A STUDY OF HIGH PERFORMANCE MULTIPLE PRECISION ARITHMETIC ON GRAPHICS PROCESSING UNITS

A Thesis Presented

by

NIALL EMMART

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Sep 2017

College of Information and Computer Sciences

A STUDY OF HIGH PERFORMANCE MULTIPLE PRECISION ARITHMETIC ON GRAPHICS PROCESSING UNITS

A Thesis Presented

by

NIALL EMMART

Approved as to style and content by:

Charles C. Weems, Chair

J. Eliot B. Moss, Member

David A. Mix Barrington, Member

Israel Koren, Member

James Allan, Chair of the Faculty
College of Information and Computer Sciences

DEDICATION

To Zoya

ACKNOWLEDGMENTS

I want to express my deeply-felt thanks to my thesis advisor, Professor Charles Weems, for his enthusiasm and warm encouragement. This thesis would not have been possible without his generosity and thoughtful guidance throughout my graduate studies. I also thank the other members of my committee: Professor J. Eliot B. Moss, for his encouragement, and careful reading of the text, and Professors David A. Mix Barrington and Israel Koren for their feedback and helpful comments.

I am grateful to Justin Luitjens, Cliff Woolley, and the team at NVIDIA for their support and sharing their deep knowledge of GPU architectures, compilers and software tuning and to Professor Yang Chen for putting me on the path toward research in multiple precision arithmetic. Thanks to Arjun Jayadev for his advice and encouragement and to Fabricio Murai Ferreira and Mostafa Dehghan for their comradery.

Finally, I wish to thank my family: to my parents, Martini Niedbalski-Emmart and Bob Niedbalski for their encouragement and for instilling an interest in academics, and to my wife and daughter, Sunitha and Zoya, for giving me the time and unwavering support to pursue a Ph.D.

This thesis is based in part upon work supported by the National Science Foundation under Award Numbers CCF-1217590 and CCF-1525754.

ABSTRACT

A STUDY OF HIGH PERFORMANCE MULTIPLE PRECISION ARITHMETIC ON GRAPHICS PROCESSING UNITS

SEP 2017

NIALL EMMART

B.A., UNIVERSITY OF MASSACHUSETTS, AMHERST

M.Sc., UNIVERSITY OF MASSACHUSETTS, AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Charles C. Weems

Multiple precision (MP) arithmetic is a core building block of a wide variety of algorithms in computational mathematics and computer science. In mathematics MP is used in computational number theory, geometric computation, experimental mathematics, and in some random matrix problems. In computer science, MP arithmetic is primarily used in cryptographic algorithms: securing communications, digital signatures, and code breaking. In most of these application areas, the factor that limits performance is the MP arithmetic. The focus of our research is to build and analyze highly optimized libraries that allow the MP operations to be offloaded from the CPU to the GPU. Our goal is to achieve an order of magnitude improvement over the CPU in three key metrics: operations per second per socket, operations per watt, and operation per second per dollar. What we find is that the SIMD design and balance of compute, cache, and bandwidth resources on the GPU is quite different from the CPU, so libraries such as GMP cannot simply be ported to the GPU. New

approaches and algorithms are required to achieve high performance and high utilization of GPU resources. Further, we find that low-level ISA differences between GPU generations means that an approach that works well on one generation might not run well on the next.

Here we report on our progress towards MP arithmetic libraries on the GPU and propose enhancements in three areas: (1) large integer addition, subtraction, and multiplication; (2) high performance modular multiplication and modular exponentiation (the key operations for cryptographic algorithms) across generations of GPUs; (3) high precision floating point addition, subtraction, multiplication, and division. We have also developed a new short division algorithm, which we prove is asymptotically optimal on EREW and CREW PRAMs, which is discussed in the last chapter.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Multiple precision (MP) arithmetic is a field of growing importance. It is a key component in cryptography (e.g., securing communications, digitial signatures, and next generation secure internet architectures), experimental mathematics (e.g., finding integer relations in real-valued infinite series), computational number theory (e.g., prime testing, prime proving, factoring), random matrix theory (e.g., least eigenvalues), quantum field theory (e.g., zeta function identities), analysis of chaotic functions (e.g., logistic map), and computing high precision constants.

A significant increase in the speed of MP calculations is needed to enable breakthroughs in these areas. In addition, if a significant performance boost can be achieved with a modest increase in cost and power, then we open the door to both commercial applications (such as enabling wider use of encrypted communication channels and longer keys) and to a larger community of researchers who can make use of MP arithmetic on affordable platforms.

Graphics processors (GPUs) have been applied in a wide range of areas of high performance computing because their performance considerably exceeds CPU performance for certain kinds of fine-grained data-parallel algorithms, at costs far lower than custom processors, and with a very attractive performance per watt. Their performance is such that 65 machines in the November 2016 TOP500 list use NVIDIA GPU accelerators. The major problem is that it can be difficult to write and tune the software to fully exploit the potential performance.

Developing and maintaining an MP library for GPUs faces a major hurdle in that successive generations of GPU architecture change in ways that necessitate non-standard

optimizations and alternate algorithmic approaches. To be useful, a library must be upwards portable so that users can continue to rely on it as they upgrade hardware. In addition, it must support a wide range of MP value sizes, which imposes significant variations in the demands that are placed on the limited resources of the GPU's thread engines. These challenges are further complicated by aspects of the vendor supplied development tools that stand in the way of the necessary optimizations. Even so, developing a scalable, portable MP arithmetic library for GPUs offers the potential for performance increases of up to two orders of magnitude.

1.1 Application Areas for Multiple Precision Arithmetic

Multiple precision arithmetic is one of the fundamental building blocks of cryptographic algorithms from key exchange (RSA, Diffie-Hellman, ECHD) to digitial signature standards (DSA, ECDSA), to prime testing (Miller-Rabin) and proving (AKS), and factoring (ECM, QS, SNFS, GNFS). These algorithms are used to secure communications and to authenticate authorship/ownership of data, and underpin significant areas of the modern economy: e-commerce, secure WWW browsing through Public Key Infrastructure (PKI), secure inter-bank transfers, peer-to-peer networking, crypto-currencies, smart cards, digital signatures for software (e.g., apps), secure utility grids, command and control systems, military communications, etc. Significantly reducing the cost and increasing the performance of cryptographic algorithms could lead to wider deployment of cryptography with better privacy and security, and entirely new applications with the potential for widespread benefits.

New internet architectures such as content centric networking (CCN) [63], which use a digital signature with each packet, will require very high performance, low power, and low cost modular exponentiation support. In addition, research into fully homomorphic encryption (FHE) [51] depends on very high precision arithmetic.

Experimental mathematics is yielding high-value results using MP arithmetic. In particular, integer relation algorithms were named among the top ten algorithms of the 20^{th} century. The most widely used, PSLQ [45], searches for relations between high precision real values for a costly function, such as an infinite series, and a set of mathematical constants, to establish the likelihood of a closed form expression of the function. Among its well known applications was finding a formula to compute any digit of pi in base two [3]. It has also found closed forms for multiple zeta functions from quantum field theory, and for identifying bifurcation points in the chaotic logistic map function. Accelerating very high precision arithmetic with GPUs will enable more extensive searches for a wider range of functions.

Multiple precision arithmetic is essential in computational number theory research, aiding in the search for and proving of new primes, and for factorization. Certain problems in random matrix theory, such as our recent work in finding the least eigenvalues of ill-conditioned Hankel matrices, depend on extreme levels of precision to produce meaningful results. Lastly, of course, our package will be useful in the computation of high precision values for mathematical constants.

These are compelling applications and a number of researchers have attempted to build MP libraries for the GPU, but none of them meet all the important criteria: high performance, multiple generations of GPU, and support for a wide range of sizes.

1.2 The Thesis

The thesis that this dissertation will address is:

"Offloading multiple precision arithmetic computations from the CPU to the GPU will result in an order of magnitude boost in performance across three metrics: operations per second per socket, operations per watt, and operations per second per dollar. However, due to the SIMD design and different balance of compute, cache, and bandwidth resources

on the GPU, achieving such a high level of performance requires new approaches and algorithms."

We will validate this thesis by building and testing three multiple precision GPU libraries. The first will implement multiple precision floating point addition, subtraction, multiplication, division, and square root with mantissas that can range in size from 1024 bits to 8192 bits. The second will implement modular exponentiation using integer operations on sizes that range from 256 bits to 2048 bits. The third will implement modular exponentiation using double precision floating point arithmetic. We will compare the GPU libraries to GMP [54] and MPFR [48] two high performance libraries on the CPU.

CHAPTER 2

LITERATURE SURVEY

The literature survey is organized as five sections. Section 2.1 covers number representation. Section 2.2 covers the relevant sequential multiple precision arithmetic algorithms, giving both the literature review and an overview of how the algorithms work. These algorithms serve as core background knowledge for the proposal. In Section 2.3 we cover the relevant parallel multiple precision algorithms that operate in a manner that is different from their sequential counterparts. Section 2.4 covers parallel CPU implementations of multiple precision arithmetic and Section 2.5 covers GPU implementations.

2.1 Number Representation

There are two common number representations for unsigned high precision positive integers, positional fixed radix number systems (FRNS) and residue number systems (RNS). In an FRNS the radix β is typically chosen to be power of two, such as 2^{32} on 32-bit machines or 2^{64} on 64-bit machines. A high precision number, X is then represented as a sequence of n values (computer words) $x_0, x_1, \cdots, x_{n-1}$ where $X = \sum_{i=0}^{n-1} x_i \beta^i$, and $0 \leq x_i < \beta$. The x_i values are ofted referred to as *limbs*. The range of X that can be uniquely represented by n-words is $0 \mathinner{..} \beta^n - 1$. In an RNS, a set of pair-wise coprime moduli, $\{m_i\}$ are chosen. Then X is represented as a sequence of moduli, $x_i = X \bmod m_i$. The range of X that can be uniquely represented is $0 \mathinner{..} \left(\prod_{i=1}^{n} m_i \right) - 1$. In an RNS, addition, subtraction and multiplication are all linear time operations, computed component-wise:

$$(X \pm Y)_i = (x_i \pm y_i) \bmod m_i$$

$$(X \cdot Y)_i = (x_i \cdot y_i) \bmod m_i$$

Due to the component-wise nature, these operations are inherently parallel, requiring no communication between the components. However, other common arithmetic operations, such as comparison, division, and modulo (by arbitrary values) are much slower. In effect, what the implementations do is convert the RNS representation to an FRNS representation and use standard algorithms. RNS representations also suffer from two other drawbacks. First, the representation is not as compact as FRNS representations on binary computers. Second, it's quite easy to extend the range of X in an FRNS, just by adding another word and increasing n. In an RNS, one must dynamically construct an extra modulus for the set $\{m_i\}$ on the fly and precompute various constants needed to support the new modulus. For much more on conventional and unconventional number representations, we refer the reader to *Computer Arithmetic Algorithms* by Koren [75].

For the remainder of this proposal, we will use the convention that capital letters represent unsigned multiple precision values, represented in an FRNS (with radix 2^w where w is the machine word size) unless otherwise stated. Lower case letters represent single word values in the range of 0 to $\beta - 1$.

2.2 Sequential Algorithms

There are several textbooks with good introductory material on sequential multiple precision algorithms. These include Knuth's *The Art of Computer Programming, Volume 2, Seminumerical Algorithms* [73], Koren's *Computer Arithmetic Algorithms* [75] and Brent and Zimmermann's *Modern Computer Arithmetic* [16].

2.2.1 Addition and Subtraction

Addition and subtraction of two n-word multiple precision (MP) values, $A \pm B$, are performed with the traditional grade school algorithm, going from the least significant word to the most signficant word, pushing the carry/borrow along. These algorithms run in $O(n)$ time and there are no faster algorithms, because all n words of A and B must be read.

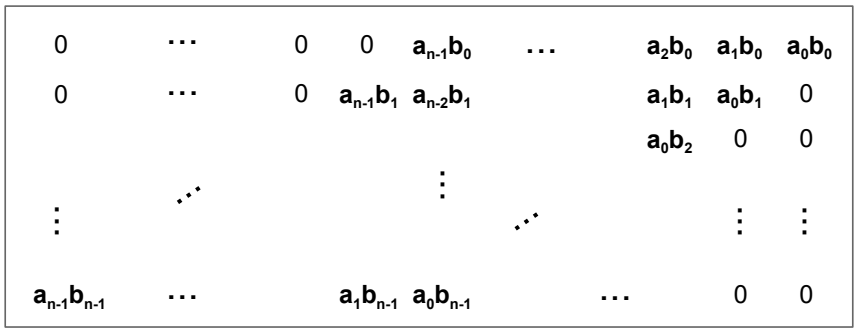

Figure 2.1. Product terms to be summed for an n-word by n-word multiply.

2.2.2 Multiplication

For multiplication there are six main algorithms for computing the product of two n-word MP values, A and B. These algorithms are presented below. The five general purpose algorithms form a ladder, with decreasing asymptotic complexity, but unfortunately, as we move up the ladder, the asymptotic notation hides increasing constants. Thus, for small n, it's actually faster to use the algorithms at the lower rungs of the ladder than the algorithm at the top. A good multiple precision library usually implements several of these algorithms, and will have some kind of tuning procedure to determine the best sizes of n (crossover points) to switch between algorithms.

Grade School: the grade school is the simplest algorithm, where the n^2 product terms of Figure 2.1 are summed along the columns from the right to the left. This algorithm runs in $O(n^2)$ time.

Karatsuba: the Karatsuba multiplication algorithm [69] is an asymptotically faster recursive divide and conquer approach. It works by splitting the A and B value into upper and lower halves, $A = A_H\beta^{\lceil n/2 \rceil} + A_L$ and $B = B_H\beta^{\lceil n/2 \rceil} + B_L$. Computing the product $A \cdot B$ requires computing three sub-products, $A_H \cdot B_H$, $A_L \cdot B_L$, and $(A_H + A_L) \cdot (B_H + B_L)$. If the length of the sub-products is short enough, it's faster to compute them with the grade school multiplier. If they are still large, then the code recursively calls the Karatsuba multiplier. The product, $A \cdot B$ is easily computed from the three sub-products.

Since Karatsuba is a recursive divide and conquer algorithm and computing the final product from the sub-products runs in time $O(n)$, we can express the Karatsuba run time as $K(n) = 3K(\lceil n/2 \rceil) + \Theta(n)$ and by the Master Theorem (see for example [27]), the run time is $\Theta(n^{log_2 3})$, i.e., $\Theta(n^{1.58})$.

There are a few variants of the Karatsuba approach, such as using $A_H - A_L$ and/or $B_H - B_L$ (instead $A_H + A_L$ and $B_H + B_L$) and using different split points instead of $\lceil n/2 \rceil$. These are described in Knuth [73] and Brent and Zimmermann [16].

Toom-Cook: Proposed by Toom [105] and improved by Cook [24], Toom-Cook multiplication is also a divide and conquer algorithm. But instead of Karatsuba's approach of dividing A and B into halves, Toom-Cook divides them into 3 or more pieces. Toom-Cook is not a single algorithm, rather it is a family of algorithms: Toom-Cook 3-way, 4-way, 5-way, etc, depending on the split count, k.

Toom-Cook works by mapping A and B to $k-1$ degree polynomials, $S(x)$ and $T(x)$, such that $S(\beta^{\lceil n/k \rceil}) = A$ and $T(\beta^{\lceil n/k \rceil}) = B$. For example, when $k = 3$, we have

$$S(x) = S_2 x^2 + S_1 x + S_0$$

and

$$S(\beta^{\lceil n/3 \rceil}) = S_2 \beta^{2\lceil n/3 \rceil} + S_1 \beta^{\lceil n/3 \rceil} + S_0 = A$$

thus S_0, S_1, S_2 split A into thirds. S_0 is the least significant $\lceil n/3 \rceil$ words of A, S_1 is the middle third and S_2 is the remaining most significant third. The S and T coefficients are analogous to the Karatsuba A_h, A_l, B_h, and B_l terms.

Next, Toom-Cook determines the product polynomial, $P(x) = S(x) \cdot T(x)$, with polynomial interpolation via a Vandermonde matrix. Since $P(x)$ is degree $2k-2$ we'll need $2k-1$ evaluation points for the interpolation. These interpolation points can be arbitrarily chosen, for example: $P(0) = S(0) \cdot T(0)$, $P(1) = S(1) \cdot T(1)$, $P(2) = S(2) \cdot T(2)$, $\cdots$, $P(2k-2) = S(2k-2) \cdot T(2k-2)$. Performing the interpolation thus requires computing $2k-1$ sub-products, but these sub-products are much shorter than

A and B because for small x, $S(x)$ and $T(x)$ will be roughly n/k words in length. If the lengths of the sub-products are small enough, then it's faster to compute them with the grade school multiplier. If they are still large, the code calls the Toom-Cook multiplier recursively.

The next step is to determine the coefficients of $P(x)$, which requires solving the Vandermonde system of $2k-1$ linear equations. Since a general linear solver would be quite slow, in practice, Toom-Cook implementations fix a set of evaluation points for each k, and implement a custom piece of code to solve the resulting specific linear system. Thus, a fast multiple precision library tends to have a few implementations of Toom-Cook, for specific values of k, rather than a general solver that works for any k. Once $P(x)$ is in hand, the product $A \cdot B$ can be computed by evaluating $P(\beta^{\lceil n/k \rceil})$.

The running time of Toom-Cook can be expressed as $TC(n) = (2k-1)TC(\lceil n/k \rceil) + \Theta(n)$, and again by the Master Theorem, we have $TC(n) = \Theta(n^{\log_k 2k-1})$.

The actual performance of a Toom-Cook implementation depends heavily on the interpolation points used. For small k, the best interpolation points have been extensively researched by Cook [24], Knuth [73], Zuras [114], Zimmermann, and Bodrato [13, 14]. Further, it has been shown (see for example Chung and Hasan [21]) when $k \geq 3$, the interpolation step requires an exact division by a non-power of two constant. Thus an efficient Toom-Cook implementation requires an efficient short division implementation.

For unbalanced multiplication, where the length of A is longer or shorter than the length of B, different split counts can be used on A and B. See Section 1.3.5 of Brent and Zimmermann [16] for details.

FFT Multiplication: FFT Multiplication splits the A and B values into fixed length pieces, $A = (a_0, a_1, \cdots, a_{k-1}), B = (b_0, b_1, \cdots, b_{k-1})$. The a_i and b_i terms can be thought of as "digits" or as "samples". The grade school algorithm tells us that the product sum of the i$^{\text{th}}$ column is $\sum_{j=0}^{i} a_j \cdot b_{i-j}$ when $i < k$ and $\sum_{j=i-k}^{k} a_j \cdot b_{i-j}$ when $i \geq k$. These sums are exactly the convolution of the a and b samples. It is well known that FFTs can be used to

compute convolutions efficiently, as follows:

$$\text{conv}(\hat{a}, \hat{b}) = \text{FFT}^{-1}(\text{FFT}(\hat{a}) * \text{FFT}(\hat{b}))$$

where $\hat{a}$ and $\hat{b}$ represent the vector of samples, asterisk represents component-wise multiplication.

The challenge for FFT multiplication is that the FFT computations must be performed in a ring, with a k^{th} primitive root of unity. The choice of ring is very important. One possibility is to use complex numbers with floating point representations. This approach is used in some FFT multiplication algorithms, but the error analysis of the round-off errors in the floating point computations is very complicated. Another option, which we use in our research, is to use finite rings or fields, represented as integers or multiple precision integers. The computations will be exact. However, for a given finite ring/field, there is a limit to the maximum length A and B that can be multiplied without the column sum overflowing the maximum ring value. Thus, FFT multiplication is not a general multiplication algorithm that can multiply any size numbers, like the previous algorithms. However, given this limitation, it is very efficient with a running time of $O(n \log n)$.

Schönhage-Strassen: In 1971 Schönhage and Strassen [94] published a fast algorithm for multiplying arbitrary length numbers, based on FFT multiplication. Here we present a slightly simplified version of the Schönhage-Strassen algorithm. Given two n-bit numbers, A and B, split A and B into k pieces where k is chosen according to a heuristic that ensures $k = \Theta(\sqrt{n})$. Next run FFT multiplication on the pieces, using the finite ring **R** consisting of the integers modulo $2^{2k} + 1$. The number 2 is a $4k^{\text{th}}$ primitive root of unity in **R**. Thus a $2k$-point FFT can be run in **R** using $\omega = 2^2$.

The FFTs and FFT^{-1} each run in $O(k \log k)$ operations in **R** and each operation (addition, subtraction, and multiplication by ω^i) in **R** runs in $\Theta(k)$ bit operations. Thus the total time to run the FFTs and FFT^{-1} is $\Theta(k \cdot k \log k)$ which is $\Theta(n \log n)$. However, the convolution algorithm still requires performing k component sub-products of

$2k$-bits times $2k$-bits each. If k is small enough, these are done using a base case algorithm (such as grade school). If not, they are done with recursive calls to the Schönhage-Strassen algorithm. Thus the total running time of the algorithm can be expressed as $T(n) = k \cdot T(2k) + \Theta(n \log n)$, where $k = \Theta(\sqrt{n})$. By the substitution method, we find that $T(n) = O(n \log n \log \log n)$ bit operations. Further, the Schönhage-Strassen algorithm takes advantage of the cyclic nature of FFTs, which allows the sub-products to be computed in $2k$-bits rather than $4k$-bits. This explanation of the algorithm and its time complexity is somewhat terse; for a full description and analysis, please refer to [94], [73], or [16].

For practical implementations, there are some helpful tricks to improve the performance described by Gaudry, Kruppa and Zimmerman [50].

Faster Multiplication: In 2007 Martin Fürer published [49] an algorithm with a running time of $O(n \log n \ 2^{O(\log^* n)})$, which is asymptotically faster than Schönhage-Strassen. In 2016 Harvey, van der Hoeven, and Lecerf published a related algorithm [56], but with a slightly better complexity of $O(n \log n \ K^{\log^* n})$, where $K = 8$. If certain conjectures about the distribution of Mersenne primes hold, then their algorithm can be shown to have complexity $O(n \log n \ K^{\log^* n})$, where $K = 4$.

However, these algorithms are mostly of theoretical interest because the constant hidden in the O-notation is very large. So in practice, Schönhage-Strassen is faster except for "astronomically" large values of A and B.

2.2.3 Fast Squaring

Computing the square of an n-word MP value A is often faster than computing the product of two n-word MP values $A \cdot B$. Here we briefly present some of the approaches. For small n, we will want to use the grade school algorithm, but we can take advantage of symmetry. Considering Figure 2.1, if a column has a term $a_i b_j$, then the column will also have a term $a_j b_i$ when $i \neq j$. Since we're squaring, these two terms will be equal, which

leads to a fast algorithm, namely, add the terms above the slew diagonal, double the value then add in the diagonal terms.

For larger n, in the Karatsuba range, we note that $A_L + A_H = B_L + B_H$ so this sum need only be computed once, and the sub-products are all squares, which can be computed by recursively calling the square routine and once the values are small enough, will take advantage of fast grade school squaring.

For Toom-Cook, $S(x)$ and $T(x)$ will be the same, which eliminates a lot of calculations. Also, like Karatsuba, all the sub-problems will be squares, and finally, Chung and Hasan [21] prove that unlike Toom-Cook multiplication, Toom-Cook squaring can be performed using only division by powers of two, thus eliminating the need for short division, which represents a significant percentage of the running time for Toom-Cook multiplication.

For large n, using the FFT based algorithms, the convolution becomes $\mathrm{FFT}^{-1}(\mathrm{FFT}(\hat{a})^2)$ which saves an FFT.

For the rest of this dissertation, we will refer to these algorithms as "fast-squaring".

2.2.4 Division

As with multiplication, there are a many algorithms for doing division. The algorithms generally fall into two categories: slow quadratic algorithms which produce a fixed number of bits of quotient per iteration, and fast sub-quadratic algorithms. Karp and Markstein's literature reivew in [21] covers many approaches to division, including some obscure ones, such as Chebyshev polynomial approximations, CORDIC methods, Sweeney, Robertson, and Tocher (SRT) division [92, 104], and Goldschmidt's sub-quadratic algorithm [53].

Here we discuss in detail some of the algorithms presented by Brent and Zimmermann in [16]. For each of these algorithms we wish to divide an n-word number, A, by an m-word divisor, B, with $n > m$. We assume that B is normalized, meaning that the most significant word of B is at least $\beta/2$. If this is not the case, we can construct A' and B' by shifting A and B by k bits to the left. k is chosen such that B' is normalized. Then the

quotient A/B is A'/B' and the remainder of A/B is the remainder of A'/B' shifted k bits to the right.

Like multiplication, the following three division algorithms form a ladder, with decreasing asymptotic complexity, but with increasing constants hidden by the Big-O notation. A good multiple precision library will typically implement all three algorithms and have a tuning routine to determine the best crossover points.

Grade School: For grade school division, we iterate over j from $n - m$ down to 0. At each iteration, we compute an estimate for the next quotient word as $q_j^* = \min\left(\lfloor (a_{m+j}\beta + a_{m+j-1})/b_{n-1} \rfloor,\ \beta - 1\right)$. This estimate is guaranteed to be greater than or equal to q_j, the next quotient word. The next step is to correct the estimate: while $A - q_j^* \cdot B \cdot \beta^j < 0$, decrement q_j^*. After this loop, q_j^* is correct and we set $q_j = q_j^*$, and update $A = A - q_j^* \cdot B \cdot \beta^j$.

Brent and Zimmermann [16] prove the correction step happens at most twice. Knuth [73] notes that if q_j^* is chosen as

$$q_j^* = \lfloor (a_{m+j}\beta^2 + a_{m+j-1}\beta + a_{m+j-2})/(b_{m-1}\beta + b_{m-2}) \rfloor$$

then only one correction step is needed and it's extremely rare.

The running time is $O(nm)$, and when $m \approx n/2$, the running time is $O(n^2)$, so the algorithm is quadratic.

Burnikel-Ziegler: The next algorithm is a recursive divide and conquer approach to division, due to Burnikel and Ziegler [17]. The basic idea is that instead of computing a single word estimate q^*, we should go to longer, multiword estimates. Then when we do the correction step, $A - q^* \cdot B \cdot \beta^j$, we can take advantage of asymtotically fast multiplication to compute $q^* \cdot B$. Without loss of generality, let's assume that A is exactly twice as long as B. We can split A into four pieces and B into two pieces as follows:

$$A = A_3\beta^{3(n/4)} + A_2\beta^{2(n/4)} + A_1\beta^{n/4} + A_0 \quad \text{and} \quad B = B_1\beta^{n/4} + B_0$$

Further, we can represent the quotient, $Q = A/B$ as $Q = Q_1\beta^{n/4} + Q_0$. We can estimate Q_1 by calling the division routine recursively, $Q_1^* = \mathrm{divide}(A_3\beta^{n/4} + A_2,\ B_1)$. The estimate will not be exact, but it will be close and the exact value is easily obtained with a few correction steps using the full values of A and B.

Next, construct $A' = A - Q_1 \cdot B \cdot \beta^{n/4}$. A' can be represented as:

$$A' = A'_2\beta^{2(n/4)} + A'_1\beta^{n/4} + A'_0$$

Q_0 can then be estimated by calling the division routine recursively again, $Q_0^* = \mathrm{divide}(A'_2\beta^{n/4} + A'_1,\ B_1)$. The estimate will be close to the exact value, which can be determined with a few more correction steps. The final Q is just $Q_1\beta^{n/4} + Q_0$.

This algorithm outperforms grade school in two ways. First, it takes advantage of asymptotically fast multiplication. Second, the total number of words affected by corrections is greatly reduced.

The running time analysis of this algorithm is complicated and depends on the multiplication algorithm being used. Brent and Zimmermann show that the running time to divide an n-word number by a $n/2$-word number is roughly 2 to 3 times the running time to multiply two $n/2$-word numbers.

Finally, as with Knuth's observation about the grade school algorithm, the correction steps can be greatly reduced by using one more word for both numerator and denominator in the recursive calls.

Newton-Raphson: For very large numbers, the fastest approach to division is to use a Newton iteration to find the reciprocal of B. The approach is similar to Burnikel-Ziegler in that it splits A into 4 pieces (A_0, A_1, A_2, A_3), and B and Q into two pieces (B_0, B_1 and Q_0, Q_1) and computes Q_1 followed by Q_0. Again we assume $n = 2m$.

The algorithm first uses a Newton iteration to compute an integer R (the reciprocal) such that $R = \left\lfloor \frac{\beta^{2n/4}}{B_1} \right\rfloor$. The algorithm follows the steps of Burnikel-Ziegler, except instead

of the recursive calls, the estimates of Q_1^* and Q_0^* come from computing the high $n/4$ words of the product $Q_1^* = high(A_3 \cdot R)$ and $Q_0^* = high(A_2' \cdot R)$.

Here we present some background on Newton's iteration. Suppose we are given a real number $0.5 \leq x < 1$, and an initial estimate y_0 for $1/x$, where $|y_0 - 1/x| < \epsilon$ and $1 < y_0 \leq 2$, then the following sequence: $y_{k+1} = y_k + y_k(1 - xy_k)$ which will rapidly converge to $1/x$. It can be proven that $|y_k - 1/x| < 4^k\epsilon^{2^k}$ (see for example Brent and Zimmermann's Section 3.4.1). In practice, this means that if an estimate y_k is accurate to c bits of precision, then y_{k+1} will be accurate to $2c - 2$ bits of precision.

Earlier we had described $R = \left\lfloor \frac{\beta^{2n/4}}{B_1} \right\rfloor$ as the "reciprocal" of B_1, which seems strange because they are both integers. Here's the story. There is a duality here: B_1 and R can be viewed as integers, but they can also be viewed as fixed point numbers with $n/4$ words after the decimal point. In the fixed point context, R and B_1 are reciprocals. Further, because B is normalized, B_1 as a fixed point number is between $1/2$ and 1 and the Newton iteration applies. Thus we can write a fixed point version Newton's iteration as follows. We start out by setting $R_0 = \beta^2/B_{m-1} \cdot \beta^{n/4-1}$ and iterate $\log m$ times:

$$R_{j+1} = R_j + R_j \cdot (\beta^{n/4} - B_1 \cdot R_j/\beta^{n/4})/\beta^{n/4}$$

This is exactly the Newton iteration translated into fixed point arithmetic. R_0 is accurate to 1 word, and after each iteration, the number of words of accuracy approximately doubles. We note that we've done one more iteration than is strictly necessary, which is to handle the 4^k term that crops up in the Newton iteration. Finally we note that the fixed point representation is just an approximation to the real numbers used in Newton's iteration. Thus, in addition to the error analysis of Newton's method, there are round off errors to contend with, and up to two correction steps may be required to get the correct final value for R.

This implementation of Newton's iteration is inefficient because the early iterations are computing with much more precision than required. Brent and Zimmermann give a more complicated, but more efficient implementation in Section 3.4.1 of [16].

Computing the Newton reciprocal R to $n/4$-words and doing the division in two steps (Q_1 followed by Q_0) is significantly faster than computing R to $2n/4$-words and computing all of Q in a single step and is due to Karp and Markstein [70]. It is sometimes called the "Karp and Markstein trick".

2.2.5 Specialized Division Algorithms

In some cases, there are special properties about the division that we can take advantage of. We use the same notation, A is the dividend and B is the divisor. However, in next two algorithms we do not assume B has been normalized.

Exact Division: Given an n-bit dividend, A, and an m-bit divisor B that exactly divides A, find Q such that $A = Q \cdot B$.

The division algorithms presented thus far all work by finding a Q^* that when multiplied by B and shifted, cancels off the most significant bits (left side) of the dividend, A. Then these algorithms construct a smaller dividend, A', such that A'/B equals the remaining bits of the quotient still to be computed. Exact division, due to Jebelean [65] works in the opposite direction. It determines the least significant bits of the quotient and cancels them off the right side of the dividend. For now, let's assume that the divisor, B is odd. We can determine Q_k, the k least significant bits of Q, as follows:

$$A = Q \cdot B \quad \Rightarrow \quad A \equiv Q \cdot B \pmod{2^k} \quad \Rightarrow \quad Q \equiv A \cdot B^{-1} \pmod{2^k}$$

and therefore

$$Q_k = Q \bmod 2^k = A \cdot B^{-1} \pmod{2^k} = (A \bmod 2^k \cdot B^{-1} \bmod 2^k) \bmod 2^k$$

where $A \bmod 2^k$ is just the least significant k bits of A and the inverse, $B^{-1} \bmod 2^k$ will exist because B was assumed to be odd.

With this computation we can compute the least significant k bits of Q. Next we want to find a smaller A' such that A'/B are the remaining high order bits of Q still to be

discovered. Thus we want $A'/B = (Q - Q_k)/2^k$. Multiplying both sides by B gives:

$$A' = \frac{Q \cdot B - Q_k \cdot B}{2^k} = \frac{A - Q_k \cdot B}{2^k}$$

We keep iterating this process, finding the next k bits of Q, until A' is zero, at which time all of Q has been determined.

This algorithm requires an efficient technique for computing $B^{-1} \bmod 2^k$. Jebelean gives an algorithm: set $I_1 = 1$ and iterate:

$$I_{j+1} = I_j(2 - B \cdot I_j) \bmod 2^{2^{j-1}}$$

$B^{-1} \bmod 2^k$ is $I_j \bmod 2^k$ when $2^{2^{j-1}} \geq k$. This algorithm is very similar to Newton's iteration in that the accuracy doubles on each iteration, which can shown as follows. Suppose I_j is a k bit modular inverse of B. Then we have $B \cdot I_j = C \cdot 2^k + 1$ for some integer C, and we have:

$$B \cdot I_{j+1} = B \cdot I_j \cdot (2 - B \cdot I_j) = (C \cdot 2^k + 1)(1 - C \cdot 2^k) = -C^2 \cdot 2^{2k} + 1$$

Thus $B \cdot I_{j+1} \bmod 2^{2k} = 1$, and therefore I_{j+1} is a $2k$ bit modular inverse of B. The origin of this approach can be traced back to Kurt Hensel's work on *p-adic* numbers, circa 1897.

We have assumed that B is odd. Suppose B is divisible by 2^p but not 2^{p+1} for some $p > 0$. Then because the division is exact, 2^p also divides A and we can compute $Q = \text{exact\_divide}(A/2^p,\ B/2^p)$ where $B/2^p$ will be odd. Dividing by 2^p to ensure that $B/2^p$ is odd is the "least significant bit first" equivalent of normalizing the divisor.

Exact division is very efficient for a number of reasons. First, it can take advantage of asymptotically fast multiplication. Second, the modular inverse step is faster than Newton reciprocal computation. Third, there are no correction steps as there are with the prior division algorithms.

Short Division / Division by Single Word Constants: In short division the divisor B is only a single word value, i.e., $B < \beta$. If the division is exact, the fastest approach is to use exact division, with k equal to the machine word size. Exact short division is required to implement Toom-Cook.

If the division is not exact, then there are two approaches. If the machine has a fast division instruction, then the grade school division algorithm will be fast and efficient, and no correction steps are required. However, some architectures, such as GPUs, do not have fast division instructions. In this case, it is often faster to compute $r = A \bmod b_0$ followed by $Q = \text{exact\_divide}(A - r, b_0)$. A fast single word remainder algorithm is given at the end of Section 2.2.7.

2.2.6 Remainder / Modulo Reduction

The division algorithms of Section 2.2.4 compute both the quotient and the remainder. However, there are a few applications where just the remainder is required. In most cases, there is a fixed modulus, M, and repeated computations modulo M are required. This is the case for modulo exponentiation and computations in a finite field $\mathbb{F}_P$, where P is a prime or a prime raised to the k^{th} power.

Barrett Reduction: The Barrett reduction is due to Paul Barrett [5]. Given a $2n$-word value X, and an n-word value M, where $\beta/2 \leq M_{n-1} < \beta$, find $X \bmod M$. Since M will be used repeatedly, we precompute I such that $I \cdot M \leq \beta^{2n} < (I+1) \cdot M$. The precomputation can be done using a Newton iteration as described in Section 2.2.4.

The algorithm computes $Q = \lfloor \lfloor X/\beta^n \rfloor \cdot I/\beta^n \rfloor$, then computes $R = X - Q \cdot M$. This algorithm can be viewed in terms of fixed point representation, where I has n words of precision after the decimal point, and is the reciprocal of M. With some careful analysis, it can be shown that $0 \leq R < 4M$. Thus, at most three correction steps are required to compute the final remainder.

The Barrett reduction is quite fast in practice because, after the precomputation step, the algorithm doesn't require any division instructions, which are quite slow on many architectures. Further, the correction steps are outside of the multiplication loops and are done at most thrice per reduction.

