows: function ModPro(a,b)

t:$=$ab

u:$=$[t$+$(tn' mod r)n]/r

if u $>=$ n then return u-n, else return u

However, we did not take into account the space required to keep the input and output values a,b,n,n_0', and u

3.1.1 Separated Operand Scanning(SOS)

In this method they use four steps to compute the final values.

```
t=0
for i = 0 to s -1
        C:= 0
        for j = 0 to s-1
                (C,S):=t[i +j] +a[j]b[i]+C
```

```
                t[i + j] := S
        t[i + s]:=C
```

This first loop computes the product $a\ b$ where we initially assume t to be zero.

```
for i = 0 to s - 1
        C:=0
        m:=t[i]n'[0] mod W
        for j = 0 to s -1
                (C,S):=t[i + j] + mn[j] + C
                t[i + j]:= S
        ADD(t[i + s],C)
```

W = 2^w

The ADD function in this segment performs a carry propagation that adds C to the input array given by the first argument, starting from the first element(t[i+s]) and propagating it until it generates no further carry. The Add function is necessary for carry propagation up to the last word of t, which increases the size of t to 2s words and a single bit.

This second loop updates t to compute $t + m*n$.

```
for j = 0 to s
        u[j]:= t[j + s]
```

We then divide the computed value of t by r, by simply ignoring the lower s words of t.

```
B:=0
for i = 0 to s - 1
        (B,D):=u[i] - n[i] - B
        t[i]:=D
(B,D):=u[s] - B
t[s]:=D
if B:=0 then return t[0],t[1],...,t[s -1]
        else return u[0],u[1],...,u[s -1]
```

Finally we obtain the number u in $s+1$ words. The algorithm then performs the multi precision subtraction from step 3 of MonPro to reduce u if necessary. This is the same way for all five algorithms.

A brief inspection of the SOS method, shows that it requires $2s^2$ +s multiplications, 4s + 4s + 2 additions, $6s^2$+ 7s + 3 reads, and $2s^2$+ 6s + 2 writes. Furthermore, the SOS method requires a total of 2s + 2 words for temporary results, which store the (2s + 1) word array t and the one-word variable m. Figure 2 on Page 13 illustrates the SOS method for s = 4.

We define n'_0 as the inverse of the least significant word of n modulo 2^w that is, $n'_0 = n_0^{-1} (\mathrm{mod}\ 2^w)$. We can compute it using a very simple algorithm from Duss and Kaliki[20]. Furthermore, the reason for separating the product computation ab from the rest of the steps for computing u is that when $a = b$, we can optimize the Montgomery multiplication algorithm for squaring. This optimization allows us to skip almost half the single-precision multiplications, since $a_i a_j = a_j a_i$. To perform optimized Montgomery squaring, we replace the first part of the Montgomery multiplication algorithm with the following simple code:

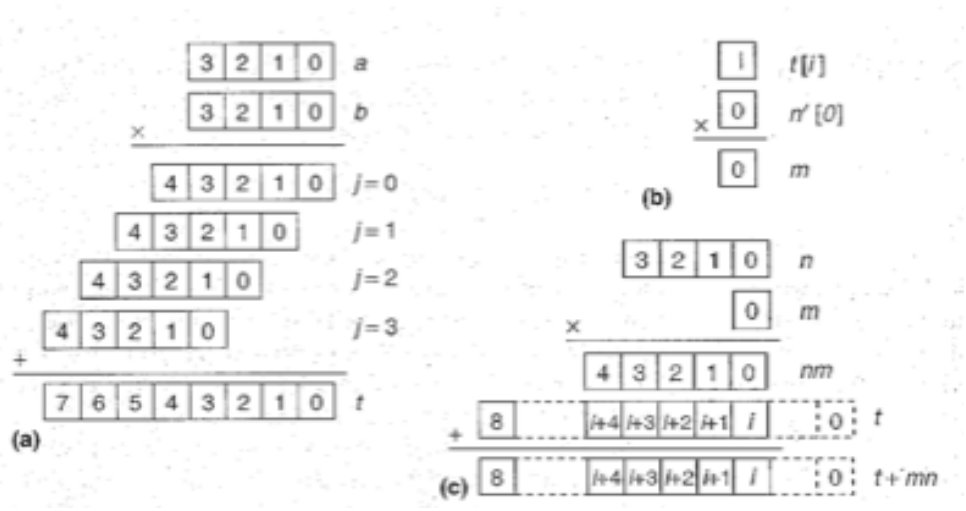

Figure 2: SOS method for s = 4. The algorithm first performs multiplication operation t=ab(a); it then multiplies n'_0, by each word of t to find m(b); it obtains the final result by adding the shifted nm to t(c).

```
for i= 0 to s - 1
        (C,S):= t[i + i] + a[i]a[i]
        for j = i + 1 to s - 1
                (C,S):= t[i+j] + 2a[j]a[i] + C
                t[i+j]:= S
        t[i+S] := c
```

One tricky part here is that value 2a[j]a[i] requires more than two words for storage. If the C value does not have an extra bit, one way to deal with this is to rewrite the loop to add the a[j]a[i] terms first,without multiplication by 2. The algorithm can then double the result and add in the a[i]a[i] terms.

3.1.2 Coarsely Integrated Operand Scanning(CIOS)

CIOS improves on SOS by integrating the multiplication and reduction steps. Specifically, instead of computing the entire product *ab* and then reducing, it alternates between iterations of the outer loops for multiplication and reduction. This is possible because the value of *m* in the *i*th iteration of the outer loop for reduction depends only on value *t[i]*, which is completely computed by the *i*th iteration of the outer loop for multiplication. This leads to the following algorithm:

```
for i = 0 to s - 1
        C:=0
        for j=0 to s - 1
                (C,S) := t[j] + a[j]b[i] + C
                t[j]:= S
        (C,S):= t[s] + C
        t[s]:= S
```

```
t[s + 1] := C
C := 0
m := t[0]n'[0] mod W
for j = 1 to s - 1
        (C,S) := t[j] +mn[j] + C
        t[j]:= S
(C,S):=t[s] + C
t[s]:= S
t[s +1]:=t[s + 1] + C
for j=0 to s
        t[j]:=[j + 1]
```

For a slight improvement, they integrate the shifting into the reduction as follows:

```
m := t[0]n'[0]mod W
(C,S) := t[0] + mn[o]
for j = 1 to s- 1
        (C,S) := t[j] + mn[j] + C
        t[j-1] := S
(C,S):= t[s] + C
t[s-1]:=S
t[s]:= t[s+1] + C
```

The CIOS method (with the slight improvement) requires $2s^2 + s$ multiplications, $4s^2 + 4s + 2$ additions, $6s^2 + 7s + 2$ reads, and $2s^2 + 5s + 1$ writes, including the final multi precision subtraction. It uses s+ 3 words of memory space, a significant improvement over the SOS method. In [20] S. R. Dusse et al. developed a cryptographic library for Motorola DSP56000 digital signal processor, the library contains three improvements of the CIOS method. In [21] M. McLoone et al. present a generic CIOS architecture that provides high speed Montgomery modular multiplication. This paper describes how to use CIOS with multiple operand sizes and achieve a high throughput. K. Zhao used in [22] CIOS as a comparison to a Karatsuba Montgomery multiplication. The Karatsuba [23] algorithm is an efficient procedure for multiplying large numbers that was discovered by Anatolii Alexeevitch Karatsuba in 1960 and published in 1962.

3.1.3 Finely Integrated Operand Scanning(FIOS)

FIOS integrates the two inner loops of the CIOS method into one by computing the multiplications and additions in the same loop. The algorithm computes multiplications $a_j b_i$ and mn_j in the same loop and then adds them to form the final t. In this case, the algorithm must compute t_0 before entering the loop, since m depends on this value. This corresponds to unrolling the first iteration of the loop for $i = 0$.

```
for i= 0 to s - 1
        (C,S):= t[0] + a[0]b[i]
        ADD(t[1],C)
        m:=Sn'[0] mod W
        (C,S):=S + mn[0]
```

The algorithm computes partial products of ab one by one for each value of i,then adds mn to the partial product. It then shifts this sum right one word, making t ready for the next i iteration.

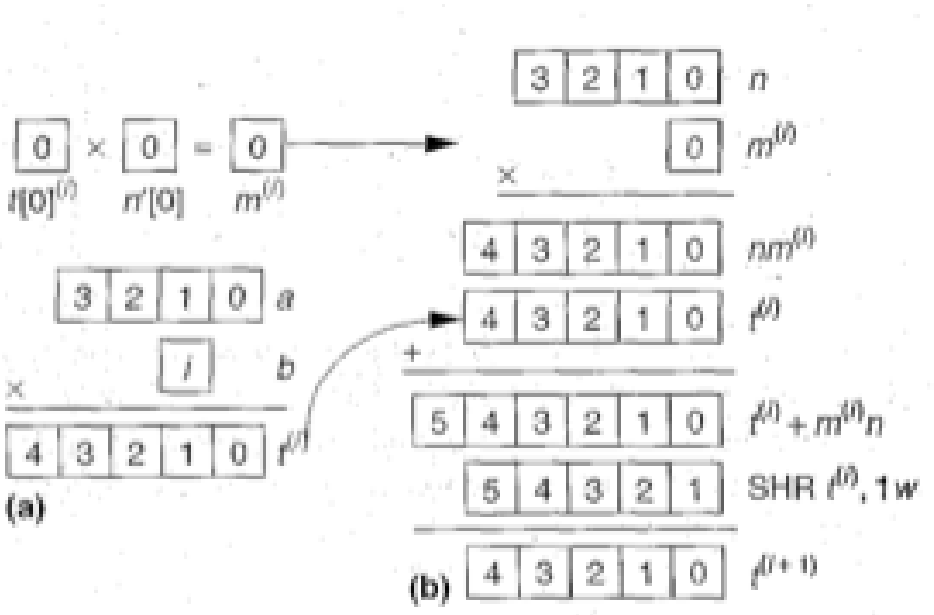

Figure 3: An iteration of the FIOS method. The computation of partial product $t^{(0)} = a * b, (a)$ enables computation of $m^{(i)}$ in that iteration. The algorithm then finds an intermediate result $t^{(j+1)}$ by adding $n * m^{(i)}$ to this partial product *(b)*.

```
for j= 1 to s - 1
        (C,S):= t[j] + a[j]b[i] + C
        ADD(t[j + 1],C)
        (C,S):=S +mn[j]
        t[j - 1]:=S
(C,S):=t[s] + C
t[s - 1]:=S
t[s]:=t[s + 1] + C
t[s + 1]:=0
```

This method differs from CIOS in that it has only one inner loop. Figure 3 on Page 15 illustrates the algorithm for s = 4. The FIOS method requires $2s^2 + s$ multiplications, $5s^2 + 3s + 2$ additions, $7s^2 + 5s + 2$ reads, and $3s^2 + 4s + 1$ writes, including the final multi precision subtraction. This is about 2^s more additions, writes and reads than the CIOS uses. Fan et al. [24] used FIOS on an programable multi core system to speed up montgomery multiplications using Very Long Instruction Word (VLIW) processor as a prototype.

3.1.4 Finely Integrated Product Scanning(FIPS)

FIPS interleaves computations *ab* and *mn*, but here both computations are in the product scanning form. The method keeps the values of *m* and *u* in the same s-word array *m*.

```
for i = 0 to s - 1
        for  j = 0 to i- 1
        (C,S) = 1[0] + a[j]b[i - j]
```

```
                ADD(t[1],C)
                (C,S):=S+m[j]n[i-j]
                t[0]:=S
                ADD(t[1],C)
        (C,S):=t[0] + a[i]b[0]
        ADD(t[1],C)
        m[i]:=Sn'[0] mod W
        (C,S):=S + m[i]n[0]
        ADD(t[1],C)
        t[0]:=t[1]
        t[1]:=t[2]
        t[2]:=0
```

This loop computes the *i*th word of m using n'_0, and then adds the least significant word of *mn* to *t*.

```
for i = s to 2s -1
        for j = i - s + 1 to s - 1
                (c,S):=t[0] + a[j]b[i - j]
                ADD(t[1],C)
                (C,S):=S + m[j]n[i - j]
                t[0]:=S
                ADD(t[1],C)
        m[i - s]:=t[0]
        t[0]:=t[1]
        t[1]:=t[2]
        t[2]:=0
```

This loop completes the computation by forming the final result *u'* word in the memory space of *m*. The FIPS method requires $2s^2 + s$ multiplications, $6s^2 + 2s + 2$ additions, $9s^2 + 8s + 2$ reads and $5s^2 + 8s + 1$ writes. The FIPS method requires s + 3 words of space.

3.1.5 Coarsely Integrated Hybrid Scanning(CIHS)

This method is a modification of the SOS method. It is called hybrid scanning method because it mixes the product-scanning method with the operand-scanning method.

```
for i = 0 to s - 1
        C:= 0
        for j = 0 to s - i - 1
                (C,S):= t[i +j] + a[j]b]i + C
                t[i + j]:=S
        (C,S):=t[s] + C
        t[s]:=S
        t[s +1 ]:= C
```

This first stage computes *(n - j)* words of the *j*th partial product of ab and adds them to *t*.

```
for i = 0 to s - 1
        m:=t[0]n'[0] mod W
        (C,S):= t[0] + mn[0]
```

Method	Multiplications	Additions	Reads	Writes	Space
SOS	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+3$	$2s^2+6s+2$	$2s+2$
CIOS	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+2$	$2s^2+5s+1$	$s+3$
FIOS	$2s^2+s$	$5s^2+3s+2$	$7s^2+5s+2$	$3s^2+4s+1$	$s+3$
FIPS	$2s^2+s$	$6s^2+2s+2$	$9s^2+8s+2$	$5s^2+8s+1$	$s+3$
CIHS	$2s^2+s$	$4s^2+4s+2$	$6.5s^2+6.5s+2$	$3s^2+5s+1$	$s+3$

Table 2: Multiplication Using Montgomery's Technique time and space requirements of the methods.

```
for j = 1 to s - 1
        (C,S):=t[j] + mn[j] +C
        t[j -1]:=S
(C,S):=t[s] + C
t[s -1]:=S
t[s]:=t[s + 1] + C
t[s + 1]=0
```

Then we interleave multiplication mn with addition $ab + mn$. It divides by r by shifting one word at a time within the i loop Since m is one word long and product $mn+C$ is two words long, total sum $t+mn$ needs at most s + 2 words.

```
for j = i + 1 to s - 1
        (C,S):=T[s - 1] + b[j]a[s - j + i]
        t[s -1]:=S
        (C,S):=t[s] + C
        t[s]:=S
        t[s + 1]:=C
```

This computes the $(s + i)$th word of ab.

CIHS requires $2s^2 + s$ multiplications. However, the number of additions decreases to $4s^2 + 4s + 2$. The number of reads is $6.5s^2 + 6.5s + 2$, and the number of writes is $3s^2 + 5s + 1$. As we mentioned earlier, this algorithm requires s + 3 words of temporary space.

3.1.6 Selecting the most promising method

From the sections above we can summarize the methods as showed in Table 2 on Page 17 the method that promises the fewest calculations is the CIOS method. CIOS is also a method that has a structure suitable for SIMD. This method is a good starting point to implement the experiments from.

3.2 Modular Arithmetic in Residue Number Systems

Advantages of residue number systems are easy parallelism, which are a result from the carry-free arithmetic and the possibility to implement multiplication without computation of sub-products. Section 2.2 introduces RNS and some of the notations used. In this section algorithms used in RNS are further analyzed to locate potential algorithms to use in an implementation. Following the argument given in [14], we assume that all currently available cryptographic protocols could be easily adopted to use RNS encoding directly and thus we disregard ways to convert between RNS and the commonly used radix-b representation. However, the advantages stated above do

not come for free: division, and as a result reduction modulo some arbitrary modulus M, is hard in residue number systems and sophisticated algorithms are needed. Almost all papers focus on hardware implementation and cannot be translated into a SIMD model without loosing at least some of their benefits.

3.2.1 Modular Multiplication Techniques

From Bajard et al. [14] we find RNS algorithms used to implement a full RSA implementation using RNS.

> Let us briefly recall the principles of Montgomery's techniques. Given two integers R;N such that gcd(R,N)=1, and $0 \leq x < RN$, the Montgomery reduction technique evaluates xR^{-1} mod N by computing the value $q < R$ such that $x + qN$ is a ultiple of R.

Algorithm 1 MM(a,b,N), RNS Montgomary Multiplication

Input: Two RNS bases $\beta = (m_1, \cdots, m_k)$, and $\beta' = (m_{k+1}, \cdots, m_{2k})$, such that M=$\prod_{i=1}^{k} m_i$, $M' = \prod_{i=1}^{k} m_{k+i}$ and gcd(M,M')=1; a redundant modulus m_r, $\gcd(m_r, m_i)=1$, $\forall i = 1 \cdots 2k$; a positive integer N represented in RNS in both bases and for m_r such that two positive integers a,b represented in RNS in both bases and for m_r, with *ab*<MN.
Output: a positive integer $\hat{r}$ represented in RNS in both bases and m_r, such that $\hat{r} \equiv abM^{-1} (\mathrm{mod} N)$, and $\hat{r}<(k+2)N$.

t=*ab* in $\beta \cup \beta' \cup m_r$

$q=t(-n^{-1})$ in β

[q in β] $\rightarrow$ [$\hat{q}$ in $\beta \cup m_r$]

$\hat{r} = (t + \hat{q}N)M^{-1}$ in $\beta' \cup m_r$

[$\hat{r}$ in β] $\leftarrow$ [$\hat{r}$ in β']

Algorithm 1 summarizes the computations of the RNS Montgomery multiplication. It computes the Montgomery product abM^{-1} mod N, where *a,b*, and *N* are represented in RNS in both bases β and β'. β and β' is defined as $\beta = m_1$, ldots, m_k and $\beta'=m_{k+1},\ldots,m_{2k}$.

Algorithm 2 First (approximated) RNS Base extension

Input: ($q_1, \cdots, q_k$), the RNS representation of *q* in the base β,
Output: ($\hat{q}_{k+1}, \cdots, \hat{q}_{2k}$) and $\hat{q}_r$, the RNS representation of $\hat{q}$ in $\beta' \cup m_r$.

$\sigma_i = q_i \mid M_i^{-1} \mid_{m_i} \mathrm{mod}\ m_i$

$t_0 = 0$

for i = $1 \cdots k$ do

$t_i=(t_{i-1} + \sigma_i |M_i|_{m_j}) \mathrm{mod}\ m_i$

$\hat{q}_j=t_k$

In [25] J. Yang et al. builds on P. Leong et al. algorithms in [26] to implement an algorithm for modular multiplication using RNS.

Algorithm 3 Second RNS Base extension

Input: $(\hat{r}_{k+1}, \cdots, \hat{r}_{2k})$ and $\hat{r}_r$, the RNS representation of $\hat{r}$ in $\beta' \cup m_r$
Output: $(\hat{r}_1, \cdots, \hat{r}_k)$, the RNS representation of $\hat{r}$ in β.

$\xi_j = \hat{r}_j |M_j^{t-1}|_{m_j} \bmod m_j$

$t_0 = 0$

for $j = 1 \cdots k$ do

$t_j = (t_{j-1} + \xi_{k+1} |M_j'|_{m_r}) \bmod m_r$

$\beta = |M^{t-1}|_{m_r} (t_k - |\hat{r}|_{m_r}) \bmod m_r$

$t_0 \leftarrow 0$

for $j = 1 \cdots k$ do

$t_j = (t_{j-1} + \xi_{k+j} |M_j'|_{m_i}) \bmod m_i$

$\hat{r}_i = (t_k - |\beta M'|_{m_i}) \bmod m_i$

First, we convert x, 2^w, 2^n, and 2^{n+j} to RNS representations with the selected moduli, where j = 0, 1, . . . , w. Note that the RNS representations of 2^w, 2^n, and 2^{n+j} can be precomputed and stored in a table such that the conversion cost can be reduced. The inputs of our algorithm, x and y, are expressed in RNS and the binary system representations, respectively. Besides, the output value z is expressed in binary system. The algorithm is shown as follows:

Input: (x: is expressed in RNS, y: is expressed in binary system)

Output: z: is expressed in RNS.

Set z = 0 expressed in RNS.

Divide the binary representation of y into dn/we parts and each part is denoted as yi. Convert each yi into an RNS representation.

Compute z = x ·RNS yi +RNS z ·RNS 2w, where i = dn/we.

Set j=0.

Compute h(z'). If h(z') = n + j, compute $z = z_{-\mathrm{RNS}\,2^{n+j}+\mathrm{RNS}\,d[j]}$

Compute h(z') again. If h(z') = n, let $z = z_{-\mathrm{RNS}\,2^{n+j}+\mathrm{RNS}\,d[0]}$

Compute j=j+1. If j≤w, go to Step 5.

Compute i=i-1. If i>0, go to Step 3.

Convert z into the binary system. If z ≥ N, compute z = (z + 2^n - N) mod 2^n.

In Step 2, y_i is a partial binary representation of y. For example, if the binary representation of y is $(110011)_2$ and w = 3, we have $y_2 = (110)_2$ and $y_1 = (011)_2$. We must convert each y_i into its corresponding RNS representation with the selected moduli. However, the RNS representation of each y_i can be also precomputed and stored. In the simplified case (i.e., w = 1), it requires no cost to convert y_i's into RNS representations since all RNS digits of y_i can be directly set to be 0 or 1 (i.e., y_i = (0,0,...,0) or (1,1,...,1).