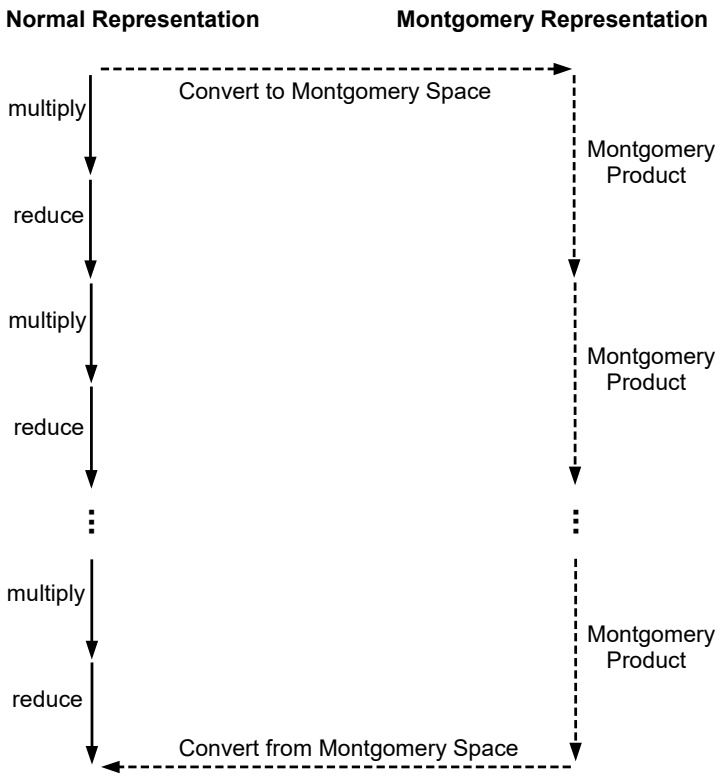

Figure 2.2. Computation using Montgomery Representation

Montgomery Reduction: The main idea of a Montgomery reduction is to transform the input to a different representation (Montgomery Representation), perform the computation in the new representation and at the end transform the output back to the original FRNS representation. This process is shown in Figure 2.2, where the dashed lines represent computation in Montgomery representation.

The mapping is straightforward. In the FRNS domain, we have an n-word MP value, A. In the Montgomery domain, it's the n-word MP value $A \cdot \beta^n \bmod M$ where M is the common modulus for the reductions. We note that M must be odd and less than β^n. We also use the convention that $\overline{A}$ means the value A mapped to the Montgomery domain.

We define two functions, $\mathrm{ToMont_M}(A)$ maps values from FRNS representation to Montgomery representation and $\mathrm{FromMont_M}(\overline{A})$ maps them back. Formally:

$$\overline{A} = \mathrm{ToMont_M}(A) = A \cdot \beta^n \bmod M$$

and

$$A = \mathrm{FromMont_M}(\overline{A}) = \overline{A} \cdot \beta^{-n} \bmod M$$

Next, we wish to define the product in the Montgomery domain $\mathrm{MontProd_M}(\overline{A},\ \overline{B})$ such that it preserves modular multiplication, i.e.,

$$\mathrm{MontProd_M}(\overline{A},\ \overline{B}) = \mathrm{ToMont_M}(A \cdot B \bmod M)$$

because $A = \overline{A} \cdot \beta^{-n} \bmod M$ and $B = \overline{B} \cdot \beta^{-n} \bmod M$, we have:

$$\begin{aligned}
\mathrm{MontProd_M}(\overline{A},\ \overline{B}) &= \mathrm{ToMont_M}(A \cdot B \bmod M) \\
&= \mathrm{ToMont_M}(\overline{A} \cdot \beta^{-n} \cdot \overline{B} \cdot \beta^{-n} \bmod M) \\
&= \overline{A} \cdot \beta^{-n} \cdot \overline{B} \cdot \beta^{-n}\beta^n \bmod M \\
&= \overline{A} \cdot \overline{B} \cdot \beta^{-n} \bmod M
\end{aligned}$$

In [83] Montgomery gives two efficient algorithms for computing $\mathrm{MontProd_M}(\overline{A},\ \overline{B})$. The clever idea behind the Montgomery algorithm is to note that if we can find Q such that $\overline{A} \cdot \overline{B} + M \cdot Q$ is exactly divisible by β^n where $0 \leq Q < \beta^n$, then we can define $\overline{P} = \frac{\overline{A} \cdot \overline{B} + M \cdot Q}{\beta^n}$ and we have:

$$\begin{aligned}
\mathrm{MontProd_M}(\overline{A},\ \overline{B}) &\equiv \overline{A} \cdot \overline{B} \cdot \beta^{-n} && (\mathrm{mod}\, M) \\
&\equiv (\overline{A} \cdot \overline{B} + M \cdot Q) \cdot \beta^{-n} \\
&\equiv \overline{P} \cdot \beta^n \cdot \beta^{-n} \\
&\equiv \overline{P}
\end{aligned}$$

and if we assume $\overline{A} < M$ and $\overline{B} < M$, then $\overline{P} = \frac{\overline{A} \cdot \overline{B} + M \cdot Q}{\beta^n} < \frac{M^2 + \beta^n \cdot M}{\beta^n} < 2M$. At a high level, the Montgomery algorithm runs the steps given in Figure 2.3.

1. Compute $P = \overline{A} \cdot \overline{B}$
2. Find Q such that $P + Q \cdot M$ is evenly divisible by β^n
3. Compute $\overline{P} = (P + M \cdot Q)/\beta^n$
4. If $\overline{P} > M$ then $\overline{P} = \overline{P} - M$ (correction step)
5. Return $\overline{P}$

Figure 2.3. Montgomery Product Algorithm

Montgomery gives two versions of the algorithm to find Q. The first, which we'll call the "three-products" approach, precomputes $I = \beta^n - \left(M^{-1} \bmod \beta^n\right)$. Then $Q = \overline{A} \cdot \overline{B} \cdot I \bmod \beta^n$ and $\overline{P} = \frac{\overline{A} \cdot \overline{B} + M \cdot Q}{\beta^n}$. This algorithm does three n-word by n-word multiplies and can be implemented with asymptotically fast multiplications.

The second algorithm uses a quadratic "word-by-word" approach. It first precomputes $i = \beta - \left(M^{-1} \bmod \beta\right)$, then sets $\overline{P}_0 = \overline{A} \cdot \overline{B}$ and loops for $j = 1$ to n:

$$\begin{aligned} q_j &= \overline{P}_{j-1} \cdot i \bmod \beta \\ \overline{P}_j &= \frac{\overline{P}_{j-1} + q_j \cdot M}{\beta} \end{aligned}$$

the final result is $\overline{P} = \overline{P}_n$. This algorithm is implemented using an n-word by n-word multiplication to compute $\overline{P}_0$ and then $n^2 + n$ word multiplications to reduce it. For very large n, the three-products approach is faster. For small to moderate size n, the word-by-word approach is faster.

There are a few implementation tricks. Conversion to and from Montgomery representation can be implemented by precomputing $C = \beta^{2n} \bmod M$, then $\mathrm{ToMont_M}(A) = \mathrm{MontProd_M}(A,\ C)$ and $\mathrm{FromMont_M}(\overline{A}) = \mathrm{MontProd_M}(\overline{A},\ 1)$. The modular inverse can be computed using the *p-adic* algorithm from Section 2.2.5.

The correction step is slow and can lead to non-uniform running times (which is a serious issue in cryptosystems). Several papers explore relaxing the normalization requirement that $\overline{X}$ be less than M. In [110] Yanik, Savaş and, Koç show that if $\overline{X}$ is allowed to range

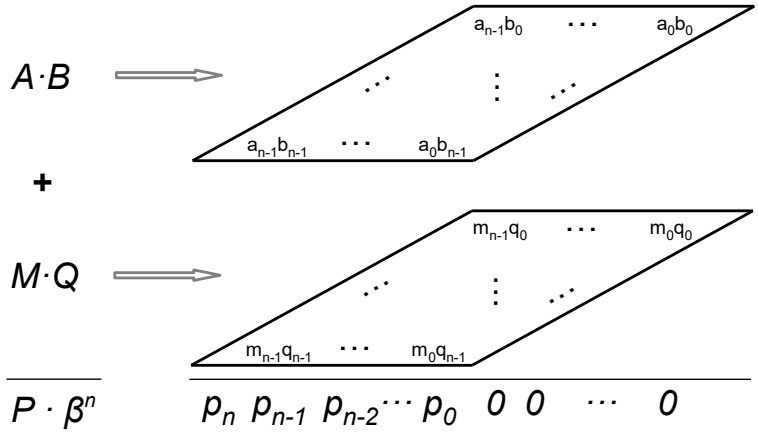

Figure 2.4. Product terms in the word-by-word approach to a Montgomery reduction

from 0 to $\beta^n - 1$, then the correction step can be simplified to if $\overline{P} \geq \beta^n$ then $\overline{P} = \overline{P} - M$. Allowing $\overline{X}$ to range from 0 to $\beta^n - 1$ is sometimes referred to as *Almost Montgomery* representation. Orup [87] and later Walter [107] show that if $\overline{X}$ is allowed to range from 0 to $2M - 1$ then the correction step can be removed entirely. Walter also shows that if $\overline{X} < 2M$, then $\mathrm{FromMont_M}(\overline{X}) < M$ provided that $M < \beta^n - 1$.

In [74] Koc, Acar, and Kaliski look at five different strategies for ordering the summation of the terms in the Montgomery word-by-word approach. The terms are shown in Figure 2.4. The authors call these five strategies: Separated Operand Scanning (SOS), Coarsely Integrated Operand Scanning (CIOS), Finely Integrated Operand Scanning (FIOS), Finely Integrated Product Scanning (FIPS) and Coarsely Integrated Hybrid Scanning (CIHS).[1] In the *separated* approach, the product $A \cdot B$ is computed in full and then reduced by adding $M \cdot Q$. Shifting off the n zeros is always the last step. In the *coarsely integrated* approach, we alternate computing a row or column of the product, $A \cdot B$, followed by a row or column of the reduction, $M \cdot Q$. In the *finely integrated* approach, we alternate, adding a single term

[1] CIHS is rarely used and we omit it for brevity

SOS: $P = \underbrace{b_0 A + b_1 A\beta + \cdots}_{A \cdot B} + \underbrace{q_0 M + q_1 M\beta + \cdots}_{M \cdot Q}$

CIOS: $P = \underbrace{b_0 A}_{\text{row}} + \underbrace{q_0 M}_{\text{row}} + \underbrace{b_1 A\beta}_{\text{row}} + \underbrace{q_1 M\beta}_{\text{row}} + \cdots$

FIOS: $P = \underbrace{b_0 a_0 + q_0 m_0 + b_0 a_1 \beta + q_0 m_1 \beta + \cdots}_{\text{finely integrated row}} + \underbrace{b_1 a_0 \beta + q_1 m_0 \beta + b_1 a_1 \beta^2 + q_1 m_1 \beta^2 + \cdots}_{\text{finely integrated row}} + \cdots$

FIPS: $P = \underbrace{a_0 b_0 + q_0 m_0}_{\text{finely integrated col}} + \underbrace{a_0 b_1 \beta + q_0 m_1 \beta + a_1 b_0 \beta + q_1 m_0 \beta}_{\text{finely integrated col}} + \cdots$

Figure 2.5. Summary of SOS, CIOS, FIOS, FIPS

of the product and single term of the reduction. In *operand scanning* the outermost loop moves through the words of one of the arguments (usually B) and in *product scanning* the outer loop moves through the words of the product. Geometrically, *operand scanning* goes across the rows of Figure 2.4 from right to left in top to bottom order, while *product scanning* goes down the columns from right to left. Figure 2.5 summarizes the computations for SOS, CIOS, FIOS, and FIPS. The importance of [74] is not so much for its particular experimental results, but that it raises awareness of the different ways to organize the Montgomery computation and these will turn out to be very important in GPU implementations. The best strategy is highly dependent on the features of the underlying hardware, and performance can vary significantly between strategies.

For moderately large n, Montgomery reductions can be accelerated using Svoboda preconditioning, also known as Montgomery folding. This is well described by Brent and Zimmermann [16] in their Section 2.4.2. For very large n, Montgomery reductions can be significantly accelerated using clever FFT multplication tricks as discovered by McLaughlin [88].

The Montgomery reduction is very popular because it is fairly easy to implement and is quite fast. It doesn't require any division instructions after the precompute phase and there is at most one fast correction step per reduction.

2.2.7 Special Moduli

There are several special moduli forms that have very fast modulo reduction algorithms. The most common example, for a binary FRNS, is to replace "$\bmod 2^k$" with a bit-wise *logical and* with $2^k - 1$. But there are other less well known examples that frequently appear in multiple precision arithmetic. Here we look at a few of them.

Special Modulus $2^k - 1$: we split X into k bit chunks, $X = (x_0, x_1, \cdots, x_{n-1})$ such that $0 \leq x_i < 2^k$ and $X = \sum_{i=0}^{n-1} x_i 2^{ik}$. Since $2^{ik} \bmod (2^k - 1) = 1$, for all i, we can rewrite the equation as:

$$X \bmod (2^k - 1) = (x_0 + x_1 + x_2 + ... + x_{n-1}) \bmod (2^k - 1)$$

Primes of this form are called Mersenne primes.

Special Modulus $2^k + 1$: again, split X into k bit chunks, $X = (x_0, x_1, \cdots, x_{n-1})$. Here we note $2^k \bmod (2^{ik} + 1)$ is -1 when i is odd and +1 when i is even. Thus

$$X \bmod (2^k + 1) = (x_0 - x_1 + x_2 - x_3 + ...x_{n-1}) \bmod (2^k + 1)$$

This is used extensively in the Schönhage-Strassen algorithm. Primes of this form are called Fermat primes.

Special Modulus $2^{2k} - 2^k + 1$: Here we consider the case where X is exactly $4k$ bits in length, split into 4 chunks of k bits, $X = (x_0, x_1, x_2, x_3)$. For brevity, let $m = 2^{2k} - 2^k + 1$, and we note $2^{3k} \bmod m = -1$ and $2^{2k} \bmod m = 2^k - 1$. Thus we can rewrite X as follows:

$$\begin{aligned} X &\equiv 2^{3k}x_3 + 2^{2k}x_2 + 2^k x_1 + x_0 \qquad (\bmod m) \\ &\equiv (-1)x_3 + (2^k - 1)x_2 + 2^k x_1 + x_0 \\ &\equiv 2^k(x_1 + x_2) - x_3 - x_2 + x_0 \end{aligned}$$

It turns out, for $k = 32$, the number $2^{2k} - 2^k + 1$ is prime! Since $k = 32$, x_0, x_1, x_2 and x_3 are all 32-bit values. Thus we can do extremely efficient computations in the finite field $\mathbf{F}_p$ (where $p = 2^{64} - 2^{32} + 1$) on a 32-bit machine.

Generalized Mersenne Moduli: There are many primes of the form $2^n \pm 2^m \pm 1$. These are referred to as generalized Mersenne primes, and they are frequently used in cryptographic algorithms. Several standard ones are used by NIST because they have fast modulo operations and are computationally efficient. See Solinas [95] for a thorough discussion.

Single Word Remainder: Here we wish to compute $X \bmod m$, for an n-word dividend X and a single word modulus m. There are a number of ways to do this. If the machine has a fast division instruction, then simply use the grade school algorithm, noting that correction steps are not needed, and keep the final remainder. If division is slow, as is the case on GPUs, then the following algorithm is efficient. Precompute $c = \beta^2 \bmod m$. Next, while X is 3 or more words in length, run the following steps:

1. Compute $T = x_{n-1} \cdot c + x_{n-2}\beta + x_{n-3}$
2. If $T \geq \beta^2$ then set $T = T - \beta^2 + c$
3. Replace the top 3 words of X with T

We note that after step 2, $T < \beta^2$, thus each iteration reduces the length of X by one word, without changing the value of $X \bmod m$. After the loop, return $X \bmod m$, which is fast to compute because X is at most 2 words in length. This algorithm has many similiarites to Svoboda preconditioning, see [98].

2.2.8 Square Root Algorithms

The square root algorithms are quite similar to the division algorithms. There is the classic $O(n^2)$ algorithm, which computes a fixed number of bits of the square root with each iteration, and is presented below. In addition there is asymptotically faster divide and conquer algorithm, which is essentially an integer version of a Newton iteration. In the interest of brevity, we omit the divide and conquer algorithm and refer the interested reader to Brent and Zimmermann's [16] Section 1.5.

We begin with a $2n$-word value, X and we wish to find the n-word square root, $S = \lfloor\sqrt{X}\rfloor$. As with division, square root has a normalization criteria, where we assume that $\beta^{2n}/4 \leq X < \beta^{2n}$. This algorithm is an iterative approach, where each iteration solves for the next word of the square root. We introduce the following notiation, let X_k be the value of the k most significant words of X, i.e.,

$$X_k = x_{2n-1}\beta^{k-1} + x_{2n-2}\beta^{k-2} + ... + x_{2n-k+1}\beta + x_{2n-k}$$

and likewise, S_k as the k most significant words of S:

$$S_k = s_{n-1}\beta^{k-1} + s_{n-2}\beta^{k-2} + ... + s_{n-k+1}\beta + s_{n-k}$$

Thus on iteration k, the algorithm has solved $S_k = \lfloor\sqrt{X_{2k}}\rfloor$.

At each step we define the remainder, R_k, to be $X_{2k} - S_k^2$, where $0 \leq R_k \leq 2S_k$. The algorithm works as follows, given $S_k = \left\lfloor\sqrt{X_{2k}}\right\rfloor$, find the next word $s < \beta$ such that $S_{k+1} = S_k\beta + s = \lfloor\sqrt{X_{2k+2}}\rfloor$. We can write X_{2k+2} in terms of X_{2k} or in terms of S_{k+1} and we have:

$$\begin{aligned}
X_{2k+2} &= X_{2k}\beta^2 + C && \text{where } C < \beta^2 \\
&= (S_k^2 + R_k)\beta^2 + C \\
&= S_k^2\beta^2 + R_k\beta^2 + C \\
X_{2k+2} &\approx S_{k+1}^2 \\
&\approx S_k^2\beta^2 + 2S_k s\beta + s^2
\end{aligned}$$

Thus $R_k\beta^2 + C \approx 2S_k s\beta + s^2$ and a good approximation for s is $s^* = \frac{R_k\beta}{2S_k}$. Further, since $s^* < \beta$ and $R_k \leq 2S_k$ we have:

$$s^* = \frac{R_k\beta}{2S_k} \approx \frac{\lfloor R_k/\beta^{k-2}\rfloor}{2\lfloor S_k/\beta^{k-1}\rfloor} \approx \frac{\lfloor R_k/\beta^{k-2}\rfloor/2}{S_1}$$

This is very efficient to compute, it's just a two word numerator divided by a single word divisor. Empirically, we find that if X is normalized, then s^* is never more than 2 correction steps from s. Since the k^{th} iteration relies on S_1 and R_k, an initial computation is required to compute $S_1 = \left\lfloor\sqrt{x_{2n-1}\beta + x_{2n-2}}\right\rfloor$ and $R_1 = x_{2n-1}\beta + x_{2n-2} - S_1^2$. This can be achieved

using a floating point square root and appropriate correction steps or a simple binary search on the bits of the square root.

2.2.9 Modular Exponentiation Algorithms

Modular exponentiation is the process of computing $A^K \bmod M$ where A, and M are n-bit numbers, and K is a m-bit number. We describe three algorithms: the classic exponentiation by squaring algorithm, fixed window exponentiation and sliding window exponentiation. All three algorithms have the same asymptotic complexity, as each requires m squaring steps, but the windowed algorithms use fewer multiplication steps and are faster in practice.

The two windowed algorithms both precompute a table, T, with 2^w window entries, where w is called the window size. The i^{th} entry of the table is initialized with $T[i] = A^i \bmod M$.

Exponentiation by Squaring: The exponentiation by squaring algorithm begins by initializing the result, R, to 1 and the current square term, S, to $A \bmod M$. It loops through the bits of the exponent from least significant to most significant. If the bit of the exponent is set, then it updates $R = R \cdot S \bmod M$. The algorithm then updates $S = S \cdot S \bmod M$. When the last bit of the exponent has been processed, the algorithms returns R. This algorithm does m squaring steps and on average $m/2$ multiplication steps.

Fixed Window Exponentiation: This algorithm starts by precomputing the window table, then breaks K into w-bit chunks, i.e., $K = (k_0, k_1, \cdots, k_{\lceil m/w \rceil - 1})$ where $K = \sum_{i=0}^{\lceil m/w \rceil - 1} k_i 2^{iw}$, and $0 \leq k_i < 2^w$. Next, the algorithm sets R to 1 and loops for $i = \lceil m/w \rceil - 1$ down to 0, updating $R = R^{2^w} \cdot T[k_i] \bmod M$. Once the exponent has been processed, the algorithm returns R. This algorithm runs with roughly $2^{w-1} + m$ squaring steps and $2^{w-1} + m/w$ multiplication steps.

Sliding Window Exponentiation: The sliding window algorithm is similar to the fixed window algorithm, in that it processes the exponent from the most significant bit to least

significant. However, it uses a slightly different approach. If the next bit of the exponent is a zero, then the algorithm can immediately square R and move on to the next bit, however, if it's a one, the algorithm extracts w bits from the exponent, which are used as an index of T, and updates $R = R^{2^w} \cdot T[i]$. Once the exponent has been processed, it returns R.

In practice, the algorithm is a bit more complicated to ensure that it doesn't grab bits past the end of the exponent. But when implemented carefully, this algorithm does roughly m squaring steps and $2^{w-1} + m/(w+1)$ multiplication steps.

2.3 Parallel Algorithms for Multiple Precision Arithmetic

By and large, parallel multiple precision arithmetic algorithms are just parallel versions of the sequential algorithms, but here we present a few parallel algorithms that are surprising and operate in a different way than their sequential counterparts. In these algorithms, we assume p is the number of processors and that the processors are connected with some sort of efficient inter-process communication without worrying about the details of the exact mechanism (shared memory, mesh network, packet switched network, shared bus, etc).

2.3.1 Addition and Subtraction / Carry Resolution

Parallel multiple precision addition and subtraction are interesting problems and the solutions have been well studied in the guise of hardware circuits. Here we wish to compute the sum of two n-word values, $S = A \pm B$ (using two's complement in the case of subtraction), where the A and B arguments have been split into p (the number of processors) pieces of size n/p. We assume that processor i has been assigned A_i and B_i and is responsible for computing $S_i = A_i \pm B_i$.

The first step is for all the processors to compute S_i in parallel, using the sequential algorithm, and we note that the sum might carry out, which we store in a separate variable c_i. To get the final (non-redundant) S, we need to resolve the carries.

There are two common algorithms for resolving carries. The simplest algorithm is ripple carry resolution, and it works like this: while any $c_i \neq 0$, each processor computes $S_i = S_i + c_{i-1}$ and updates their c_i according to the new carry out. In the average case, this will resolve the carry in just a few iterations. However, in the worst case, a carry can ripple from the first processor all the way to the last, and thus the algorithm is $O(p-1)$. This algorithm is quite similar to a hardware Ripple Carry Adder.

The second algorithm uses a generate and propagate approach. Set $g_i = c_i$, set p_i to one if $S_i = \beta^{n/p} - 1$ and zero otherwise. Next, we iterate for $k = 1, 2, 4, 8, \cdots, p/2$: update $g_i = g_i \vee (p_i \wedge g_{i-k})$ and update $p_i = p_i \wedge p_{i-k}$. After the loop, the final step is to set $S_i = S_i + g_{i-1}$. This algorithm runs in $\Theta(\log p)$ steps, and is equivalent to a Hierarchial Carry Look-ahead Adder in hardware.

Finally, we note that if there are k multiple precision values to be summed, $X_1 + X_2 + \cdots + X_k$, then we need not resolve the carries after each addition. Instead we can accumulate the carries locally and then resolve them all at once after the last addition. In other words, we leave the sum S in redundant form, and wait to resolve the carries until the sum is needed in normal form, which is sometimes referred to as lazy carry resolution. In hardware, it's called a Carry Save Adder [34].

2.3.2 Multiplication

The multiplication algorithms of Section 2.2.2 are easily parallelized using standard techniques and there are no special tricks or unique parallel algorithms other than to use a redundant representation with local carry accumulation (as described above), with a final carry resolution step at the end of the multiplication. Several parallel multiplication implementations are discussed in sections 2.4 and 2.5.

2.3.3 Division

The sequential division algorithms (Section 2.2.4) are much more difficult to parallelize. They all involve an iterative process: compute a portion of the quotient, then con-

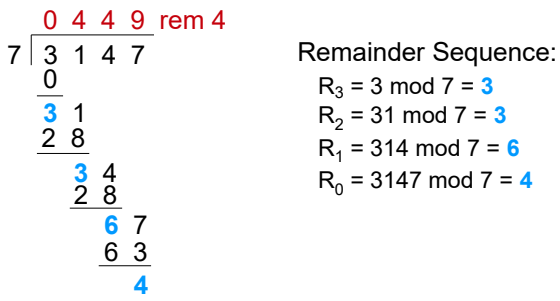

Figure 2.6. Short Division Example

struct a smaller division problem A'/B for the remaining quotient bits, which is inherently sequential. That said, division algorithms are generally constructed out of addition, subtraction, and multiplication of MP values. These sub-operations can all be parallelized.

The two special case division algorithms, Exact Division and Short Division, do have interesting parallel algorithms.

Bidirectional Exact Division: Krandick and Jebelean [76] propose a bidirectional exact division algorithm. In this approach, two processors work in parallel on an exact division instance. One processor finds the lower half of the quotient, Q_L, using Jebelean's least significant bit first exact division algorithm (described in Section 2.2.5) and the other processor uses a most significant bit first algorithm (similar to grade school division, but computing only a truncated remainder) to find the upper half of the quotient, Q_H.

Parallel Short Division: In [101] Takahashi gives a parallel algorithm for short division. Given an n-word dividend X divided by a single word divisor $d < \beta$ find an n-word Q and r such that $X = d \cdot Q + r$. Consider Figure 2.6 which shows an example of a short division, with $n = 4$. We begin by assuming that p (the number of processors) is n. Next, let's define the remainder sequence as the remainders that you get from doing grade school division, these are shown in blue on the figure. Takahashi realized that the quotient, q_k could be computed from the remainder sequence r_k in a single step as follows:

$$q_k = \left\lfloor \frac{r_{k+1}\beta + x_k}{d} \right\rfloor$$

the trick was to find a fast parallel computation for the remainder sequence. Figure 2.6 shows that r_k can be computed from the most significant $n-k$ digits (or words) of X mod d. Formally:

$$r_k = \Big(\sum_{i=k}^{n-1} x_i \beta^{i-k} \Big) \bmod d$$

and we can write these sums out for our four term example:

$$\begin{aligned} r_0 &= (x_3\beta^3 + x_2\beta^2 + x_1\beta + x_0) \bmod d \\ r_1 &= (x_3\beta^2 + x_2\beta + x_1) \bmod d \\ r_2 &= (x_3\beta + x_2) \bmod d \\ r_3 &= x_3 \bmod d \end{aligned}$$

These equations have a nice structure and can be computed as follows: set $r_k = x_k \bmod d$. Then update $r_k = (r_{k+1}\beta + r_k) \bmod d$, where over indexing, i.e., r_4, r_5, etc. is assumed to produce zero. After the update we have:

$$\begin{aligned} r_0 &= (x_1\beta + x_0) \bmod d \\ r_1 &= (x_2\beta + x_1) \bmod d \\ r_2 &= (x_3\beta + x_2) \bmod d \\ r_3 &= x_3 \bmod d \end{aligned}$$

then update $r_k = (r_{k+2}\beta^2 + r_k) \bmod d$ and we have our desired final result. At each step double the jump $r_{k+1},\ r_{k+2},\ r_{k+4}, \cdots$ and double the power, $\beta,\ \beta^2,\ \beta^4, \cdots$. Thus, this is an $O(\log n)$ algorithm.

Next, we consider the case of $n > p$. Without loss of generality, let's assume that n is an exact multiple of p. We break X into p chunks of n/p words, thus each processor has the responsibility for n/p words of the remainder sequence r_k. Takahashi's parallel remainder sequence algorithm is presented in Figure 2.7. As we can see from the algorithm, the local n/p elements of the remainder sequence are updated log p times. Thus the running time

```
k = ... get my processor id (0 .. p-1) ...
m = n/p

// initialize my portion of the r vector
for(i=0; i<m; i++)
  r[i] = (X[k*m + i] * beta^i) % d

for(j=1; j<=p; j=2*j) {
  if(k + j >= p)
    upd = 0;
  else
    upd = ... get r[m-1] from processor k+j ...

  // update my portion of r vector with remainder from processor k+j
  upd=upd * beta^(j*(m-1)+1) % d
  for(i=0; i<m; i++)
    r[i] = (r[i] + upd*beta^i) % d
}
```

Figure 2.7. Takahashi's parallel remainder sequence algorithm

for this algorithm is $O\big((n/p)\log p\big)$ which is the overall running time for Takahashi's short division algorithm.

2.4 Parallel Multiple Precision Implementations on CPUs

In this section, we cover seven parallel multiple precision arithmetic implementations. Some implementations are used for specific interesting arithmetic computations; others are implemented as general studies of multiple precision arithmetic and parallel algorithm scaling. The papers are listed in chronological order with the first five dating from the 1990s. These papers all implement their own custom inter-process communication schemes and harken back to an earlier era of experimentation with parallel algorithms and processing, where algorithms were very much tied to one specific machine. The last two papers, Takahashi's computation of π to 2.577 trillion digits from 2010 and our smallest eigenvalue paper from 2014, are both more modern, using the MPI and OpenMP standards, and which should be easily portable to a wide range of HPC cluster machines.

In [108] Weber implements addition, subtraction, and parallel multipication and tests the implementations on two shared memory machines, an 8 processor Encore Multimax and a 26 processor Sequent Balance. Weber starts with a Fortran multiple precision package written by David Bailey, but modifies the multiplication algorithm to use a parallel FFT

implementation based on double precision floating point numbers with 16-bit FFT samples. Weber uses his routines to prove that the 15$^{\text{th}}$ Fermat number, $F_{15} = 2^{2^{15}} + 1$, has at least 4 prime factors. One thing to note: Weber does not discuss the potential for round-off error in the floating point computations to impact the multiplication results, which could be because the numbers he's multiplying are relatively small (roughly 10,000 decimal digits in length).

In [44] Fagin implements large integer multiplication on a 32K Connection Machine, using FFT multiplication in the ring modulo $2^{2^b} + 1$, where the choise of b depends on the size of the numbers being multiplied. This is the same FFT technique as the Schönhage-Strassen algorithm, except that it uses a quadratic grade-school algorithm to do the component-wise multiplications required for the FFT convolution, whereas Schönhage-Strassen uses recursive calls to itself for the multiplications. Fagin uses his code to multiply numbers that range in size from 2K-bits to 8M-bits with corresponding running times from 50 milliseconds to 2000 milliseconds.

In [19] Char, Johnson, Saunder, and Wick (CJSW) implement addition, subtraction, and parallel multiplication on a network of 27 Sun IPC workstations. Addition and subtraction are fast operations and are handled locally using the sequential algorithms. For multiplication, if both multiplicands are less than 6000 decimal digits, they run the sequential grade school algorithm locally. If not, they use a parallel version of Karatsuba. They have a couple of schemes that they evaluate, but essentially, they all work as follows. The system runs a shared work queue, where tasks can be queued. Each of the 27 workstations then runs a worker process, which gets assigned tasks to exectue. The Karatsuba multiplier queues four tasks: three sub-products (that would normally be handled with recursion) and a merge task that can only be run after the three sub-products are complete. The result of the Karatsuba multiplication is the result of the merge task. CJSW use their routines to compute $\sqrt{2}$ out to 1,000,000 decimal places, and achieve a speed up of about 15x over the computation on a single workstation.

In [18] Cesari and Maeder implement Karatsuba multiplication on an Intel Paragon with 96 processors. They explore several strategies, the two fastest are covered here. The first has a master process that forks $3^d - 1$ slave processes, where d is the depth. The master process walks the Karatsuba recursion tree to a depth of d and for each leaf kicks off a multiplication on one of the slave processes, except for the last leaf which is run locally. Once the slave processes have completed their computation (using recursive Karatsuba) and forwarded their results to the master process, the master process walks back up its Karatsuba recursion tree and generates a final result. The second strategy uses a fork/join approach where two processes are forked to compute two sub-products and the third sub-product is computed locally. When a certain cut-off criterion is reached, the algorithm switches from forking to local recursion for the sub-products. Cesari and Maeder test the performance of their approach on numbers sized from 2^{10} words to 2^{20} words.

In [66] Jebelean implements multiple precision parallel multiplication and division on an 18 processor shared memory Sequent Symmetry system. There are three implementations of the multiplication routine. The first implementation is a quadratic grade school multiplier. The second is based on Karatsuba and uses three recursive calls to the multiplier to compute the half sized products. The third implementation is also based on Karatsuba, but instead of doing three recursive calls, it forks two processes to compute two of the sub-products, and the third is computed in the local process using a recursive call to the multiplier. Jebelean explores two heuristics for choosing which implementation to call. In the first heuristic, the forking Karatsuba multiplier is used for the first two levels of Karatsuba and all the other levels use recursion. In the second heuristic, the forking Karatsuba multiplier is called if the arguments are more than 24 words long, the recursive multiplier is called if the arguments are between 6 and 24 words in length, and anything less than 6 uses the quadratic multiplier. Jebelean implements division using a recursive divide and conquer algorithm (a forerunner to Burnikel-Ziegler algorithm of Section 2.2.4). The division algorithm is sequential, but whenever it requires multiplication, it uses the parallel

multiplier. Jebelean tests his multiplication routine with numbers from 10 to 500 words in length. At 30 words, he finds a speedup of 3x compared to the single processor version and at 500 words the speedup is 8x. For similar sized divisors, the speedup ranges from 1.1x to 3.4x.

In [102] Takahashi implements five basic arithmetic operations (addition, subtraction, multiplication, division, and square root) for a cluster using Fortran, OpenMP, and MPI to compute π to 2.5 trillion decimal places. He uses two different algorithms to compute π and compares the results to ensure there were no errors in the computations. Takahashi uses an FRNS, but instead of the standard $\beta = 2^{32}$, uses a base 10 system with $\beta = 10^8$. Addition and subtraction are done in parallel with local carry/borrow accumulation. The carries are then resolved using a ripple approach. Takahashi notes that since the digits of π are essentially random, it never takes more than a few steps to resolve the carries. Multiplication is implemented using FFTs on a double precision floating point "balanced representation" of $\mathbb{R}$ to minimize errors (see Crandall and Fagin [29] for details). Takahashi puts 4 decimal digits in each FFT sample and has a maximum FFT size of $3 \cdot 5^2 \cdot 2^{33}$, allowing him to represent products up to 2.577 trillion decimal digits in length. Division and square root are implemented using Newton iterations and take advantage of the Karp and Markstein trick [70]. The Newton iteration is just a sequence of multiple precision adds, subtracts, and multiplies, and these are all done using the parallel primitives. Takahashi runs his algorithms on a 640 node Appro Xtreme-X3 machine, which completes the calculation and verification in 73 hours and 36 minutes elapsed time.

In [36] we report on a parallel algorithm for finding the smallest eigenvalue of a family of extremely ill-conditioned Hankel matrices, where the condition number grows much faster than N^N where N is the size of the matrix. Because the matrix is so ill-conditioned, the computations must be performed using arbitrary precision integer fixed point arithmetic. We explore four different algorithms for the eigensolver (Lanczos, Householder, the Jacobi Method, and a direct secant method on the characteristic polynomial). We find

that the direct secant method is by far the fastest, but it finds only the smallest eigenvalue, whereas other approaches, in particular the Jacobi method (which was second fastest), find all the eigenvalues. For this problem, we are interested in only the smallest eigenvalue and we built a parallel version of the secant algorithm using GMP for the arbitrary precision arithmetic, and OpenMP and MPI to distribute columns of the matrix across a cluster of machines. The matrices have a parameter, β, which along with N, the matrix size, determine the exact matrix elements. Our biggest run was for $\beta = 7/4$ and a 2500 by 2500 matrix. We used 12K bits of precision for the secant algorithm. The computation ran for 7 hours and 33 minutes on a 440-core Intel 3255 cluster and achieved a utilization (total GMP compute time/total wall time) of 95%.