Based on the work by Posch et al. [27], Kawamura et al. [28] created new base extension algorithm, which plays an important role in an RNS Montgomery multiplication. This new extension in turn got slightly changed by Bajard et al. [29, 11] later. The latter technique is given in Algorithm 4 . While Posch et al. [27] needed additional correction steps because of the underlying primitives, Kawamura uses accurate algorithms and adds some restrictions regarding the dynamic range needed to compute consecutive multiplications.

Algorithm 4 Modular Multiplication Algorithm for Residue Number Systems

Require: A modulus M, two RNS bases A and B, composed of n distinct moduli m_i each, i.e. A = $(m_0, m_1, \ldots, m_{n-1})$ and B = $(m_n, m_{n+1}, ..., m_{2n-1})$, with respective dynamic ranges A = $\prod_{i=0}^{n-1} m_i$ and B = $\prod_{i=n}^{2n-1} m_i$, gcd(A, B) = gcd(A,M) = 1 and B > A > 4M. Two factors X and Y, 0 leq X,Y < 2M, encoded in both bases and in Montgomery form, i.e. $\langle X \rangle_{A \cup B}$ and $\langle Y \rangle_{A \cup B}$, X = xA (mod M) and Y = yA (mod M).

Ensure: The product C = XYA^{-1} (mod M), 0 leq C < 2M, in both bases and Montgomery form, i.e. $\langle C \rangle_{A \cup B}$ and C = xyA (mod M).

1: $\langle u \rangle_{A \cup B} \leftarrow \langle X \rangle_{A \cup B} * \langle Y \rangle_{A \cup B}$

2: $\langle f \rangle_A \leftarrow \langle u \rangle_A * \langle -M^{-1} \rangle_A$

3: $\langle f \rangle_{A \cup B} \leftarrow BaseExtend(\langle f \rangle_A)$

4: $\langle u \rangle_B \leftarrow \langle u \rangle_B + \langle f \rangle_B . \langle M \rangle_B$ $\langle u \rangle_A = 0$ by construction

5: $\langle w \rangle_B \leftarrow \langle v \rangle_B * \langle A^{-1} \rangle_B$

6: $\langle w \rangle_{A \cup B} \leftarrow BaseExtend(\langle w \rangle_B)$

7: **return** $\langle w \rangle_{A \cup B}$

Freking et al. [30] created a sequential RNS Montgomery Modular Multiplication algorithm integrating mixed-radix conversion to speed up the the base extension used in [27].

Computations in residue number systems yield the advantage of being inherently parallel. According to Algorithm 4 all steps are computed in one base only, except for the first multiplication. Thus, the optimal mapping of computations to threads is as follows: each thread determines values for one modulus in the two bases. As a result, we have coarse-grained (different exponentiations) and fine grained parallelism (base size), fulfilling Criterion A1 ref. 2.3. We call n' the number of residues that can be computed in parallel, i.e., the number of threads per encryption. The base extension by Shenoy et al. [31] needs a redundant residue starting from the first base extension to be able to compute the second base extension.

3.2.2 Base Extension Using a Mixed Radix System

Assume we have the RNS representation of some integer x in base
$A = (m_0, m_1 \ldots, m_{n-1})$, $\langle X \rangle_A = \langle |X|_{m_0}, |X|_{m_1}, \ldots, |X|_{m_{n-1}} \rangle_A$
and want to compute the representation in another base
$B = (m_n, m_{n+1} \ldots, m_{2n-1})$, $\langle X \rangle_B = \langle |X|_{m_n}, |X|_{m_{n+1}}, \ldots, |X|_{m_{2n-1}} \rangle_B$
One possibility to achieve this efficiently is to derive the mixed radix system (MRS) representation of x first and compute the residues modulo the target base afterwards.

Conversion to Mixed Radix System

Szab and Tanaka in their book about residue arithmetic [32] describe base extension algorithms. Assume we define a mixed residue system using the base $(m_0, m_1, ..., m_{n-2})$, i.e. directly matching the source RNS $(m_0, m_1, ..., m_{n-2}, m_{n-1})$. Let the mixed radix representation of x be $\langle\langle X_{n-1}, X_{n-2}, \ldots, X_0\rangle\rangle$. Then the following equation holds:

$$x = x_{n-1}m_{n-2}m_{n-3}\ldots m_2m_1m_0 + x_{n-2}m_{n-3}m_{n-4}\ldots m_2m_1m_0 + \ldots + x_3m_2m_1m_0 + x_2m_1m_0 + x_0$$

Now further assume that $x_i < m_i$ for all i. Observe that $x_0 = |x|_{m_0}$, i.e. the first digit of the MRS representation equals the first residue in the RNS representation of x. By subtracting this digit from the second residue of the RNS we obtain

$|x - x_0|_{m_1} = |x_1m_0|_{m_1}$.
Thus, we can compute the next MRS digit x_1 by multiplying $|x - x_0|_{m_1}$ with the multiplicative inverse of m_0 modulo m_1.

$$||x - x_0|_{m_1} * |m_0^{-1}|_{m_1}|_{m_1} = |x_1m_0 * m_0^{-1}|_{m_1} = |x_1|_{m_1} = x_1$$

Each following MRS digit can be attained using a similar approach, for example the next digit x_2 can be derived by using all the previous steps for the third RNS digit $|x|_{m_2}$, subtracting x_1 and multiplying with $|m_1^{-1}|_{m_2}$:

$$\begin{aligned}
&||x|_{m_2} - |x_0|_{m_2}|_{m_2} = |x_2m_1m_0 + x_1m_0|_{m_2}\\
&||x - x_0|_{m_2} * |m_0^{-1}|_{m_2}|_{m_2} = |x_2m_1 + x_1|_{m_2}\\
&||(x - x_0)m_0^{-1}|_{m_2} - |x_1|_{m_2}|_{m_2} = |x_2m_1|_{m_2}\\
&||(x - x_0)m_0^{-1} - x_1|_{m_2} * m_1^{-1}|_{m_2}|_{m_2} = |x_2|_{m_2} = x_2
\end{aligned}$$

Thus, the following recursive description can be used to compute all MRS digits:

$$\begin{aligned}
&|\varphi_0|_{m_i} = |x|_{m_i}\\
&|\varphi_{k+1}|_{m_i} = |(|\varphi_k|_{m_i} - |x_k|_{m_i}) * |m_k^{-1}|_{m_i}|_{m_i} = |(|\varphi_k|_{m_i} - |x_k|_{m_i}) - A_{i,k}^{inv}|_{m_i}\\
&x_0 = |x|_{m_0}\\
&x_{k+1} = |\varphi_{k+1}|_{m_{k+1}}
\end{aligned}$$

where k = 0,1,...,n - 2 and $A_{i,k}^{inv} = |m_k^{-1}|_{m_i}$ is a matrix of pre-computed constants. Note the values needed to compute x_{k+1}: it depends on $\varphi_{k+1}, \varphi_k, \ldots, \varphi_0$, but only the RNS digit with index k + 1. The more MRS digits get computed the less RNS digits have to be taken into account: at the start all RNS digits except the first one are involved in subtraction of the MRS digit x_0 and multiplication by the inverse of m_0 modulo the respective RNS moduli, while the next k-iteration only needs the lately computed MRS digit x_1 and all RNS digits except the first two. In other words, only the residues of $\varphi_{k+1}, \ldots, \varphi_0$ with indices $i \geq k$ have to be computed in each iteration. This however is quite non-optimal for our SIMD model when using *n* tasks, each handling one of the *n* residues of φ_{k+1}.

Conversion to RNS Using the Parallel Method

A common way of converting the MRS representation to RNS is the way described by Szab and Tanaka[32]. From the Szab and Tanaka equations which holds for numbers encoded in the mixed

radix system from above. When reducing this equation modulo one of the target moduli, say m_i, we have:

$$|x|_{m_i} = |x_{n-1}m_{n-2}\dots m_0 + x_{n-2}m_{n-3}\dots m_0 + \dots + x_0|_{m_i}$$

$$= |x_{n-1}m_{n-2}\dots m_0|_{m_i} + |x_{n-2}m_{n-3}\dots m_0|_{m_i} + \dots + |x_0|_{m_i}$$

$$= ||x_{n-1}|_{m_i}|m_{n-2}\dots m_0|_{m_i}|_{m_i} + ||x_{n-2}|_{m_i}|m_{n-3}\dots m_0|_{m_i}|_{m_i} + \dots + |x_0|_{m_i}$$

$$\left|\sum_{k=0}^{n-1}\left(|x_k|_{m_i}\left|\prod_{\mu=0}^{k-1}m_\mu\right|_{m_i}\right)\right|_{m_i} = \left|\sum_{k=0}^{n-1}\left(|x_k|_{m_k}A_{i,k}^{prod}\right)\right|_{m_i}$$

While this technique is highly parallelizable and fits very well to our SIMD architecture, it has the disadvantage of using an n x n matrix of pre-computed values that need storage space and require n^2 look-ups for the full base extension from A to B.

Conversion to RNS Using the Serial Method

Bajard et al. describe a slight modification to the previous algorithm [33, 2]. This modification computes the sum involved in each conversion serially, thus trading the n^2 look-ups in the matrix $A_{i,k}^{prod}$ for the inability to compute the sum in parallel. This means no particular disadvantage in our case, as we want to use *n* threads, one for each target modulus, only.

Bajard et al. present further techniques to reduce the number of look-ups needed and to shorten the relevant operands [33, 2].

3.2.3 Base Extension Using the Chinese Remainder Theorem

The Chinese remainder theorem is a result about congruences in number theory and it's generalizations in abstract algebra. This is an efficient way to compute all residues modulo the target base, the instruction flow is highly uniform and fits our SIMD architecture well, i.e. we can use *n* threads to compute the *n* residues of *x* in the target base in parallel. Harrison et al. [18] used CRT in their paper to speed up their implementation.

Shenoy and Kumaresan's Technique

In [31] Shenoy et al. presents a technique to extend the base of a residue number system (RNS) based on the Chinese remainder theorem (CRT) and to use the redundant modulus. The technique obtains the residue(s) of a given number in the extended moduli without resorting to the traditional mixed radix conversion (MRC) algorithm. The base extension can be achieved in $\log_2 n$ table lookup cycles where n is the number of moduli in the RNS. Shenoy et al. demonstrates the superiority of the technique compared in terms of latency and hardware requirements to the traditional Szabo et al. [32] method.

Posch and Posch's Technique

Posch and Posch describe a base extension algorithm that approximates using floating or fixed point arithmetic [34]. This can be used in an approximate Chinese Remainder Reconstruction, and is found to be a fast method. More precisely, they compute a weighted sum of all x_k with the weights $w_k < 1$ being rational constants that only depend on the moduli m_k:

$$\alpha' = \left|\sum_{k=0}^{n-1} x_k w_k\right|, \text{ with } w_k = \frac{\left|\hat{A}_k^{-1}\right|_{m_k}}{m_k} < 1$$

Note that α is not principally upper-bounded by n.

In an approximate Chinese Remainder Reconstruction each term is calculated to a limited floating point precision reducing both the amount of work done and the accuracy of the reconstruction. Powell et al. [35] use this technique in their library for parallel modular arithmetic. Among other things needed to implement RSA efficiently, Pearson [36] fundamentally describes the same algorithm. However, his paper is not giving any reasoning or bounds involved and states that, given a particular set of parameters, the algorithm outputs the correct value "almost always". The author claims that exceptional cases can be easily detected and corrected without stating how this can be achieved.

Kawamura et al.'s Technique

Kawamura et al. [28] provide a new RNS base extension algorithm. Cox-Rower Architecture described in this paper is a hardware suitable for the RNS Montgomery multiplication. In this architecture, a base extension algorithm is executed in parallel by plural Rower units controlled by a Cox unit. Each Rower unit is a single-precision modular multiplier-and-accumulator, whereas Cox unit is typically a 7 bit adder. Transformation to and from radix representation is also covered in the paper.

Bajard et al.'s Technique

Finally, Bajard et al. [8] follow the most radical approach possible: they simply compute no α at all . The resulting value will still be equivalent to the exact value modulo M, but may include an offset of up to (n - 1)M. This is possible due to the fact that α is strictly bounded by n. However, this technique needs additional measures of precaution in the multiplication algorithm, which predominantly condense in the higher dynamic ranges needed.

4 Computing on the GPU

This chapter contains information on how to execute code on the GPU. There are many different ways of doing this. In [37] Cook et al. uses the the OpenGL API to utilize the power of the GPU to perform AES operations, S. Fleissner [38] implements Montgomery multiplications using OpenGL and OpenGL Shading Language. Other API that can be used includes DirectX Direct Compute[1] and ATI Close To Metal[2]. This chapter will be limited to the use of CUDA and OpenCL. CUDA and OpenCL is the newest way of doing calculations using the GPU and is gaining popularity in the marked. AMD/ATI has closed their Close to Metal project and is fully committed to using OpenCL as the way to use AMD/ATI in general computing using the GPU. CUDA is only limited described and the inclusion of CUDA is only to show how similar CUDA is to OpenCL. OpenCL is to be described in more detail as this is the programing API used to implement the modular arthritic operations in this project.

4.1 CUDA

CUDA (an acronym for Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA graphics processing units or GPUs that is accessible to software developers through industry standard programming languages. Programmers use 'C for CUDA' (C with NVIDIA extensions), compiled through a PathScale[3] Open64 C compiler, to code algorithms for execution on the GPU. CUDA architecture supports a range of computational interfaces including OpenCL and DirectCompute. Third party wrappers are also available for Python, Fortran, Java and Matlab.

A CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results. A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-Block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in Global Memory space after kernel-wide global synchronization. Figure 4 on Page 26 show the CUDA hierarchy.

[1] http://msdn.microsoft.com/en-us/directx/default.aspx
[2] http://developer.amd.com/media/gpu_assets/Hensley-Close_to_the_Metal(Siggraph07_GPGPUCourse).pdf
[3] http://www.pathscale.com/

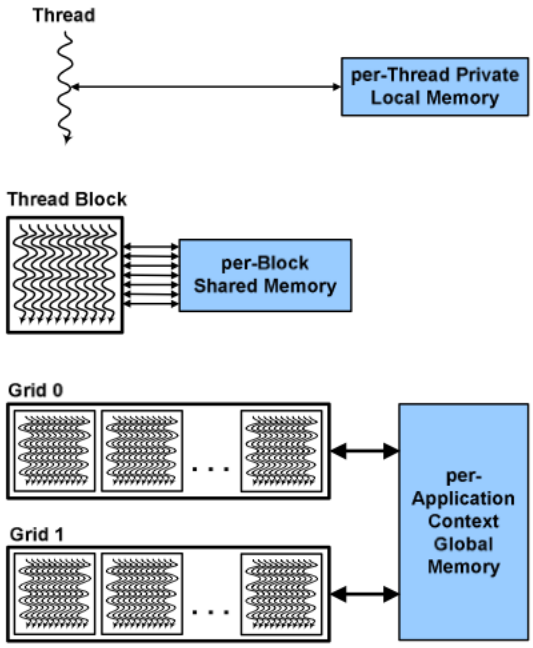

Figure 4: CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

CUDA's hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 threads called a warp. While programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

The latest video drivers from Nvidia all contain the necessary CUDA components. CUDA works with all NVIDIA GPUs from the G8X series onwards, including GeForce, Quadro and the Tesla line. Nvidia states that programs developed for the GeForce 8 series will also work without modification on all future Nvidia video cards, due to binary compatibility. CUDA gives developers access to the native instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest NVIDIA GPUs effectively become open architectures like CPUs. Unlike CPUs however, GPUs have a parallel "many-core" architecture, each core capable of running thousands of threads simultaneously - if an application is suited to this kind of an architecture, the GPU can offer large performance benefits.

CUDA comes with a software environment that allows developers to use C as a high-level programming language. As illustrated by Figure 5 on Page 27, other languages or application programming interfaces will be supported, such as FORTRAN, C++, OpenCL, and DirectX Compute.

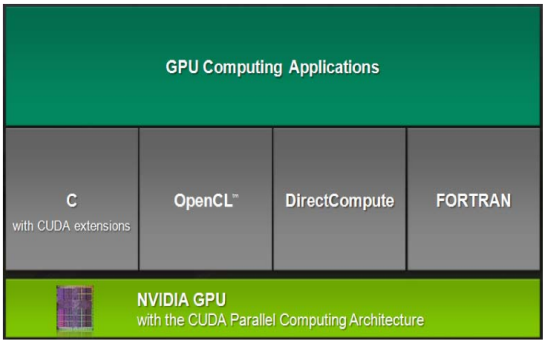

Figure 5: CUDA is Designed to Support Various Languages or Application Programming Interfaces

There have been many others that have done modular arithmetic using CUDA as the API to use the GPU as the processing unit some examples can be found in [39, 18, 40, 6, 41].

4.2 OpenCL

4.2.1 Introduction

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.

"OpenCL supports a wide range of applications, ranging from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient, close-to-the-metal programming interface, OpenCL will form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications"[4].

OpenCL consists of an API for coordinating parallel computation across heterogeneous processors; and a cross-platform programming language with a well-specified computation environment. The OpenCL standard:

- Supports both data- and task-based parallel programming models
- Utilizes a subset of ISO C99 with extensions for parallelism
- Defines consistent numerical requirements based on IEEE 754
- Defines a configuration profile for handheld and embedded devices
- Efficiently interoperates with OpenGL, OpenGL ES, and other graphics APIs

The specification is divided into a core specification that any OpenCL compliant implementation must support; a handheld/embedded profile which relaxes the OpenCL compliance requirements for handheld and embedded devices; and a set of optional extensions that are likely to move into the core specification in later revisions of the OpenCL specification.

OpenCL is being created by the Khronos Group[5] with the participation of many industry-

[4] http://www.khronos.org/opencl/
[5] http://www.khronos.org/opencl/

leading companies and institutions.

4.2.2 The Anatomy of OpenCL 1.0

The OpenCL 1.0 specification is made up of three main parts: the language specification, platform layer API and runtime API.

The language specification describes the syntax and programming interface for writing compute kernels that run on supported accelerators, such as GPUs and multi-core CPUs. The language used is based on a subset of ISO C99. C was chosen as the basis for the first OpenCL compute kernel language due to its prevalence and familiarity in the developer community. To foster consistent results across different platforms, a well-defined IEEE 754 numerical accuracy is defined for all floating point operations along with a rich set of built-in functions. The developer has the option of pre-compiling their OpenCL compute kernel or letting the OpenCL runtime compile their kernels on demand.

The platform layer API gives the developer access to routines that query for the number and types of devices in the system. The developer can then select and initialize the necessary compute devices to properly run their work load. It is at this layer that compute contexts and work-queues for job submission and data transfer requests are created.

Finally, the runtime API allows the developer to queue up compute kernels for execution and is responsible for managing the compute and memory resources in the OpenCL system.

Table 3 on Page 28, is a concise representation of the various parts of OpenCL.

OpenCL C
C-based cross-platform programming interface
Subset of ISO C99 with language extensions - familiar to developers
Well-defined numerical accuracy - IEEE 754 rounding behavior with defined maximum error
Online or offline compilation and build of compute kernel executables
Includes a rich set of built-in functions
OpenCL API
A hardware abstraction layer over diverse computational resources
Query, select and initialize compute devices Create compute contexts and work-queues
OpenCL Runtime
Execute compute kernels
Manage scheduling, compute, and memory resources

Table 3: Main parts of OpenCL

OpenCL C

OpenCL defines OpenCL C, which is a variant of the familiar C99 language optimized for GPU programming. It incorporates changes necessary to adapt the C programming language for use with GPUs and to support parallel processing. OpenCL C includes comprehensive support for vector types to streamline data flow and increase efficiency. Well-defined numerical precision requirements (based on IEEE 754-2008) are specified to provide mathematical consistency across the GPU hardware of different vendors. Developers use OpenCL C to rewrite just the

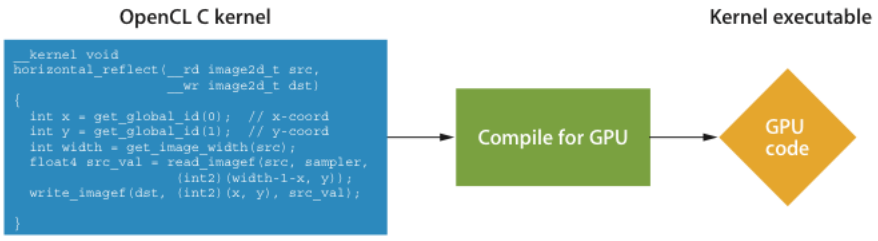

Figure 6: Kernel executing as defined by OpenCL 1.0

performance- or data-intensive routines in their applications. During the rewrite, the routine is factored down to its most elementary state: a series of discrete operations that describe the computations that can be performed in parallel over a data set. The resulting code, which is similar to a traditional C function, is called an OpenCL kernel. Unlike traditional C code, OpenCL kernels are incorporated into the application in an uncompiled state. They are compiled on the fly and optimized for the user's hardware before being sent to the GPU for processing. Figure 6 on Page 29 show how this work. There are some important limitations in the OpenCL C language that is further explained in section 4.2.5 on Page 31.

OpenCL API

The OpenCL API provides functions that allow an application to manage parallel computing tasks. It enumerates the OpenCL-capable hardware in a system, sets up the sharing of data structures between the application and OpenCL, controls the compilation and submission of kernels to the GPU, and has a rich set of functions that manage queuing and synchronization.

OpenCL Runtime

The OpenCL runtime executes tasks submitted by the application via the OpenCL API. The runtime efficiently transfers data between main memory and the dedicated VRAM used by the GPU, and directs execution of the kernels on the GPU hardware. During execution, the OpenCL runtime manages the in-order or out-of-order dependencies between the kernels, and utilizes the GPU's processing elements in the most efficient manner.