2.5 GPU Implementations

Papers concerning multiple precision arithmetic papers for GPUs generally fall into two categories. In the first, there have been many papers dedicated to accelerating cryptographic primitives. Small size arguments, up to 500 bits or so, are useful for implementing elliptic curve cryptography. Sizes from 512-bits to 4096-bits are useful for RSA and Diffie-Hellman key exchange. The second category of papers concerns building multiple precision arithmetic libraries for acceleration of general computation, such as experimental mathematics and number theory.

The remainder of this section on GPU implementations is organized as follows. In Section 2.5.1 we provide some background on multiple precision arithmetic as used in cryptographic operations. We do not attempt to explain the algorithms, as that's beyond the scope of this dissertaion, but we refer the interested reader to two textbooks, *Applied Cryptography* by Bruce Schneier [93] and *An Introduction to Mathematical Cryptography* by Hoffstein, Pipher, and Silverman [60]. Section 2.5.2 covers the early efforts, prior to 2012, to implement multiple precision cryptographic operations on GPUs, where the papers are reviewed in chronological order. Section 2.5.3 covers the later efforts, from

2012 on, which in some cases partially overlap with the work in this dissertation. Section 2.5.4 covers the papers on GPU implementations of multiple precision libraries for general computation.

As a final note, elliptic curves can be defined over a prime finite field which can be implemented with MP arithmetic as covered here. They can also be defined over finite fields of the form p^n for prime p and $n > 1$. These are implemented with polynomial arithmetic and we refer the interested reader to the following GPU papers: [9, 35, 22, 7].

2.5.1 Cryptographic Operations Requiring Multiple Precision Arithmetic

In this section we review some cryptographic operations that have compute-heavy multiple precision arithmetic at their core. These include key exchange and digital signature algorithms such as RSA [91], Diffie-Hellman (DH) [32], Kravitz's DSA algorithm [77], and elliptic curve (EC) algorithms such as ECDH [79] and ECDSA [67].

The math underpinning these algorithms is very involved, especially for the elliptic curve cryptography (ECC) algorithms. What's important for our purposes is to understand just the computational cost of the MP operation being performed at the heart of each of these algorithms, as summarized in Table 2.1. We note that some of these algorithms involve additional work, such as generating large random numbers, computing hash func-

Algorithm	On Server	On Client	typical sizes
k-bit RSA key exchange	$X^D \bmod M$	$X^E \bmod M$	k is typically 1024-4096 bits X, D, and M are k-bits in length E is typically 16-bits in length
DSA (k-bit RSA sign) DSA (k-bit RSA verify)	$X^D \bmod M$	 $X^E \bmod M$	same as above same as above
k-bit DH key exchange	$X^A \bmod P$	$X^B \bmod P$	k is typically 2048-4096 X, A, B are k-bits in length P is prime and k-bits in length
k-bit ECDH	2x ECPM: $N \cdot P$	2x ECPM: $N \cdot P$	k is typically 256-521 bits in length N is typically a k-bit number P is an EC point
k-bit ECDSA (sign) k-bit ECDSA (verify)	ECPM: $N \cdot P$	 2x ECPM: $N \cdot P$	same as above same as above

Table 2.1. Dominant computations at the heart of important cryptography algorithms

tions, and in some cases additional multiple precision operations inside a k-bit finite field, but these are generally much less time consuming than the dominant operation.

For RSA, DSA, and DH, the computation is a modular exponentiation, with the modulus $M = P \cdot Q$, where P and Q are prime and $k/2$-bits in length. On the server, the factorization of M is known (it's related to the private key), and therefore $X^D \bmod M$ can be computed using the Chinese Remainder theorem with $X^D \bmod P$ and $X^D \bmod Q$, in roughly 1/4 the time. This is known as the CRT trick and is due to Quisquater and Couvreur [90].

For ECDH and ECDSA, the core operation is an elliptic curve scalar point multiplication (ECPM). At a very high level, we can define an elliptic curve E as a set points, (X, Y), where $Y^2 \equiv X^3 + a \cdot X + b \pmod M$ for a set of parameters (a, b, M) where a and b are typically small constants and M is a k-bit prime.[2] It turns out that E (with the inclusion of a zero point) forms a finite Abelian group, where the group operator is point addition. That is to say, that if we have two points, P_1 and P_2 in E, then $P_1 + P_2$ will also be in E and there is an algorithm to perform point addition (a black box for now). Further, if P is in E, there will exist a point $-P$ in E such that $P + (-P) = 0$. In essence, E (which is defined by parameters a, b, M) forms a number system in which we can do interesting computations.

Since we have point addition, we can also define scalar multiplication (ECPM): $N \cdot P = P + P + ... + P$, where the sum has N copies of P. The naive algorithm to compute $N \cdot P$ is set $T = 0$ and repeat N times: $T = T + P$, which runs in $O(N)$ steps, but we can do better. We note that if $N = 2^j$, then we can compute $N \cdot P$ by setting $T = P$ and repeating j times: $T = T + T$. This algorithm runs in j steps rather than 2^j steps. Since any N can be decomposed into a sum of powers of two, $N \cdot P$ can be calculated by computing the powers of two and adding to the total if the corresponding bit of N is set, which runs in $O(\log N)$,

[2]There is another case where M can be of the form p^n for prime p. These are Galois fields, GF(p^n), which require polynomial arithmetic.

i.e., $O(k)$ steps. This is called the double and add algorithm and it is the exact analogue of the exponentiation by squaring algorithm of Section 2.2.9, where squaring replaces doubling and multiplication replaces addition. Likewise, the fixed window exponentiation and sliding window exponentiation algorithms also have exact analogues in ECPM.

The running time of ECPM using the double and add algorithm is k calls to point double and on average $k/2$ calls to point addition. Returning to the black box for point addition and point doubling, the details of these algorithms are presented in [60], but for our purproses, we just need to know how many MP operations are needed. It turns out point addition and point doubling do all of their computing in the finite field $\mathbf{F_M}$, where M is one of the curve parameters and is k bits in length. Table 2.2 gives the MP operation counts for various EC operations. Note, the fixed window algorithm can be faster than the double and add algorithm, but it involves computing 2^w modular inverses in $\mathbf{F_M}$ where w is the window size.

Operation	Computations in the finite field $\mathbf{F_M}$
k-bit point doubling	7 modular multiplications 5 modular squares
k-bit point addition	9 modular multiplications 2 modular squares
k-bit point multiplication double and add algorithm	11.5 k modular multiplications 6 k modular squares 1 modular inverse

Table 2.2. Operations in the finite field $\mathbf{F_M}$ EC point doubling, addition, and scalar multiplication

As a final note, since M is a fixed curve parameter, it can be chosen to have a special form where the modulo operation can be much faster than general division (see Section 2.2.7 on Special Moduli). Picking the right M can greatly improve the performance of ECC. This is why NIST publishes several standard curves, with preselected special moduli.

In the event that M is not a special modulus, computations in the finite field $\mathbf{F_M}$ can still be done efficiently using a Montgomery representation.

2.5.2 GPU Based Asymmetric Cryptography, Early Papers

The GPGPU computing era was kicked off in the early 2000s. By this time it was clear that GPUs were improving in performance at a rate much faster than CPUs and that GPUs could be used for tasks other than graphics, in particular, scientific computing. At the same time, the importance of the internet and secure web browsing through HTTPS/TLS and public key infrastructure (PKI) had been well established. So an obvious application for GPUs was to offload the computationally heavy MP arithmetic operations required to support HTTPS/PKI.

Here we review the papers covering these early attempts. We note that the GPU is inherently a batched device. So all of the papers run batches of cryptography operations in each kernel. Some of the papers assign a problem instance to each thread, others use groups of threads in a warp or block to handle problem instances.

In [84] Moss, Page, and Smart report on one of the first efforts to use GPUs for multiple precision arithmetic. They use an RNS to implement 1024-bit modular exponentiation on an NVIDIA 7800 GTX GPU, using OpenGL and GLSL (the shader language). They chose an RNS over an FRNS so they could avoid resolving carries across GPU threads, or in this case, pixels. They use a Cox-Rower architecture for Montgomery multiplication and Szabo-Tanaka for base extension (see [72] and [99] for details). Moss, Page, and Smart explore 5 different schemes for the moduli. The first scheme uses a 12-bit prime modulus per word, stored as a single precision floating point value, and 88 words to represent a 1024 bit value. The idea is that two 12-bit (integer) values can be multiplied without overflowing a single precision floating point value, and GLSL has a fast floating point **mod** function which makes the computation on the RNS components fast. The other schemes use 24 bits per word (either a single 24-bit modulus, or two 12-bit moduli packed together) and 44 words per 1024 bit value. In essense, their approach spreads a 1024-bit modular exponentiation across either 44 or 88 computing elements. The throughput results range from 46 modular exponentiations per second (12-bits per word) to 175 modular exponentiations

per second for their best tuned 24-bits per word implementation. Unfortunately, they only achieve these throughputs for batch sizes over 1000, with a corresponding latency of over 5 seconds. The high latency makes the approach impractical for HTTPS acceleration.

In [47] Fleissner implements a 192-bit modular exponentiation routine using Montgomery products and an FRNS with $\beta = 2^{24}$. Each multiple precision value consists of 8 single precision floating point numbers, each of which contains a 24-bit integer, stored in texture memory. It's not entirely clear from the paper how Fleissner computes the required 24-bit full products, but it appears that the 24-bit values are sampled into bytes and the computations are done on bytes, and finally the results are packed back into 24-bit floats and stored in the output. In contrast to Moss et al., Fleissner uses a single computing element per 192-bit modular exponentiation. To test the performance, Fleissner runs batches of 100K 192-bit modular exponentiations on three NVIDIA GPUs: a GeForce 6500, a GeForce FX 5900 Ultra, and a 7800 GTX and compares the results to the same computation on the host computers, running on Intel Pentium 4 processors. Fleissner reports a speedup of 136x to 169x. Unfortunately, Fleissner does not report the GPU running times, only the speedups compared to the CPU, so it's difficult to compare this result to others. Also, we note that 192-bit modular exponentiation is not particularly useful.

In [100] Szerwinski and Güneysu implement three sets of algorithms. The first set performs 1024-bit and 2048-bit modular exponentiation using an FRNS with $\beta = 2^{32}$ and Montgomery products using the CIOS approach (see Section 2.2.6) with an exponentiation instance per thread. The second set also performs 1024-bit and 2048-bit modular exponentiation using a warp parallel RNS approach (an exponentiation instance per warp) with 32-bit moduli. They use a Cox-Rower architecture for the Montgomery multiplication and explore several different base extension algorithms. The third set is for elliptic curve cryptography and implements 224-bit ECMP using NIST's P-224 curve (P-224 is a generalized Mersenne prime with a fast modulo operation). Szerwinski and Güneysu implement their algorithms using CUDA and run their experiments on an NVIDIA 8800 GTS GPU. They

achieve a performance of 813 1024-bit modular exponentiations per second using the CIOS approach and 439 using the RNS approach. The 2048-bit results are roughly 8x slower than the 1024-bit. For 224-bit EC routines, they achieve 1412 ECMP operations per second.

Harrison and Waldron's paper [55] focuses on RSA-1024 decryption using two 512-bit modular exponentiations and the CRT trick. They test three implementations, two are based on an FRNS with $\beta = 2^{32}$ and Montgomery reductions. The third uses an RNS approach. The first implementation runs a 512-bit modular exponentiation in each thread, but it does so using the three-products approach to Montgomery reductions, where each product runs a 16-word by 16-word multiplication. The second implementation does the same computation, but uses 16 threads working together for each instance. The threads each take one row of the multiplication and then they run a parallel reduction to sum the rows. The third approach they implement uses a parallel RNS. They explore six different schemes for the moduli that range from 12 to 32 bits in length. Their experiments on an NVIDIA 8800GTX GPU show that the instance per thread implementation achieves the highest throughput at 5536 1024-bit RSA decrypts/second, but also the highest latency. Of the parallel approaches, the best of the RNS schemes outperforms the parallel Montgomery by about 4 to 1, which is most likely due to the poor design of the parallel reductions. There are much faster approaches, as we'll see in Jang et al.'s work. One important contribution of this paper is the observation that a good implementation should use a parallel approach when the batch size is small (more overhead, but lower latency) and switch to an instance per thread approach when the batch size is large (higher latency, but higher throughput).

In [10] Bernstein et al. implement a modular multiplication routine on a GTX 295. They give performance numbers for 210-bit modular multiplication but their code supports a range from 160 to 320 bits. The paper doesn't give too many details about exactly how their GPU algorithm is implemented, but we can read between the lines. For the 210-bit result they use an FRNS with 15 limbs (i.e., $n = 15$) of 14-bits per limb (i.e., $\beta = 2^{14}$) and assign a 210-bit modular multiplication instance to each thread. Although the paper

doesn't say it explicitly, we assume they keep two or three 210-bit numbers in registers in each thread. Then they take advantage of the fact that a GTX 295 has a multiply-and-accumulate instruction that multiplies two 24-bit arguments, computes a 48-bit product, then adds a third 32-bit argument and stores the lower 32-bit of the result into a destination register. This instruction can be dispatched in a single cycle. Consider Figure 2.1 from Section 2.2.2. Using the multiply-and-accumulate instruction, Bernstein et al. can sum down the column (which has at most 15 product terms) into a 32-bit accumulator knowing that the sum cannot overflow because $15 \cdot 2^{14} \cdot 2^{14} + c_{in} < 2^{32}$ where c_{in} is the carry from the previous column and is less than 2^{18}. With enough values stored in registers and complete unrolling of the loops, a 210-bit times 210-bit multiplication can be handled in something like 285 device instructions (225 for the multiply and accumulate instructions and approximately 60 instructions to handle initialization and column post processing). Using this technique, Bernstein et al. are able to achieve a massive 481 million 210-bit modular multiplications per second on a GTX 295. The authors also report a rate of 4928 ECMP operations per second but these are for special Edward's curves and a much longer, 11797-bit scalar suitable for elliptic curve method (ECM) factoring.

In [52] Giorgi, Izard, and Tisserand build a GPU library for computing $A \pm B \bmod P$ and $A \cdot B \bmod P$ for A, B and P in the range of 160 to 600 bits, assigning an instance per thread. These computations are suitable for implementing elliptic curve cryptography. They use an FRNS representation with Montgomery products and an FIOS approach. They explore two limb sizes, $\beta = 2^{16}$ and $\beta = 2^{32}$ and a variety of strategies to locate temporary variables in registers, local memory, and shared memory. Giorgi, Izard, and Tisserand also implement EC point addition, EC point doubling and ECMP routines, using Montgomery representation for the finite field computations. One important contribution is that they identfy register pressure and the compiler's poor register management as one of the key issues affecting performance. They run their experiments on a 9800GX2 and achieve throughputs of 45.5 million 160-bit modmuls/sec, 30.3 million 192-bit modmuls/sec, 18.2

million 224-bit modmuls/sec and 12.3 million 256-bit modmuls/sec and achieve a throughput of 1972 ECMP operations/sec. Bernstein et al.'s results in [10] are roughly 20 to 25 times faster, but this is offset a bit by the fact that the GTX 295 has approximately two to six times the performance of a 9800GX2, depending on the exact code.

In [1] Antão, Bajard, and Sousa implement ECMP based on an RNS for a 224-bit underlying finite field. ECMP requires implementing $A \pm B \bmod P$ and $A \cdot B \bmod P$ where A, B, and P are 224-bit numbers. They use a Cox-Rower architecture for Montgomery multiplication and use a CRT based approach to base extension. Antão, Bajard, and Sousa test their algorithms on a GTX 285 and achieve a throughput of 9990 ECMP operations per second. Unfortunately they do not provide their modular multiplication throughput. However, Giorgi, Izard, and Tisserand [52] found that 224-bit ECMP was about 9200 times slower than 224-bit modular multiplication. If we use that as a yardstick, we can estimate that their performance is probably in the neighborhood of 92 million mod mul operations per second. Even accounting for the differences in cards and sizes, this is significantly slower than Bernstein et al.'s result.

In [86] Neves and Araujo implement 512-bit modular exponentiation using a thread per instance. They use an FRNS representation with $\beta = 2^{32}$ and Montgomery reductions. They implement and test three variants: CIOS, FIOS, and FIPS. They apply four optimization techniques to maximize throughput: 1) try to keep as much of the computation in registers as possible; 2) extensive use of CUDA PTX assembly; 3) they use school fast-squaring (grade school) to reduce the number of multiplications; and 4) extensive use of hand unrolled loops.[3] Neves and Aurajo test their implementation on a GTX 260 GPU and find that the FIOS method is fastest and achieves a throughput of 41.4K 512-bit modular exponentiations per second. Neves and Araujo also estimate a performance of 26M 512-bit modmuls per second. It's not clear if these are all modular multiplications or some com-

[3]The Open64 compiler did not allow inline PTX inside a loop, so the loops had to be unrolled by hand.

bination of modular multiplications and modular squares. Adjusting for card and size, this is probably in the neighborhood of 386 million 210-bit modular multiplications per second on a GTX 295, while slower than Bernstein et al.'s result they are at least in the ballpark.

In [64] Jang et al. implement four sizes of RSA decryption (512-bits, 1024-bits, 2048-bits, 4096-bits). They use the CRT trick, and run two half sized modular exponentiations, i.e., 256-bits, 512-bits, 1024-bits, 2048-bits respectively. They explore two parallel approaches, an RNS and an FRNS both using Montgomery multiplications. They found that the RNS implementation was always significantly slower, even with extensive optimizations. For their FRNS implementation they use $\beta = 2^{64}$ and parallelize the computation across a warp. For the small sized modular exponentiations (256-bits to 1024-bits), they pack multiple mod exp instances into each warp and they use an FIOS approach to computing the product. Carries are accumulated locally and after each Montgomery multiplication, they run a ripple carry resolution across the threads responsible for an instance. Jang et al. test their implementation on a GTX 580 and report decrypt performance of 322K/sec for RSA-512, 74.7K/sec for RSA-1024, 12.0K/sec for RSA-2048 and 1.66K/sec for RSA-4096. Comparing this work to Bernstein et al., we find that the 322K RSA-512 decrypts/sec requires a throughput of 199.6 million 256-bit modular multiplications per second. Scaling this to a GTX 295 by number of cores, clock rates, and $\left(\frac{256}{210}\right)^2$ to adjust for size, gives 223 million 210-bit modular multiplications per second. This is clearly slower than Bernstein et al. without even considering the fact that the GTX 580 has a more powerful multiplier. Comparing to Neves and Araujo's result, 74.7K RSA-1024 decrypts/sec requires 149.4K 512-bit mod exps/sec. Scaling this to a GTX 260 by number of cores and clock rate gives 47K 512-bit mod exps/sec. This is slightly faster than Neves and Araujo's result, but it's worth noting that a GTX 580 has more than twice the throughput per core per clock than a GTX 260 running 32-bit full multiplications. There are two likely explanations for the performance gaps. First, even though this approach maps well to parallel processing of a warp, there is still some overhead in inter-thread communication and carry resolution

that is not required in the instance-per-thread model. Second, there is no efficient way to implement warp parallel fast-squaring because the algorithm is irregular and some threads would have to do more processing than others. Since the majority of operations to do modular exponentiation are squares, this has a big impact on final performance.

2.5.3 GPU Based Asymmetric Cryptography, Recent Papers

In [57] Henry and Goldberg look at a different facet of the cryptography space: code breaking. They use the Pollard Rho algorithm to solve the discrete logarithm problem in a cyclic group modulo M with a smooth totient. In particular, they show that they can solve a discrete logarithm problem for a 1536-bit RSA in less than 2 minutes on a system with two M2050 GPUs, provided $P - 1$ and $Q - 1$ are $2^{55} - smooth$, which is to say $P - 1$ and $Q - 1$ have no factors greater than 2^{55}. These are particularly weak RSA keys that would not be used in practice by a good RSA implementation. It turns out the heart of their code breaking routine is high performance modular multiplication, implemented with an FRNS with $\beta = 2^{32}$ and Montgomery reductions. They assign an instance per thread and use Koç et al.'s CIOS approach. Henry and Goldberg implement their code using PTX assembly and take advantage of features such as the carry flag and the 32-bit multiply and accumulate instruction. To get around the compiler issues, Henry and Goldberg wrote Perl scripts to unroll loops containing PTX inlines. Henry and Goldberg report modmul throughputs of 840M/sec at 192 bits, 505M/sec at 256 bits, 137M/sec at 512 bits, 52M/sec at 768 bits, and 11.1M/sec at 1024 bits on a Tesla M2050 graphics card. They state that the GPU cores are clocked at 1.55 GHz, but we believe that's an error in their paper. The M2050 memory is clocked 1.55 GHz, but the cores are clocked at 1.15 GHz. If we scale the 192 bit result to a GTX 295 for number of cores, clock rate and size, we get a result of 812M 210-bit modmuls per second. If we take into consideration that an M2050 has twice the 32-bit full multiplication throughput on a per core per clock basis, then we see these results are only a bit slower than Bernstein et al.'s.

In [15] Bos investigates ECMP on a GTX 295 and a GTX 580 using the NIST P224 curve, where $P = 2^{224} - 2^{96} + 1$, which has a fast modulo reduction. The author has two concerns in his paper, throughput and latency and he tries to balance them. The approach with the highest throughput would be to assign an ECMP operation to each thread, but this would also have very high latency. Earlier, in Jang, et al., the approach was to take a single MP value and spread it across multiple threads. In this paper, the author uses a novel scheme to parallelize the computation at a less granular level. What he does is organize the field finite operations of point addition into a complex workflow (rather like a dataflow machine) and then use 7 threads to process the workflow in parallel. The parallelization isn't perfect, and there are a number of gaps where threads are idle, but it's an intruiging approach. One advantage of this approach is that there are no carry resolution issues as there are in Jang et al.'s work. Bos runs his codes on several GPUs, and achieves a performance of 79K ECMP operations/sec on a GTX 295 and 290K ECMP operations/sec on a GTX 580. These are very fast results compared to Giorgi et al. [52] and Antão et al. [1], but still not as good as Bernstein et al., who achieved a scaled throughput of somewhere in the neighborhood of 250K ECMP operations/sec on a GTX 295.

Leboeuf, Muscedere, and Ahmadi published one paper in 2012 [80] and a follow on paper in 2013 [81]. The first paper deals with high speed computations of $A \cdot B \bmod P$, where P is a NIST defined prime with size 192, 224, 256, or 384 bits. These four primes all have fast modulo operations. The second paper uses generic primes which range in size from 112 to 521 bits. Both papers use an FRNS with $\beta = 2^{32}$ with grade school multiplication. The former uses custom fast modulo routines to reduce the product. The latter uses Montgomery reductions with a CIOS design and surprisingly outperforms the fast modulo routines. Leboeuf, Muscedere, and Ahmadi are the first (in the context of MP arithmetic) to explicitly discuss one of the major flaws of the compiler. When we write MP arithmetic code in CUDA C, the compiler generates PTX using a static single assignment form. Then the PTX code gets assembled to device code using far more registers than

strictly required. The excess register usage severely impacts performance of the device code. They work around this by implementing a C-based code generator for their modular multiplication routine, which allows them to generate the entire kernel in PTX with all loops entirely unrolled. They have also carefully thought through their storage design, keeping A entirely in registers, B in global memory, P in constant memory, and n words of the product in registers. They also take advantage of the full 32-bit multiplier, using the multiply and accumulate with carry in and carry out instruction. The authors then test their results on a GTX 480 and report the following performance (mod mul/sec): 1563M at 192 bits, 1282M at 224 bits, 1031M at 256 bits, and 294M at 512 bits. This is a well thought out approach and has higher performance than any previous paper. However, there are two minor caveats. First, the prime P must be constant as it gets generated into the PTX code, which would be unsuitable for RSA. Second, the paper implements only multiplication and they did not consider fast-squaring.

In [113] Zheng et al., implement a 256-bit modular multiplication routine to compute $A \cdot B \bmod P$ for a fixed $P = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$. This prime is used in the Chinese public key ECC algorithm *SM2*. The authors use an FRNS with $\beta = 2^{32}$ and a grade school multiplication algorithm that computes the product by going across the rows rather than down the columns. Once they have the product, they reduce it using a fast reduction algorithm based on P. In addition to multiplication, the authors introduce a novel row oriented algorithm for fast-squaring that requires significantly fewer add steps. It's clear from the paper that the authors are are using inlined PTX, because their algorithms are described in terms of "madc.cc.lo" and "madc.cc.hi", which are PTX instructions. But they don't discuss code generation or unrolling loops by hand. At some point between 2012 and 2014, NVIDIA updated the compiler to allow PTX inlines within unrolled loops. So it's possible that this paper uses straight CUDA C with no code generation. Another novel contribution of this paper is they introduce a utilization metric. They measure the number of 32-bit multiplications per second their code runs and divide the clock rate times

the number of multipliers on the chip, which gives them a utilization rate. They run their code on a GTX Titan, and achieve excellent results. At 256-bits, they report throughputs of 3.412B mults/sec, 3.384B mod muls/sec, 6.001B squares/sec and 5.993B mod sqrs/sec. They also report 392K ECMP operations per second and a utilization of 96%. Adjusting for clock rate and card, we see that this performance would correspond to somewhere around 1288M 256-bit mod mults (with a generic P) on a GTX 480. This is roughly 20% faster than the results of Leboeuf, Muscedere, and Ahmadi. Perhaps some of this discrepancy can be explained by clock boost and better overlap of instructions on the GTX Titan.

In [112] Zheng et al., implement a 2048-bit RSA implementation using two 1024-bit modular exponentiations and the CRT trick. They implement modular exponentiation with an FRNS and an unusual β of 2^{23} and 45 limbs per MP value. They use a Montgomery representation and a column oriented approach to sum up the product terms and Montgomery reduction terms. Since they are using 45 limbs of 23-bits, the maximum sum in any column will be $90 \cdot 2^{23} \cdot 2^{23}$. It's 90 because 45 are from computing A limbs times B limbs and 45 are from computing M limbs times Q limbs (as required for a Montgomery reduction). Note, this maximum sum is less than 2^{53}, which allows the authors to do all of their limb computations using double precision floating point. They don't have to worry that round off will poison the result because the mantissa is 53 bits in length (52 bits plus an implicit one). Unfortunately, the approach would require too many registers to use a thread per instance. So the authors employ a parallel technique, similar to Jang et al.'s, and divide a mod exp across 4, 8, or 16 threads. They then test their implementation on GTX Titan (which is one of the few GTX cards with high double precision floating performance) and achieve an RSA-2048 throughput of 39.0K decrypts/sec with 4 threads per mod exp, 36.3K with 8 threads per mod exp, and 27.5K with 16 threads per mod exp. Unlike their previous paper, the authors do not compute a utilization. It's also unclear if they could have achieved better performance in the integer domain.

In [33] Dong, Zheng, et al. improve upon their earlier work [112] and again use double precision floating point arithmetic to implement RSA-2048, RSA-3072 and RSA-4096. For RSA-2048 they use 23 bit samples, and for RSA-3072 and RSA-4096 they use 22 bit samples. Their new code achieves a throughput of 42.2K RSA-2048 decrypts/sec, 12.2K RSA-3072 decrypts/sec and 5.8K RSA-4096 decrypts/sec. The RSA-4096 result is noteworthy, when scaled for the differences in cards, it has the highest throughput of any RSA-4096 implementation published to date.

2.5.4 GPU Based Multiple Precision Libraries

In [111] Zhao and Chu implement a wide range of operators (comparison, addition, modular addition, subtraction, modular subtraction, multiplication, modular multiplication, division, and modular exponentiation) with precisions of 256 bits, 512 bits, 1024 bits, and 2048 bits in a library called GPUMP. They run their algorithms on a GTX 280 and report the results for various operations and sizes. Zhao and Chu have not done a careful study: there are many problems with the data and limited analysis. For example, in Figure 3, they report 3.8 million 512-bit multiplications per second, in Figure 7, they report 18 million 512-bit modular multiplications per second (modular multiplications should be slower than regular multiplications). Then in Figure 9 they report 8500 256-bit modular exponentiations per second. If the 18 million number were right, this should be somewhere in the neighborhood of 240K 256-bit modular exponentiations per second. Even if the data is interpreted in the most optimistic light, Neves and Araujo's and Bernstein et al.'s results are much faster on similar hardware.

In [103] Thall investigates implementing 64-bit floating point and 128-bit floating point operations represented as a sequence of native 32-bit floating point values. His library supports addition, subtraction, multiplication, division, square root, and comparison. There are three major limitations of the library: it doesn't scale past 128 bits, performance is quite limited, and it predates CUDA (it is written using Cg-based shader code).

In [82] Lu, He, and Luo port David Bailey's QD and ARPREC libraries to the GPU, calling them GQD and GARPREC. The GQD library supports double double and quad double. The GARPREC library supports arbitrary precision. Like QD and ARPREC, both implementations use a sequence of floating point values to represent an MP value. The authors compare the performance of GQD and GARPREC running on a GTX 280 to QD and ARPREC on an Intel Core 2 Q6600. The GQD library achieves a speedup of 10 to 20 over the CPU. GARPREC achieves a speedup of 7 to 12 depending on the operation and precision over the CPU. Unfortunately, the authors do not report raw performance, in operations per second, they only report performance as a speedup versus the CPU, which makes it difficult to compare this work with other papers. It's also worth noting that ARPREC is quite slow compared to GMP on the CPU.

In [85] Nakayama and Takahashi implement multiple precision floating point routines for addition and multiplication on the GPU, which they call CUMP. Unlike GARPREC, the authors represent the MP value with a sequence of 64-bit integers. They test their implementation on a Tesla C2050 GPU and compare the results of their implementation to GARPREC running on the same GPU and to GMP running on a 8-core AMD Opteron 6134. They run large batches and test at sizes of 100, 1000, and 10000 decimal digits. At 100 digits (roughly 333 bits) they achieve 143 million multiplies per second. At 1000 digits (3322 bits), they achieve 3.6 million multiplies per second, and at 10K digits (33220 bits), they achieve roughly 40K multiplications per second. For multiplication at 100 digits the GPU is faster than the CPU. However, for 1000 and 10000 digits, the CPU outperforms the GPU. Nakayama and Takahashi report that their implementation is between 1.7 and 2.6 times faster than GARPREC at all sizes. At 1000 digits, we estimate that the authors are achieving about 25% utilization of the multipliers on a C2050.

In [68] Joldes et al. construct a multiple precision floating point library called CAMPARY for the GPU using sequences of floating point numbers, similar to the approach used by QD and ARPREC. They implement addition, subtraction, multiplication, division, and

square root. One important feature of their library is that they give rigorous proofs of the accuracy and implement rounding modes with correct rounding in their routines. They also provide a set of "quick-and-dirty" routines that are faster, but do not come with the accuracy guarantees or the rounding modes. They test their quick-and-dirty routines on a Telsa C2075 GPU and compare their results to a multithreaded Intel i7-3820 running QD and MPFR. They report speedups of 2 to 3x vs QD and up to 20x faster than MPFR. Unfortunately, they don't give throughput numbers.

In [61] Honda, Ito, and Nakano investigate batched integer multiplication where the values to be multiplied range from 1024 bits to 32768 bits. Their approach is to assign a multiplication instance to each warp and use a warp synchronous approach to computing the product. For the smaller sizes, they use a grade school multiplication. For larger sizes they have the option to use one or two levels of Karatsuba multiplication. They run their experiments on a GTX 980 and achieve 65.8 million 1024-bit multiplication per second, 13.3M/sec at 2048 bits, 3.8M/sec at 4096 bits, 971K/sec at 8192 bits, 229K/sec at 16384 bits and 60.5K/sec at 32768 bits. They find that using Karatsuba improves the performance by less than 5%, even at 32768 bits. Comparing the 4K result to Nakayama and Takahashi's 3322 bit result, we see this is about 1.6 times faster and the 32K result is roughly 1.5 times faster than Nakayama and Takahashi's 33220 bit result. However, it's worth noting that the GTX 980 is a much faster card than a Tesla C2050.

2.5.5 Literature Survey – Conclusions

This section summarizes some open questions that have not been adequately addressed by the existing literature. It is organized in three subsections: parallel algorithms, asymmetric cryptography primitives on the GPU, and MP libraries on the GPU.

2.5.6 Parallel Algorithms

Takahashi's parallel short division algorithm has a complexity of $O\big((n/p)\log p\big)$, but this algorithm is not asymptotically optimal. If we cast short division as a parallel prefix

operation, it should be possible to achieve $O(n/p + \log p)$. This is an important problem area because a good Toom-Cook implementation requires a fast short division algorithm.

2.5.7 Asymmetric Cryptography Primitives on the GPU

Even though there have been so many papers published, we believe that not a single paper has reached peak performance at any size on any card:

1. For the 1.x cards (GTX 260, GTX 280, GTX 295) cards, no one has built a 48-bit accumulator using the 24-bit instructions. In theory, this should lead to significantly better performance on these cards.

2. For the 2.x and 3.x cards (GTX 580, M2050, GTX Titan), no one has explored Karatsuba multiplication, which should be faster at the sizes of interest.

3. None of the algorithms to date will work well on the 5.x and 6.x cards (GTX 750Ti, GTX 980, GTX 1080), which have 16-bit multipliers.

In general, all the papers are limited in scope. Some have just one algorithm, others support only one size, some support only one card. However, to build a good library requires achieving good performance across multiple generations of GPU and the full range of cryptographically useful sizes. No one has accomplished this.

Finally we note that Zheng's notion of a utilization metric is very important. Utilization metrics should be developed for all cards and algorithms and applied uniformly.

2.5.8 MP Libraries on the GPU

The MP libraries for the GPU fall into two categories, the low precision routines range from 60-bit mantissas to about 240-bit mantissas. At these scales, the best approach is probably to store the MP value as a sequence of floating point values and leverage the device level FP hardware to handle round off. This is the approach taken by Thall, and

the GQD and CAMPARY libraries. The CAMPARY library is the only one that supports rounding modes with correct rounding.

For mantissas larger than 256 bits, GPUMP and Honda, Ito, and Nakano implement integer libraries, while GARPREC and CUMP implement floating point libraries. At these sizes, it's probably best to represent the limbs as sequences of integer values and this is the area where our research strength lies.

Weaknesses of GPUMP, GARPREC, CUMP, and Honda et al.'s work:

1. They test their implementations only on a single GPU.
2. Honda et al. have the only implementation that supports more than one multiplication algorithm, but their Karatsuba implementation achieve only a modest 5% performance improvement over the grade school algorithm.
3. The only library to support division is GPUMP, but the performance is lacking.
4. The floating point libraries do not support important IEEE 754 features such as multiple rounding modes with correct rounding, or special values such as NaN and $\pm\infty$.

In summary, we believe these four libraries are slow, performing significantly below peak utilization on their respective cards, and are lacking important operations and features. Considerable work still needs to be done in the area of MP libraries.

CHAPTER 3

ASYMPTOTICALLY OPTIMAL PARALLEL SHORT DIVISION / DIVISION BY CONSTANTS

There has been a great deal of research studying PRAM models and proving upper and lower complexity bounds across a broad range of algorithms, including arithmetic, in the 1980s. This work is well summarized in two surveys, *A survey of parallel algorithms for shared-memory machines* by Karp and Ramachandran [71] and *The complexity of computation on the parallel random access machine* by Fich [46]. Since the late 90s, it's rare to find novel upper or lower bounds on PRAM arithmetic, but the research still continues. For example, *Modular exponentiation via the explicit Chinese remainder theorem* by Bernstein and Sorenson [8] in 2007 and Sorenson's paper [96], *A randomized sublinear time parallel GCD algorithm for the EREW PRAM*, from 2010. In this Chapter we present the results of two of our papers: [40] gives a new parallel short division algorithm that's asymptotically faster than Takahashi's algorithm (discussed in Section 2.3.3) and [43] which uses Cook, Dwork and Reischuk's [25] result on the complexity of Boolean functions to prove the new short division algorithm is asymptotically optimal on EREW and CREW PRAMs. The results are significant because they present an algorithm that achieves the lower bound for two fundamental MP arithmetic operations: short division and division by constants on an important class of parallel machine models.

Short division is the process of dividing a MP integer X by a single precision divisor d to find a multiple precision quotient Q and a remainder r such that $X = dQ + r$ where $0 \leq r < d$. We assume an FRNS with β to be a power of two and define X and Q in the

usual way:

$$X = \sum_{i=0}^{n-1} x_i \beta^i \quad \text{and} \quad Q = \sum_{i=0}^{n-1} q_i \beta^i$$

where $0 \leq x_i, q_i < \beta$. Throughout this chapter, we will use upper case variables to denote MP values, l for the length of X in bits, n for the length of X in words, thus $l = n \log_2 \beta$, and p for the number of processors.