4.2.3 OpenCL Execution Model

The AMD/ATI web site has an article containing information on the OpenCL Execution Model[6]
In short the article explains the execution model like this:

> OpenCL has a flexible execution model that incorporates both task and data parallelism. Data movements between the host and compute devices, as well as OpenCL tasks, are coordinated via command queues. Command queues provide a general way of specifying relationships between tasks, ensuring that tasks are executed in an order that satisfies the natural dependences in the computation. The OpenCL runtime is free to execute tasks in parallel if their dependencies are satisfied, which provides a general-purpose task parallel execution model. Tasks themselves can be comprised of data-parallel kernels, which apply a single function over a range of data elements, in parallel, allowing only restricted synchronization and communication during the execution of a kernel.

[6]http://developer.amd.com/documentation/articles/pages/opencl-and-the-ati-stream-v2.0-beta.aspx

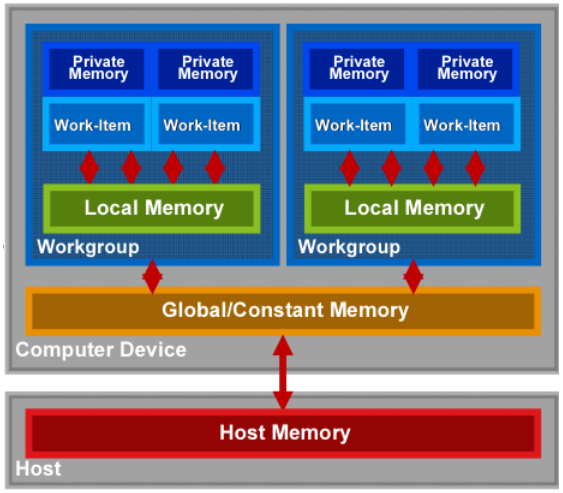

Figure 7: Memory hierarchy as defined by OpenCL 1.0

Parts of this article is copied to Appendix A on Page 53 to further explain the execution model.

4.2.4 The Memory Model

OpenCL defines a multi-level memory model with memory ranging from private memory visible only to the individual compute units in the device to global memory that is visible to all compute units on the device. Depending on the actual memory subsystem, different memory spaces are allowed to be collapsed together.

OpenCL 1.0 defines 4 memory spaces: private, local, constant and global. Figure 7 on Page 30, shows a diagram of the memory hierarchy defined by OpenCL.

Private memory is memory that can only be used by a single compute unit. This is similar to registers in a single compute unit or a single CPU core.

Local memory is memory that can be used by the work-items in a work-group.

Constant memory is memory that can be used to store constant data for read-only access by all of the compute units in the device during the execution of a kernel. The host processor is responsible for allocating and initializing the memory objects that reside in this memory space.

Finally, global memory is memory that can be used by all the compute units on the device. Each compute device has a global memory space, which is the largest memory space available to the device, and typically resides in off-chip DRAM. There is also a read-only, limited-size constant memory space, which allows for efficient reuse of read-only parameters in a computation. Each compute unit on the device has a local memory, which is typically on the processor die, and therefore has much higher bandwidth and lower latency than global memory. Local memory can be read and written by any work-item in a work-group, and thus allows for local communication between work-groups. Additionally, attached to each processing element is a private memory, which is typically not used directly by programmers, but is used to hold data for each work-item that does not fit in the processing element's registers.

As OpenCL has a relaxed consistency model, different work-items may see a different view

of global memory as the computation progresses. Within a work-item, reads and writes to all memory spaces are consistently ordered, but between work-items, synchronization is necessary in order to ensure consistency. This relaxed consistency model is an important part of OpenCL's efforts to provide parallel scalability: parallel programs that rely on strong memory consistency for synchronization and communication usually fail to execute in parallel, because memory ordering requirements force a serialization of such programs during execution, hindering scalability. Requiring explicit synchronization and communication between work-items encourages programmers to write scalable code, avoiding the trap often seen in parallel programming where code looks parallel, but ends up executing in serial due to frequent and implicit synchronization induced by reliance on a strict memory ordering model.

Additionally, OpenCL views the global memory space of each compute device as private and separate from host memory. Moving data between compute devices and the host requires the programmer to manually manage communication between the host and the compute devices. This is done through the use of explicit memory reads and writes between devices.

4.2.5 Limitations in the OpenCL C language

When writing kernels there are some important limitations on what is allowed in the code. The limitations can be found on the Khronos[7] web page. The list below is a collection of some of the limitations found on the web page:

- The use of pointers is somewhat restricted. The following rules apply:
  - Arguments to __kernel functions declared in a program that are pointers must be declared with the __global, __constant or __local qualifier.
  - A pointer declared with the __constant, __local, or __global qualifier can only be assigned to a pointer declared with the __constant, __local, or __global qualifier respectively.
  - Pointers to functions are not allowed.
  - Arguments to __kernel functions in a program cannot be declared as a pointer to a pointer(s). Variables inside a function or arguments to non __kernel functions in a program can be declared as a pointer to a pointer(s).
- Bit-fields are currently not supported.
- Variable length arrays and structures with flexible (or unsized) arrays are not supported.
- Variadic macros and functions are not supported.
- The C99 standard headers assert.h, ctype.h, complex.h, errno.h, fenv.h, float.h, inttypes.h, limits.h, locale.h, setjmp.h, signal.h, stdarg.h, stdio.h, stdlib.h, string.h, tgmath.h, time.h, wchar.h, and wctype.h are not available and cannot be included by a program.
- The extern, static, auto, and register storage-class specifiers are not supported.
- Predefined identifiers are not supported.
- Recursion is not supported.

[7] http://www.khronos.org/opencl/

- The function using the __kernel qualifier can only have return type void in the source code.

4.2.6 Performance considerations

In [6] the authors provide a valuable list that shows some of the key areas in how to create a fast implementation. This list shows key points in making code run fast on a GPU:

Maximize use of available processing power

A1 Maximize independent parallelism in the algorithm to enable easy partitioning in threads and blocks.

A2 Keep resource usage low to allow concurrent execution of as many threads as possible, i.e., use only a small number of registers per thread and shared memory per block.

A3 Maximize arithmetic intensity, i.e., match the arithmetic to band- width ratio to the GPU design philosophy: GPUs spend their transistors on ALUs, not caches. Bearing this in mind allows to hide memory access latency by the use of independent computations (latency hiding). Examples include using arithmetic instructions with high throughput as well as re-computing values instead of saving them for later use.

A4 Avoid divergent threads in the same warp.

Maximize use of available memory bandwidth

B1 Avoid memory transfers between host and device by shifting more computations from the host to the GPU.

B2 Use shared memory instead of global memory for variables.

B3 Use constant or texture memory instead of global memory for constants.

B4 Coalesce global memory accesses, i.e., choose access patterns that allow to combine several accesses in the same warp to one, wider access.

B5 Avoid bank conflicts when utilizing shared memory, i.e., choose patterns that result in the access of different banks per warp.

B6 Match access patterns for constant and texture memory to the cache design.

To achieve these goals on a GPU we need to consider the memory access patterns in the application. Nvidia has published best practice guide for OpenCL[8]. From this guide there are some areas that is needed to be looked at to be successful in implementing fast modular arithmetic on a GPU.

- **Memory Optimizations:** Correct memory management is one of the most effective means of improving performance. This chapter explores the different kinds of memory available to OpenCL applications, and it explains in detail how memory is handled behind the scenes.
- **NDRanges Optimizations:** How to make sure your OpenCL application is exploiting all the available resources on the GPU.
- **Instruction Optimizations:** Certain operations run faster than others. Using faster operations and avoiding slower ones often confers remarkable benefits.
- **Control Flow:** Carelessly designed control flow can force parallel code into serial execution; whereas thoughtfully designed control flow can help the hardware perform the maximum amount of work per clock cycle.

[8] http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

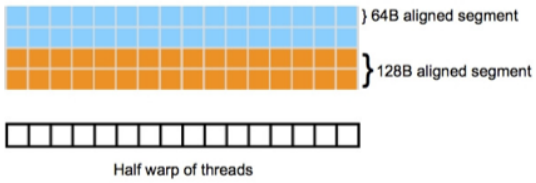

Figure 8: Linear memory segments and threads in a half warp

Memory Optimizations

Perhaps the single most important performance consideration in programming for the CUDA architecture is coalescing global memory accesses. Global memory loads and stores by threads of a half warp (16 threads) are coalesced by the device in as few as one transaction (or two transactions in the case of 128-bit words) when certain access requirements are met. To understand these access requirements, global memory should be viewed in terms of aligned segments of 16 and 32 words. Figure 8 on Page 33 helps explain coalescing of a half warp of 32-bit words, such as floats. It shows global memory as rows of 64-byte aligned segments (16 floats). Two rows of the same color represent a 128-byte aligned segment. A half warp of threads that accesses the global memory is indicated at the bottom of the figure.

- **High Priority:** Ensure global memory accesses are coalesced whenever possible.
- **Medium Priority:** Use shared memory to avoid redundant transfers from global
- **Low Priority:** For kernels with long argument lists, place some arguments into constant memory to save shared memory.

NDRange Optimizations

Thread instructions are executed sequentially in OpenCL, and, as a result, executing other warps when one warp is paused or stalled is the only way to hide latencies and keep the hardware busy. Some metric related to the number of active warps on a multiprocessor is therefore important in determining how effectively the hardware is kept busy. This metric is occupancy. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that are actively in use. Higher occupancy does not always equate to higher performance—there is a point above which additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.

Instruction Optimizations

Awareness of how instructions are executed often permits low-level optimizations that can be useful, especially in code that is run frequently (the so-called hot spot in a program). Best practices suggest that this optimization be performed after all higher-level optimizations have been completed. Single-precision floats provide the best performance and their use is highly encouraged. The throughput of single-precision floating-point add, multiply, and multiply-add is 8 operations per clock cycle. Integer division and modulo operations are particularly costly and should be avoided or replaced with bitwise operations whenever possible.

- **High Priority:** Minimize the use of global memory. Prefer shared memory access where possible.

Control Flow

Any flow control instruction (if, switch, do, for, while) can significantly affect the instruction throughput by causing threads of the same warp to diverge; that is, to follow different execution paths. If this happens, the different execution paths must be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path. To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps.

- **High Priority:** Avoid different execution paths within the same warp. where possible.

4.2.7 Installing OpenCL

In order to use OpenCL on a system we first need to install an OpenCL enabled driver. Both Nvidia and AMD/ATI have delivered drivers that enable developers to utilize OpenCL in programs running on Windows, Linux and OS X. OS X support is delivered by Apple and is included in OS X 10.6 Snow Leopard[9] with support for the following GPUs:

- Nvidia
  - NVIDIA GeForce 9400M
  - GeForce 9600M GT
  - GeForce 8600M GT
  - GeForce GT 120
  - GeForce GT 130
  - GeForce GTX 285
  - GeForce 8800 GT
  - GeForce 8800 GS
  - Quadro FX 4800
  - Quadro FX5600
- ATI
  - ATI Radeon HD 4670
  - ATI Radeon HD 4850
  - ATI Radeon HD 4870

[9] http://www.apple.com/macosx/specs.html

In order to use OpenCL on a Nvidia GPU the system needs to install the needed software downloaded from the Nvidia web site[10] the download includes OpenCL drivers, OpenCL Visual Profiler, OpenCL code samples, and OpenCL Best Practices Guide. Nvidia supports installation on Windows and Linux, support for OS X is included in OS X 10.6 Snow Leopard[11].

To utilize AMD/ATI GPU the system needs to install the needed software from AMD/ATI downloaded from the AMD/ATI web site[12]. AMD/ATI supports Windows and Linux, support for OS X is included in OS X 10.6 Snow Leopard[13].

4.3 OpenCL or CUDA

CUDA is developed by Nvidia to use the Nvidia GPU as a processing unit in programs running on a system with a supported Nvidia GPU. Some differences on the two platforms:

- CUDA
  - Runs programs only on Nvidia GPUs.
  - Large community of developers[14].
  - Highly optimized for Nvidia GPUs.
  - Nvidia controls the development of CUDA.
  - Supported from the G8X series onwards(G8X series released in 2006)
- OpenCL
  - Runs on all supported accelerators, CPUs GPUs or specialized hardware.
  - Is an open standard developed by the Khronos Group.
  - Community of developers from the leading GPU companies, AMD(ATI)[15] and Nvidia[16]
  - Shorter life time (December 8 2008) than Nvidia CUDA and the support in OS is just starting to appear.

OpenCL addresses the need for a cross-platform, industry standard approach to development for heterogeneous architectures. This can enable more developers to take advantage of GPGPU acceleration in their applications, but what is even more compelling is the opportunity to build applications that leverage all of the system's compute resources – CPUs and GPUs – to provide a superior user experience.

[10] http://developer.nvidia.com/object/opencl-download.html
[11] http://www.apple.com/macosx/specs.html
[12] http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx
[13] http://www.apple.com/macosx/specs.html
[14] http://forums.nvidia.com/index.php?s=b61a17fcf475bc0e0bd56d8518c76ac2&showforum=62
[15] http://forums.amd.com/devforum/categories.cfm?catid=390&entercat=y
[16] http://forums.nvidia.com/index.php?showforum=134

5 Experiments

This chapter will include all work done in OpenCL and results from experiments run on the GPU.

5.1 Equipment used

In order to be able to run modular exponential calculations on the GPU we need equipment capable of running code on the GPU using OpenCL. In this thesis the selected equipment was limited to the support of OpenCL when we started this thesis. At the start time of writing the software the only operating system supporting OpenCL was Apple OS X 10.6. This situation has changed(15.04.2010) and AMD and Nvidia have delivered support for both Microsoft Windows and Linux in their graphics cards drivers. Due to this fact we decided to buy an Apple Mac Pro to be used to run all experiments on. The hardware is specified in more detail in Table 5.1 on Page 37. All the code for all the experiments is written in the C programing language the code running on the GPU is also written in C with the inclusion of the OpenCL extensions.

Software used in the experiments is software supported on the Apple OS X platform, the Table 5.1 on Page 38 has more details on the software and versions used in the experiments.

5.2 Modular Arithmetic on the GPU

5.2.1 Selecting method to use in the experiments

GPU

From the knowledge learned in Chapter 3 we selected to try to solve the challenge of modular arithmetic using a Montgomery approach as described in Sections 3.1.2 and 3.2.2. The main body of the OpenCL code then looks like this:

```
u32  a[N], n[N], workspace[(N+2)];
u32 t[N];
int id = get_local_id(0); // To get the thread id
u32 i,j;
u32 ex;
u32 blockId = get_group_id(0)*STRIDE; //To get the block id
t[id] = x[id + blockId ]; //Assign values to be calculated in this thread
```

Computer equipment	
CPU	Two 2.26GHz Quad-Core Intel Xeon 5500 series processors
GPU	Nvidia GeForce GTX 285 with 1024MB of GDDR3 memory, PCI Express 2.0
Memory	12GB
Hard Disk	4TB

Table 4: Computer equipment used in the experiments

Software	
Operating System	Apple OS X 10.6.3
IDE	Apple Xcode 3.2.1
Compiler	GCC 4.2
External libraries	OpenSSL 0.98

Table 5: Software used in the experiments

```
a[id] = r2[id + blockId];
n[id] = mod[id + blockId];

d = inv2adic(n[0]);

mulredc(t, t, a, n, d, workspace, N); // t = xR mod N

redc(a, a, n, d, workspace, N); // a = R mod N

for(i=0; i < BITS/32; ++i) //need to work in wraps of 32 threads to fully utilize the wrap s
{
  ex = exponent[i + blockId];

  for(j=0; j < 32; ++j)
  {
    if(ex&1)
      mulredc(a, a, t, n, d, workspace, N);
    mulredc(t, t, t, n, d, workspace, N);
    ex >>= 1;
  }
}
```

To run a kernel on the GPU using OpenCL there is a list of steps that has to be performed:

1 Get a list of available devices.

   OpenCL is capable of running the code on different devices, we need to obtain a list of supported devices on the system at hand. This is done using the OpenCL API.

2 Select device.

   We need to select the device we like to run the kernel on. This enables us to select the best device for the work at hand. We can also read out data about the device if we like to further analyze the capabilities of the device.

3 Create a context.

   This creates a "connection" to the device we have selected to use. This context is used to read/write data to the device.

4 Create command queue.

The command queue is responsible for scheduling the running of kernels on the GPU. We can schedule to run more than one kernel and create dependencies between the kernels in order to have them executed in an specific order.

5 Read the kernel file.

All kernel code to run on the device is written in a separate file and read at runtime.

6 Create program object.

Add the kernel code file to the program object in order to enable the device to compile the program.

7 Compile kernel.

Use the newly created program object to compile the kernel code. The compilation is done on the device and there is a possibility to read back the compile log to check for errors if the compile failed.

8 Create memory objects.

Create the memory objects responsible for storing the data on the device.

9 Write memory object to device memory.

Copy memory objects from the system memory to the device memory.

10 Set kernel arguments.

Set the appropriate kernel argument, this is often the memory objects created earlier.

11 Execute kernel.

Use the command queue to schedule execution of the kernel program created using the program object.

12 Read memory object.

Read back to the system memory from the device memory, this is the only way to get the results from running a kernel.

13 Free objects.

We need to free objects created on the device in order to enable it to run more kernels without running out of memory.

In arithmetic computation, Montgomery reduction is an algorithm introduced in 1985 by Peter Montgomery [5] that allows modular arithmetic to be performed efficiently when the modulus is large (typically several hundred bits). A single application of the Montgomery algorithm is faster than a "naive" modular multiplication. The code for modular reduction in OpenCL is included to show how the code is constructed to do the calculations using the GPU:

```
void mulredc(u32 *z, u32 *x, u32 *y, u32 *n, const u32 d, u32 *t,int N)
{
```

```
  int i, id = get_local_id(0);
  u32 m, u;
  t[id] = 0;
  t[N] = t[N+1] = 0;

  for(i=0; i < N; ++i)
  {
    // multiply
    addmul_1(t, x, y[i], N);

    // reduce
    m = d*t[0];
    addmul_1(t, n, m, N);

    // shift
    u = t[id+1];
    t[id] = u;
    u = t[N+1];
    t[N] = u;
    t[N+1] = 0;
  }

if(cmp_ge_n(t, n, N)){
sub_n(t, t, n,N);
}

  z[id] = t[id];
}
```

The main focus of the GPU program was to align the threads into wraps of size 32. This to maximize the utilization of the Nvidia GPU properties. Using this method we found that it was better to have threads do nothing than break the wrap size of 32. Even more important was it to follow the rules from [6] and use the rules to optimize the GPU program. M. Welschenbach book "Cryptography in C and C++" [42] contains c code samples on how to create Montgomery multiplication and exponentiation modulo n. The samples have been used to create a starting point for the OpenCL code used in the experiment. But in the end none of this code was used in the final program. As a reference in C the book "The C programming language" was used to solve C language questions the book "Programming in C" [43] was used for the same purpose. When starting to create the GPU code the article by S Fleissner [38] contains valuable points in how to create a GPU program that does Montgomery exponentiation. The article contains a list of steps how this can be done:

1. Use n to pre-compute n' and r.
2. Calculate $\bar{a} := axr \bmod n$.

3. Calculate $\bar{x} := 1xr \bmod n$. item For i:=|b|-1 down to 0 do
   (a) Calculate $\bar{x} := \text{MonPro}(\bar{x}, \bar{x})$
   (b) If the *i*-th bit of b is set, then calculate $\bar{x} := \text{MonPro}(\bar{a}, \bar{x})$
4. Calculate $x = \text{MonPro}(\bar{x}, 1)$.

The OpenCL coding was done using these online resources as references[44, 45, 46] and the work of Samuel Neves[47]. To create the values of n and e OpenSSL[1] was used to create the values using the RSA struct of OpenSSL. The use of OpenSSL as the generator of values ensures that the values are of high quality and with the correct length.

The RSA structure consists of several BIGNUM components. It can contain public as well as private RSA keys:

```
struct
        {
        BIGNUM *n;              // public modulus
        BIGNUM *e;              // public exponent
        BIGNUM *d;              // private exponent
        BIGNUM *p;              // secret prime factor
        BIGNUM *q;              // secret prime factor
        BIGNUM *dmp1;           // d mod (p-1)
        BIGNUM *dmq1;           // d mod (q-1)
        BIGNUM *iqmp;           // q^-1 mod p
        // ...
        };
RSA
```

The function RSA_generate_key is used to fill the RSA struct with the values we need to do the modular arithmetic using OpenCL. The function is part of the OpenSSL Library and is found used in the C programing language using this include statment: #include <openssl/rsa.h>

CPU

One of OpenCL's properties is to be multi device, meaning that the same code should be able to run on different devices without any changes to the code. The experiments indicates that this is not completely true, after starting to optimize the code for running on the GPU with the highest speed possible the code would no longer compile on the the CPU. This challenge lead to the creation of to separate code projects one for the GPU using OpenCL and one on the CPU. The question was then how to select the best path for creating a fast code for modular arithmetic on the CPU, the decision landed on doing the same as Harrison et al. in [18] and use OpenSSL as the library for RSA calculations. OpenSSL library is highly optimized for RSA operation and other attempts to create something from scratch showed that this was the fastest solution possible when using the CPU.