It is often convenient to think of β as defined by a machine word, and we frequently refer to x_i and q_i as *words*, but this need not be the case. β can be defined as 2^k for an arbitrary fixed k, regardless of the underlying machine word size and the operations a *op* b, where *op* is addition, subtraction, multiplication, division, and modular inverse all run in $O(1)$ time when $0 \leq a, b < \beta$. Likewise, we frequently rely on the fact that d can be factored into $d = d_o \cdot d_e$ where d_o is odd and d_e is a power of two in $O(1)$ time.

For division by a fixed constant integer, such as $X/3$ or $X/31415926535$, we simply pick a β that is larger than the constant divisor and run the algorithm. The running time and lower bound analyses apply, because as stated above, the primitive operations all run in constant time.[1] Thus short division and division by constants are essentially the same problem and the proofs that follow are applicable to both.

The rest of this chapter is organized as follows. Section 3.1 gives a short overview of prior work. Section 3.2 presents the algorithm and proves correctness. Section 3.3 discusses the connections between short division and parallel prefix/suffix sum. Section 3.4 proves optimality of our short division algorithm and Section 3.5 covers our experimental setup and results.

---

[1]This argument does not apply to the general case of X/Y for arbitrary precision values X and Y because no matter what β is chosen, there will always be a larger Y.

3.1 Prior Work

The literature survey chapter covers the important multiple precision sequential and parallel division algorithms. Here we present a brief overview of some important results related to this chapter from other research areas.

Division By Constants: For small constants, several authors [62, 2, 97] use the following approach. Find a small k, such that the constant c divides $2^k \pm 1$. Multiply X by $\frac{2^k \pm 1}{c}$ and use a combination of logical right shifts and sums to divide the result by $2^k \pm 1$. The division step is no faster than the algorithm presented here and this approach does not work for all constants. Another approach is to compute a fixed-point reciprocal of c. This approach is quite slow because it involves computing the product of $1/c \cdot X$, where $1/c$ must be evaluated to roughly n word. The n-word by n-word multiplication has a minimum running time of $O(n \log n)$, which is at least as slow as the algorithm presented here.

Circuit Complexity: A number of papers [6, 20, 58, 30] study the circuit complexity of A/B where A and B are both MP values of length n. They show that a polynomial sized circuit in n can compute A/B in $O(\log n)$ time. Although this solves the problem with a hardware circuit, the same approach can be simulated with a PRAM. Thus, a circuit complexity argument using simulation could be made to establish an upper bound for short division on a PRAM, but such a simulation would be less efficient than the algorithm presented here, requiring either a polynomial number of processors (in n), or more than $O(\log n)$ time. Further, since PRAMs are more powerful than bounded fan in / bounded fan out circuits, a circuit complexity argument cannot be used to establish a tight lower bound for the complexity of short division on a PRAM (see for example [25]).

Parallel Prefix: The heart of our short division algorithm is a parallel prefix computation. Blelloch [12] has an extensive survey on the applications of parallel prefix, but division is not mentioned. In [78] Ladner and Fischer show how parallel prefix can be applied to

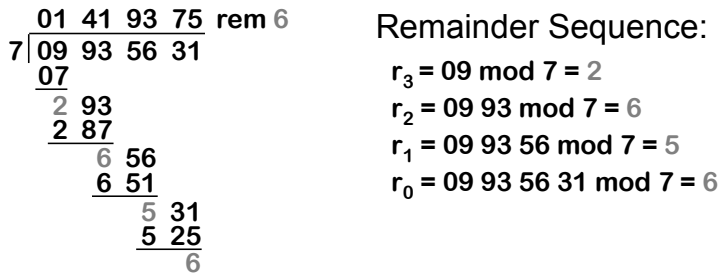

Figure 3.1. Example division and remainder sequence for $X = 9935631$, $d = 7$, and $\beta = 100$

addition of binary numbers with fixed size, and states that division by a constant is possible with a prefix computation but no construction or analysis is given, and short division is not mentioned at all.

3.2 Short Division Algorithm

We begin by considering an easier problem, where we assume that d is relatively prime to β and that $p = n$. See Figure 3.1 for an example. As in Takahashi's algorithm, the idea is to break the short division into two steps. First compute the remainder sequence, denoted r_k and then compute the quotient. We note that the quotient can be computed from the remainder sequence (2, 6, 5, 6 in the example) in a single parallel step with n processors as follows:

$$q_k = \left\lfloor \frac{r_{k+1}\beta + x_k}{d} \right\rfloor$$

Further, as shown in Figure 3.1, each element of r_k can be computed from all the digits of X to its left, i.e., x_i where $i \geq k$:

$$r_k = \left(\sum_{i=k}^{n-1} x_i \beta^{i-k} \right) \bmod d$$

When d and β are relatively prime (as in the example), $\beta^{-k} \pmod d$ will exist, and can be factored out of the sum as follows:

$$r_k = \left(\beta^{-k} \sum_{i=k}^{n-1} x_i \beta^{i} \right) \bmod d$$

and the remaining sum, $\sum_{i=k}^{n-1} x_i \beta^i \bmod d$, is just a suffix sum problem — prefix and suffix sums are classic problems with well know fast parallel algorithms (see for example Blelloch [12]).

Next, we relax the two assumptions and consider the full short division algorithm, where there are no restrictions on d, n, or p, provided that $d < \beta$. In the case where d and β are not relatively prime, d can be factored into $d_o \cdot d_e$ where d_o is odd and d_e is a power of two. The sub-division can be accomplished with two divisions, i.e.,

$$Q = \left\lfloor \frac{X}{d} \right\rfloor = \left\lfloor \frac{\lfloor \frac{X}{d_o} \rfloor}{d_e} \right\rfloor$$

since β is required to be a power of two, d_o and β will be relatively prime, and the division by d_o can be done using the remainder sequence approach. The result is then divided by d_e using logical bit-shift operations.

When $n > p$, the algorithm partitions X across the processors as follows. First pad X with zeros until the length of X is evenly divisible by p. Then break X into equal sized chunks each of length $m = \lceil \frac{n}{p} \rceil$. The k^{th} chunk will consist of the words x_{km} through x_{km+m-1} and will be assigned to processor k (where $k \in 0 \,..\, p-1$). The partitioning of X across the processors motivates the following definitions:

Definition 1. *Let X_i^j be the value corresponding to the words from chunk i through chunk j, i.e., x_{im} through x_{jm+m-1} and we have*

$$X_i^j = \sum_{k=im}^{jm+m-1} x_k \beta^{k-im} \quad \textit{and} \quad Q_i^j = \sum_{k=im}^{jm+m-1} q_k \beta^{k-im}$$

when $i > j$, the sum is empty and we define $X_i^j = 0$ and $Q_i^j = 0$.

Definition 1 means that the chunk assigned to processor k is X_k^k. Likewise, processor k will be responsible for computing Q_k^k. Further, we have $X_0^{p-1} = X$, and $Q_0^{p-1} = Q$.

For a formal PRAM algorithm, we must define the input and output memory structure. In this case, it's quite simple, we put X in memory cells M[0] through M[$n-1$] and the store the divisor, d, in M[n]. The output of the algorithm stores Q in M[0] through M[$n-1$]

1. If d is one, set the final remainder r to zero and exit, otherwise factor d into $d = d_o \cdot d_e$ where d_o is odd and d_e is a power of two.
2. If d_o is 1, then d is a power of two, use logical bit-shifting operations to compute Q and compute the final remainder, $r = X \bmod d = x_0 \bmod d_e$, then exit.
3. Pad X with zeros so it can be evenly split into p chunks. Let m be the length of each chunk, $m = \lceil n/p \rceil$. Assign the k^{th} chunk to processor k.
4. Each processor $k \in 0 \,..\, p-1$ computes $v_k = X_k^k \beta^{mk} \bmod d_o$.
5. Compute the parallel suffix sum of the terms,
$$t_k = \sum_{i=k}^{p-1} v_i \quad \Big(\bmod d_o\Big)$$
6. Next each processor computes the remainder sequence r_k:
$$r_k = t_k \beta^{-mk} \bmod d_o$$
7. Each processor then computes the quotient for its chunk
$$Q_k^k = \Big\lfloor \frac{r_{k+1}\beta^m + X_k^k}{d_o} \Big\rfloor$$
8. At this point, $Q = \lfloor \frac{X}{d_o} \rfloor$. Next, use logical bit-shift operations to divide Q by d_e.
9. The final remainder r is computed as $r = x_0 - d * q_0 \ \big(\bmod \beta\big)$.

Figure 3.2. parallel short division algorithm

and the final remainder in M[n]. Because X and Q are overlaid, when the divisor is one, we need not copy X to Q.

To clarify, we describe the computation on processor k in terms of X_k^k and Q_k^k rather than in terms of memory locations. The full short division algorithm is presented in Figure 3.2. We note that if $p > n$, then we only use the first n processors and leave the others idle.

Theorem 1. *Let $T_d(n,p)$ represent the running time of the short division algorithm (Figure 3.2) on an EREW PRAM.*

$$T_d(n,p) \in \begin{cases} \Theta(1) & \text{if } d = 1 \\ O(\lceil n/p \rceil) & \text{if } d = 2^i \text{ and } i > 0 \\ O(n/p) + O(\log p) & \text{if } d \neq 2^i \text{ and } p < n \\ O(\log n) & \text{if } d \neq 2^i \text{ and } p \geq n \end{cases}$$

Proof. Step 1 requires at most $\log_2 \beta$ bit tests and runs in constant time. If $d_o = 1$ then step 2 runs in $O(\lceil n/p \rceil)$ time.[2] Step 3 runs in constant time since at most $p-1$ words must be written using p processors. In step 4, $X_k^k \bmod d_o$ and $\beta^m \bmod d_o$ can be computed in $O(\lceil n/p \rceil)$ time. Computing $\beta^{mk} \bmod d_o$ requires an additional $O(\log p)$ steps. For step 5, we use the standard EREW PRAM two pass parallel suffix sum algorithm, which requires $O(\log p)$ time. Since $\beta^{mk} \bmod d_o$ was computed in step 4, in step 6, $\beta^{-mk} \bmod d_o$ the modular inverse can be computed in constant time. Steps 7 runs in $O(\lceil n/p \rceil)$ time. Steps 8 and 9 run in constant time.

Thus, if $d = 1$, then the algorithm stops after step 1 and $\mathrm{T}_d(n,p) \in \Theta(1)$. If d is a power of two, then the algorithm stops after step 2 and $\mathrm{T}_d(n,p) \in O(\lceil n/p \rceil)$. If d is not a power of two then all steps of the algorithm are run, and there are two cases. If $p < n$, we have $O(\lceil n/p \rceil) = O(n/p)$, and the run time $\mathrm{T}_d(n,p) \in O(n/p) + O(\log p)$. Lastly, if $p \geq n$ then only the first n processors are used and the running time for steps 4 and 5 will be $O(\log n)$. All other steps run in constant time and we have $\mathrm{T}_d(n,p) \in O(\log n)$. □

As a final note in this section, there are several variants of the algorithm in Figure 3.2, such as replacing the r_k computation with a parallel cyclic reduction (similar to Takahashi's algorithm), all with the same running time complexity. We chose this variant because the proof of correctness is easier and because it highlights the connections between short

[2] Since p could be much larger than n, we use $\lceil n/p \rceil$ to ensure it takes at least one time step.

division and parallel prefix/suffix sums. In the experimental results section we also explore some of the other variants.

3.2.1 Proof of Correctness

For the correctness proof, if $d = 1$, then $Q = X$, which completes after the first step. If not, factor d into $d = d_o \cdot d_e$, where d_o is odd and d_e is a power of two. If $d_o = 1$ then d is a power of two and we can compute Q using logical shift operators and r will be $x_0 \bmod d$. The challenging part is to show that steps 4-7 compute $Q = \lfloor X/d_o \rfloor$, when $d_o > 1$.

Lemma 1. $X_i^j = \sum_{k=i}^{j} X_k^k \beta^{(k-i)m}$.

Proof.

$$\begin{aligned} X_i^j &= \sum_{k=im}^{jm+m-1} x_k \beta^{k-im} \\ &= \sum_{k=i}^{j} \sum_{h=km}^{km+m-1} x_h \beta^{h-im} \\ &= \sum_{k=i}^{j} \sum_{h=km}^{km+m-1} x_h \beta^{h-km+(k-i)m} \\ &= \sum_{k=i}^{j} X_k^k \beta^{(k-i)m} \end{aligned}$$

□

Intuitively this can be understood by thinking of X_k^k (the chunk assigned to processor k) as a digit in the base β^m and X_i^j as a sequence of these digits. It follows trivially that $X_i^j = \sum_{k=i}^{j} X_k^k \beta^{(k-i)m}$

Lemma 2. $X_i^j = X_i^i + \beta^m X_{i+1}^j$, *holds for all* $i, j \in 0 \,..\, p-1$.

Proof. This follows directly by pulling the first term from the sum in Lemma 1. When $i = p-1$ and $j = p-1$, we have $X_{p-1}^{p-1} = X_{p-1}^{p-1} + \beta^m X_p^{p-1}$ which holds because X_p^{p-1} was defined to be zero. □

Corresponding lemmas hold for Q_i^j.

Theorem 2. *Steps 4-6 of the short division algorithm compute* $r_k = X_k^{p-1} \bmod d_o$.

Proof.

$$
\begin{aligned}
v_k &= X_k^k \beta^{mk} \bmod d_o \\
t_k &= \Big(\sum_{i=k}^{p-1} v_k \Big) \bmod d_o = \Big(\sum_{i=k}^{p-1} X_i^i \beta^{mi} \Big) \bmod d_o \\
r_k &= t_k \beta^{-mk} \bmod d_o = \Big(\beta^{-mk} \sum_{i=k}^{p-1} X_i^i \beta^{mi} \Big) \bmod d_o \\
&= \Big(\sum_{i=k}^{p-1} X_i^i \beta^{m(i-k)} \Big) \bmod d_o \\
&= X_k^{p-1} \bmod d_o
\end{aligned}
$$

□

Theorem 3. *Step 7 of the short division algorithm computes* Q *such that* $X = d_o \cdot Q + r_0$.

Proof. We use a reverse induction to show that $X_i^{p-1} = d_o \cdot Q_i^{p-1} + r_i$ holds for all $i \leq p$. For the base case, $i = p$ holds trivially since X_p^{p-1} and Q_p^{p-1} and $r_p = X_p^{p-1} \bmod d_o$ are all zero. For the inductive step, we assume the statement holds for $i + 1$ and prove it is true for i. Assume:

$$X_{i+1}^{p-1} = d_o Q_{i+1}^{p-1} + r_{i+1}$$

Multiplying both sides by β^m and adding X_i^i yields:

$$
\begin{aligned}
\beta^m X_{i+1}^{p-1} + X_i^i &= \beta^m (d_o Q_{i+1}^{p-1} + r_{i+1}) + X_i^i \\
X_i^{p-1} &= d_o \beta^m Q_{i+1}^{p-1} + \Big(\beta^m r_{i+1} + X_i^i \Big) \\
X_i^{p-1} &= d_o \beta^m Q_{i+1}^{p-1} + d_o \underbrace{\Big\lfloor \frac{\beta^m r_{i+1} + X_i^i}{d_o} \Big\rfloor}_{\text{definition of } Q_i^i} + \Big((\beta^m r_{i+1} + X_i^i) \bmod d_o \Big) \\
X_i^{p-1} &= d_o (\beta^m Q_{i+1}^{p-1} + Q_i^i) + \Big((\beta^m X_{i+1}^{p-1} + X_i^i) \bmod d_o \Big) \\
X_i^{p-1} &= d_o Q_i^{p-1} + \underbrace{\Big(X_i^{p-1} \bmod d_o \Big)}_{\text{definition of } r_i} \\
X_i^{p-1} &= d_o Q_i^{p-1} + r_i
\end{aligned}
$$

By induction, we have $X_i^{p-1} = d_o \cdot Q_i^{p-1} + r_i$ for all $i \in 0 \,..\, p$. Since $X_0^{p-1} = X$, $Q_0^{p-1} = Q$, we have $X = d_o \cdot Q + r_0$, and since $0 \leq r_0 < d_o$, we can conclude $Q = \lfloor X/d_o \rfloor$.

Step 8 of the algorithm sets Q to Q divided by d_e, thus computing the final quotient:

$$Q = \left\lfloor \frac{\lfloor X/d_o \rfloor}{d_e} \right\rfloor = \left\lfloor \frac{X}{d_o \cdot d_e} \right\rfloor = \lfloor X/d \rfloor$$

In step 9 the algorithm computes the final remainder, r:

$$r = X \bmod d = X - d \cdot Q = X - d \cdot Q \pmod{\beta} = x_0 - d \cdot q_0 \pmod{\beta}$$

Since $Q = \lfloor X/d \rfloor$ and $r = X \bmod d$, we can conclude that $X = d \cdot Q + r$ and the algorithm is correct. □

3.3 Connections to Parallel Prefix/Suffix Sum

We have already seen that the heart of the remainder sequence computation is a parallel suffix sum. Now we show that exclusive suffix sum problems can be solved with the short division algorithm.

Given a sequence of y_0, y_1, ... y_{n-1} where $0 \leq y_i \leq \hat{y}$ for some maximum value $\hat{y}$, the exclusive suffix sum is defined as:

$$s_k = \sum_{i=k+1}^{n-1} y_i$$

for each $k \in 0 \,..\, n-1$.

We wish to construct a short division problem instance consisting of X (x_0, x_1, ... x_{n-1}), β and d such that the resulting quotient words match the exclusive suffix sum, i.e., $q_k = s_k$ for all $k \in 0 \,..\, n-1$ and prove the construction is correct. We choose β to be a power of two greater than $n \cdot \hat{y} + 1$ and we choose $d = \beta - 1$ and we set all the x_i values to be equal to the y_i values.

We know from the correctness proof that for an odd d, when we have solved for Q and r such that $X = dQ + r$, there exists a remainder sequence, r_k that relates X_k^k to Q_k^k. By

taking $n = p$ and thus $m = 1$, we have $X_k^k = x_k$ and $Q_k^k = q_k$, and from their definitions, we have:

$$r_k = \Big(\sum_{i=k}^{n-1} x_i \beta^{i-k}\Big) \bmod d \quad \text{and} \quad q_k = \Big\lfloor \frac{r_{k+1}\beta + x_k}{d} \Big\rfloor$$

Since $d = \beta - 1$ we have $\beta \bmod d = 1$ and therefore $\beta^{i-k} \bmod d = 1$ and the β^{i-k} term drops out of the r_k equation and we are left with:

$$r_k = \Big(\sum_{i=k}^{n-1} x_i\Big) \bmod d$$

since $\sum_{i=k}^{n-1} x_i \leq n \cdot \hat{y} < d$ for all k, we have:

$$r_k = \Big(\sum_{i=k}^{n-1} x_i\Big) \bmod d = \sum_{i=k}^{n-1} x_i = \sum_{i=k}^{n-1} y_i = s_{k-1}$$

Combining this result with our expression for q_k and simplifying, we get:

$$\begin{aligned}
q_k &= \Big\lfloor \frac{r_{k+1}\beta + x_k}{d} \Big\rfloor \\
&= \Big\lfloor \frac{r_{k+1}(\beta - 1) + r_{k+1} + x_k}{\beta - 1} \Big\rfloor \\
&= r_{k+1} + \Big\lfloor \frac{x_k + r_{k+1}}{\beta - 1} \Big\rfloor \\
&= r_{k+1} + \Big\lfloor \frac{x_k + \sum_{i=k+1}^{n-1} x_i}{\beta - 1} \Big\rfloor \\
&= r_{k+1} + \Big\lfloor \frac{r_k}{\beta - 1} \Big\rfloor \\
&= r_{k+1} \\
&= s_k
\end{aligned}$$

We also note that the final remainder is $r = r_0 = s_{-1} = \sum_{i=0}^{n-1} y_i$. We can conclude, with an appropriate β and d, short division can be used to solve exclusive suffix sum problems.

3.4 Optimality Proof

In this section, we will show that parallel short division is subject to two distinct lower bounds, i.e., $\mathrm{T}_d(n,p) \in \Omega(\lceil n/p \rceil)$ and $\mathrm{T}_d(n,p) \in \Omega(\log n)$. The first bound comes from the

fact that some processor must read every word of X. This is easy to prove by contradiction. Suppose Q and r could be computed without reading some word, x_w. Then there would be two values, X and X' that differ at word w but have the same Q and r. But $X = d \cdot Q + r$ and $X' = d \cdot Q + r$, yet $X \neq X'$. Contradiction. Therefore some processor must read $\lceil n/p \rceil$ words, and we have proven the first bound. The second bound is more difficult.

It is well known that computing the PARITY of n bits, b_0, b_1, ... b_{n-1} has a lower bound of $\Omega(\log n)$ on a CREW PRAM (see for example Fich [46] Theorem 21.42). Using the construction in the prior section, PARITY_n can be computed using short division as follows: Set $x_i = b_i$, run short division with $d = \beta - 1$. Provided that $n < \beta - 1$, the parity will be the least significant bit of the final remainder r_0, because $\sum b_i = \sum x_i = r = r_0$. Since PARITY_n can be solved with short division, the lower bound for some short division instances on a CREW PRAM must be $\Omega(\log n)$. This approach to the second lower bound has some significant shortcomings. First, we would like to establish the bound for all divisors, not just $d = \beta - 1$. Second, the restriction that $n < \beta - 1$ is problematic since we are interested in the asymptotic behavior as $n \to \infty$. But it does give insight that the way to attack the lower bound is through the least significant bit of the remainder.

In [25] Cook, Dwork and Reischuk (CDR) study the time complexity of OR_n, the time it takes to compute the logical OR of n true/false values stored in memory on a CREW PRAM. Their work establishes a lower bound for the computation of any Boolean function of n variables, $f{:}\{0,1\}^n \to \{0,1\}$. The bound can be expressed in terms of the *critical complexity* $\mathrm{c}(f)$ also known as the *sensitivity* of f. Since the least significant bit of the remainder is a Boolean function of all the bits of X, we can use critical complexity for a lower bound on short division. We will use the definition of critical complexity adopted by Wegener in [109].

Definition 2. *Let $\mathrm{c}(f, V)$, the critical complexity of f at point v (where v is a vector of Boolean values) be the number of neighbors W of V with Hamming distance 1, where*

$f(W) \neq f(V)$. *And let* $\mathrm{c}(f)$*, the critical complexity of* f *be the maximum of* $\mathrm{c}(f, v)$ *over all* $V \in \{0,1\}^n$.

For example, consider the critical complexity of OR_n. There are three cases. If V is a vector with multiple bits set, then $\mathrm{c}(\mathrm{OR}_n, V) = 0$. If V has exactly one bit set, then $\mathrm{c}(\mathrm{OR}_n, V) = 1$. If V is a vector of zeroes, then all bits are critical and $\mathrm{c}(\mathrm{OR}_n, V) = n$. $\mathrm{c}(\mathrm{OR}_n)$ is the max of the three cases and is thus n.

Cook, Dwork and Reischuk require that the n input bits to f are stored in memory locations M[0] through M[$n-1$] and that the result of f is to be written to M[0] once computed. They prove that the minimum number of steps to evaluate any boolean function, f, on a CREW PRAM is at least $\log_b\big(\mathrm{c}(f)\big)$, where $b = \frac{1}{2}(5+\sqrt{21}) \approx 4.79$. This lower bound holds regardless of the number processors available. Parberry and Yan [89] improve the lower bound and show it holds when $b = 4$.

3.4.1 Asymptotic lower bound for REMPAR on a CREW PRAM

In this section, we define the Boolean function $\mathrm{REMPAR}_{l,d}(V)$ to be the least significant bit (parity) of the remainder of the integer represented by V divided by d, where l is the length of the dividend in bits and d is the divisor. We will show that $\mathrm{REMPAR}_{l,d}(V)$ requires $\Omega(\log l)$ steps to evaluate on a CREW PRAM when d is odd and greater than one. We begin with two helper functions that map integers to their standard bit representation (a vector of Booleans) and bit representation back to integers.

Definition 3. $X = \mathrm{Z}_l(V)$ *maps a Boolean vector* v *of length* l *to its corresponding integer* X *and* $V = \mathrm{Z}_l^{-1}(X)$ *is the inverse function, that maps* X *back to vector representation. Thus,*

$$\mathrm{Z}_l(v_0, v_1, ...v_{l-1}) = \sum_{i=0}^{l-1} 2^i v_i \quad \textit{and} \quad \mathrm{Z}_l^{-1}(\mathrm{Z}_l(V)) = V.$$

clearly there is a one to one mapping between X and V and we can use them somewhat interchangeably.

Definition 4. $\mathrm{REMPAR}_{l,d}(V)$*:* $\{0,1\}^l \rightarrow \{0,1\}$ *is a Boolean function of* l *variables whose result is zero if* $\mathrm{Z}_l(V)$ *mod* d *is even and one if odd.*

To simplify the discussion, we introduce a new operator $\otimes$, and define $V \otimes i$ to mean the vector V with the i[th] bit flipped. Further, the i[th] bit is said to be *critical* if and only if $\mathrm{REMPAR}_{l,d}(V) \neq \mathrm{REMPAR}_{l,d}(V \otimes i)$. Thus the critical complexity, $\mathrm{c}(\mathrm{REMPAR}_{l,d}, V)$ is just the count of the critical bits in V.

Next, given an odd divisor $d > 1$, we wish to construct an instance V of length l, where a significant majority of the bits are critical. The construction is done in two steps. First we'll assign values to the i[th] element of V, where $i \in [\log_2 \beta, l)$, according to the following rule: if $2^i \bmod d$ is even, we assign a one, otherwise a zero. Once these bits have been fixed, we assign the remaining bits in the range $i \in [0, \log_2 \beta)$, such that $\mathrm{Z}_l(V)$ is smallest value that is exactly divisible by d. This construction is shown in Figure 3.3. We note that once the first step is complete, there are $\log_2 \beta$ bits to assign in the second step. Thus the second assignment gives rise to a set of possible values, $\{\mathrm{Z}_l(V)\}$, containing β consecutive integers. Since $\beta > d$, the interval must contain at least one value where $\mathrm{Z}_l(V)$ is exactly divisible by d. If there are multiple values exactly divisible by d, we choose the assignment

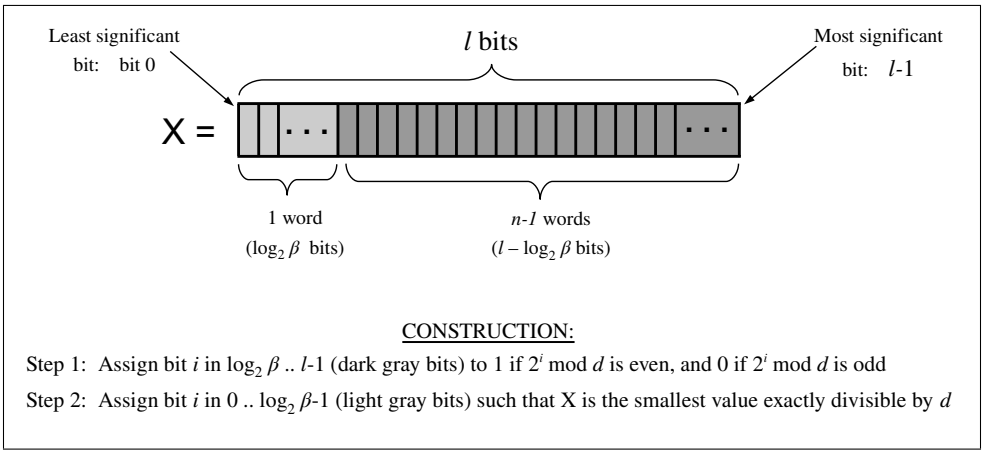

Figure 3.3. X represented as an array of bits

that produces the smallest $\mathrm{Z}_l(V)$. We can conclude, for any l and odd d greater than 1, this construction produces a unique vector, henceforth called $V_{l,d}$ or equivalently $X_{l,d}$.

Claim 1. *For odd d greater than 1, the i[th] bit of $V_{l,d}$ will be critical for all $i \in [\log_2 \beta, l)$.*

Proof. By the construction $\mathrm{Z}_l(V_{l,d})$ is evenly divisible by d, thus $\mathrm{REMPAR}_{l,d}(V_{l,d}) = 0$ and the i[th] bit will critical if and only if $\mathrm{REMPAR}_{l,d}(V_{l,d} \otimes i) = 1$. There are two cases to check.

Case 1 when $2^i \bmod d$ is odd:

In accordance with the construction, the i[th] bit of $V_{l,d}$ is zero, and $V_{l,d} \otimes i$ flips the i[th] bit to a one, thus we have $\mathrm{Z}(V_{l,d} \otimes i) = X_{l,d} + 2^i$, and:

$$\begin{aligned} \mathrm{REMPAR}_{l,d}(V_{l,d} \otimes i) &= \Big((X_{l,d} + 2^i) \bmod d\Big) \bmod 2 \\ &= \Big(2^i \bmod d\Big) \bmod 2 \end{aligned}$$

But since $2^i \bmod d$ is odd, we have $\mathrm{REMPAR}_{l,d}(V_{l,d} \otimes i) = 1$ and the i[th] bit is critical.

Case 2 when $2^i \bmod d$ is even:

According to the construction, the i[th] bit of $V_{l,d}$ is one and $V_{l,d} \otimes i$ flips the i[th] bit to a zero, thus we have $\mathrm{Z}_l(V_{l,d} \otimes i) = X_{l,d} - 2^i$, and we have

$$\begin{aligned} \mathrm{REMPAR}_{l,d}(V_{l,d} \otimes i) &= \Big((X_{l,d} - 2^i) \bmod d\Big) \bmod 2 \\ &= \Big((0 - 2^i) \bmod d\Big) \bmod 2 \\ &= \Big(d - (2^i \bmod d)\Big) \bmod 2 \end{aligned}$$

Since d is odd and greater than one, 2^i is not evenly divisible by d, thus, $0 < 2^i \bmod d < d$. Further, since d is odd and $2^i \bmod d$ is even, $d - (2^i \bmod d)$ must be odd, and therefore, $\mathrm{REMPAR}_{l,d}(V_{l,d} \otimes i) = 1$ and the i[th] bit is critical.

We can conclude that each bit in the range $[\log_2 \beta, l)$ is critical. □

It is worth noting that this construction only works for odd divisors greater than 1. If d is even, then $\mathrm{Z}_l(V)$ and $\mathrm{Z}_l(V \otimes i)$ will either both be even or both will be odd, and as a

result, none of the bits in the range $[\log_2 \beta, l)$ will be critical. If d is one, then $2^i \bmod d$ will always be zero, and again, none of the bits in the range will be critical.

It follows immediately from Claim 1 that $\mathrm{c}(\mathrm{REMPAR}_{l,d}) \geq \mathrm{c}(\mathrm{REMPAR}_{l,d}, V_{l,d}) \geq l - \log_2 \beta$ and by Cook, Dwork, and Reischuk's and Parberry and Yan's bounds, we can conclude that the minimum number of time steps to evaluate $\mathrm{REMPAR}_{l,d}$ on a CREW PRAM is in $\log_4(l - \log_2 \beta)$, even if there are an unlimited number of processors available. Since β is constant, $\mathrm{REMPAR}_{l,d} \in \Omega(\log l)$, or equivalently $\Omega(\log n)$, when d is odd and greater than one.

3.4.2 Asymptotic lower bounds for short division on a CREW PRAM

In the previous section, we established a lower bound on $\mathrm{REMPAR}_{l,d}$ for odd d greater than one. Here we show that $\mathrm{REMPAR}_{l,d}$ can be reduced to short division (for all non-power of two divisors) and therefore short division must be in $\Omega(\log n)$.

Definition 5. *Let $S_d(n, p)$ be the running time of an optimal short division algorithm.*

Theorem 4. *Let $S_d(n, p)$ represent a lower bound on the running time of an optimal short division algorithm on an CREW PRAM for three cases:*

$$S_d(n,p) \in \begin{cases} \Omega(1) & \textit{if } d \textit{ is one} \\ \Omega(\lceil n/p \rceil) & \textit{if } d \textit{ is a power of two greater than 1} \\ \Omega\big(\max(n/p, \log n)\big) & \textit{if } d \textit{ is not a power of two} \end{cases}$$

Proof. When $d = 1$, then Q=X and the only work to be done is to set $r = 0$. This can be done in constant time if X and Q are overlaid in the same memory cells or in time $\lceil n/p \rceil$ if X must be copied.

Next, if $d > 1$, as we have shown, every word of X must be read by some processor, and therefore a lower bound is $\lceil n/p \rceil$.

The algorithm starts with V, a vector of l bits, in memory cells M[0] through M[$l-1$], where $l = n \log_2 \beta$.

1. Use the available processors to pack the bits of V into the first n memory cells so that:
$$\mathrm{Z}_l(v) = \sum_{i=0}^{n-1} \mathrm{M}[i]\beta^i$$
In essence, this step constructs X from input vector V. Since β is a constant and each processor is responsible for packing $\lceil n/p \rceil$ words, the run time is O($\lceil n/p \rceil$).
2. Run the optimal short division algorithm that computes Q and r from X and d using the available processors in as few time steps as possible.
3. Factor d into $d_o \cdot d_e$ where d_o is odd and d_e is a power of two. Compute $r' = r \bmod d_o$.
4. If r' is even, write zero to M[0] else write one to M[0].

Figure 3.4. $\mathrm{REMPAR}_{l,d_o}(V)$ algorithm implemented using short division

The tricky case is when d is not a power of two. In this case, d can be factored into $d_o \cdot d_e$ where d_o is odd and greater than one and d_e is a power of two. Consider the algorithm presented in Figure 3.4. This algorithm starts with a vector V of l Boolean values in memory and constructs an X such that $\mathrm{Z}_l(V) = X$. It uses short division to compute $r = X \bmod d$, then computes $r' = r \bmod d_o$. Since d_o divides d, we have

$$r' = X \bmod d_o = \mathrm{Z}_l(V) \bmod d_o$$

Thus the algorithm computes REMPAR_{l,d_o} where d_o is odd and greater than one. Let $\mathrm{R}_d(n,p)$ be the run time of this algorithm on a CREW PRAM. By the bound of the previous subsection, we know $\mathrm{R}_d(n,p) \in \Omega(\log n)$. Further, since step 1 of the algorithm takes $O(\lceil n/p \rceil)$ time, step 2 takes $\mathrm{S}_d(n,p)$ time, steps 3 and 4 run in constant time, we have

$$\mathrm{R}_d(n,p) = \mathrm{S}_d(n,p) + O(\lceil n/p \rceil) + O(1)$$

Since $d > 1$, the short division algorithm must read every word of X, and we have $\mathrm{S}_d(n,p) \geq \lceil n/p \rceil$, therefore

$$\mathrm{R}_d(n,p) \leq c \cdot \mathrm{S}_d(n,p)$$

for some $c > 0$. Thus we have $\mathrm{S}_d(n,p) \in \Omega(\log n)$ and $\mathrm{S}_d(n,p) \in \Omega(\lceil n/p \rceil)$. The $\log n$ bound holds regardless of the number of processors available. From these two bounds we can conclude $\mathrm{S}_d(n,p) \in \Omega\big(\max(n/p, \log n)\big)$ and we have established $\mathrm{S}_d(n,p)$ as a lower bound for short division. □

We conclude this section with two asides. It is tempting to try to simplify the $\log n$ bound using a circuit height argument, i.e., since each word of X can change the remainder, the height of the expression tree for the remainder must be O(log n). Unfortunately, this is not possible. As Cook, Dwork, and Reischuk observe, a PRAM is fundamentally more powerful than a circuit. It has constant time read indexing and can pass information by both writing into memory and by not writing into memory, whereas a circuit must always "write" its result. The importance of CDR's work is that it places a strict limit on the amount of information that can be passed by not writing.

The lower bound was derived from the least significant bit of the remainder, so perhaps it is possible to compute $Q = \lfloor X/d \rfloor$ without r and beat this bound? Alas, the answer is no. If Q and X are known, r can be computed in constant time. Thus it is the computation of Q that drives the complexity.

3.4.3 Short division algorithm is asymptotically optimal for all *d*

Claim 2. *The short division algorithm presented in Figure 3.2 is asymptotically optimal for all divisors on a CREW PRAM, i.e.,* $T_d(n,p) \in \Theta\big(S_d(n,p)\big)$.

Proof. If $d = 1$ is one, then the short division algorithm runs in constant time. If d is a power of two, then the short division algorithm runs in $\Theta(\lceil n/p \rceil)$ time which matches the lower bound of $\Omega(\lceil n/p \rceil)$.