[1] http://www.openssl.org/docs/crypto/rsa.html

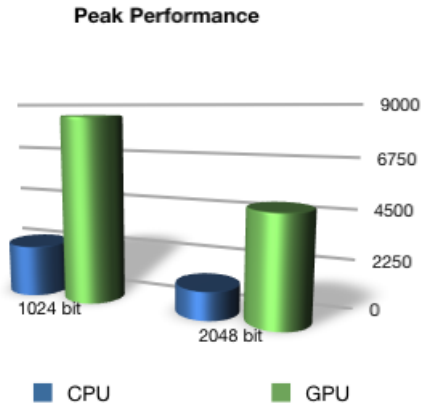

Figure 9: Performance of Nvidia GPU vs Intel CPU. Performance is messages per second.

5.2.2 Results

Given the results from [18] where Harrison et al. achieved a peak performance of 5536.75 using a G80 series Nvidia GPU the peak performance of 8533 from the top of the line G200 series GPU used in this project is in line with the results from Harrison et al. From the Figure 1 on Page 8 we can read that the G80 has a single precision performance just over 500 GFlops/s and the G200 is just below 1000 GFlops/s so the peak performance of 8533 is not twice the performance from Harrison el al. but this is near impossible to achieve. So a peak performance of 8533 is in line with the performance from Harrison et al. Szerwinski et al. [6] achieved a peak performance of 813 using the CIOS method on an older Nvidia 8800GTS GPU. As shown in Figure 1 on Page 8 there has been a great leap in performance for GPU in the resent years and this is not stopping, there has been even greater performance leaps in the last 6 months with the release for ATI Radeon HD 5000 series[2] and the Nvidia Geforce 400 series[3] moving the performance up in the 3000 GFlops/s range.

The results from running RSA public encrypt is showed in table 5.2.2 on Page 43

The peak performance in Table 5.2.2 on Page 43 is measured by doing many RSA encryptions and measuring the total time and dividing by the number of messages to get the peak performance for doing RSA encryption. This has an large impact on the result of the GPU as the startup cost(transfer of data to the GPU memory) is divided on many RSA encryption operations. The CPU have a smaller startup cost. The CPU also increase it performance as we schedule to do more RSA Encryption, indicating that OpenSSL is able to utilize the wast CPU power of the machine running the experiments. The time of doing just one RSA encryption is 59 ms for the GPU and 33 ms for the CPU the time is the same for both 1024 and 2048 bits.

[2] http://www.amd.com/uk/products/desktop/graphics/ati-radeon-hd-5000/Pages/atiradeonhd5000.aspx
[3] http://www.nvidia.com/object/geforce_family.html

Peak Performance		
Processing uint	Bits	Result
GPU	1024	8533
	2048	4830
CPU	1024	2371
	2048	1272
Min Performance		
Processing uint	Bits	Result
GPU	1024	8089
	2048	4272
CPU	1024	2360
	2048	1265
Average Performance		
Processing uint	Bits	Result
GPU	1024	8463
	2048	4753
CPU	1024	2368
	2048	1270

Table 6: Results from running the code. Result column is measured by RSA encryptions per second

6 Conclusion

6.1 Conclusion

Over the course of the last few years, GPUs have risen in relevance in many computational areas. Their impressive computing power in floating-point allowed GPUs to provide significant speedups in computational finance, chemistry, etc[1]. In fact, a GPU-based cluster has reached the Top 500 Supercomputer list[2], at 170 TFlops[3]. With this in mind, we started this project to harvest the computing power to perform modular arithmetic operations in the GPU. One of the main obstacles, is the overhead incurred in copying data to the GPU and back. From this results, using the GPU for cryptography is most fruitful when aiming for high-throughput.

OpenCL is a relatively young API and the implementation is still fresh so some of the promises made by the API is still in working progress to be fulfilled. Especially the promise of write once run everywhere is a truth with exceptions. From the experiments run the code written worked fine on both CPU and GPU until we started tuning for better performance, once we started using the different memory areas to speed up the modular arithmetic operations the code would no longer compile on the CPU, we hope this will change as the implementations mature. Also is the issue of different memory models on different devices will impact the performance of the kernel executed. So to create a truly universal program the developer needs either to stay away from trying to optimize performance or creating different version of the kernels and start by querying the device for supported extensions and running the kernel version that best suits the device at hand. So there is still some development needed before OpenCL is on route to deliver what it has promised[4].

We believe that we were successful: The best algorithm was selected and implemented ref Sections 3.1.2 and 3.2.2. The resulting performance figures are in line with the state of the art implementations, ref Sections 2.2 and 2.3, in the literature. This was accomplished by a careful study and development of the algorithm using the literature available and experiment with different settings on the GPU.

6.2 Future work

In this work we have not looked into the Elliptic Curve Cryptography, there is some work of doing ECC on a GPU but there is no work done on implementing this using OpenCL to research the performance of ECC on the GPU. In the timeline of this project there has been a larger development of support of OpenCL from being implemented in Apple OS X 10.6 only to being included in Microsoft Windows and Linux as well. There are also new suppliers of OpenCL drivers

[1] http://www.nvidia.com/object/cuda_apps_flash_new.html#
[2] http://www.top500.org/list/2009/11/100
[3] http://www.nvidia.com/object/io_1226945999108.html
[4] http://www.khronos.org/opencl/

emerging, one of them is FOXC from Fixstars[5]. This is a fast runtime for OpenCL, according to the book "The OpenCL Programming Book" [48] FOXC beats the AMD/ATI and Nvidia in memory copy and kernel execution. When starting this project the selection of HW was limited to Apple and there was a hope that the AMD/ATI Radeon HD 5870 GPU was to be available on the Apple platform before the experiments were run, this did not happen and the experiments were run on a Nvidia 285GTX GPU. ATI has delivered support for Windows and Linux and it will be interesting to see if someone in the future will do research on how the AMD/ATI GPU will perform with modular arithmetic operations.

Further research in using larger key values is also an area that can be of interest for further work, in this project we limit the key size to 1024bit and 2048bit and optimized for these key sizes. Larger key sizes can possible need a new approach and needs to be further researched in order to locate the optimal implementation.

[5] http://www.fixstars.com/en/foxc/

List of abbreviations

AMD Advanced Micro Devices

API Application Programming Interface

ATI ATI Technologies Inc.

CIHS Coarsely Integrated Hybrid Scanning

CISO Coarsely Integrated Operand Scanning

CPU Central Processing Unit

CRT Chinese Remainder Theorem

CUDA Compute Unified Device Architecture

ECC Elliptic Curve Cryptography

FIOS Finely Integrated Operand Scanning

FIPS Finely Integrated Product Scanning

GPGPU General-Purpose computation on Graphics Processing Units

GPU Graphics Processing Unit

HW Hardware

IBM International Business Machines

MRS Mixed Radix System

OpenCL Open Computing Language

RNS Residue Number System

RSA Rivest, Shamir and Adleman

SDK Software Development Kit

SOS Separated Operand Scanning

Bibliography

[1] Boeing, A. Survey and future trends of efficient cryptographic function implementations on gpgpus. *Security Research Centre conferences*, 21.04.2010. `http://scissec.scis.ecu.edu.au/secauconfs/proceedings/2008/forensics/Boeing%20Survey%20and%20Future%20Trends%20GPGPUs.pdf`.

[2] Bajard, J. & Plantard, T. 2004. Rns bases and conversions. *Advanced Signal Processing Algorithms, Architectures, and Implementations XIV,*, (5559), 60–69.

[3] Knuth, D. E. Jan 1971. *The art of computer programming vol 2: Seminumerical Algorithms*. Addison-Wesley Professional.

[4] Garner, H. Jan 1959. The residue number system. *AFIPS Joint Computer Conferences*.

[5] Montgomery, P. Jan 1985. Modular multiplication without trial division. *Mathematics of computation*, 44(170), 519–521.

[6] Szerwinski, R. & Guneysu, T. Jan 2008. Exploiting the power of gpus for asymmetric cryptography. *Lecture notes in computer science*.

[7] Bajard, J., Didier, L., & Kornerup, P. Jan 1997. An iws montgomery modular multiplication algorithm. *Proceedings of the 13th IEEE Symposium on Computer Arithmetic (ARITH-13 '97)*.

[8] Bajard, J., Didier, L., & Kornerup, P. Jan 2001. Modular multiplication and base extension in residue number systems. *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*.

[9] Bajard, J., Duquesne, S., & Ercegovac, M. Jan 2006. Residue systems efficiency for modular products summation: Application to elliptic curves cryptography. *SPIE the international society for optical Engineering*.

[10] Bajard, J., Kaihara, M., & Plantard, T. Jun 2009. Selected rns bases for modular multiplication. *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, 25 – 32.

[11] Bajard, J., Didier, L., & Kornerup, P. Jun 2001. Modular multiplication and base extensions in residue number systems. *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, 59 – 65.

[12] Bajard, J., Imbert, L., & Jullien, G. Jan 2005. Parallel montgomery multiplication in gf (2 k) using trinomial residue arithmetic. *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05)*.

[13] Bajard, J., Imbert, L., & Plantard, T. Modular number systems: Beyond the mersenne family. *Selected areas in cryptography*.

[14] Bajard, J. & Imbert, L. Jan 2004. A full rns implementation of rsa. *IEEE Transactions on Computers*.

[15] Wu, C., Hong, J., & Wu, C. Jan 2001. Rsa cryptosystem design based on the chinese remainder theorem. *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*.

[16] Walter, C. Jan 1999. Montgomery's multiplication technique: How to make it smaller and faster. *Lecture notes in computer science*.

[17] Moss, A., Page, D., & Smart, N. Jan 2007. Toward acceleration of rsa using 3d graphics hardware. *Lecture notes in computer science*.

[18] Harrison, O. & Waldron, J. Jan 2009. Efficient acceleration of asymmetric cryptography on graphics hardware. *Proceedings of the 2nd International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009*.

[19] Koc, C. K., Acar, T., & Kaliski, B. Jun 1996. Analyzing and comparing montgomery multiplication algorithms. *Micro, IEEE*, 16(3), 26 – 33.

[20] Dussé, S. & Jr, B. K. Jan 1990. A cryptographic library for the motorola dsp56000. *Advances in Cryptology—EUROCRYPT*.

[21] McLoone, M., McIvor, C., & McCanny, J. Jan 2004. Coarsely integrated operand scanning (cios) architecture for high-speed montgomery modular multiplication. *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, 185 – 191.

[22] Zhao, K. Implementation of multiple-precision modular multiplication on gpu. online, 20.04.2010. `http://www.comp.hkbu.edu.hk/~pgday/2009/10th_papers/kzhao.pdf`.

[23] Karatsuba, A. & Ofman, Y. Jan 1963. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, (145), 293–294.

[24] Fan, J., Sakiyama, K., & Verbauwhede, I. Jan 2007. Montgomery modular multiplication algorithm on multi-core systems. *2007 IEEE Workshop on Signal Processing Systems*.

[25] Yang, J., Chang, C., & Wang, C. Jan 2005. An iterative modular multiplication algorithm in rns. *Applied Mathematics and Computation*.

[26] Leong, P., Tan, E., & Tan, P. Jan 2002. An iterative modular multiplication algorithm. *Computers and Mathematics with with Applications*, (44), 175–180.