If d is not a power of two and $p < n$ then the run time $\mathrm{T}_d(n,p) \in \Theta(n/p) + \Theta(\log p)$ and we have:

$$\begin{aligned} \mathrm{T}_d(n,p) &\leq c_1 \cdot n/p + c_2 \log p \\ &\leq c_1 \cdot n/p + c_2 \log n \\ &\leq (c_1 + c_2) \max(n/p, \log n) \end{aligned}$$

for some $c_1, c_2 > 0$ and we have $\mathrm{T}_d(n,p) \in \Omega\big(\max(n/p, \log n)\big)$, which matches the lower bound.

Finally, if d is not a power of two and $p \geq n$, then the run time is $\Theta(\log n)$ which matches the lower bound since $\max(n/p, \log n) = \log n$. We can conclude that our short division algorithm is asymptotically optimal and the lower bound $\mathrm{S}_d(n,p)$ is tight for all divisors. □

3.5 Experiments and Results

To evaluate the performance of a variety of parallel short division algorithms, we reran the code developed for the 2011 HiPC paper [40] with some minor updates to support the Maxwell architecture. In Section 3.5.1 we describe the algorithms tested, in Section 3.5.2 we describe the experimental setup and in Section 3.5.3 we present the results and analysis.

3.5.1 Parallel Short Division Algorithms Tested

We have implemented and tested four different algorithms:

Takahashi: the first algorithm is Takahashi's parallel short division algorithm, which is described earlier in Section 2.3.3. Takahashi's algorithm has a running time complexity of $O((n/p)\log p)$, where n is the length of the dividend and p is the number of processors. This algorithm is performs well when $n = p$, but as we will see, when the number of limbs per processor increases, the performance suffers.

RTL Quotient: this is essentially the algorithm presented in Figure 3.2, where r_k is computed using $\beta^k \bmod d$, a parallel suffix sum and $\beta^{-k} \bmod d$. However, instead of using the classic left-to-right division to compute Q_k, we use a Jebelean right-to-left (RTL) approach, as described in Section 2.2.5 on exact division. The right-to-left approach has the advantage of swapping multiplication instructions for division instructions. Division is quite slow on the GPU. The running time of this algorithm is still $O(n/p + \log p)$.

Hybrid: this algorithm is a hybrid approach. It turns out the $\beta^k \bmod d$ and $\beta^{-k} \bmod d$ computations are expensive on the GPU. These can be avoided by using a parallel cyclic reduction to compute r_k. This is similar to Takahashi's algorithm, except it runs in $O(n/p + \log p)$ time. It then uses the RTL approach to compute Q_k.

Constant: for this algorithm, we explore doing short division by the constant 360. 360 is of interest because it's suitable for implementing 3-way and 4-way Toom-Cook multiplication. This algorithm follows the approach described in Figure 3.2, first divide by 8 using logical shifting, then divide by 45. Further, we take advantage of two optimizations. First, we can replace the $\beta^k \bmod d$ and $\beta^{-k} \bmod d$ computations with small precomputed tables. Second, we can replace $x \bmod 45$ where x is a 32-bit value, with $\lfloor x/2^{16} \rfloor \cdot 16 + x \bmod 2^{16}$. This latter approach reduces a 32-bit value to roughly a 20-bit value and is very fast on the Maxwell architecture. This algorithm also has a running time complexity of $O(n/p+\log p)$.

3.5.2 Experimental Setup

To evaluate our algorithms we ran our experiments on an NVIDIA GeForce GTX 980 GPU with a clock running at 1.22 GHz. The host machine is an Intel Core i5-7400 clocked at 3.0 GHz with 16 GB of memory, running 64-bit Ubuntu Server version 16.04.1 LTS and CUDA 8.0 with version 375.26 of the NVIDIA driver. For the CPU tests, we use the GNU Multiple Precision Library (GMP) version 6.1.1.

The GTX 980 has 16 Streaming Multiprocessors (SMs). Each SM consists of 96KB of shared memory, a 64K word register file (divided amongst the resident threads) and 128

CUDA cores, for a total of 2048 cores on the GPU. GPUs are based on a Single Instruction, Multiple Threads (SIMT) paradigm, where computations are organized into threads, warps, blocks and grids. Each thread has a thread ID and a private set of registers. 32 consecutive threads are then grouped together into a warp. Instructions are dispatched to a warp, and each thread in the warp executes the instruction in parallel. Thus at the hardware level, dispatching an instruction to a warp is equivalent to executing an instruction on a 32-way vector unit. Warps are further grouped into blocks where the warps in a block are co-located on the same SM. Finally, blocks are grouped into grids which are scheduled across the available SMs and resources on a GPU. Threads that reside in the same block can be synchronized and can communicate via shared memory, but communication between threads in different blocks is prohibited (except in very limited circumstances). For a full description of the GPU programming model, we refer the reader to the NVIDIA CUDA C Programming Guide [28].

To simplify our experiments and timing analysis, we designed our code around warp parallelism. Each warp is assigned a short division problem instance and the 32 threads of the warp work in parallel to solve the instance. Assigning an instance per warp is beneficial because warps are inherently synchronized and thus no special synchronization operations are required to communicate between threads in the warp. In our experiments, we run 4 warps (128 threads) in each block, and therefore, each block handles 4 separate instances. For each algorithm, we perform a series of three tests, the first uses $n = 32$ (32 limbs) with $\beta = 2^{32}$ for a total of 1024 bits and $p = 32$ (each thread in the warp counts as a "processor"). In the second set of tests, we use 2 limbs per thread (2048 bits), and in the third set, 4 limbs per thread (4096 bits). For each set, we proceed by generating 10,000 random multi-word dividends and 10,000 random divisors where each divisor is less than 2^{16}. The choice of 16 bit divisors was somewhat arbitrary, but it has the advantage that $a \cdot b \bmod d$, where a and b are 32-bit values, can be implemented efficiently on the GPU

Algorithm	$n = p$ (1024 bits)		$n = 2p$ (2048 bits)		$n = 4p$ (4096 bits)	
	time	ops/sec	time	ops/sec	time	ops/sec
Takahashi	0.194 ms	258.3 M/s	0.501 ms	99.9 M/s	0.936 ms	53.4 M/s
RTL Quotient	0.518 ms	96.6 M/s	0.589 ms	84.7 M/s	0.703 ms	71.2 M/s
Hybrid	0.171 ms	293.0 M/s	0.223 ms	224.3 M/s	0.263 ms	190.0 M/s
Divide by 360	0.071 ms	701.7 M/s	0.081 ms	613.7 M/s	0.120 ms	416.4 M/s
GMP (CPU based)		13.9 M/s		7.4 M/s		3.5 M/s

Table 3.1. Run time results for algorithms where $n = 2p$

as $(a \bmod d) \cdot (b \bmod d) \bmod d$ using only 32-bit arithmetic, which is the native size of the ALUs on the GPU.[3]

The test procedure is straightforward: generate the random data, copy it to the GPU, run the appropriate short division kernel, copy the results back to the CPU and verify against the results computed using GMP on the CPU. We measure the time for the kernel to run, but do not include the time to generate the data, copy it to or from the GPU, or to verify the results.

In our initial experiments, we discovered that some of the algorithms were so fast they actually become memory bound instead of computation bound. To work around this problem, we perform 5 consecutive divisions for each dividend, i.e., for each X, we compute $X/d/d/d/d/d$. This means that we perform 5 divisions for each load of the dividend and store of the result, which reduces the bus bandwidth requirements by a factor of 5.

3.5.3 Results and Discussion

For each algorithm, we test the three sizes (1024, 2048 and 4096 bits) and report the average running time for 10 runs, and from that compute the average throughput (operations per second). These results are presented in Table 3.1. For the CPU, we report the throughput of a single thread running the mpz_div_ui function, which is the GMP equivalent of short division.

[3]On Maxwell, the multiplications are actually based on a 16-bit hardware multiplier.

At 1024 bits, where $n = p$, we see that Takahashi's algorithm is approximately 2.7 times faster than RTL Quotient. When $n = 2p$, then the Takahashi's algorithm is just 17% faster and when $n = 4p$, RTL Quotient algorithm is 33% faster. When n is roughly p, the RTL Quotient algorithm has the overhead of computing both $\beta^k \bmod d$ and $\beta^{-k} \bmod d$. When $n \geq 4p$, this extra overhead is outweighed by the better asymptotic complexity of the RTL Quotient.

The Hybrid algorithm has the best of both worlds, having an $O(n/p + \log p)$ complexity without needing to compute $\beta^k \bmod d$ and $\beta^{-k} \bmod d$. At 1024 bits it is 13% faster than Takahashi's algorithm. At 2048 bits it is 2.2 times faster and at 4096 bits it is 3.6 times faster.

The 4th algorithm, division by the constant 360, is significantly faster than the others. This is due to the fact that $\beta^k \bmod d$ and $\beta^{-k} \bmod d$ can be precomputed into small tables which improves performance of the short division algorithm significantly.

Finally we note that at 4096 bits running the Hybrid algorithm, the GPU is about 54x the performance of a single Intel core and the divide by 360 is about 119x the performance of a single core. These represent good performance gains over a CPU.

CHAPTER 4

HIGH PRECISION FLOATING POINT ARITHMETIC

In this chapter we present our work on a high precision floating point library for the GPU, which provides the basic arithmetic operations, and is a small subset of the MPFR [48] library that runs on the CPU. This chapter is organized as follows: Section 4.1 presents an overview of the library, including supported features and APIs. Section 4.2 describes the implementation and important algorithms in the library and Section 4.3 presents our experimental setup, testing, results and comparisons to prior papers. Finally, Section 4.5 gives our conclusions and future directions for the library.

4.1 Library Feature Overview and API

GPUs provide extremely high performance for many applications of single precision (24-bit mantissa) floating point (FP) arithmetic and some GPUs support high performance double precision (53-bit mantissa) FP arithmetic. In this library, we support much higher precision arithmetic, where the size of the mantissa (henceforth n) can range from 1K bits to 8K bits in 1K increments.

The library provides an array oriented API, where each operation works on an array of floating point values. Each value in the array must have the same size mantissa. The operations all support six special values: zero, NaN (not a number), positive infinity, negative infinity, positive 1/infinity and negative 1/infinity. The library will detect overflows and return positive or negative infinity and will detect underflows and return positive or negative 1/infinity. Division by zero, square roots of negative numbers, and a variety of special cases such as infinity minus infinity all return a NaN value.

The IEEE 754 standard does not support ± 1/infinity, instead underflows get mapped to ± 0 and optionally raise an exception. This is fine for sequential computation, where it is clear which computation underflowed. However, since this is an array based API, we felt it was better to have explicit values, different from zero, to represent the underflow condition, so there is no ambiguity about where in the array the underflow(s) occurred.

Internally, the library uses a standard radix 2 floating point representation with a sign bit (s), an exponent (e), and a mantissa (M). The sign bit, exponent, and the special values (zero, NaN, infinities) are encoded in a 32-bit value called the *exponent word*. The mantissa is an unsigned multiple precision fixed point value, with an implicit decimal point immediately to the left of the most significant bit of the mantissa. The mantissa is always normalized, which means the most significant bit is always one, and therefore $1/2 \leq M < 1$. Formally, a non-special floating point value is defined as:

$$\text{fp value} = (-1)^s \cdot 2^{(e-bias)} \cdot M$$

The bias is a fixed value which allows an unsigned exponent to represent values less than 1. Unlike some hardware implementations, the library does not support subnormal (also known as denormalized) floating point values, where $M < 1/2$. Since we have a 32-bit exponent word, we can represent numbers as small as $10^{-323,228,491}$ and the additional overhead and complexity of supporting subnormal values was not justified by the small increase in range. In the IEEE 754 standard, the most significant one bit of the mantissa is usually not stored — it's an implicit bit. In our implementation, the most significant one bit is always explicit and is stored.

The library has support for the five standard IEEE rounding modes: *round to nearest with ties to even*, *round to nearest with ties away from zero*, *round up* (towards positive infinity), *round down* (towards negative infinity) and *truncate* (round towards zero). Since *round to nearest with ties to even* is the most common and most complex mode, we use it as the default mode for testing and timing results.

API Call	Description
fpa_init(*fpa_t* **r**, *uint32_t* **count**, *uint32_t* **precision**)	Construct a FP array on the GPU of **count** elements each with **precision** bits of mantissa
fpa_clear(*fpa_t* **x**)	Free the FP array and associated GPU memory
fpa_set_ui(*fpa_t* **r**, *uint64_t* ***ui_array**)	Set the corresponding elements of the array to the unsigned values in **ui_array**
fpa_set_si(*fpa_t* **r**, *int64_t* ***si_array**)	Set the corresponding elements of the array to the signed values in **si_array**
fpa_set_float(*fpa_t* **r**, *float* ***float_array**)	Set the corresponding elements of the array to the single precision values in **float_array**
fpa_set_double(*fpa_t* **r**, *double* ***double_array**)	Set the corresponding elements of the array to the single precision values in **double_array**
fpa_set(*fpa_t* **r**, *fpa_t* **x**, *fpa_rm_t* **mode**)	Copy the FP values from **x** to **r**. Currently, **x** must have the same size mantissa as **r**, but in a future version of the library will support extension and rounding.
fpa_sgn(*int32_t* ***results**, *fpa_t* **x**)	Return an array of *int32*s, containing the signum of each element of the FP array **x**.
fpa_cmp(*int32_t* ***results**, *fpa_t* **a**, *fpa_t* **b**)	Return an array of *int32*s, containing 1 if the corresponding element of **a** is greater than the **b** element, 0 if equal, and -1 if less.
fpa_neg(*fpa_t* **r**, *fpa_t* **x**)	Set each element of **r** to the negative of the corresponding element of **x**.
fpa_add(*fpa_t* **r**, *fpa_t* **a**, *fpa_t* **b**, *fpa_rm_t* **mode**)	Add the corresponding elements of **a** and **b** and put the results in **r**
fpa_sub(*fpa_t* **r**, *fpa_t* **a**, *fpa_t* **b**, *fpa_rm_t* **mode**)	Subtract the corresponding elements of **a** and **b** and put the results in **r**
fpa_mul(*fpa_t* **r**, *fpa_t* **a**, *fpa_t* **b**, *fpa_rm_t* **mode**)	Multiply the corresponding elements of **a** and **b** and put the results in **r**
fpa_div(*fpa_t* **r**, *fpa_t* **a**, *fpa_t* **b**, *fpa_rm_t* **mode**)	Divide the corresponding elements of **a** and **b** and put the results in **r**
fpa_sqrt(*fpa_t* **r**, *fpa_t* **x**, *fpa_rm_t* **mode**)	Compute the square root of each element of **x** and store the results in **r**

Table 4.1. Arrays based floating point library API for the GPU

Table 4.1 gives the list of supported API calls. The API calls are modeled on MPFR, but extended for arrays. There are two basic types, *fpa_t* represents an array of floating point numbers, where each element of the array has the same precision, i.e., the same length mantissa. *fpa_rm_t* is a C enum containing the five rounding modes. The API methods are briefly described in Table 4.1. If an API method has multiple *fpa_t* arguments, those arguments must have the same number of elements in the arrays and all the elements across all the arrays must have the same precision.

At this point, the API has been designed to support research into the MP floating point performance that can be attained with a GPU. To be used in practical applications, the API would need to be enhanced to support more arithmetic operations and better data import/export APIs. Further enhancements are discussed at the end of the chapter.

4.2 Implementation and Important Algorithms

The algorithms that are used to implement the floating point library fall into three categories. Handling special values (such as NaN, zero, plus/minus infinity), correct rounding for the five rounding modes, and low level algorithms for performing math on the mantissas.

We begin with the special value handling. For performance reasons, we implement the special value handling using a table driven approach. We have a very simple fast function that classifies the exponent word of an FP number as a 0 through 7 value as follows:

0	positive value
1	negative value
2	zero value
3	special value -1/infinity
4	special value +1/infinity
5	special value -infinity
6	special value +infinity
7	special value NaN

The classify function is then used to index an action table, for example see Table 4.2, which presents the action table for the *fpa_add* API function. The table encodes four possible actions: *add*, *sub*, *copy_a*, and *copy_b*, and the rest just set the result to a special value. Each of the arithmetic APIs in the library has its own action table, which drives the rest of

	a=+*value*	*-value*	*zero*	*-1/inf*	*+1/inf*	*-inf*	*+inf*	*nan*
b=+*value*	add	sub	copy_b	copy_b	copy_b	-inf	+inf	nan
-value	sub	add	copy_b	copy_b	copy_b	-inf	+inf	nan
zero	copy_a	copy_a	zero	-1/inf	+1/inf	-inf	+inf	nan
-1/inf	copy_a	copy_a	-1/inf	-1/inf	nan	-inf	+inf	nan
+1/inf	copy_a	copy_a	+1/inf	nan	+1/inf	-inf	+inf	nan
-inf	-inf	-inf	-inf	-inf	-inf	-inf	nan	nan
+inf	+inf	+inf	+inf	+inf	+inf	nan	+inf	nan
nan	nan	nan	nan	nan	nan	nan	nan	nan

Table 4.2. Special value handling for floating point addition

the computation. This approach to special values is extremely efficient on the GPU and all special values are handled in about a dozen instructions.

The next set of algorithms handles correct rounding. The idea behind correct rounding is that we wish to compute each operation to *infinite* precision, and then round the result, according to the rounding mode, to the nearest value exactly representable with a finite mantissa and exponent. Correct rounding is, in some sense, the gold standard — it's the best that you can do given the representation and finite precision. In practice, the computation need never be carried out to infinite precision, in fact, usually all that's required is two extra *guard bits*. These are usually referred to as the round bit and the sticky bit. The round bit represents one half of the least significant bit of the finite precision mantissa and the sticky bit is computed as the logical OR of all bits less significant (in the infinite precision computation) than the round bit. Consider rounding the following 16 bit mantissa to 8 bits:

$$x = \underbrace{0.1000\ 1010}_{\text{desired precision}}\ \underbrace{1000\ 0001}_{\text{to be rounded}}$$

the round bit is 1 (it's the MSB of the rounding portion), and the sticky bit would be 1 (it's the logical OR of the bits 000 0001). Applying this technique to floating point addition, assume we have two floating point values a and b to be added. Without loss of generality, assume a's exponent is greater than b's. The first step is to align the decimal points. Thus, we must shift the b mantissa to the right by the difference of the exponents. The round bit and the sticky bit are computed from the bits that are shifted out of b's mantissa. This

```
// we assume the FP values representing a and b have been loaded into unsigned integer
// variables a_exp, b_exp, a_mantissa and b_mantissa, and that a_exp > b_exp.

round_bit = 0;
sticky_bit = 0;
shift = a_exp - b_exp;

for(bit = 0; bit<shift-2; bit++) {
  if(get_bit(b_mantissa, bit)==1)
    sticky_bit = 1;
}
if(shift>0)
  round_bit = get_bit(b_mantissa, shift-1);

b_mantissa = b_mantissa >> shift;
```

Figure 4.1. Computing the guard bits for floating point addition

is shown in Figure 4.1. Once the mantissa has been shifted and the guard bits have been computed, the correct rounding algorithm for addition proceeds according to the algorithm in Figure 4.2. The algorithm calls *fpa_round* which takes five arguments: the rounding mode, the sign bit of the result value, a flag indicating the mantissa is odd, the round bit,

```
// we assume that the round_bit and sticky_bit have been computed, n is the size
// of the mantissa (in bits), mode is one of the five rounding modes, and sign is
// the sign of the result.

r_mantissa = a_mantissa + b_mantissa;
if(r_mantissa >= 2^n) {
  // r_mantissa is too long, normalize it
  sticky_bit = sticky_bit | round_bit;
  round_bit = r_mantissa & 0x01;
  r_mantissa = r_mantissa >> 1;
  r_exp = a_exp + 1;
}

// round bit will be 1 for round up, 0 for round down
round_up = fpa_round(mode, sign, r_mantissa & 0x01, round_bit, sticky_bit);

if(r_mantissa==2^n-1 && round_up==1) {
  // r_mantissa is all ones, and we have a round up --- need to normalize
  r_mantissa = 2^(n-1);
  r_exp = a_exp + 1;
}
else if(round_up==1) {
  r_mantissa++;
  r_exp = a_exp;
}

// handle overflow
if(r_exp > MAX_EXPONENT)
  r_exp = sign ? NEG_INFINITY : POS_INFINITY;
```

Figure 4.2. Correct rounding for floating point addition

and the sticky bit, and returns 1 if the mantissa should be rounded up and a 0 if the mantissa should be rounded down.

We use the same approach for all of the arithmetic operations — operate on the mantissas in the integer domain and use the two guard bits for the correct rounding. For multiplication, the round and sticky bits come from the lower half of the product of the mantissas. For division, the mantissa of the numerator is multiplied by 2^n and divided by the mantissa of the denominator. The round bit is set if the remainder is at least half of the denominator, and the sticky bit is set if the remainder is non-zero. For square root, we multiply the mantissa by 2^n, thus $X = 2^n \cdot M$ and compute the integer square root, $S = \lfloor \sqrt{X} \rfloor$. The remainder is $R = X - S^2$ and the round bit is set if $R > S$ and the sticky bit is set if R is non-zero. The algorithms for handling correct rounding are well studied and we refer the reader to [16, 73, 75, 48] for further information. For implementation on the GPU, it's generally faster to use a round word and a sticky word (32-bit values) instead of round and sticky bits.

For the computations on the mantissas, we use a warp parallel approach, where a problem instance is assigned to a warp of 32 threads which work together in parallel to solve the problem. This is similar to the approach taken for Short division (see Chapter 3). We begin by splitting the mantissa into 32 contiguous *slices* one slice per thread. For a 1024 bit mantissa, each slice is 1 limb (32 bits) in length. For a 2048 bit mantissa, each slice consists of 2 consecutive limbs (64 bits). For a 3072 bit mantissa, each slice consists of 3 consecutive limbs, etc, through 8192 bits and 8 consecutive limbs. In the code, the *Slice* type holds an unsigned value between 0 and $2^{n/32} - 1$, where n is the length of the manissa. Addition and subtraction of slices wrap around, from $2^{n/32} - 1$ back to 0, just like unsigned types in C.

The code makes extensive use of two instructions that allow threads in the same warp to communicate: the *ballot* instruction takes a single bit (true / false) from each thread in a warp and concatenates them together into a 32 bit value (the bits are ordered, where thread

0 contributes the LSB through thread 31 which contributes the MSB). The resulting 32-bit value is then distributed to each thread as the result of the ballot. The ballot operation has the following format:

$$x = \text{\_\_ballot}(\textit{true / false expression})$$

The second instruction is *shuffle* which uses a hardware cross-bar that allows an arbitrary exchange of 32-bit values between all the threads in a warp. It has the following format:

$$x = \text{\_\_shfl}(v, src)$$

The shuffle operation is equivalent to a shared memory store followed by a shared memory load:

$$\text{SHM[threadIdx \% 32]} = v;$$

$$x = \text{SHM}[src\%32];$$

In the algorithns that follow, we take advantage of the *ballot* and *shuffle* instructions and other low level features of the architecture to build efficient warp parallel unsigned multiple precision integer arithmetic algorithms. The algorithm in Figure 4.3 shows how to compare MP values represented as slices across a warp. First the algorithm compares the local slices in the variables X and Y, then ballot operations are used to build *greater* and

```
int slice_compare(Slice X, Slice Y) {
  int32_t  compare;
  uint32_t greater, lesser;

  // each thread computes signum(X-Y);
  compare=signum(X-Y);

  // next we construct two ballots for threads where X>Y and threads where X<Y
  greater=__ballot(compare==1);
  lesser=__ballot(compare==-1);
  if(greater > lesser)
    return 1;    // X>Y
  else if(lesser > greater)
    return -1;   // Y>X
  else
    return 0;    // X==Y
}
```

Figure 4.3. Comparing unsigned MP values represented as slices

lesser bit masks. Suppose $X > Y$, then there must be a thread, t, such that the local slices for thread t, $X_t > Y_t$, and for all threads, $s > t$, we have $X_s = Y_s$. Thus the *greater* ballot will have bit t set to one and all more significant bits will be zero, whereas the *lesser* ballot will have bit t and all more significant bits set to zero. We can conclude that *greater* will be greater than *lesser*. Finally, we note that the function will return the same result to all threads in the warp.

In Figure 4.4 we present the algorithm for counting the leading zeros in an unsigned MP value represented as slices across a warp. The algorithm sets *mask* to the ballot of threads where the local slice is non-zero. Next we set *count* to the clz(*mask*). Suppose t is the most significant thread with a non-zero slice, X_t. Then we have $X_s = 0$ for $s > t$, and *count* is the count of such threads. If the *count* is zero, then t must be 31. If the *count* is 1, t must be 30, etc. Thus $t = 31 - \textit{count}$. Finally, the total number of leading zeros is $\textit{count} \cdot n/32 + \text{clz}(X_t)$, which is the value returned to all threads in the warp.

```
int slice_clz(Slice X) {
  uint32_t local_count, mask;

  mask=__ballot(X!=0);
  count=clz(mask);
  if(count==32)
    return n;
  local_count=clz(X);
  return count*n/32 + __shfl(local_count, 31-count);
}
```

Figure 4.4. Count the leading zeros of an unsigned MP integer represented as slices

In Figure 4.5 we present an algorithm for incrementing a value represented as slices. The tricky case is when the carry ripples from slice to slice. The algorithm works by first computing a bit mask of the critical threads, i.e., threads where a carry in will lead to a

```
void slice_increment(Slice &X) {
  uint32_t critical, lane=1<<warp_thread;

  critical=__ballot(X==2^n/32 - 1);
  if((critical ^ critical + 1) & lane)
    X++;
}
```

Figure 4.5. Increment an unsigned MP integer represented as slices

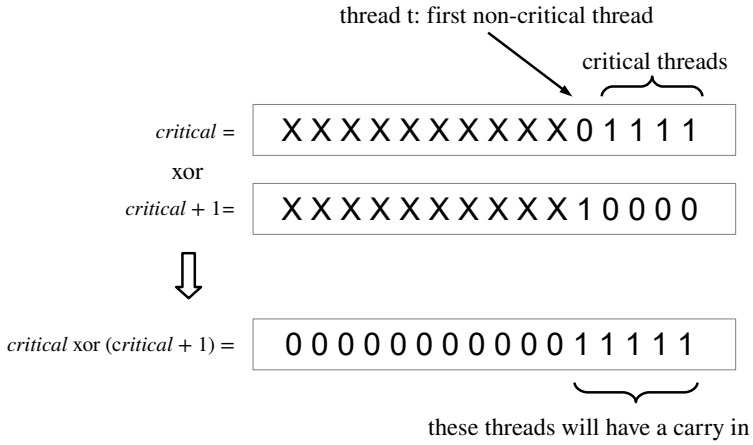

Figure 4.6. X represented as an array of bits

carry out. The ballot is stored in the variable *critical*. Next, suppose t is the least significant thread where X_t is not critical. Then incrementing the value will ripple the carry through threads 0, 1, 2, etc, up to thread t, where it will stop. Computing *critical* xor *critical* $+ 1$ builds a mask for how far the carry will ripple, for example, see Figure 4.6 which shows 4 critical threads. Each thread then tests its bit in the mask to determine if it should increment its local slice. Note, the local slices behave like unsigned values in C, so incrementing a critical slice will wrap around to zero.

```
void slice_increment(Slice &X) {
  uint32_t check, critical, lane=l<<warp_thread;

  check = X[0];
  #pragma unroll
  for(int word=1; word<n/1024; word++)
    check = check & X[word];

  critical=__ballot(check == 0xFFFFFFFF);
  critical=(critical ^ critical + 1) & lane;

  add_cc(critical, 0xFFFFFFFF);  // sets the carry flag if critical is non-zero
  #pragma unroll
  for(int word=0; word<n/1024; word++)
    X[word] = addc_cc(X[word], 0);
}
```

Figure 4.7. Low-level implemation of increment

The increment is written in terms of MP slices. Figure 4.7 shows how the algorithm is implemented at the level of unsigned 32-bit words. X[index] refers to a word within the slice X. It also makes use of three add routines, *add_cc* which adds two 32-bit unsigned values and has a side effect of setting the hardware carry flag, *addc* adds two values with carry in, and *addc_cc* takes carry in and sets carry out. These are inlined functions and the result is just a single instruction in the compiled kernel. Corresponding methods, *sub_cc*, *subc*, and *subc_cc* are used for subtract with carry in/carry out.

Negating a slice is quite similar to the above, except a thread is critical if its slice is 0 instead of $2^{n/32} - 1$ and is presented in Figure 4.8.

```
void slice_negate(Slice &X) {
  uint32_t critical, lane=l<<warp_thread;

  // computes X=1 + ~X
  // carry will ripple across least significant slices that are zero
  critical=__ballot(X == 0);
  critical=(critical ^ critical + 1) & lane;

  add_cc(critical, 0xFFFFFFFF);  // sets the carry flag if critical is non-zero
  #pragma unroll
  for(int word=0; word<n/1024; word++)
    X[word] = subc_cc(0, X[word]);
}
```

Figure 4.8. Negate an unsigned MP integer represented as slices

We find that there are many places in the library where it is beneficial to accumulate a local "carry out" word in each thread, rather than immediately and repeatedly pushing the carry out to the thread above. This is sometimes called lazy carry resolution and is discussed in Section 2.3.1. Figure 4.9 gives a fast algorithm for resolving the carries across a warp using ballot operations and a generate/propagate scheme similiar to a carry look-ahead adder.

The multiplication algorithm in Figure 4.10 computes the full product of X and Y and places the high half of the result into H and the low half into L. X, Y, H, and L are all represented as slices spread across the warp. The algorithm appears complicated, but in fact all the main loop is just iterating over the limbs of X and computing $ACC = ACC + X[i] * Y$. After each limb product is accumulated, the code shifts ACC one limb

```
void slice_resolve(uint32_t cw, Slice &X) {
  uint32_t carry, critical, lane=1<<warp_thread;

  // shift the carry word to the next higher thread, where it can be added to X
  carry=__shfl(cw, warp_thread-1);
  if(warp_thread==0)
    carry=0;

  // X=X + carry_word
  X[0]=add_cc(X[0], carry);
  #pragma unroll
  for(int word=1; word<n/1024; word++)
    X[word]=addc_cc(X[word], 0);

  // grab the hardware carry flag, set if the add carried out
  carry=addc(0, 0);

  generate=__ballot(carry!=0);
  propagate=__ballot(X == 2^(n/32)-1);

  carry=(generate*2+propagate ^ propagate) & lane;
  if(carry!=0)
    X++;
}
```

Figure 4.9. Resolve the carry words across a warp

to the right, placing the shifted out limb into the next position of L. At the end of the loop, the product is complete, but ACC can have lazy carries that need to be resolved, hence the final call to *slice_resolve*. We note, that for the Maxwell architecture, the performance of

```
void slice_multiply(Slide X, Slice Y, Slice &H, Slice &L) {
  uint32_t words_per_thread=n/1024;
  Slice    ACC(words_per_thread + 2);  // two extra words are needed for high and carries

  ACC=0;
  for(int index=0; index<n/32; index++) {
    temp = __shfl(X[index % words_per_thread], index / words_per_thread);
    ACC = temp * Y + ACC;

    // shift off the least significant word into L
    if(warp_thread == index / words_per_thread)
      L[index % words_per_thread] = __shfl(ACC[0], 0);

    temp = __shfl(ACC[0], warp_thread + 1);
    if(warp_thread == 31)
      temp=0;

    ACC = ACC + (temp<<n/32);
    ACC = ACC>>32;
  }

  for(int word=0; word<words_per_thread; word++)
    H[word]=ACC[word];
  slice_resolve(ACC[words_per_thread], H);
}
```

Figure 4.10. Multiplication algorithm for two MP values represented as slices

this algorithm could probably be improved by broadcasting and multiplying a full slice at time rather than a limb at a time, which could take better advantage of the 16-bit multiplier on Maxwell.

Division instructions on the GPU are very slow, and the algorithms that we implement for MP division attempt to avoid them as much as possible. We begin with two low level algorithms used to accelerate division of a 64-bit unsigned value by a common 32-bit normalized divisor. Common meaning the same divisor is used repeatedly, normalized meaning the most significant bit of the divisor must be a one. The approach is similar to a Barrett reduction (see Section 2.2.6), where we precompute an approximate inverse of the divisor, d, as follows:

$$\textit{approx inv} = \min\left(\left\lceil 2^{64}/d \right\rceil - 2^{32},\ 2^{32} - 1\right)$$

with the code shown in Figure 4.11. Once the approximation has been computed, division can be performed with the code in Figure 4.12. The proof of correctness is quite involved and is beyond the scope of this thesis. What is important about these routines is that they run very efficiently on the NVIDIA hardware, and use only two correction steps, versus the three required in the usual Barrett reduction. These routines also are much faster than what the compiler generates because we can assume normized divisors. Finally, we note there is nothing that limits these algorithms to 32 bits, in fact, the same algorithms can be applied to any length values, by replacing the *uint32_t* type with higher precision types, including multiple limb values such as the value of a slice on a particular thread. These algorithms

```
uint32_t approx_inv32(uint32_t divisor) {
  uint64_t approx;

  // assert(divisor>=0x80000000);

  if(divisor == 0x80000000)
    return 0xFFFFFFFF;

  approx = (-(uint64_t)divisor) / divisor;
  return 2+(uint32_t)approx;
}
```

Figure 4.11. Construct an approximation of the inverse of a 32-bit value

```
uint32_t div32(uint32_t hi, uint32_t lo, uint32_t divisor, uint32_t approx_inv) {
  uint32_t q, add, ylo, yhi;

  add = (lo<divisor) ? 1 : 2;

  // computes q=MIN(__umulhi(approx, hi) + add + hi, 0xFFFFFFFF)
  q = __umulhi(approx_inv, hi) + add + hi;
  if(q<hi)
    q = 0xFFFFFFFF;

  // computes yhi and ylo as full 64-bit product of q and divisor
  ylo = q*divisor;
  yhi = __umulhi(q, divisor);

  // first correction step
  lo = sub_cc(lo, ylo);
  hi = subc_cc(hi, yhi);
  add = subc(0, 0);
  q = q + add;

  // second correction step
  lo = add_cc(lo, divisor);
  hi = addc_cc(hi, 0);
  q = addc(q, add);

  return q;
}
```

Figure 4.12. 64 bits divided by 32 bits

are one of the most important contributions for achieving high performance and are used repeatedly in the library.

The next algorithm, shown in Figure 4.13 computes $Q = \lfloor NUM \cdot 2^n / DENOM \rfloor$, where *Q*, *NUM*, and *DENOM* are unsigned MP values represented as slices across a warp. The algorithm sets *Q* to the quotient and returns the remainder in *NUM*, and is just the classic sequential MP division algorithm described in Section 2.2.4 implemented with slices. The parallelism comes from the fact that the sub-operations (add, subtract, multiply) on slices all run in parallel with local carry/borrow accumulation. The division routine makes use of two subroutines, *APPROX_INV* and *DIV*, which are the algorithms discussed above, implemented on the limbs of a particular thread's slice. The implementation also assumes a generalized *__shfl* that works on all the limbs of a slice.

The last algorithm in this section is square root, which computes $S = \lfloor \sqrt{X \cdot 2^n} \rfloor$ where S and X are unsigned MP values represented as slices across a warp. The corner cases in the square root code make it significantly longer and more complex than other algorithms,

```
void slice_divide(Slide &NUM, Slice DENOM, Slice &Q) {
  Slice   APPROX, EST, PROD_HI, PROD_LO;
  int32_t carry_word;

  // Compute APPROX = min(ceil(2^(n/16) / DENOM_31) - 2^(n/32), 2^(n/32) - 1)
  APPROX=APPROX_INV(__shfl(DENOM, 31));

  for(index=31; index>=0; index--) {
    // Compute EST = (NUM_31 · 2^(n/32) + NUM_30)/DENOM_31, using APPROX
    EST = DIV(__shfl(NUM, 31), __shfl(NUM, 30), __shfl(DENOM, 31), APPROX);

    PROD_LO = LO(EST * DENOM);     // compute the low half of EST * DENOM
    PROD_HI = HI(EST * DENOM);     // compute the high half of EST * DENOM

    // each thread has a local carry word (cw) to track lazy carries/borrows
    cw=0;
    NUM = sub(cw, NUM, PROD_HI);        // NUM=NUM - PROD_HI, using cw for local borrows
    NUM = __shfl(NUM, warp_thread-1);   // equivalent to NUM=NUM << n/32;
    cw = __shfl(cw, warp_thread-1);     // shift cw (special code is req'd for thread 31)
    NUM = sub(cw, NUM, PROD_LO);        // NUM=NUM - PROD_LO, using cw

    while(slice_signed_resolve(cw, NUM)<0) {  // at most two corrections are req'd
      EST--;
      NUM = add(cw, NUM, DENOM);    // NUM=NUM + DENOM, using cw for local carry out
    }
    if(warp_thread == index)
      Q=EST;    // store next slice of Q
  }
}
```

Figure 4.13. Division algorithm for two MP values represented as slices

so we omit the pseudo-code. However, it follows the iterative approach from Section 2.2.8. It is quite similar to the division algorithm in Figure 4.13, in that it uses 32 iterations where each iteration generates the next slice of the result by computing an estimate of the slice and using correction steps. In both algorithms, the estimate is computed by dividing the most significant two slices of the remainder by a common divisor (in the square root case, the divisor comes from the initial approximation of the square root). To accelerate the divisions, the square root routine also uses *APPROX_INV* and *DIV*.

4.3 Experimental Testing and Results

To evaluate our algorithms we ran our experiments on an NVIDIA GeForce GTX 980 GPU running at 1.22 GHz. The host machine is an Intel Core i5-7400 running at 3.0 GHz with 16 GB of memory and is running 64-bit Ubuntu Server version 16.04.1 LTS and CUDA 9.0 with version 384.81 of the NVIDIA driver. For the CPU tests, we use the GMP

version 6.1.1 for integer arithmetic and MPFR version 3.1.5 for floating point arithmetic. For all of our experiments, we use *nvidia-smi* to set the persistence mode to 1 and set the memory clock rate to 3505 MHz and the graphics cores clock rate to 1392 MHz. These are the maximum values supported on this GTX 980 card.

The first set of tests we perform is simply to confirm that our low level *approx_inv32* and *div32* routines of Figures 4.11 and 4.12 outperform the built-in compiler routines. To test this, we launch a large number of threads where each thread divides 1000 random 63-bit numbers by a normalized common 32-bit divisor using the compiler division. When we perform the same test using the *approx_inv32* and *div32* algorithms. The compiler version runs in 20.6 ms vs 3.36 ms in our optimized approach, which is roughly a 6x speedup.

For the performance testing of the library, we generate arrays of random values. For 1024-bit mantissas, we generate 1M instances. For 2048-bit through 4096-bit mantissas we generate 500K instances and for 5120-bit through 8192-bit mantissas, we generate 100K instances. Then for each API, we run the API function 10 times on the random input and average the resulting run times, which we report Table 4.3. After the 10th run, we copy the data back to the CPU and verify it using MPFR, to ensure GPU computation

GPU Run Time (ms)								
bits:	1024	2048	3072	4096	5120	6144	7168	8192
instances:	1M	500K	500K	500K	100K	100K	100K	100K
set_ui	1.79	1.14	1.36	1.69	0.51	0.58	0.64	0.70
set_si	1.72	1.18	1.36	1.69	0.51	0.57	0.64	0.70
set_float	1.90	1.29	1.35	1.51	0.50	0.56	0.62	0.69
set_double	3.44	1.81	1.84	1.82	0.49	0.54	0.60	0.66
sgn	0.04	0.02	0.02	0.02	0.01	0.01	0.01	0.01
cmp	0.27	0.14	0.14	0.14	0.03	0.03	0.03	0.03
neg	1.54	1.35	1.97	2.64	0.67	0.81	0.96	1.10
add	2.69	1.94	2.91	3.88	0.97	1.16	1.36	1.55
sub	2.73	1.94	2.91	3.88	0.97	1.16	1.36	1.55
mul	7.11	11.83	23.82	38.70	11.31	15.80	20.95	27.02
div	64.24	63.72	81.00	102.40	25.41	31.90	38.72	47.83
sqrt	57.91	76.63	95.86	118.65	29.51	35.80	43.05	51.48

Table 4.3. GPU running time in milliseconds

CPU Run Time (ms)								
bits:	1024	2048	3072	4096	5120	6144	7168	8192
instances:	1M	500K	500K	500K	100K	100K	100K	100K
set_ui	27.85	21.29	29.82	38.29	10.93	11.02	12.70	14.33
set_si	25.18	21.27	29.68	38.25	9.39	11.05	12.71	14.32
set_float	26.00	21.34	29.83	38.29	9.87	11.09	12.81	14.47
set_double	24.44	20.47	28.44	36.34	8.92	10.52	12.14	13.64
sgn	2.97	1.68	1.67	1.66	0.62	0.62	0.62	0.62
cmp	5.11	2.68	2.80	2.78	0.84	0.84	0.84	0.84
neg	36.77	31.77	44.86	57.91	14.45	17.12	19.73	22.37
add	48.78	41.96	59.07	76.18	19.00	22.35	25.72	29.08
sub	48.82	42.02	59.19	76.17	18.98	22.34	25.73	29.07
mul	50.04	64.82	110.92	189.15	52.64	69.60	87.85	105.71
div	105.15	115.04	203.53	297.24	85.87	111.19	142.66	178.44
sqrt	147.41	123.84	176.59	256.31	66.02	84.44	100.81	123.25

Table 4.4. CPU running time in milliseconds

matches MPFR. The random values explore a range of signs, exponents and mantissas, but no special values.

For comparison, we also implement a multithreaded version of the same APIs built on top of MPFR. The threaded version uses OpenMP and runs with 8 threads on the CPU (an Intel core i5-7400) to handle the arrays. The test results are thus a socket to socket comparison between the GPU and the CPU. As on the GPU, we run each API 10 times and average the running time. These results are presented in Table 4.4. It's important to note that the arrays are quite large, and do not fit in the CPU cache, thus we are measuring *memory* $\Rightarrow$ *compute* $\Rightarrow$ *memory*, as we are on the GPU.

In Table 4.5 we present the speed-ups, where we divide the CPU running time by the GPU running time for the same APIs with the same size mantissas. The last column is the average of the speed-ups across all the sizes in each row. For the set APIs, the GPU is roughly 17x faster. For the comparisons, the GPU is over 20x faster. For *neg*, *add*, and *sub*, the GPU is roughly 19x faster. For the compute intensive APIs, *mul* is 4.9x faster, *div* is 2.9x faster, and *sqrt* is 2.2x faster.

bits:	1024	2048	3072	4096	5120	6144	7168	8192	
instances:	1M	500K	500K	500K	100K	100K	100K	100K	average
set_ui	15.6	18.7	21.9	22.7	21.4	19.0	19.8	20.5	19.9
set_si	14.6	18.0	21.8	22.6	18.4	19.4	19.9	20.5	19.4
set_float	13.7	16.5	22.1	25.4	19.7	19.8	20.7	19.9	19.9
set_double	7.1	11.3	15.5	20.0	18.2	19.5	20.2	20.7	16.6
sgn	74.3	83.8	83.4	82.8	62.0	61.7	61.5	61.5	71.4
cmp	18.9	19.1	20.0	19.9	28.0	28.0	27.9	27.9	23.7
neg	23.3	24.2	22.5	21.7	21.5	20.7	20.4	20.2	22.0
add	18.1	21.6	20.3	19.6	19.6	19.3	18.9	18.8	19.5
sub	17.9	21.7	20.3	19.6	19.6	19.3	18.9	18.8	19.5
mul	7.0	5.5	4.7	4.9	4.7	4.4	4.2	3.9	4.9
div	1.6	1.8	2.5	2.9	3.4	3.5	3.7	3.7	2.9
sqrt	2.5	1.6	1.8	2.2	2.2	2.4	2.3	2.4	2.2

Table 4.5. Speedup table: CPU running time / GPU running time

4.4 Comparison to Prior Work

There are several papers that have implemented multiple precision integer or floating point libraries on the GPU which were discussed in Section 2.5.4. Here we compare our results to the three most recent papers. The most straightforward comparison is to Honda, Ito, and Nakano's [61] MP integer multiplier. They use a warp synchronous approach (much like our own) to implement their multiplier and test at power of two sizes from 1024 bits through 32768 bits on a GTX 980. We use the same card and support some of the sames sizes:

	Throughput Ops/Sec			
	1024 bits	2048 bits	4096 bits	8192 bits
Honda et al. [61]	65.8 M	13.3 M	3.8 M	970 K
Ours	142.7 M	42.2 M	13.5 M	3.7 M
Speed Up	2.2	3.2	3.6	3.8

Table 4.6. A comparison of this work to Honda, Ito, and Nakano's

Nakayama and Takahashi [85] implement array oriented high precision floating point addition and multiplication in CUMP. They evaluate their library running at three sizes, 384 bits (roughly 100 decimal digits), 3328 bits (roughly 1000 decimal digits) and 33280

bits (roughly 10,000 decimal digits) on a Tesla C2050 GPU card. The C2050 is quite an old card which does not support the *__shfl* instruction. However, with some changes to our library, we were able emulate the *__shfl* operations using shared memory and were able to get our library up and running on a C2050. We compare their reported performance to our library at 384 bits and 3328 bits.

	Addition			
	size	throughput	size	throughput
Nakayama and Takahashi	384 bits	240 M	3328 bits	41 M
Ours	1024 bits	64.6 M	4096 bits	42.4 M

	Multiplication			
	size	throughput	size	throughput
Nakayama and Takahashi	384 bits	145 M	3328 bits	3.6 M
Ours	1024 bits	16.9 M	4096 bits	4.2 M

Table 4.7. A comparison of this work to Nakayama and Takahashi's

At 3328 bits, our library has greater precision and greater performance for both addition and subtraction. If we scale the 384 bit results by 2.67 for addition, we find our performance is roughly 172 M/sec and by 2.67^2 for multiplication, our performance is roughly 120 M/sec. This is less than what is achieved by Nakayama and Takahashi, but there are several things to note. First, the lack of a *__shfl* instruction significantly impacts the performance of our library on the C2050. Second, our library supports a rich set of features, such as correct rounding and special values that are not supported in Nakayama and Takahashi's library. The associated overheads are especially burdensome at small sizes.

Finally we compare our library to CAMPARY by Joldes et al. [68]. CAMPARY is an MP floating point library that supports correct rounding and a *quick and dirty* mode which will be close to correctly rounded result, but is not guaranteed in all cases. CAMPARY represents a floating point value as a sequence of double precision values, similar to David Bailey's QD library. CAMPARY is very fast at low precision (106-212 bits) but performance suffers at larger sizes. In [68] Joldes et al. implement a Hénon map routine using

CAMPARY and measure the performance at sizes from 106 bits to 424 bits on a Tesla C2075 card. These results are presented in Table 4.8:

Precision	Hénon Map Iterations per second
106 bits	227 M
159 bits	76 M
212 bits	37 M
318 bits	15 M
424 bits	8 M

Table 4.8. Performance of the Hénon map implmented with CAMPARY running on a Tesla C2075 GPU

We have implemented the same Hénon map iteration using our library. Unfortunately, our library doesn't support sizes less than 1024 bits at this time, so we compare our library at 1024 bits to COMPARY at 424 bits. We see from Table 4.8 that each doubling of the precision results in roughly a factor of 5 slowdown, thus we would expect CAMPARY at 848 bits to achieve roughly 1.6 M iterations per second. At 1024 bits, our library achieves a throughput of 4.87 M iterations per second on a C2075, about a factor of 3 faster.

4.5 Conclusion and Future Work

The floating point library is a good start toward a high precision library for the GPU. It has rich feature set, including support for all five IEEE 754 rounding modes and supports a superset of the IEEE 754 special values. The performance of the library is also quite strong. It outperforms all of the GPU MP libraries with one small exception, Nakayama and Takahashi's CUMP at very small sizes. We believe that comparing our library to CAMPARY and CUMP on a more recent Tesla GPU with warp shuffle support, such as a K40, would show even further gains.

The socket to socket comparison against an Intel processor running MPFR is also quite strong. For linear operations: addition, subtraction, comparison, the GPU library is over

19x faster than the CPU library. For more compute intensive operations: multiply is 5.4x faster, divide is 3x faster and square root is 2.3x faster.

However there are some important caveats. To date the library is best classified as *research-ware*. We have tested the library with random data, but that does not perform a thorough test of the corner cases which exist in an FP library, especially with the complex carry chains that can occur using a warp oriented approach with local carry accumulation. The code has been written carefully and we have done limited ad-hoc testing for many cases, but to thoroughly test the library would be a very big project, on the order of the time to write the software in the first place. We leave this testing for the future.

In addition, there are a number of enhancements that should be done in the future, which would improve the usability of the library. These include:

- Support fused multiply and add / fused multiply and subtract.
- Support support a dot product operation, based on fused multiply and add, which would enable an efficient matrix multiply routine.
- Allow multiple instances per warp, specifically, 1, 2, 4, 8 and 16 instances per warp.
- Improve the internal algorithms for better performance.
- Better import/export and input/output routines.

CHAPTER 5

LARGE UNSIGNED INTEGER ADDITION, SUBTRACTION AND MULTIPLICATION

In this chapter we present our work on large unsigned integer addition, subtraction, and multiplication. This chapter is organized as follows: Section 5.1 presents our work on large integer addition and subtraction. This section describes the problem, algorithms used, and experimental results. In Section 5.2 we report on our prior efforts to implement an FFT approach to large integer multiplication, published in [39]. We present the basics of FFT multiplication, the memory layout, implementation details, optimizations, and experimental setup and results.

5.1 Large Integer Addition and Subtraction

In this section, we look at the problem of adding two very large integer values, stored in contiguous blocks of memory and writing the resulting sum to a third block of memory. The scales that we are interested in range in size from 1 megabit to 8 gigabits. On the CPU, this is quite an easy problem, just run through the blocks from least significant word to most significant word, pushing the carry along. A single CPU core can easily saturate the memory bus, so there is no need to use a multi-core/multi-threaded approach. On the GPU, this not the case. It's not possible for a single thread, warp, block or even streaming multiprocessor (SM) to saturate the GPU memory bus. To achieve saturation, a large number of blocks must be run across multiple SMs. Thus on the GPU the adder will need to break the problem up into a number of chunks that are run in parallel. These chunks are then assigned to blocks, or warps within a block. Since communication between blocks is

very restricted, it's best to structure the adder as two kernels, where the first kernels computes the sums and carry outs, and the second kernel resolves the carries that cross block boundaries. Since addition is not a compute intensive operation, our goal for the kernels is to achieve as close to full memory bandwidth as possible, even in the presence of very long carry chains across block boundaries.

Our first attempt to implement the kernels for large addition had terrible performance. The bandwidth achieved was just a small fraction of what we had expected. It was so bad that we thought there must be a bug in the code. After checking it carefully, we realized the code was fine, but something about the access pattern of the kernel was causing the memory subsystem to significantly underperform. To understand the influence of access patterns on bandwidth, we looked at a much simpler problem – computing the logical XOR of two blocks of memory of 8 gigabits (256M words) each and writing the result to a third block. We implemented and tested 5 kernels (described below) based on common GPU design patterns. The 5 kernels all use 128-byte aligned and coalesced global memory read and writes and our expectation was that we should see only minor differences in the bandwidth achieved, but that is not what we found.

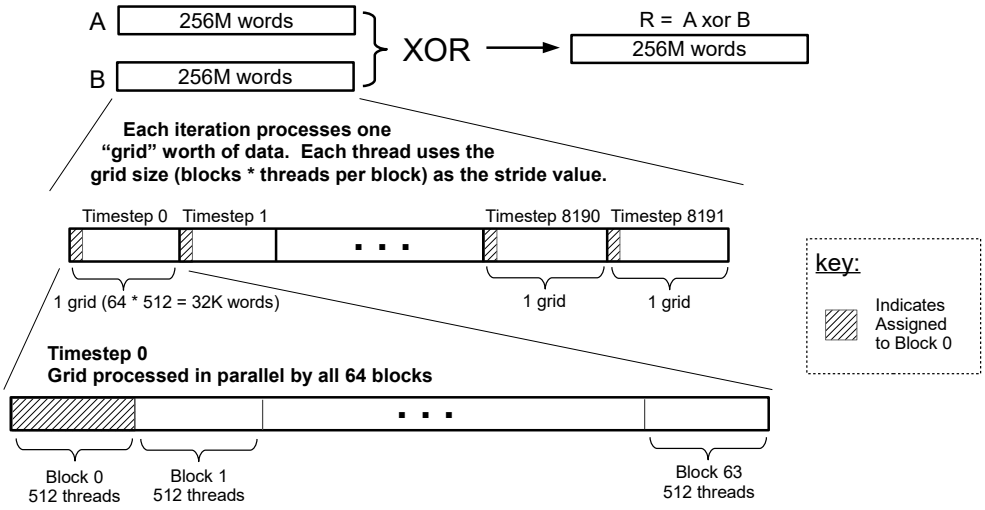

Figure 5.1. Grid Stride Loop processing using 64 blocks and 512 threads per block

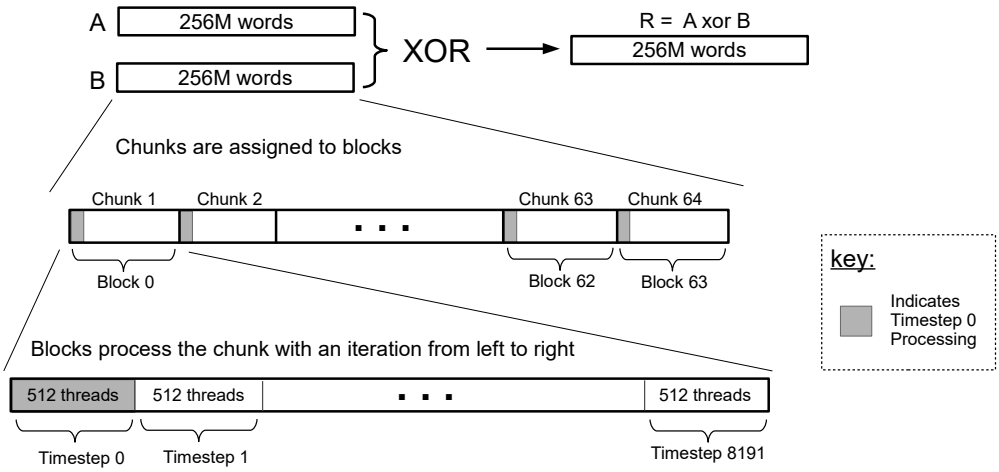

Figure 5.2. Block Stride Loop processing using 64 blocks and 512 threads per block

Thread per Word: here we use the standard approach of launching enough blocks so that we have a thread per result word. Each thread loads a single word from A and B, computes the exclusive OR and writes the resulting word to R and then terminates.

Grid Stride Loop: this another common technique. Instead of launching a thread per word, we launch a much smaller fixed grid of threads, whose size is typically chosen such that all threads can be run simultaneously on the device. The threads iterate through the arrays using the grid size as the stride value. This is generally quite an efficient approach. This is shown in Figure 5.1 which uses 64 blocks and 512 threads per block.

Block Stride Loop: in this approach, a fixed number of blocks are launched. The arrays are then split into equal sized contiguous chunks, and exactly one chunk is assigned to each block. The blocks then run in parallel and each block iterates through its chunk using the block size as a stride value for the iteration. This is shown in Figure 5.2.

Warp Stride Loop: this is similar to a block stride loop. The array is again split into equal sizes contiguous chunks, and exactly one chunk is assigned to each warp. The warps all run in parallel and each warp iterates through its chunk using a stride value of 32 (the number of threads in each warp).

Approach	BW (gigabytes/sec) no synchthreads	bandwidth (gigabytes/sec) with syncthreads
Thread per Word	203.2	203.2
Grid Stride Loop	170.9	197.8
Block Stride Loop	169.0	197.2
Warp Strde Loop	22.5	22.5
Grid Stride Loop (4 word Read/Write)	190.2	196.5

Table 5.1. XOR kernels and memory bandwidth achieved

Grid Stride Loop using 4 Word Read/Writes: the last approach makes use of grid stride loop, but instead of processing a single word on each iteration, each thread processes 4 words at a time. The memory system supports 1 word, 2 word and 4 words per thread for reads and writes. This kernel makes use of the 4 word reads and writes.

For each kernel, we run two tests, the first runs the kernel as described above. The second test adds a *syncthreads* operation after the write step. The syncthreads is not required for correctness, but as we'll see, it can have an impact on performance. For each test, we do 10 runs with 4 blocks of 512 threads per SM and we calculate the average running time. The bandwidth is then computed as 3 GB divided by the running time and these results are shown in Table 5.1. The tests are performed on a GTX 980, which has 16 SMs, so a total of 64 blocks and 1024 warps (16 warps per block) are launched per run.

On the GPU, there are two well known issues that can cause the memory bandwidth to drop, the first is uncoalesced reads or writes, which occurs when the threads of a warp issue reads or writes to non-consecutive addresses. The second well known issue is when the warp is accessing addresses that are not 128-byte aligned. The cache tends to hide some of these issues, but they can still impact the bandwidth significantly. However, in the 5 kernels above, all reads and writes are 128-byte aligned and warp coalesced. Further, since we are streaming through extremely large arrays, virtually all reads and writes go directly to global memory and the cache has a negligible impact on performance.

The results in Table 5.1 can be clusted in the three groups. On the low end, *Warp Stride Loops* have terrible performance of 22.5 GB/s or roughly 1/9th of the potential bandwidth. On the high end, above 195 GB/s, achieved by *Thread per Word*, *Grid Stride Loops with Syncthreads* and *Block Stride Loops with Syncthreads* almost fully saturates the memory bandwidth. Then there are the results in the middle that range from 165 GB/s to roughly 190 GB/s. The middle range are all derived from kernels without syncthreads. Collectively, these results suggest that as the warps are spread further and further apart, i.e., with less locality of reference in global memory, the bandwidth drops. This explains the terrible performance of *Warp Stride Loops* where each of the 1024 warps are accessing different regions of memory (3096 different regions to be precise, 1024 for each of the A and B arguments, and 1024 regions for the result), versus the excellent performance of *Thread per Word* and *Grid Stride Loops with Syncthreads*, where all threads are accessing the same regions of memory. The interesting cases are the kernels without syncthreads. We believe that what's happening is that as the kernel is running, some of the warps get further and further out of sync with the others. So the kernel starts off with excellent locality of reference, but as it runs, the warps get out of sync and spread out in memory, and the locality of reference drops to the point where it starts to impact the bandwidth. The other interesting case is the excellent performance of *Block Stride Loops with Syncthreads*, in which the 64 blocks are each accessing different regions of the arguments and result. This implies that the memory subsystem can efficiently handle accesses to 192 regions of memory, but somewhere between 192 regions and 3096 regions, the memory subsystem hits a big performance cliff. NVIDIA does not describe the hardware internals in any detail, but there are two scenerios that seem plausible. First, the GPU uses a paged memory system, so perhaps we're thrashing the page table translation look-aside buffer. Another possibility is that modern DRAMs architectures employ a system of open pages. Accesses to the open pages are much faster than to other pages. Maybe we're actually passing a page threshold in the DRAMs which causes performance to drop significantly. In any case, the

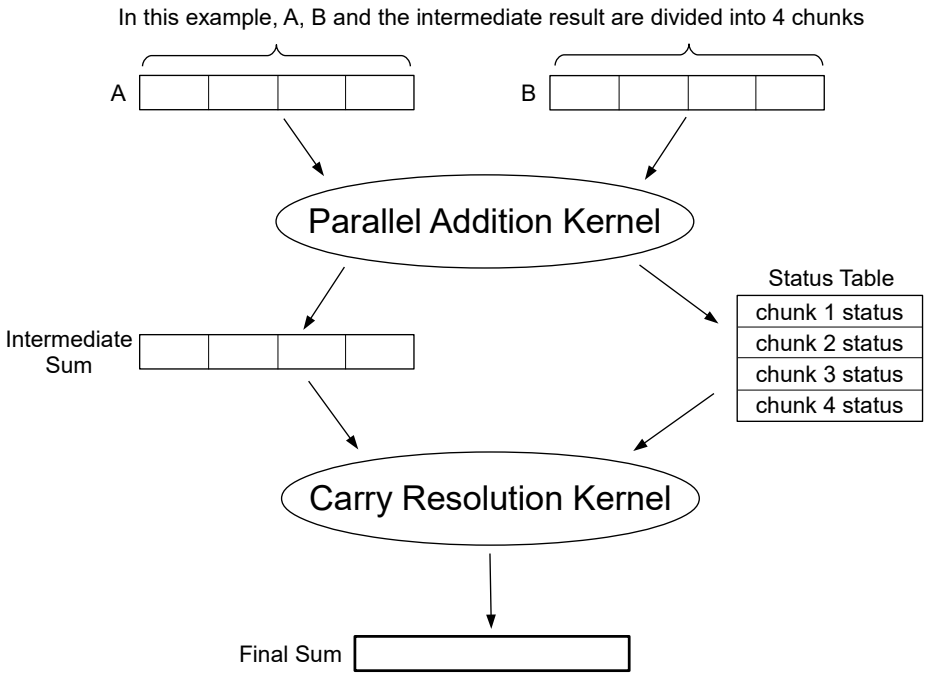

Figure 5.3. Large addition using two kernels: Parallel Chunk Addition followed by Carry Resolution

simplistic view that GPU main memory is flat, i.e., a uniform access time regardless of locality of reference, is clearly wrong. There can be a wide variance in achieved bandwidth and this needs to be factored into the design of certain memory bound kernels, such as large addition.

Next we turn our attention to the construction of the large adder. As mentioned earlier, due to the restrictions in communication between GPU blocks, it's best to perform the addition using two kernels. The first breaks the A and B arguments into chunks, then corresponding chunks from A and B are added. The chunk sums are all run in parallel. The second kernel then resolves the carries that cross the chunk boundaries. This approach is shown in Figure 5.3. The carry resolution kernel needs to know which chunks generate carry-outs and which chunks are *critical*, i.e., a carry-in will produce a carry-out. This is tracked in the *Status Table*, where each entry is one of three states *No carry out*, *Critical*, and *Carry out*. It's worth noting that carries can ripple across critical chunks, thus to

compute the carry in for the k^{th}, potentially requires examining the chunk status for all chunks less than k. For this reason, it's preferable to keep the number of chunks relatively small.

The the *Thread per Word* and *Warp Stride Loop* approaches both achieve excellent memory saturation, however, it can be seen in Figure 5.1 that every 512 words will be handled by a different GPU block, and thus the A and B arguments must be broken into 512 word chunks. For a 256M-word problem, we'll have 512K chunks and a corresponding number of status entries. Thus, the carry resolution kernel potentially has to do a lot of computation just to determine the carry in for each chunk, which makes the *Thread per Word* and *Warp Stride Loop* approaches unattactive for the large adder.

```
void addition_kernel(uint32_t *SUM, uint32_t *status, uint32_t *A, uint32_t *B) {
  int  carry_forward=0, sum_word, a_word, b_word;
  bool critical=true;

  block_size = ... the size of the GPU block, i.e., 512 ...
  chunk_size = ... compute the size of my chunk ...
  chunk_offset = ... compute the offset (in A and B) of my chunk ...

  // we assume the chunk_size is evenly divisible by the block size
  for(int index=0; index<chunk_size / block_size; index++) {
    a_word = A[chunk_offset + block_size*index + thread_id];
    b_word = B[chunk_offset + block_size*index + thread_id];

    sum_word = a_word + b_word;
    if(thread_id==0 && carry_forward!=0)
      sum_word++;

    ... resolve carries across the threads in the block ...
    ... set carry_forward to the carry out of the last thread ...

    if(sum_word!=0xFFFFFFFF)
      critical=false;

   SUM[chunk_offset + block_size*index + thread_id] = sum_word;
  }

  // since the initial carry in was 0, a block can either be CARRY_OUT or
  // critical, but it can't be both
  if(carry_foward==1)
    status[block_id]=CARRY_OUT;
  else if(block_id>0 && critical is true in all threads)
    status[block_id]=CRITICAL;
  else
    status[block_id]=NO_CARRY_OUT;
}
```

Figure 5.4. Parallel Chunk Addition kernel pseudo-code

Using the *Warp Stride Loop* or *Block Stride Loops* approaches keeps the status table to a much more managable size, 1024 entries for WSL, and 64 entries for BSL. Our first attempt to implement large addition used a WSL approach, but we ran into the bandwidth limitations as described above. Our second implementation uses a BSL approach and it achieves good bandwidth saturation across a range of size. The pseudo-code for the chunk addition kernel is presented in Figure 5.4. Carries that are internal to the chunk are resolved using *__ballot* operations within warps (this technique is described in depth in Chapter 4, see Figure 4.9), shared memory across warps in the block, and with the *carry_forward* variable across iterations of the block stride loop. When the addition for the chunk is complete, the routine writes its entry to the status table and terminates. The pseudo-code for inter-chunk carry resolution is presented in 5.5. This routine also uses *__ballot* operations and shared memory to handle communications within the warp and shared memory across warps in the block.

```
void resolution_kernel(uint32_t *SUM, uint32_t *status) {
  __shared__ uint32_t shared_status[64];
  uint32_t            carry = CRITICAL;
  int                 index;

  if(thread_id<64)
    shared_status[thread_id]=status[thread_id];

  __syncthreads();

  for(index=block_id-1;  carry==CRITICAL ; index--)
    carry=status[index];

  // if the prior block did not carry out, we're done
  if(carry==NO_CARRY_OUT)
    return;

  for(int index=0; index<chunk_size / block_size; index++) {
    sum_word = SUM[chunk_offset + block_size*index + thread_id];

    ... if all threads in the block less than my thread_id have
        sum_word==0xFFFFFFFF, then increment sum_word ...

    SUM[chunk_offset + block_size*index + thread_id] = sum_word;

    ... if any thread has sum_word != 0, then break ...
  }
}
```

Figure 5.5. Parallel Chunk Resolution kernel pseudo-code

To test the performance of the large adder, we generate blocks of random data for the A and B arguments and measure the total running time of the two kernels. We perform 10 runs for each size and average the running times together. From the running times, we compute the bandwidth achieved. These results are presented in Table 5.2, along with the bandwidth achieved by XOR using block stride loops for the same size blocks and the bandwidth achieved by the system *cudaMemcpy* function. As we can see, from the table, at the small sizes (128K, 256K, and 512K words), the Large Adder is considerably slower (35%, 25%, and 16% respectively) than the XOR kernel. At larger sizes, the Large Adder performs almost as well as the XOR kernel and cudaMemcpy function. Most of the performance difference at the small sizes seems to be driven by the fact that the Large Adder runs two kernels instead of one, and thus has twice the kernel launch overhead. To test this, we run only the first adder kernel and measure the bandwidth performance achieved. At 128K words, the first kernel achieved 121.1 GB/sec and at 256K words, the first kernel achieved 151.1 GB/sec, which closes the performance gap considerably. At the larger sizes, the overhead of two kernel launches is amortized over a much larger number of bytes which which shrinks the performance gap.

	Bandwidth Gigabytes/sec		
Size in Words	Large Adder	XOR	Memory Copy
128K	86.9	134.6	127.6
256K	119.3	159.4	155.4
512K	143.7	170.6	165.8
1024K	165.1	182.9	179.8
2048K	179.7	189.1	189.0
4096K	188.7	194.0	193.7
8192K	193.0	195.9	195.7
16M	195.7	197.1	197.3
32M	197.0	197.8	197.4
64M	198.2	198.3	197.9
128M	197.4	197.4	198.1
256M	197.0	197.0	197.8

Table 5.2. Bandwidth achieved by the Large Adder kernels, the XOR kernels, and the CUDA memory copy routines for various sizes

There is still one thing to note. These kernels are running on random data. Thus on average, carries rarely propagate far and it's extremely improbable that a block could be critical. However, not all computations are on random data. For example, we might be computing $X - Y$ where X and Y are very large integers, and Y happens to be equal to X. In this case, the carry will literally ripple through the entire large integer from the first block to the last. Thus, the first kernel will read $2n$ words and write n words (all words will be 0xFFFFFFFF), and the second kernel (carry resolution) will read the n words and overwrite all of them with zeros. The total memory reads and writes will be $5n$ words, where n is the length of the arguments. The problem here is that the first kernel is writing out critical words at the start of the block and the carry in isn't available until the second kernel, which might need to flip all those words. We can improve the kernels as follows. Instead of the first kernel writing least significant critical words, it should count them, but not write them. Then it writes this count into the status table. The second kernel, which has the block carry-in information, can then write the critical ones if there was no carry-in, or write zeros and increment the next word if there was a carry-in. This brings the total reads and writes down to $3n$ (plus a small number of words for the increments) from $5n$ words. We implement this approach in our third and final set of kernels.

To test this, we generate random data for the A argument, and set the B argument to the bit-wise complement of A. We then overwrite each k^{th} word of B with random data. Thus, on average, we have k length carry chains, and on average half of them have carry-ins and half do not.

We compare the performance of the *version 2* of the large adder which does not implement the critical word counting to *version 3* which does on a 256M-word addition with carry chain lenghts (k) that range from 10 to 10^9 words. The results are presented in Table 5.3. The *Effective Bandwidth* is computed as 3 GB divided by the kernel running time. For short carry chains, only a small number of words need updating by the second kernel and the performance of the two versions is roughly equal. But when the carry chains get to

	Effective Bandwidth Gigabytes/sec	
k, the carry chain length	Large Adder Version 2	Large Adder Version 3
10	197.0	196.9
100	196.6	196.3
1,000	196.8	196.7
10,000	196.7	195.7
100,000	195.5	196.0
1,000,000	185.0	196.8
10,000,000	147.2	199.0
100,000,000	151.2	196.2
10^9 with carry in	116.9	195.3
10^9 no carry in	196.8	195.7

Table 5.3. Effect of long carry chains on achieved bandwidth

about 1M words, *version 2* performance starts to drop (because it's reading and writing far more than the required $3n$ words). When k is greater than the length of the arguments, then the entire addition is critical. In the case where there is a carry in, *version 2* must flip every word and the effective bandwidth is only 60% of *version 3*.

Large subtraction is implemented using a twos complement adder. The code is almost identical to large addition, the only differences are that in the first kernel, B is complemented after loading, and the Carry Resolution kernel forces a carry in to the first block. As expected, the bandwidth performance of large subtraction routine matches that of large addition routing.

We conclude this section with a short discussion of bandwidth utilization. According to NVIDIA specifications, the GTX 980 uses a GDDR5 memory subsystem with a 256-bit interface bus running at 3.5 GHz, with a theoretical bandwidth of 224 GB/sec. The large addition and subtraction kernels achieve between than 190 and 200 GB/sec when the arguments are larger than 8M words. This is better than 85% of peak theoretical bandwidth. It's also worth noting that not all of the bandwidth is available for data in CUDA applications. It's shared with the frame buffers, the virtual memory paging system, various caches (instruction, texture, constant), various on chip microcontrollers/microprocessors that run

the GPU operating system, etc. During our various tests, the best bandwidth we've seen delivered to an application is 203 GB/sec, so the large addition and subtraction routines are within a few percent of the best that's possible for large arguments on a GTX 980.

5.2 Large Unsigned Integer Multiplication

In this section we report on our earlier work [39], where we explore FFT multiplication on the GPU using a 64-bit finite field, $\mathbf{F}_p$, where $p = 2^{64} - 2^{32} + 1$. The prime, p, is a special modulus and supports a very fast modular reduction that is suitable for 32-bit ALUs. Moduli of this form were discussed earlier in Section 2.2.7. The code base for this work was designed for the Fermi architecture, but the same code still runs without change on later generations and in Section 5.2.5 we report results of the original code running on a GTX 980. There are a number of optimization that could be made to better exploit later architectures and these changes are described in Section 5.2.6, Conclusions and Future Work.

One of the operations that GPUs are able to perform quickly is an FFT. Thus, it's reasonable to expect that a multiplication algorithm based on the FFT would be an effective means of obtaining good performance on a GPU. In 1971 Schönhage and Strassen published [94], an arbitrary precision multiplication algorithm that uses a recursive approach where each level of the recursion runs an FFT multiplication in the ring of integers modulo $2^{k_i} + 1$, where i is the level in the recursion with $k_{i+1} \approx \sqrt{k_i}$. The idea to use FFT convolutions for multiplication is older and generally attributed to Strassen circa 1968, however, there are some limitations to using a single FFT: for any fixed finite ring / finite field, there is a maximum size integer that is guaranteed not to overflow the ring/field. However, for practical purposes, as we'll see, a 64-bit field can efficiently support very large integers, up to sizes of 4 GB times 4 GB, which is sufficient for our needs on the GPU.