[27] Posch, K. & Posch, R. May 1995. Modulo reduction in residue number systems. *Parallel and Distributed Systems, IEEE Transactions on*, 6(5), 449 – 454.

[28] Kawamura, S., Koike, M., Sano, F., & Shimbo, A. Jan 2000. Cox-rower architecture for fast parallel montgomery multiplication. *Lecture notes in computer science*.

[29] Bajard, J., Didier, L., & Kornerup, P. Jan 2000. Montgomery modular multiplication in residue arithmetic. online, 20.04.2010. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.7076&rep=rep1&type=pdf.

[30] Freking, W. & Parhi, K. Oct 1999. Montgomery modular multiplication and exponentiation in the residue number system. *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, 2, 1312 – 1316 vol.2.

[31] Shenoy & Kumaresan. 1989. Fast base extension using a redundant modulus in rns. *Computers, IEEE Transactions on*, 38, 292–297.

[32] Szabó, N. S. & Tanaka, R. I. Jan 1967. *Residue arithmetic and its applications to computer technology*. McGraw-Hill.

[33] Bajard, J., Meloni, N., & Plantard, T. Efficient rns bases for cryptography. *Proceedings of IMACS 2005 World Congress*.

[34] Posch, K. & Posch, R. Jan 1993. Base extension using a convolution sum in residue number systems. *Computing*.

[35] Power, D. & Bradford, R. Jan 1999. A library for parallel modular arithmetic. *Lecture notes in computer science*.

[36] Pearson, D. A parallel implementation of rsa. online, 20.04.2010. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.5526&rep=rep1&type=pdf.

[37] Cook, D., Ioannidis, J., Keromytis, A., & Luck, J. Secret key cryptography using graphics cards. online, 20.04.2010. http://www.cs.columbia.edu/techreports/cucs-002-04.pdf.

[38] Fleissner, S. Jan 2007. Gpu-accelerated montgomery exponentiation. *Lecture notes in computer science*.

[39] Manavski, S. Jan 2007. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. *Proc. IEEE International Conference on Signal Processing and Communications (ICSPC 2007), 24-27 November 2007, Dubai, United Arab Emirates*.

[40] Szerwinski, R. & Paar, I. Efficient cryptography on graphics hardware. online, 20.04.2010. http://www.crypto.rub.de/imperia/md/content/texte/theses/da_szerwinski.pdf.

[41] Bækkelund, Ø. Implementation of public key algorithms in cuda. Master's thesis, NTNU, 2008.

[42] Welschenbach, M. Jan 2005. *Cryptography in C and C++*. Apress.

[43] Kochan, S. G. Jan 1988. *Programming in C*. Sams Publishing.

[44] Inc., A. 2009. Opencl programming guide for mac os x. online, 20.04.2010. http://developer.apple.com/mac/library/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCL_MacProgGuide.pdf.

[45] Nvidia. 2009. Opencl programming guide for the cuda architecture. online, 01.04.2010. http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf.

[46] Nvidia. 2009. Nvidia opencl jumpstart guide. online, 01.04.2010. http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf.

[47] Neves, S. 2010. Cryptography in gpus. online, 10.04.2010. http://eden.dei.uc.pt/~filipius/arquivo/2009/gpucrypto.pdf.

[48] Tsuchiyama, R. 2010. *The OpenCL Programming Book*. Fixstars Corporation.

A OpenCL Execution Model

OpenCL has a flexible execution model that incorporates both task and data parallelism. Data movements between the host and compute devices, as well as OpenCL tasks, are coordinated via command queues. Command queues provide a general way of specifying relationships between tasks, ensuring that tasks are executed in an order that satisfies the natural dependences in the computation. The OpenCL runtime is free to execute tasks in parallel if their dependencies are satisfied, which provides a general-purpose task parallel execution model. Tasks themselves can be comprised of data-parallel kernels, which apply a single function over a range of data elements, in parallel, allowing only restricted synchronization and communication during the execution of a kernel. These concepts will be further explained in this section.

Kernels

As mentioned, OpenCL kernels provide data parallelism. The kernel execution model is based on a hierarchical abstraction of the computation being performed. OpenCL kernels are executed over an index space, which can be 1, 2 or 3 dimensional. In Figure 10 on Page 53, we see an example of a 2 dimensional index space, which has G_x * G_y elements. For every element of the kernel index space, a work-item will be executed. All work items execute the same program, although their execution may differ due to branching based on data characteristics or the index assigned to each work-item. The index space is regularly subdivided into

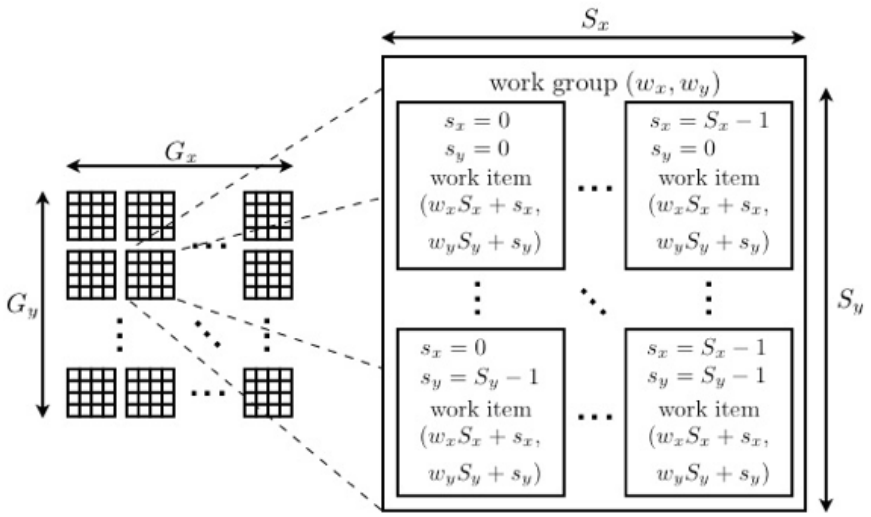

Figure 10: SEQ Executing Kernels - Work-Groups and Work-Items

work-groups, which are tilings of the entire index space. In Figure 10 on Page 53, we see a work-group of size s_x * s_y elements. Each work-item in the work group receives a work-group id, labeled (w_x, w_y) in the figure, as well as a local id, labeled (s_x, s_y) in the figure. Each work-item also receives a global id, which can be derived from its work-group and local ids.

The work-items may only communicate and synchronize locally, within a work-group, via a barrier mechanism. This provides scalability, traditionally the bane of parallel programming. Because communication and synchronization at the finest granularity is restricted in scope, the OpenCL runtime has great freedom in how work-items are scheduled and executed.

Command Queues

To execute a kernel, the kernel is pushed onto a particular command queue. Enqueueing a kernel is done asynchronously, so that the host program may enqueue many different kernels without waiting for any of them to complete. When enqueueing a kernel, the developer optionally specifies a list of events that must occur before the kernel executes. Events are generated by kernel completion, as well as memory read, write, and copy commands. This allows the developer to specify a dependence graph between kernel executions and memory transfers in a particular command queue or between command queues themselves, which the OpenCL runtime will traverse during execution. Figure 11 on Page 54 shows a task graph illustrating the power of this approach, where arrows indicate dependencies between tasks. For example, Kernel A will not execute until Write A and Write B have finished, and Kernel D will not execute until Kernel B and Kernel C have finished. The ability to construct arbitrary task graphs is a

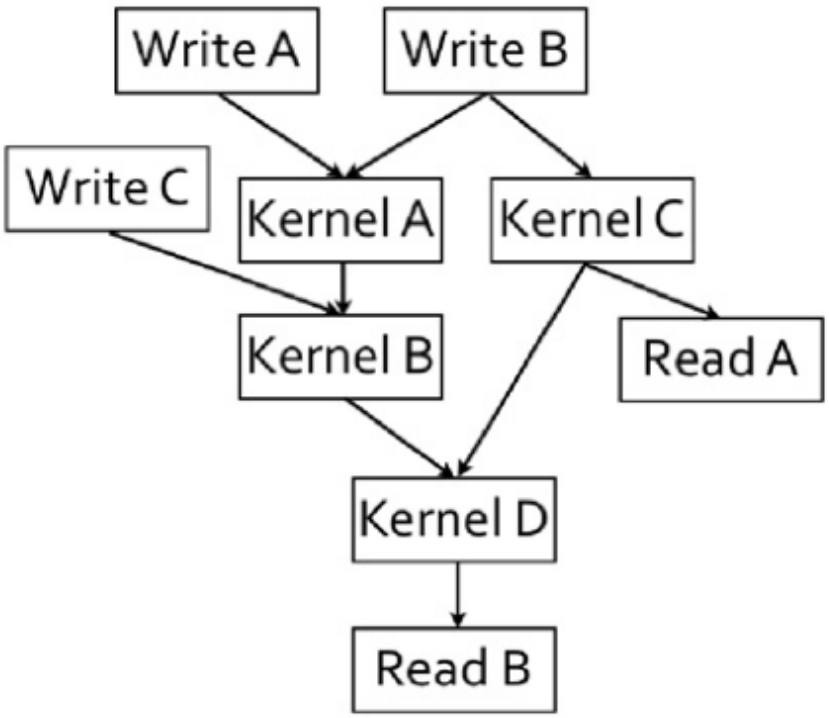

Figure 11: Task Parallelism within a Command Queue

powerful way of constructing task-parallel applications. The OpenCL runtime has the freedom to execute the task graph in parallel, as long as it respects the dependencies encoded in the task graph. Task graphs are general enough to represent the kinds of parallelism useful across the spectrum of hardware architectures, from CPUs to GPUs.

Developers are also free to construct multiple command queues, either for parallelizing an application across multiple compute devices, or for expressing more parallelism via completely independent streams of computation. OpenCL's ability to use both data and task parallelism simultaneously is a great benefit to parallel application developers, regardless of their intended hardware target.

Synchronization

Besides the task parallel constructs provided in OpenCL which allow synchronization and communication between kernels, OpenCL supports local barrier synchronizations within a work-group. This mechanism allows work-items to coordinate and share data in the local memory space using only very lightweight and efficient barriers. Work-items in different work-groups should never try to synchronize or share data, since the runtime provides no guarantee that all work-items are concurrently executing, and such synchronization easily introduces deadlocks.

Work-items in different work-groups may coordinate execution through the use of atomic memory transactions, which are an OpenCL extension supported by some OpenCL runtimes, such as the ATI Stream SDK OpenCL runtime for the x86 multi-core compute devices. For example, work-items may append variable numbers of results to a shared queue in global memory. However, it is good practice that work-items do not, generally, attempt to communicate directly, as without careful design scalability and deadlock can become difficult problems. The hierarchy of synchronization and communication provided by OpenCL is a good fit for many of today's parallel architectures, while still providing developers the ability to write efficient code, even for parallel computations with non-trivial synchronization and communication patterns.