Figure 5.6 presents Strassen's FFT convolution multiplication algorithm. The FFT computations can be done in $\mathbf{C}$ (the complex numbers) with floating point operations, however,

```
1: Given a base b, compute the fast Fourier transform of the digits (with respect
   to the base) of A and B, treating each digit as an FFT sample.

2: Multiply the FFT results, component by component:  set C[i] = FFT(A)[i]*FFT(B)[i]

3: Compute the inverse fast Fourier transform:  set C' = invFFT(C)

4: Resolve the carries: whenever C'[i] >= b:
      set C'[i+1] = C'[i+1] + (C'[i] div b)
      set C'[i] = C'[i] mod b.

5: return C'
```

Figure 5.6. Strassen Multiplication

the error analysis is very difficult. Instead, most implementations perform the computation using a finite field $\mathbf{F}_p$ consisting of the integers modulo p, for some prime p. Using integers, the FFT computations are exact, however, there are several important restrictions on b (the base), p (the size of the finite field), and k (the size of the FFT):

- The field $\mathbf{F}_p$ must have a k^{th} primitive root of unity.
- The length (in base b digits) of the product of A times B must be less than k.
- The maximum value in the convolution must fit in the field, that is, $\frac{k}{2}(b-1)^2 < p$.
- Multiplication in the field $\mathbf{F}_p$ requires computing modulo p. Therefore, the existence of a fast modulo p operator is desirable.

Our first attempt to implement large integer multiplication on the GPU [38] used a 32-bit field, with $p =$ 0xFFF00001, $k = 65536$, $b = 256$ (for byte sized samples), and a single word Montgomery reduction to avoid the slow modulo operations on the GPU. Unfortunately, the largest values that could be multiplied were 32KB times 32KB and for larger size, the code switched to an implementation of Karatsuba built on top of the 32KB "digit" multiplier. The performance was lacklaster and worse yet, as the size increased, GMP using a comparable 64-bit CPU outperformed our GPU implementation. The main problem was the choice of a 32-bit field which required another algorithm for very large sizes.

For the next implementation, we investigated a variety of 64-bit primes for finite fields, but we found that $p = 0$xFFFFFFFF00000001, i.e., $2^{64} - 2^{32} + 1$ has a number of especially useful properties:

- The field $\mathbf{F}_p$ supports power of two FFT sizes up to 2^{32}. It also supports FFT sizes of 3, 5, and 7.
- On 32-bit processors, there is a *very* fast direct method to compute x modulo p for any x (without Montgomery reductions).
- The number 8 is a 64^{th} root of unity. This means that 64-point FFTs can be done with shifts rather than requiring 64-bit by 64-bit multiplications.

5.2.1 Fast Modulo

The computation of an FFT in the field $\mathbf{F}_p$ requires the three field operators: addition, subtraction and multiplication. Addition and subtraction modulo p are straightforward. However, multiplication requires computing z modulo p where z is an arbitrary 128-bit number. Fortunately, choosing $p = 2^{64} - 2^{32} + 1$ simplifies the computation. The 128-bit number z can be represented as $z = 2^{96}a + 2^{64}b + 2^{32}c + d$ (where a, b, c and d are each 32-bits). Using two identities of p, namely, $2^{96} \bmod p = -1$ and $2^{64} \bmod p = 2^{32} - 1$ we can rapidly reduce z as follows:

$$\begin{aligned} z &\equiv 2^{96}a + 2^{64}b + 2^{32}c + d \qquad (\bmod p) \\ &\equiv (-1)a + (2^{32} - 1)b + (2^{32})c + d \\ &\equiv (2^{32})(b + c) - a - b + d \end{aligned}$$

Although we discovered this prime and the modulo algorithm on our own, it turns out that it is well known in the cryptography community and there are whole families of primes of the form $2^m \pm 2^n \pm 1$ with fast modulo reductions. The seminal paper is by Jerome Solinas [95] hence they are oftened called Solinas Primes.

5.2.2 Multi-byte Sample Sizes

The first step of the Strassen algorithm is to break A and B into digits. In theory, b (the base), could be any number, but for performance reasons b should be a power of 2. To keep the complexity of the load and store routines reasonable, samples should be byte-aligned. Thus, where we formerly used 1-byte samples, we are now able to support 2-byte and 3-byte samples, i.e., $b = 2^{16}$ or 2^{24}. Recall the restriction that the maximum convolution value must fit in the field, $\frac{k}{2}(b-1)^2 < p$. Thus, for 2-byte samples, k, (the size of the FFT) must be less than or equal to 2^{33} and for 3-byte samples, k must be less than or equal to 2^{17}. The more bits in each sample, the shorter the FFTs and the more efficient the computation. Therefore, if the product of A times B is less than $3 \cdot 2^{17}$ bytes in length, our implementation will use the more efficient 3-byte samples. Otherwise it uses 2-byte samples.

Although k can be up to 2^{33} for 2-byte samples, there is no corresponding primitive root of unity in the field. The largest power of two FFT size supported by the field is 2^{32}. Therefore the largest product that can be computed (using power of two FFTs) with 2-byte samples is $2 \cdot 2^{32}$, or 8 GB. However, memory limitations alone will often preclude working with values this large, without resorting to alternative algorithms that stage data from tertiary storage.

5.2.3 FFT Layout and Implementation

We had three key goals in designing the FFT layout for the GPU:

1. Avoid launching kernels whose sole function is to transpose global memory.
2. Always use coalesced accesses for GPU global memory.
3. Maximize use of the fact that 64-point FFTs can be done with shifts rather than multiplies.

Avoiding Transpose Operations: there is an elegant way to lay out each FFT size that accomplishes all three goals. We begin by factoring k (the FFT size) into $k = 8xy$ where $1 \leq x \leq 64$ and $y = 64^n$. Figure 5.7 shows the layout for several sizes of FFT. As

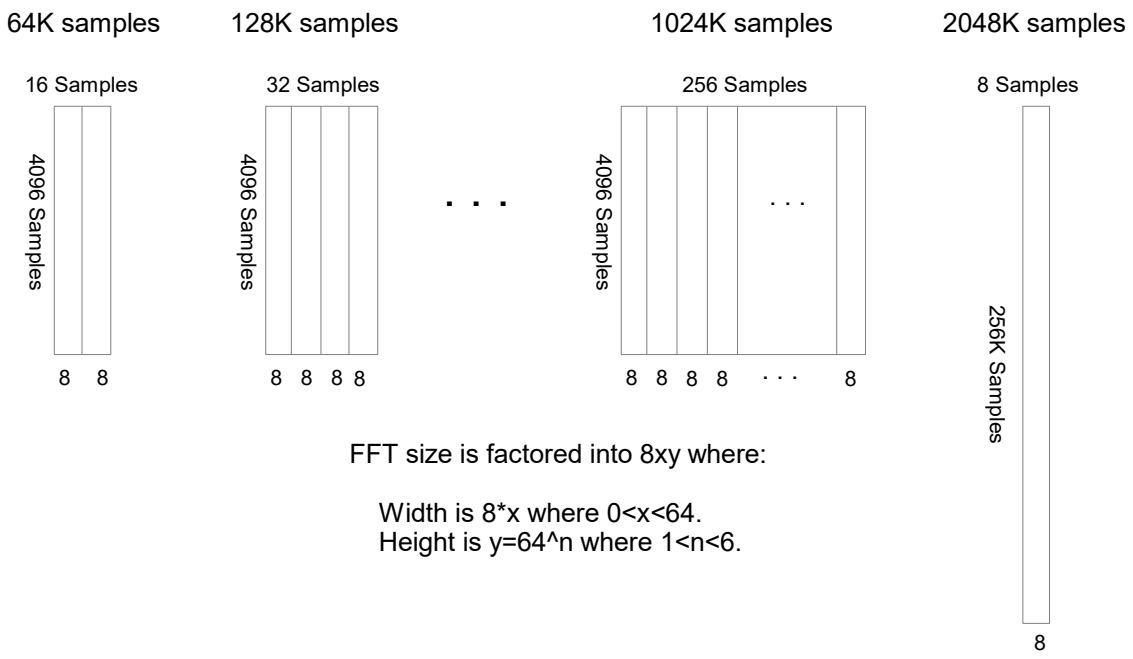

Figure 5.7. FFT layouts as the number of samples is doubled

the number of samples doubles, the width of the matrix doubles, until the matrix is 256 samples wide. Upon the next doubling, the width is shrunk to 8 samples wide and the height is expanded by a factor of 64. We store the data in row-major order, i.e., the rows of the samples are stored consecutively in memory. At the top level, we use the standard four-step Cooley-Tukey [26] algorithms to compute the convolution of the samples. Consider the code in Figure 5.8 that computes the convolution of two 64K samples. Doing the computation in this straightforward manner would involve three transpose operators (steps 1d, 2d, and 4d). We can avoid the transposes by reversing the order of the FFTs in step 4. Instead of treating the samples as 16 columns of 4096 rows, we will use 4096 columns of 16 rows. Thus we can compute the inverse FFT with an alternate four step process: a) transpose the matrix from 4096 columns by 16 rows to 16 columns by 4096 rows; b) compute inverse FFTs of the rows [4096 16-point FFTs]; c) multiply by twiddle factors; d) compute the inverse FFTs of the columns [16 4096-point FFTs]. Using this approach for step 4 allows us to drop all the transpose operators, leading to the much more effi-

```
1) Compute the 64K point FFT of the X samples [16 cols, 4096 rows]
   a) compute the FFTs of the columns (16 4096-point FFTs)
   b) multiply by twiddle factors
   c) compute the FFTs of the rows (4096 16-point FFTs)
   d) transpose the matrix to 4096 columns by 16 rows

2) Compute the 64K point FFT of the Y samples [16 cols, 4096 rows]
   a) compute the FFTs of the columns [16 4096-point FFTs]
   b) multiply by twiddle factors
   c) compute the FFTs of the rows [4096 16-point FFTs]
   d) transpose the matrix to 4096 columns by 16 rows

3) Multiply the samples of X by the samples of Y

4) Compute the 64K point inverse FFT samples [16 cols, 4096 rows]
   a) compute the inverse FFTs of the columns [16 4096-point FFTs]
   b) multiply by twiddle factors
   c) compute the inverse FFTs of the rows [4096 16-point FFTs]
   d) transpose the matrix to 16 columns by 4096 rows
```

Figure 5.8. 64K-Point Convolution using the Cooley-Tukey 4-step algorithm

```
1) Compute the 64K point FFT of the X samples [16 cols, 4096 rows]
   a) compute the FFTs of the columns (16 4096-point FFTs)
   b) multiply by twiddle factors
   c) compute the FFTs of the rows (4096 16-point FFTs)
   ~~d) transpose the matrix to 4096 columns, 16 rows~~

2) Compute the 64K point FFT of the Y samples [16 cols, 4096 rows]
   a) compute the FFTs of the columns [16 4096-point FFTs]
   b) multiply by twiddle factors
   c) compute the FFTs of the rows [4096 16-point FFTs]
   ~~d) transpose the matrix to 4096 columns, 16 rows~~

3) Multiply the samples of X by the samples of Y

4) Compute the 64K point inverse FFT samples [4096 cols, 16 rows]
   ~~a) transpose the matrix from 4096 columns, 16 rows to 16 columns, 4096 rows~~
   b) compute the inverse FFTs of the rows [4096 16-point FFTs]
   c) multiply by twiddle factors
   d) compute the inverse FFTs of the columns [16 4096-point FFTs]
```

Figure 5.9. 64K-Point Convolution using the Cooley-Tukey 4-step algorithm, without transposes

cient algorithm in Figure 5.9. The key point about this algorithm is that during the entire computation, the samples are always organized as 16 columns by 4096 rows and stored in row-major order.

Coalesced Accesses to GPU Global Memory: the large FFTs (along the columns) will not fit in a single CUDA kernel (they require too much shared memory). So we process them using the technique described by David Bailey [4]. The large FFTs have 64^n samples and thus we handle them as a series of n radix-64 steps. Bailey's paper describes indexing

formulas originally due to Swartztrauber and Stockham. The indexing formulas allow a large FFT to be broken into steps with the transpose operators folded into the indexing. In the GPU implementation, we load 512 samples in each CUDA kernel block. The 512 samples are from 8 different consecutive columns (64 samples from each columns). Once the samples are loaded, the GPU computes 8 64-point FFTs, multiplies by twiddle factors and stores the results, often to different locations rows. Although it may seem somewhat counterintuitive, each block computes eight independent 64-point FFTs. This is done to ensure that accesses to GPU memory are always coalesced. Loads will always load 8 consecutive samples and stores will always store 8 consecutive samples. Also, note, because the load and store indexes can differ, the matrix must be double buffered, i.e., the data is loaded from buffer A and written to buffer B. Returning to our example of a 64K point FFT, Figures 5.10 and 5.11 show the implementation of the 4096-point column FFTs, using

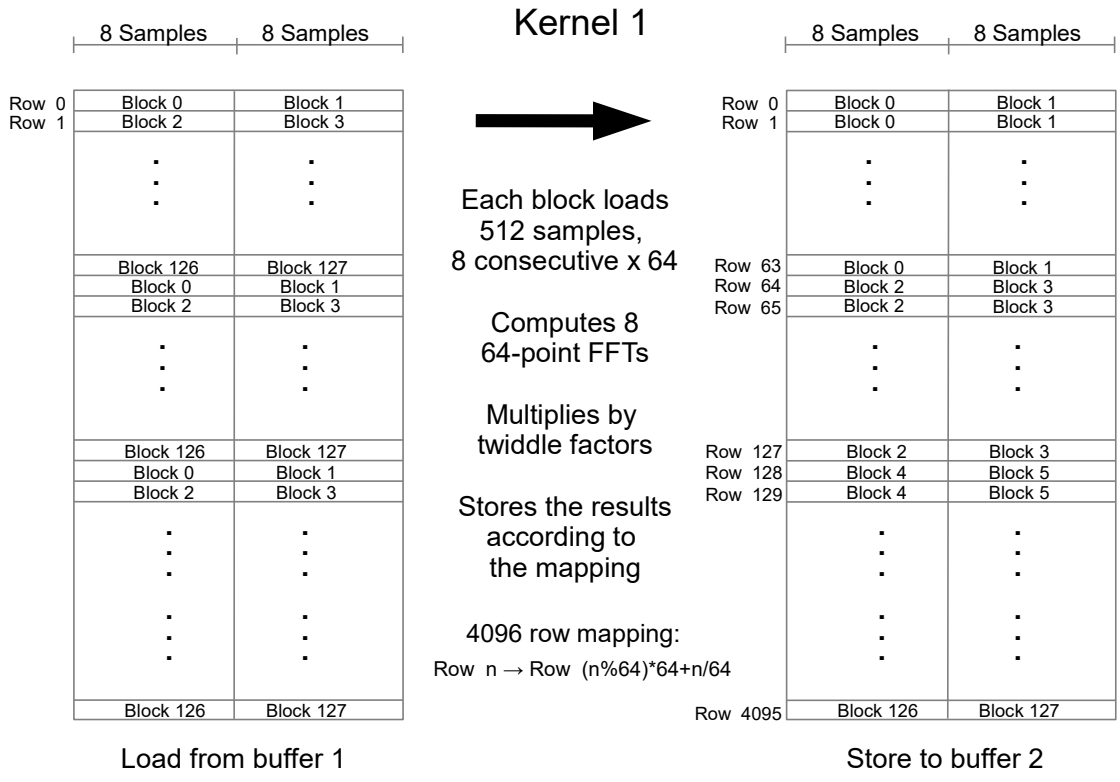

Figure 5.10. 64K-Point FFT example: Column FFTs - Step 1

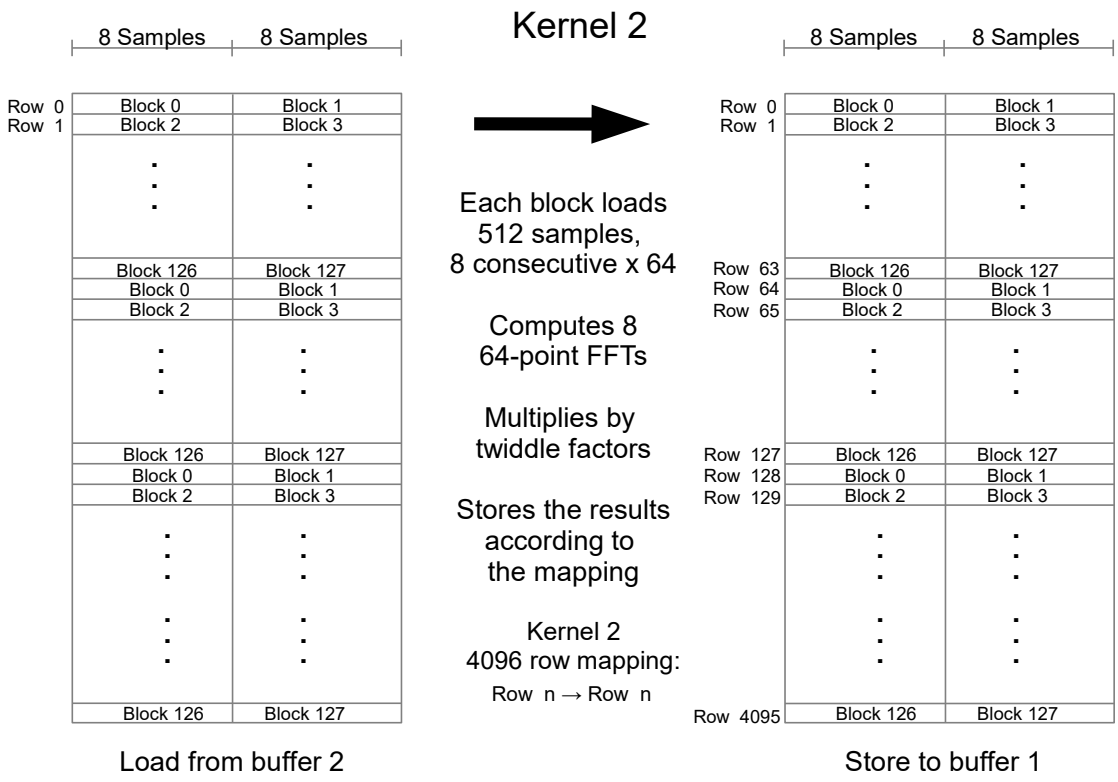

Figure 5.11. 64K-Point FFT example: Column FFTs - Step 2

variant 2 indexing described by Bailey. Since each block handles 512 samples (8 64-point FFTs), it requires 64K/512=128 blocks to run each radix-64 step. The two figures show the mapping of rows to blocks for a 64K-point convolution. More detail on these mapping functions can be found in Bailey's paper.

Once the column FFTs have been computed, the row FFTs need to be computed. Figure 5.12 shows the how the rows are mapped to blocks on the GPU. Since each CUDA block handles 512 samples, each block will handle multiple rows. For the 64K example, each block will run 32 16-point FFTs.

5.2.4 CUDA Implementation and Optimizations

General purpose GPU code must be well thought out and optimized in order to achieve anywhere near its maximum performance. If the code is poorly designed, it can run much

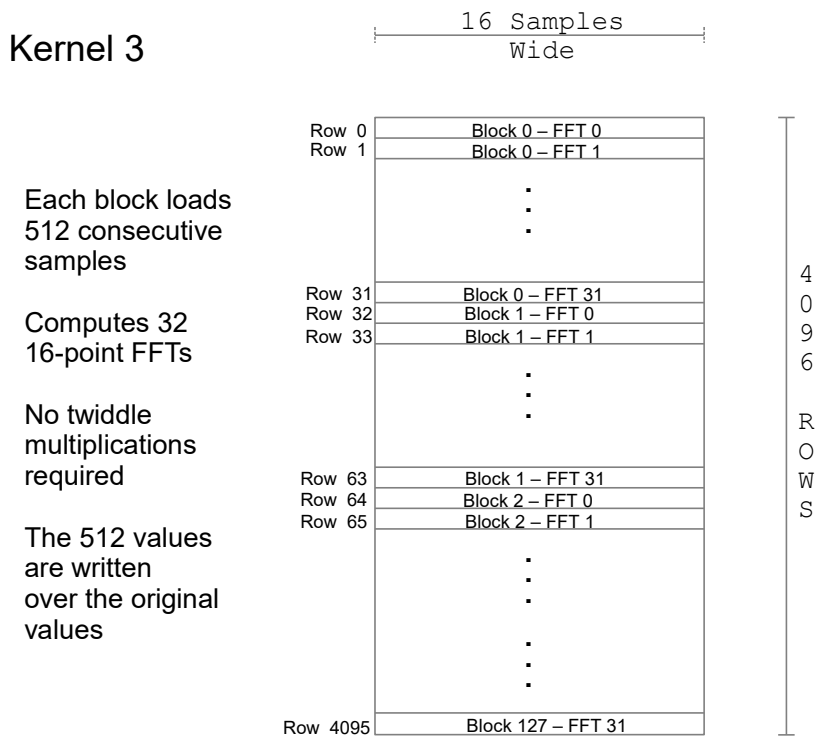

Figure 5.12. 64K-Point FFT example: Row FFTs

slower than on the CPU. With CPU code, the way a problem is partitioned will impact performance, but because the cache is so large in a modern system, partitioning is typically much less significant than in the design of GPU code. With the GPU the programmer must carefully think through each of the following constraints:

- Accesses to global memory should be coalesced
- Accesses to shared memory should minimize bank conflicts
- The problem must be partitioned into a large number of blocks
- Minimize the number of kernel launches, which are very expensive
- Multiple blocks (preferably 8 blocks) must concurrently fit on each streaming multiprocessor (SM), which implies a small number of registers and very small shared memory footprint

- Each kernel needs to have enough warps so that GPU code can make maximum use of the global memory bandwidth

In addition to the memory layout described in Section 5.2.3, our CUDA implementation was heavily influenced by Volkov and Kazian's [106] FFT work. We follow their design principals: (a) aggressive use of registers to store FFT samples so that small FFTs can be computed locally in each thread without the need for synchronization; (b) shared memory is then used to handle the inter-warp communication required for transpose operators. In our implementation, each block has 64 threads and each thread holds 8 samples in registers for a total of 512 samples. To perform eight 64-point FFTs:

1. Each thread runs a local 8-point FFT
2. Transpose the samples using shared memory to pass data between warps
3. Each thread then twiddles the 8 samples using shifting and fast reduction
4. Each thread then runs another local 8-point FFT

This process requires only a single *__syncthreads* call to compute eight 64-point FFTs.

With this basic design in mind, we looked at a number of specific optimizations to increase the overall performance:

Finite Fields: we explored a number of different primes as the basis for the field computations. We found $p = 2^{64} - 2^{32} + 1$ to have the fastest modulo operator.

Computation of Roots: the FFT routines require various powers of the primitive root of unity. The roots are computed at the start of each kernel and cached in shared memory. The usual algorithm to compute the power of the primitive root is exponentiation by squaring, as shown in Figure 5.13. Since the primitive root is fixed, we can improve the performance by unrolling the loop and precomputing the powers of two of the primitive root, see Figure 5.14. Next, since the ProductModP code requires about 25 instructions, it is faster still to process the bits two at a time and reduce the number of ProductModP calls. This version

```
uint64_t PowModP(uint32_t k) {
  uint64_t current=1, square=root;

  while(k>0) {
    if(k & 0x0001)
      current=ProductModP(current, square);
    sqr=ProductModP(square, square)
    k=k>>1;
  }
  return current;
}
```

Figure 5.13. Exponentiation by Squaring

```
uint64_t PowModPUnrolled(uint32_t k) {
  uint64_t current=1;

  if(k & 0x0001)
    current=root;
  if(k & 0x0002)
    current=ProductModP(current, root_2)
  if(k & 0x0004)
    current=ProductModP(current, root_4)
      . . .
  if(k & 0x8000)
    current=ProductModP(current, root_8000)
      . . .
  return current;
}
```

Figure 5.14. Unrolled Exponentiation by Squaring

proved to be the fastest and is shown in Figure 5.15. We tested processing the bits three at a time, but it was slower than the two bits at a time version.

Assembly Language Add, Subtract and Multiply: we have implemented all of the core math routines to do computation in the field $\mathbf{F}_p$ in PTX assembly language because they are significantly faster than the corresponding C versions. The performance of C versions is limited by the lack of direct access to the carry flag and MAD instructions.

Custom Transpose Operators: the implementation of FFT routines requires a variety of transpose operators for samples within a block. Each transpose operator uses shared memory to pass samples between different threads in the block. The basic structure of a transpose to write the samples to shared memory, synchronize the threads, then read the

```
uint64_t PowModPUnrolled2(uint32_t k) {
  uint64_t current, mult;

  current=1;
  if((k & 0x0003)==0x0001)
    current=root_1;
  if((k & 0x0003)==0x0002)
    current=root_2;
  if((k & 0x0003)==0x0003)
    current=root_3;
  mult=1;
  if((k & 0x000C)==0x0004)
    mult=root_4;
  if((k & 0x000C)==0x0008)
    mult=root_8;
  if((k & 0x000C)==0x000C)
    mult=root_C;
  current=ProductModP(current, mult)
     . . .
  mult=1;
  if((k & 0xC000)==0x4000)
   mult=root_4000;
  if((k & 0xC000)==0x8000)
   mult=root_8000;
  if((k & 0xC000)==0xC000)
   mult=root_C000;
  current=ProductModP(current, mult)
     . . .
  return current;
}
```

Figure 5.15. Unrolled Exponentiation, 2-bits at a Time

values back using a mapping. Consider the example of the transpose in the middle of a 64-point FFT:

```
__shared__ uint64_t buffer[512];
uint64_t            samples[8];

void blockTransposeSamples() {
  uint32_t low = threadIdx.x % 8, high = threadIdx.x / 8;

  #pragma unroll
  for(int index=0; index<8; index++)
    buffer[threadIdx.x + index*64] = samples[index];
  __syncthreads();
  #pragma unroll
  for(int index=0; index<8; index++)
    samples[index]=buffer[low + index*8 + high*64];
}
```

Figure 5.16. Transpose - 5 Shared Memory Cycles per warp per sample

In the first part of the transpose (writing to shared memory), there are no bank conflicts. In the second part (reading shared memory), there are 4-way bank conflicts. So this transpose takes 5 shared memory cycles per warp per sample. But the question is: can this transpose be done with fewer shared memory cycles? What is the optimum?

In our approach, we break the transpose into two mappings. We use the first mapping to write to shared memory and the second mapping to read from shared memory. Each mapping is a permutation of the 9 bits used to index shared memory. The mappings come in pairs, e.g., once you pick a write mapping, there is only one read mapping that will produce the desired transpose. Since there are 9 factorial permutations of the 9 index bits, there are exactly 362,880 permutation pairs. We developed a small utility that takes the desired transpose and searches through all the permutation pairs to find the one with the smallest

```
__shared__ uint64_t buffer[512];
uint64_t            samples[8];

void TransposeSamples() {
 uint32_t tid=threadIdx.x, to, from;

 to=(tid & 0x07) | ((tid & 0x38)<<1);

 // 2-way bank conflicts
 buffer[to+0x000]=samples[0];
 buffer[to+0x008]=samples[1];
 buffer[to+0x080]=samples[2];
 buffer[to+0x088]=samples[3];
 buffer[to+0x100]=samples[4];
 buffer[to+0x108]=samples[5];
 buffer[to+0x180]=samples[6];
 buffer[to+0x188]=samples[7];

 __syncthreads();

 from=(tid & 0x0F) | ((tid & 0x30)<<3);

 // 2-way bank conflicts
 samples[0]=transpose[from+0x000];
 samples[1]=transpose[from+0x010];
 samples[2]=transpose[from+0x020];
 samples[3]=transpose[from+0x030];
 samples[4]=transpose[from+0x040];
 samples[5]=transpose[from+0x050];
 samples[6]=transpose[from+0x060];
 samples[7]=transpose[from+0x070];
}
```

Figure 5.17. Optimum - 4 Shared Memory Cycles per warp per sample

number of memory cycles and with simple bit permutations. The resulting transpose code is shown in Figure 5.17. As can be seen from this example, the permutations are not obvious and would have been difficult to discover by hand. This is the optimum implementation where the read and write mappings are based on permutations of the address bits.

We encountered one serious difficulty during the development – each kernel must run a number of steps, for example our first FFT kernel does the following: computes the powers of the roots, loads the X data 24 bits per sample, computes the 64-point FFT on the X data, multiplies by the twiddle factors, stores the X data, then does the same steps for the Y data. We could split this into two kernels, one for the X data and one for the Y data, but performance would suffer because we would be incurring two expensive kernel launches and more significantly, the same roots would need to be computed twice.

Unfortunately, processing the X and Y samples in a single kernel causes other performance problems. The CUDA C compiler does a great job of generating fast assembly, unrolling loops, reusing subexpressions, etc. But it does not optimally schedule the register file. For example, in our code, we call a set of functions to process the X data, and then call exactly the same functions to process the Y data. In theory the same registers could be used to process both data sets. But the CUDA C compiler/assembler does not reuse the same registers. What we observe is that the number of real registers required to process both X and Y data in the same kernel increases over the number of registers required to just process the X data. This may be due to bugs in the compiler/assembler or it may be due to the compiler/assembler aggressively reusing subexpressions. Either way, the cost of using more registers far outweighs any benefits from the optimized code. If we specify a __launch_bounds__ directive, the compiler just spills some of those registers to local memory. The only way we were able to force the compiler/assembler to stay within the physical register file without any spilling to local memory was to write all of the core FFT routines in PTX assembly language.

5.2.5 Experimental Setup and Results

For our experiments, we compare an NVIDIA GeForce GTX 980 running at 1.22 GHz against its host machine, an Intel Core i5-7400 running at 3.0 GHz with 16 GB of memory running 64-bit Ubuntu Server version 16.04.1 LTS. On the GPU, we run our original large integer multiplier software (which had been tuned for the Fermi architecture) and compare the performance to running GMP on all of the cores of the host processor. The GPU software was compiled with CUDA 8.0 with version 375.26 of the NVIDIA driver. For the CPU test we use a small test program based on OpenMP and GMP 6.1.1 for the large integer multiplications.

Although the finite field we have chosen (with $p = 2^{64} - 2^{32} + 1$) supports a wide range of FFT sizes, each size requires a significant amount of customized assembly code. At this time we have implemented seven different FFT sizes: 32K, 64K, 128K, 256K, 512K, 1024K, and 2048K samples. These correspond to multiplications of 48K bits by 48K bits through 16384K bits by 16384K bits. For the 32K, 64K and 128K p oint FFTs we use three bytes per sample and for the other sizes, two bytes per sample.

Our test procedure is straightforward. For the GPU tests, we generate a pair of random numbers of the required size and copy them to the GPU global memory. We then start a timer on the GPU, run the sequence of kernels needed to multiply the numbers, then stop the timer. After each run we copy the results back to the CPU verify that the computation was correct using GMP. For each size we do five runs and report the average running time in milliseconds. The throughput is then computed as one over the average running time. On the CPU side, we generate 1000 random pairs of numbers of the specific size, and then use an OpenMP parallel for loop to process the array of multiplications. By default, OpenMP uses the environment variable OMP_NUM_THREADS to control the number of threads to run. The best number of threads turned out to be 4, which makes sense since the Core i5-7400 is a 4 core processor. To generate the time for a single multiply on the CPU, we simply test with a single thread, and divide the running time by 1000.

Size in Bits	Single Multiply (ms)		Throughput (Mults/Sec)		
	GTX 960	Core i5-7400	GTX 960	Core i5-7400	speed-up
384K	0.156	1.173	6410.3	2960.6	2.17
768K	0.300	2.569	3333.3	1458.4	2.28
1536K	0.450	5.654	2222.2	651.6	3.41
2048K	0.746	7.995	1340.5	450.6	2.97
4096K	1.785	18.102	560.2	190.9	2.93
8192K	3.545	48.441	282.1	69.7	4.05
16384K	5.691	103.338	175.7	30.3	5.79

Table 5.4. Large integer multiplication performance on the GTX 980 and a Core i5-7400

The results are presented in Table 5.4. The first column is the size in bits. The next two columns report the running time to multiply a pair of random numbers. The next two columns report throughput and the final column gives speedup (computed as the GPU throughput over the CPU throughput).

As we can see from Table 5.4 the single multiplication performance on the GPU ranges from roughly 7.5x faster at 384K bits to roughly 18x faster at 16384K bits. The throughput measurements compare socket to socket and we see that the GPU is between 2.2x at 384K bits to 5.8x faster at 16384K bits. We note that at 1536K the GPU is 3.4x faster than the CPU but at the next size up, 2048K bits, the speed-up drops to 2.97x. This is most likely because the GPU algorithm has switched from 3 byte samples to 2 byte samples so more computation is needed per bit of the result. The performance improvements are quite modest, but it's worth noting that the implementation was designed and tuned for Fermi and doesn't take advantage of many of the recent architectural improvements such as the funnel shifter, the warp shuffle, and the larger shared memories.

5.2.6 Conclusion and Future Work

The original implementation of the multiplier was in 2011. A lot has changed since then. The compiler's register management has improved significantly, so CUDA code with some inlines should be as fast as the monolithic PTX code implemented here, but it would be much easier to maintain. The GPU architecture has also evolved, the new shuffle in-

structions could be used to support 512-point FFTs within a warp (without the need for a transpose through shared memory) and a 4096-point FFT within a block. The small FFT routines could also be significantly improved by taking advantage of the byte permute instructions and hardware funnel shifter that was introduced in compute capability 3.5. Finally, we note there are better approaches to the shared memory transposes. With a small increase, of about 10%, in the size of the shared memory transpose buffer, it's possible to implement mappings with no bank conflicts. These changes would considerably improve the performance of the large multiplier on recent GPU architectures.

CHAPTER 6

MODULAR EXPONENTIATION ACROSS MULTIPLE GENERATIONS OF GPU

Multiple precision (MP) modular exponentiation, in which a value M is raised to a power K and reduced modulo P, is a central component of many cryptographic operations such as key exchange (RSA, DH, and ECDH) digital signatures (DSS and ECDSA) and prime testing algorithms such as Miller-Rabin. Given that the values being computed are hundreds to thousands of bits in length, the computational cost of modular exponentiation is high. When servers are handling thousands of sessions at once, the cost can be overwhelming.

While this effort can be offloaded to custom hardware, recent research has shown that graphics processors (GPUs) offer an effective low-cost alternative. Neves and Araujo[86] gives an excellent overview of cost-performance and architectural issues related to using GPUs for modular exponentiation.

In this chapter we go beyond prior work to obtain throughput over batches of modular exponentiations that is very close to the theoretical peak capability of NVIDIA GPU architectures ranging from early Compute Capability 1.1 designs to the most recent. Our efforts improve on prior results by factors ranging from 2.6 to 24 times and leave little room for further gains.

A key insight we offer is that the commonly used approaches proposed by Koç et al. [74] in the uniprocessor context are not appropriate for GPUs. It is necessary to completely rethink the layout of the computation. Koç was targeting a small register file with a fast cache, and showed several methods to implement the Montgomery multiplication at the

core of modular exponentation that trade off cache reads and writes against computation and temporary storage. Since the computation fits entirely in cache, the access pattern of the MP values has little impact on the overall performance.

On the GPU, however, best throughput is obtained via maximum parallelism, with one instance computed per thread. When an instance doesn't fit the resources available to a thread, then the goal is to at least maximize the instances per warp. On the GPU, there is a large register file and only a small amount of cache (on a per thread basis) available. Register to register computation is fast and inexpensive, but due to the high latency of memory access, another goal is to minimize the shuffling of data through main memory. Thus, the most effective approach on the GPU is to entirely avoid loads and stores within the inner loops of the square, multiply, and reduce operations. It is essential to keep entire MP values in registers and carefully manage the access patterns.

Attaining these goals is directly dependent on the particular configuration of resources within a given GPU architecture. Different generations shift the balance of available threads, registers, and shared memory. A secondary effect is that each generation changes the available operations and the rates at which they execute. For example, in some GPUs, 24 bit arithmetic can be substantially faster than 32 bit arithmetic because there are more floating point units than integer units. Thus, more 24 bit integer operations can be passed through the mantissa ALUs in parallel than through the dedicated integer ALUs.

Obtaining high throughput therefore depends primarily on mapping the MP operands to the available registers so as to maximize instances per warp. Secondarily, optimization must take maximal advantage of the given computational capabilities. We have identified four distinct parameterized mapping models that are needed to address the range of GPU generations and problem sizes.

Our experiments have found that the optimal mapping is not at all obvious on the basis of an architectural specification. The many potential combinations of problem size, mapping model and parameters, launch geometry, exponentiation window size, and GPU

architecture form a multidimensional design space that must be searched via generating and running actual code. We have developed tools that facilitate the search, which we employ to obtain our results.

The remainder of the chapter is organized as follows. In Section 6.1, we present necessary background material. Section 6.2 reviews prior work. Sections 6.3 through 6.6 explain the different mapping models. In section 6.7 we describe the setup for our experiments and propose a utilization metric to analyze the efficiency of the software. In section 6.7.2 we give the results and related discussion and we conclude with Section 6.9.

6.1 Background

GPU architectures are now commonly accepted for use in parallel computing, and their architectures and programming models are well known. We refer the reader to [28] for details and in the experimental setup section we summarize the capabilities of the specific architectures used in our experiments.

Throughout this chapter, we use uppercase to represent MP values and lowercase for single precision scalar values. We follow the typical RSA exponentiation naming convention where $M^K \bmod P$ refers to message (M), key (K), and prime modulus (P). M, K, and P have the same length, denoted b in bits and n when measured in words. For windowed exponentiation algorithms, we use w as the bit size of the window. For RSA decryption, we assume the use of Quisquater's "CRT trick" [90], where a message of length $2b$ is handled using the Chinese Remainder Theorem (CRT) and two exponentiations of length b.

When implementing MP arithmetic, and modular exponentiation in particular, there are many choices for representation and algorithms. For example, numbers can be represented with a fixed radix or in a residue number system. Since Harrison and Waldron [55] have shown that the former significantly outperform the latter, we explore only algorithms that use a fixed radix number system, where β, the base, is a power of two.

There are three common modular exponentiation algorithms, discussed in Section 2.2.9. Each of these rely on two sub-operators to square and multiply the MP numbers. All three require roughly the same number of squaring steps, but the sliding window algorithm uses fewer multiplications and is somewhat faster. However, only the fixed window algorithm is immune to timing side channel attacks, since the execution time of the other two depends on the bit pattern in the key. Thus, we use only the fixed window algorithm. To accelerate the computations on the GPU we use a standard Montgomery reduction, except we allow the values in the Montgomery domain, $\overline{X}$, to range from 0 to $2^b - 1$, which simplifies the correction step. Throughout the chapter, we refer to this as Almost Montgomery representation. For further information on Montgomery reductions, see Section 2.2.6.

As previously noted, we have identified four mapping models for placing MP values in registers. We refer to these as Three-N, Two-N plus Local, Sampled, and Four-N Distributed. These are each described in a later section. Here we briefly summarize them.

Three-N keeps 3 n-word values in registers, and can perform all the key operations in registers. With K and P stored in shared memory.

Two-N plus Local places 2 n-word values in registers, and enables separate computation of low and high order portions of the MP operations. It requires four times as many memory accesses, but conserving registers can enable greater parallelism on some architectures.

Sampled uses fewer bits per register (between 20 and 22) to take advantage of architectures where 24-bit multiplication is faster than 32-bit multiplication. It trades faster computations for the cost of using more registers.

Four-N Distributed uses a group of 2, 4, 8, or 16 threads for a single exponentiation. It keeps four MP values in registers but distributed across threads instead of being local to a single thread. For t threads, it uses $4n/t + 1$ registers. Although it uses registers efficiently, it suffers inter-thread communication overhead, and cannot take advantage of fast squaring or multiplication.

Much prior work has been done using the NVIDIA CUDA tools, but our experience is that they are inadequate for developing highly tuned libraries. Although their optimizations are improving, their register allocation scheme often wastes registers. For our work, careful management of the register usage is critical to obtaining maximum performance. In addition, MP arithmetic often involves long chains of add with carry and multiply-add with carry instructions which are cumbersome to implement with CUDA inlines. Thus we have developed custom tools to generate large blocks of PTX assembly language giving us much better control over the register allocation. Our tools also target a simulator we use for debugging and performance estimation.

Based on the preceding factors, our design space has the following dimensions that we search for the optima:

1. GPU card (compute capability, number of SMs, etc.)
2. Size of the exponentiation (b)
3. Window size for the exponentiation algorithm (w)
4. Model and parameters (e.g., sample size, # of threads, etc.)
5. Launch geometry (i.e., the number of threads per block)
6. Max reg count (limit assembler's use of extra registers).

6.1.1 Code Generator

The main tool in our environment is the code generator. In a traditional compiler, source code is translated into assembly language. Instead, our code generator takes an XML specification of parameters for the desired algorithm (target card, size of exponentiation, window size, model/parameters, launch geometry) and generates PTX assembly. The code generator is somewhat analogous to a template system, but instead of processing template files, we invoke Java objects that are responsible for generating the code for a particular algorithm. We call these Java objects *coders*. See Figure 6.1.

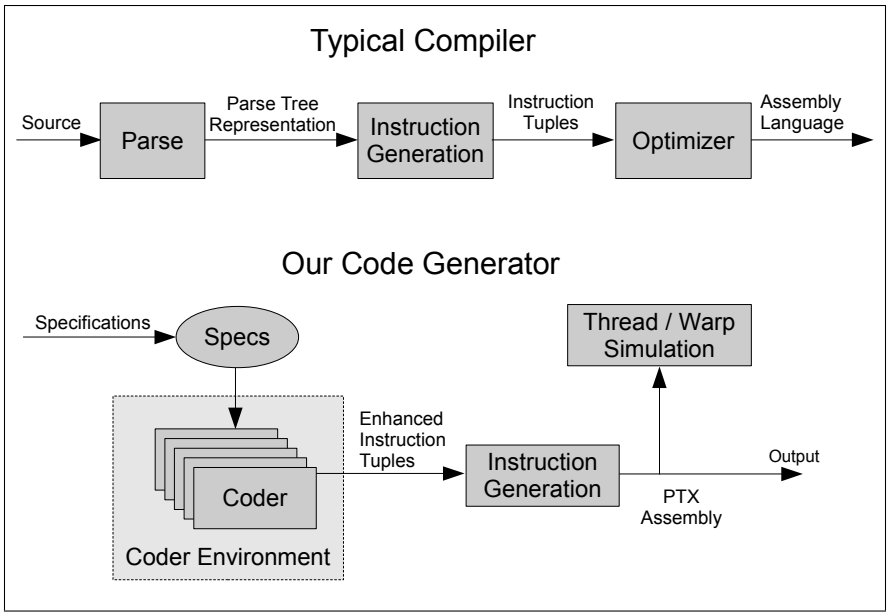

Figure 6.1. Typical Compiler vs. Our Code Generator.

Throughout the remainder of this chapter, we present pseudocode for a variety of algorithms and routines. However, it is important to note that the pseudocode presented is actually the output of the corresponding *coders*, which is quite distinct from how the *coders* are implemented. For a full explanation of the code generator and its features, see [41].

Our *coders* are quite general and provide a sophisticated environment for generating PTX code. An important feature is that *coders* can dynamically invoke each other, resulting in an assembly language equivalent to inlined function calls. Further, since *coders* are full fledged Java objects, they benefit from many of Java's features including *subclassing* and *interfaces*. The latter is especially useful because it enables polymorphism, where multiple *coders* all supply the same set of APIs. For example, consider the fixed window exponentiation (FWE) coder in Figure 6.2. The FWE coder takes a model coder as an argument. The idea here is that the FWE coder is agnostic about the register layout and representation of the MP values. Whenever it needs to perform an operation, it invokes the model coder to generate the actual code for the operation. This approach cleanly separates the code generation for the exponentiation algorithm from the code generation specific

```
procedure exponentiate(ExponentiationModel modelCoder)
  // the values K, P, and R2 are in shared memory
  // where R2=2^(2b) mod P

  modelCoder.loadArgument()  // loads M
  modelCoder.multiplyByConstant(R2)

  .. CONSTRUCT THE WINDOW TABLE ..

  i = b - ((b % w==0) ? w : b % w)
  j = extract bits i to b-1 from K
  modelCoder.loadCurrentFromWindow(j)
  while i>0 do
    for j=0 to w-1 do
      modelCoder.square()
    end
    j = extract bits i to i+w-1 from K
    modelCoder.multiplyByWindow(j)
    i = i-w
  end
  modelCoder.multiplyByConstant(1)
  modelCoder.storeResult()
end
```

Figure 6.2. Fixed Window Exponentiation

to the register layout and MP value representation, which are managed by the model. A further advantage of this approach is that the FWE coder is generic and it works with any model coder that implements the *ExponentiationModel* interface.

Fixed window exponentiation starts by loading M and converts it to Montgomery form. Next it constructs the window table of values, table$[k] = M^k \bmod P$, for $k \in 0, 1, ..., 2^w - 1$. Then the main loop processes K from the most significant bit to the least significant bit, breaking it into chunks of w bits. The algorithm keeps a "current" value, and at each step it either squares the current value or multiplies it by a value from the window table. The FWE routine does not manipulate the MP values directly. Instead it delegates the work to the *model* according to the *ExponentiationModel* interface. The function of each of the APIs is described in Figure 6.3. In subsequent sections, we describe each model in terms of how they implement the *ExponentiationModel* APIs.

6.2 Related Work

Several papers have looked at modular multiplication and modular exponentiation on the GPU, see for example: [84, 47, 100, 55, 10, 86, 64, 52, 22]. Jang et al [64] give

METHOD	FUNCTION
loadArgument()	loads M from global memory into the "current" value
storeResult()	store the "current" value into global memory as the final result
loadCurrentFromWindow(i)	loads a value from the window table into the "current" value
storeCurrentToWindow(i)	stores the "current" value into the window table
multiplyByConstant(i)	multiply the "current" value by a constant value in shared memory and reduce modulo P
multiplyByWindow(i)	multiply the "current" value by a value from the window table and reduce modulo P
square()	square the "current" value and reduce modulo P

Figure 6.3. The API methods of the ExponentiationModel interface

throughput rates for a GTX 580 running RSA decryption of various key sizes, which involves two half-size modular exponentiation operations and an application of the CRT. Neves and Araujo [86] present results for a GTX 260 running 512-bit modular exponentiation. These last two papers report the best performance we are aware of. Because we are able to generate code for the architectures they use, we can make a direct comparison in our results section, as well as showing how performance improves with later architectures. For the same architectures, we improve upon their results by factors of from 2.6 to 8.8. On a more recent architecture, we show results ranging from 3.1 to 24 times faster.

6.3 Three N Model

There are two standard approaches to implementing O(n^2) multiplication, $X \cdot Y$, where X and Y are each n words in length. The first is column oriented: for $k = 0$ to $n - 1$, compute the column sum $\sum_{i,j} X[i] \cdot Y[j]$ where $i + j = k$. While straightforward to code, handling carries during the sum can be slow. The second approach is row oriented: the product is the sum of the rows $X[i] \cdot Y \cdot \beta^i$ where β is 2^{32}. If the architecture has instructions that directly handle carry propagation, row oriented is usually preferred.

The NVIDIA 2.x and 3.x architectures provide 32-bit by 32-bit integer multiply and accumulate with carry instructions, which are ideal for the row oriented approach, because

they propagate carry with no extra instructions. Thus each row requires $2n$ *madc* instructions and one *addc* instruction.

The Three N model takes advantage of these instructions. It reserves space to hold 3 n-word values in registers, which we call A, B and C. A stores the "current" value while B and C are used during the multiplication, squaring and reduction steps. We also define R as a $2n$ word long alias for the concatenation of B and C, i.e., $R[i]$ refers to $B[i]$ when $i < n$ and to $C[i - n]$ when $i \geq n$.

The implementation of *loadArgument* reads the M value from global memory into A and *storeResult* writes the result from A to global memory. Likewise, *loadCurrentFromWindow* and *storeCurrentToWindow* read and write A to the given index in the window table for the current thread.

The multiplies, *multiplyByWindow* and *multiplyByConstant*, read or copy the value into C, then compute A times C, storing the result in R. They use a row oriented approach as follows.

```
temp = C[0]
R[0 .. n] = temp * A
for i=1 to n-1 do
  temp = C[i]
  R[i .. i+n] = temp * A + R[i .. i+n-1]
end
```

Figure 6.4. Three N Multiplication

Note that, because C and R overlap, care must be taken to ensure that updates to R don't overwrite values of C that are still needed (hence the temp variable).

The squaring operation *square*, computes A^2 and stores the result in R. It runs in about half the time of the multiply operation. The algorithm works by exploiting the symmetry in each column of the product terms. The algorithm computes the sum of the terms above the main diagonal, then doubles the sum (because each term appears twice in each column), then adds in the products on the diagonal. We frequently refer to this as fast squaring. For the Three N model, this algorithm can be implemented with a row oriented approach.

Once the result of the multiplication or squaring is in R, we reduce the value modulo P. P is loaded from shared memory into A followed by an Almost Montgomery row oriented reduction. The result is stored in A, as the "current" exponentiation value.

The Three N model also implements Karatsuba squaring and limited Karatsuba multiplication as options. The Karatsuba [69] algorithm uses a divide and conquer approach that is asymptotically faster than the $O(n^2)$ multiplication algorithm. However, for small sizes (fewer than about 16 words), there is at best a modest improvement in performance. Typically Karatsuba is implemented with a threshold parameter – below a certain size, it switches to the $O(n^2)$ algorithm. However, in our environment it is more appropriate to specify the number of levels of recursion before switching algorithms. Thus, the Three N model has two model parameters, KS specifies the levels of recursion to use for squaring and KM is the number of levels for multiplication. Specifying KS=0 and KM=0 disables Karatsuba and uses only the regular multiplication and squaring algorithms.

In the interest of brevity, we omit the psuedocode for our Karatsuba square and multiply routines, but they are quite similar to those presented by Brent & Zimmermann [16]. We use their *additive* method for multiplication and *subtractive* method for squaring.

One significant drawback of Karatsuba is that it tends to require many temporary registers. With careful reuse, however, an arbitrary number of levels of Karatsuba squaring can be done with $3n$ registers plus one additional temporary per level. However, Karatsuba multiplication seems to require at least $4n$ registers plus one temporary per level. Our code generator supports a single level of Karatsuba multiplication, which can be done with $3.5n$ registers and one temporary.

6.4 Two N Plus Local Model

The Two N plus Local model (or just Local model) places two n-word values in registers and one in GPU local memory. The registers are A and B. Like the other models, Local

keeps the "current" exponentiation value in the A registers and uses the B registers for multiplication, squaring and reduction operations.

The implementations of *loadArgument*, *storeResult*, *loadCurrentFromWindow* and *storeCurrentToWindow* all read/write their respective values to/from the A registers. The *multiplyByWindow* and *multiplyByConstant* routines operate as follows: load the window or constant value into the B registers. Compute the high order n words of the product of $A \cdot B$ using a row oriented approach that overwrites the words of B as the product is computed. When complete, write the high order words to local memory. Next, reload the window or constant value into B and compute the low order n words of $A \cdot B$. One extra register holds all of the carries out of the low order n words of the product.

The *square* routine is analogous. Compute the high portion of A^2 in B, write the result to local memory. Then compute the low order words and an extra word to hold the carries. This algorithm uses optimized squaring for both the low and high halves, which saves a significant amount of computation.

Once the product is computed, we run Almost Montgomery, which assumes the lower order n words of the product are in B, there is a register holding carries (which has to be added to the high order words), and the high order words of the product are in local memory.

The Montgomery reduction loop depends only on the low order n words of the product. Because we computed the low order words last, we can immediately start the reduction. We load P into the A registers and run the following:

```
// Assume the A registers contain P and the B register
// the n low order words of the product
for i=0 to n-1 do
  temp = B[0] * np0
  B = (temp * A + B)>>32    // no carry out
end

.. add the carry outs word to B
.. read the high order words from local memory
..    and add the value to B
.. if there was a carry out, then subtract P
..    from B
```

Figure 6.5. Local Model Almost Montgomery Reduction

At this point, the result has been reduced to an Almost Montgomery value, and we copy it back to the "current" exponentiation value (the A registers).

6.5 Sampled Model

On the NVIDIA 1.x architectures, the GPU lacks a native 32-bit by 32-bit integer multiply. Instead 32-bit multiplies are implemented with a sequence of 4 instructions based on 16-bit multiplies. However, the hardware does support single cycle 24-bit integer multiply and accumulate instructions. There are two instructions *mad24.lo* and *mad24.hi*. Both are four argument opcodes of the form $d \leftarrow \text{lo}(a \cdot b) + c$ and $d \leftarrow \text{hi}(a \cdot b) + c$ where each of a, b, c, and d are 32-bit registers. They function as follows. The .lo instruction computes the 48-bit product of a times b then extracts the least significant 32-bits of the product and adds c. The .hi instruction computes the 48-bit product, then extracts the most significant 32-bits of the product and adds c. These are rather unusual instructions in that the high order 16 bits of the .lo variant overlap the low order 16 bits of the .hi variant. But we use them to construct a fast 48-bit accumulator as follows. Given a set of A samples $(a_0, a_1, \ldots a_{n-1})$ and B samples $(b_0, b_1, \ldots b_{n-1})$ where the samples are each less than 24-bits, the following code will compute the 48-bit dot product, $\sum_{i=0}^{n-1} a_i \cdot b_i$, provided the sum does not exceed 2^{48} and $n < 2^{14}$. We discuss the second condition below.

```
// computes the 48-bit dot product of A and B,
// low 32 bits will be in acclo, high 16 bits in acchi
acclo = 0
acchi = 0
for i=0 to n-1 do
  mad24.lo.u32   acclo, A[i], B[i], acclo
  mad24.hi.u32   acchi, A[i], B[i], acchi
end
// post processing step
if acclo<0x80000000 then
  acchi = acchi + n
end
acchi = acchi>>16
```

Figure 6.6. 48-bit Dot Product

The *if* statement in the post processing resolves a corner case. Consider A =(0x0FFFF, 0x00001) and B =(0x10001, 0x00001). Without the *if*, the code would return zero for the

sum, which is clearly incorrect. The reason is that a carry is generated out of the low order 16 bits of acclo, but an equivalent carry is not injected into acchi. This condition can happen at most $n - 1$ times, so adding n to acchi resolves the problem, provided $n < 2^{14}$. Since we have at most a few hundred registers to work with, this is not a significant limitation.

The Sampled model takes advantage of this fast dot product. The model has a parameter, s, to specify the number of bits to use in each sample, which can range from 20-22. It represents a b-bit value as a sequence of $n = \lceil b/s \rceil$ samples and the model uses $3n$ registers to hold three MP values, A, B, and C. Since $1.5s \approx 32$, the Sampled model requires roughly 50% more registers than the Three N model for the equivalent size. As before, R is the concatenation of B and C, A holds the "current" value, and B and C are used for squaring, multiplication and reduction. The implementation of *loadArgument* reads the MP value M from global memory, which has 32-bits of data per word and samples it into A with s bits per sample. *storeResult* is similar, except it compresses the samples of A back into 32-bit words and writes the result to global memory. The window APIs (*loadCurrentFromWindow, storeCurrentToWindow, multiplyByWindow*) all work on sampled MP values.

To implement *multiplyByWindow* and *multiplyByConstant* we first load the sampled window value or constant value into C. Then we use column oriented multiplication with the fast 48 bit dot product, as shown in Figure 6.7. In this algorithm we refer to the post processing after the sum as *resolving the high word*. When implemented carefully, it takes

```
int acclo=0, acchi

for k=0 to 2n-2 do
  acchi = 0
  .. use fast dot product to sum
  ..    A[i]*C[j] where i+j=k
  R[k] = acclo & ((1<<s)-1)

  // resolve the high word
  if acclo<0x80000000 then
    acchi = acchi + n
  end
  acclo = ((acchi>>16)<<32-s) | (acclo>>s)
end
R[2n-1] = acclo
```

Figure 6.7. Sampled Multiplication

as few as 5 single cycle instructions. As in the Three N model, we implement multiply so the R values only overwrite C values that are no longer required.

Sampled squaring is similar to multiply in that it works on a sampled MP value in A and stores the result into B and C, using a column oriented approach. It works on the same principal as the standard fast squaring algorithm (doubling the off diagonal terms). Because samples are at most 22 bits, we can double each sample and it is still shorter than 24 bits. So we first double each element of A and store it in C. Then whenever we need to compute $2 \cdot A[i] \cdot A[j]$, we instead compute $A[i] \cdot C[j]$. Again, care must be taken to ensure the values of R don't overwrite values of C while they are still needed.

For the Almost Montgomery reduction, we use a column oriented approach similar to Koç et al's FIPS method, see [74]. First, we load the modulus P (kept in sampled form) from shared memory into A. Then for each column, $k = 0$ to $n - 1$ we use the fast dot product to compute:

$$B[k] = \left(R[k] + \text{acchi}_{k-1} + \sum_{i=0}^{k-1} B[i] \cdot A[k-i]\right) \cdot \text{np0}$$

Where acchi_{k-1} is the carry from the prior column. For columns $k = n$ to $2n - 1$, we compute:

$$R[k] = R[k] + \text{acchi}_{k-1} + \sum_{i=0}^{2n-1-k} B[i] \cdot A[k-i-n]$$

The reduced result is in C. Finally, following the Almost Montgomery rule, if the value in C is greater than 2^{sn} we subtract the modulus P, and store the result back in A.

6.6 Distributed Model

The distributed model is different from the others in that it uses groups of threads to solve an instance instead of an instance per thread. The model parameter t configures the number of threads per group. t must be 2, 4, 8 or 16. The distributed model breaks each MP value into t chunks and assigns consecutive chunks to consecutive threads in the group. This approach significantly reduces the number of registers per thread, which allows our

tools to code much larger sizes. The main disadvantage of the approach is the lack of a fast squaring algorithm. Thus, squaring takes almost twice as much computation as the other models. The second disadvantage is there is a small overhead for communication between threads.

The distributed model reserves enough registers to store 3 n-word values and an $n + 1$ word value, using roughly $4n/t + 1$ registers. Beside A, B, C, it defines T (which is $n + 1$ words in length). The model uses A for the "current" exponentiation value. The others are used during multiplication/reduction. The model is based on Koç et al's CIOS method [74], but with distributed values and is similar to Jang et. al's approach [64].

The implementations of *loadArgument*, *storeResult*, *loadCurrentFromWindow* and *storeCurrentToWindow* all read/write their respective values to/from the A registers.

The implementations of *multiplyByWindow*, *multiplyByConstant* and *square* all work the same way. They leave the "current" value in A, load the multiplicand into B, and load

```
.. t = the number of threads in each group ..
.. g = my thread index within the group (0 .. t-1) ..

T = 0
for i=0 to n-1 do
  x = BROADCAST(g==i/t, A[i % t])
  T = T + x * B

  if g!=0 then .. transmit T[0] to g-1 ..
  x = 0
  if g!=t-1 then .. receive x from g+1 ..
  T = T + x<<(n/t-1)*32

  x = BROADCAST(g==0, T[0])
  T = (T + LO(x * np0) * C)>>32
end

// resolve any carries
do
  x = BROADCAST(g==t-1, T[n/t])
  if x!=0 then T = T - C
  if g!=t-1 then
    .. transmit T[n/t] to g+1 ..
    T[n/t] = 0
  end
  if g!=0 then .. receive x from g-1 ..
  T = T + x  // note: x can be negative
until ALL(T[n/t]==0)
```

Figure 6.8. Distributed Multiply and Reduce

P into C, then multiply A times B and reduce by C, placing the running result in T. The psuedocode is presented in Figure 6.8

The pseudocode is written with message passing to make the algorithm more understandable. The actual code uses shared memory, with a word per thread. The BROADCAST routine takes a boolean and a value. If the boolean evaluates to true, then this thread is the broadcaster, otherwise it's a recipient. All broadcasts are written to/read from group thread zero's shared memory.

6.7 Experimental Setup and Results

To evaluate our algorithms, we used six different NVIDIA GeForce GPU cards: 9800 GT, GTX 260, GTX 580, GTX 680, GTX 780Ti, and GTX 750Ti. These cover a wide range of NVIDIA micro-architectures. The host is an ASRock X79 Extreme11, configured with a quad core Intel Core i7-4820K, clocked at 3.7 GHz, with 16GB of memory, running 64-bit Ubuntu Server (version 14.04.1), using CUDA version 6.5 and version 340.29 of the NVIDIA driver. We use GMP version 6.0.0a with GCC version 4.8.2 to verify the exponentiation results from the GPUs.

A main application of modular exponentiation is RSA public key/private key encryption. An RSA key of length l is based on the product of two primes, P and Q, each of length $l/2$. To decrypt an RSA message it is necessary to compute M^K mod PQ where M is the message and K is a constant based on P and Q. Since P and Q are prime, it is faster to do two smaller modular exponentiations (mod P and mod Q) and combine their results using the CRT. For useful key sizes, CRT reconstruction time is less than 1% of exponentiation time. Thus, the exponentiation time is the metric of interest and we can ignore the CRT time. For RSA, the most common key lengths are 1024 and 2048 bits, so performance of the 512 bit and 1024 bit modular exponentiations is especially important.

Our test procedure is straightforward. We use the code generator to build a CUDA kernel from a set of parameters (target compute capability, launch geometry, exponentia-

tion window size, model, and parameters specific to the model such as bits per sample and number of levels of Karatsuba, etc). The CUDA kernel is compiled with standard ptxas options (-v and -abi=no) and linked with the tester. It always has the same calling interface regardless of the model or parameters so we can use a generic testing tool across all implementations. It essentially serves as a wrapper for the inline PTX.

The tester first generates a large number of messages, either 128000 for 256 and 512 bit exponentiations or 32000 for 1024 and 2048 bit exponentiations. Each message consists of a random number between 1 and $2^b - 1$ where b is the size of the exponentiation. Then, for every 32 message (one warp) the tester generates a random modulus P and random exponent K also between 1 and $2^b - 1$. We ensure the modulus is odd, but not necessarily prime. From the modulus, we compute the Montgomery constant $R^2 \bmod P$ and then copy the data to the GPU.

We run the GPU kernel 12 times on the same data: 2 for warm-up and 10 for timing. On every run the GPU computes $M^K \bmod P$ for each message. The times are combined with an arithmetic mean to give the average throughput (exponentiations per second). The time we report is for the GPU to load the data (from GPU global memory), perform the exponentiation, and store the result. It does not include the time to generate the data on the CPU, to copy the data to and from the GPU, or to verify the results on the CPU.

6.7.1 Utilization

One of the key questions we wish to answer with our results is: what is the efficiency of the software? With floating point programs, this is easy to assess: it's the delivered floating point operations per second divided by the theoretical maximum sustainable by the architecture. Of course, this metric isn't a complete picture of performance, but it is still useful. If utilization is high, it means the code is well tuned for the hardware. If the utilization is low, there might be opportunities to further tune the software. Although a

Model	256-bit (w=4)			512-bit (w=5)			1024-bit (w=6)			2048-bit (w=6)		
	madc.lo	madc.hi	add/sub	madc.lo	madc.hi	add/sub	madc.lo	madc.hi	add/sub	madc.lo	madc.hi	add/sub
Three N, KS=2, KM=1	31.59K	31.59K	51.78K	234.77K	234.77K	200.06K	1.780M	1.780M	778.75K	13.73M	13.73M	3.040M
Three N, KS=1, KM=1	32.36K	32.36K	33.13K	244.22K	244.22K	127.61K	1.868M	1.868M	494.98K	14.48M	14.48M	1.931M
Three N, KS=2, KM=0	32.72K	32.72K	48.52K	242.32K	242.32K	189.91K	1.831M	1.831M	745.22K	14.11M	14.11M	2.918M
Three N, KS=1, KM=0	33.50K	33.50K	29.87K	251.77K	251.77K	117.46K	1.920M	1.920M	461.45K	14.86M	14.86M	1.809M
Three N, KS=0, KM=1	33.92K	33.92K	21.22K	258.92K	258.92K	80.36K	1.994M	1.994M	307.90K	15.51M	15.51M	1.196M
Three N no karatsuba	35.05K	35.05K	17.95K	266.47K	266.47K	70.21K	2.046M	2.046M	274.39K	15.89M	15.89M	1.074M
Local	35.05K	34.79K	18.96K	266.47K	265.95K	73.14K	2.046M	2.045M	282.70K	15.89M	15.89M	1.102M
Distributed, T=1	45.02K	42.37K	21.85K	340.03K	329.73K	83.72K	2.601M	2.568M	323.53K	20.24M	20.08M	1.260M
Distributed, T=2	47.66K	42.37K	38.40K	350.34K	329.73K	146.83K	2.648M	2.568M	566.81K	20.39M	20.08M	2.206M
Distributed, T=4	52.96K	42.37K	71.50K	370.94K	329.73K	273.06K	2.729M	2.568M	1.053M	20.70M	20.08M	4.098M
Distributed, T=8	63.55K	42.37K	137.70K	412.16K	329.73K	525.50K	2.889M	2.568M	2.026M	21.33M	20.08M	7.882M
Sampled, S=20	95.81K	91.51K	3.37K	716.13K	699.39K	13.65K	5.451M	5.385M	54.65K	41.34M	41.08M	213.93K
high word resolves		*16.27K*			*65.11K*			*257.17K*			*1.003M*	
Sampled, S=21	95.81K	91.51K	3.37K	662.98K	646.88K	13.13K	4.844M	4.783M	51.50K	overflows 2^{48}		
high word resolves		*16.27K*			*62.54K*			*242.13K*				
Sampled, S=22	82.06K	78.09K	3.11K	overflows 2^{48}			overflows 2^{48}			overflows 2^{48}		
high word resolves		*14.94K*										

Table 6.1. Instruction counts to perform modular exponentiation

low utilization can also indicate that the problem has low computational intensity or that it intrinsically maps poorly to the hardware.

For our analysis, we develop a similar metric. For each of our models we track the exact number of *madc.lo* operations, *madc.hi* operations (in most cases, 32-bit by 32-bit multiplies with carry in/out) and *add/sub* operations that are required to compute X^2 mod P and $X \cdot Y$ mod P. For the Sampled model, targeting 24-bit arithmetic units, the counts are for *mad24.lo* and *mad24.hi* instructions and because carries are not done in hardware, we also track the number of times *high word resolve* is run, since it represents a significant percentage of the computation time for small b. Multiplying these counts by the number of squaring and multiplication steps executed during the fixed window exponentiation algorithm, we derive the exact number of computation instructions required for an exponentiation.

The instruction counts for various models and sizes are in Table 6.1. The leftmost column gives the model and model parameters: KS is the number of levels of Karatsuba squaring, KM is the levels of Karatsuba multiplication, S is the sample size, and T is the number of threads in the group. Each column to the right represents a different size of b. Since the algorithms are roughly O(b^3), as we move from left to right, each doubling of b results in the counts increasing by nearly a factor of 8.

CARD	Micro Architecture	Compute Capability	Freq GHz	Number of SMs	Cores Per SM	Cycles/Instructions per Operation					
						madc.lo	madc.hi	add/sub	mad24.lo	mad24.hi	carry resolution
9800 GT	G92	1.1	1.500	14	8	5	5	1	1	1	5
GTX 260	G200	1.3	1.296	24	8	5	5	1	1	1	5
GTX 580[1]	Fermi	2.0	1.544	16	32	2	2	1	–	–	–
GTX 680[1]	Kepler	3.0	1.110[2]	8	192	1	1	1	–	–	–
GTX 780Ti[1]	Kepler	3.5	1.046	15	192	1	1	1	–	–	–
GTX 750Ti[1]	Maxwell	5.0	1.084	5	128	4	6	1	–	–	–

Table 6.2. Instructions/Cycles for each madc.lo, madc.hi and add/sub across the generations of GPU cards

Note that the Three N model with Karatsuba has the fewest multiplications and thus we would expect it to perform best. But it also requires more registers per thread than the Local or Distributed models. Although the instruction counts increase for those models, the number of registers per thread decreases, enabling more threads to execute in parallel. The Sampled model uses the most registers and has the highest instruction counts, so we would expect it to always have the worst performance. But as we will see, it outperforms the other models on the 1.x architectures for small b. For large b, however, its performance is always worse. Our implementation would need to be further extended to handle an overflow that occurs with larger samples on larger instances of b, but since there is nothing to gain from doing so, we simply note it in those table entries.

To establish the theoretical maximum sustained instruction throughputs on each architecture, we turn to four sources. Section 5.4 of the CUDA C Developers Guide (Maximizing Instruction Throughput) [28], the cuobjdump tool, the deviceQuery tool. and our own micro-benchmarks. Our findings are summarized in Table 6.2.

On the 9800 GT, the GTX 260 and on the GTX 750Ti, there are no native 32-bit by 32-bit multiply instructions. Instead, the code generator and ptxas assembler emulates them with a sequence of simpler instructions based on 16-bit multiplies. These instructions all

[1] 24-bit multiplications are emulated in software on these cards, but we haven't determined the number of instructions used to emulate them

[2] device query reports the clock rate as 1056 MHz, but when we actually measure the rate, it is 1110 MHz

run in a single cycle. On the 9800 and 260, each madc.lo and each madc.hi is emulated with 5 instructions. On the 750Ti, madc.lo takes 4 instructions and madc.hi takes 6 instructions.

The 580, 680 and 780Ti all support native madc.lo and madc.hi instructions. However, these architectures are asymmetric in the sense that they have fewer 32-bit integer multipliers than floating point units or add/sub units. Thus a 680 or 780Ti can dispatch 6 instructions to 4 different warps in a single cycle, but only one of those warps can be running a multiply instruction. Since our software does so many multiplications as compared to adds and subtracts, in most cases the latter can be completely overlapped with the multiplications and we find the overall throughput is dominated by the time for the multiplications. Thus for the 580, 680 and 780Ti, we define utilization as the measured multiplications per second divided by the theoretical maximum number of multiplications per second.

In the 9800GT and GTX 260, adds and subtracts cannot be issued in parallel with multiplies. In the 750Ti, two instructions can be issued at once (per warp scheduler), but only one integer instruction can be issued per cycle. Thus, adds and subtracts also compete with multiplies in this most recent architecture. In recognition of this serialization of integer operations, for these architecture we define utilization as the measured integer instructions dispatched per second divided by the theoretical maximum integer dispatch rate.

To clarify, we give an example of how we calculate utilization. On the GTX 580, the Three N with no Karatsuba achieves 713,913 512-bit modular exponentiations per second. Each modular exponentiation requires 532,940 multiplications. Thus the software achieves a rate of 380.5 billion multiplications per second. The card is capable of 395.3 billion multiplications per second (1.544 GHz $\cdot$ 16 SMs $\cdot$ 32 cores / 2 cycles). Thus we achieve a utilization of 96.3%.

Model	Exponentiations Per Second (thousands) / Utilization					
	9800	260	580	680	780Ti	750Ti
Three N	28.56	45.20	713.91	515.83	894.42	**201.90**
no karatsuba	46.5%	49.7%	96.3%	96.7%	94.9%	**79.6%**
Three N	29.33	46.61	726.50	540.24	956.17	198.11
KM=0, KS=1	46.0%	49.4%	92.6%	95.7%	95.9%	75.3%
Three N	29.52	47.39	700.34	550.78	972.22	178.27
KM=0, KS=2	45.9%	49.8%	85.9%	93.9%	93.9%	67.2%
Three N	28.80	46.10	729.10	527.79	942.16	177.27
KM=1, KS=0	45.8%	49.5%	95.5%	96.2%	97.2%	68.2%
Three N	29.40	47.85	**740.00**	554.51	967.72	181.64
KM=1, KS=1	45.0%	49.4%	**95.3%**	95.3%	94.1%	67.3%
Three N	**29.55**	**48.27**	710.14	**566.49**	**976.48**	177.00
KM=1, KS=2	**44.8%**	**49.4%**	84.4%	**93.6%**	**91.3%**	65.0%

Table 6.3. Impact of Karatsuba on Performance - 512 bits with $w = 5$ and a launch geometry of 128 threads per block

6.7.2 Results and Discussion

In this section we present our results and analysis. We first examine the impact of the different algorithms and model parameters on the performance of 512 bit exponentiation with a window size of 5 bits and 128 threads per block. In Table 6.3 we present the impact of Karatsuba on Three N performance. For each architecture, we mark the parameters that produced the best performance in bold.

As can be seen from Table 6.1, going from KM=0, KS=0 to KM=1, KS=0, saves roughly 15K multiplications at the cost of 10K additions and subtractions (and some control instructions). On the 9800 GT and the GTX 260, each multiplication takes 5 instructions. Thus, as one would expect, using Karatsuba multiplication is faster than not. Going from KM=1, KS=1 to KM=1, KS=2, saves roughly 19K mulitplications at a cost of 62K additions. Again, because because multiplication is more expensive than addition, this is still profitable, but the returns are diminishing.

On the 580, 680 and 780Ti, multiply, add and subtract each require only a single instruction dispatch. However, as mentioned earlier, multiply and add/subtract can be dispatched simultaneously. On the 580 one multiply and one add/subtract can be dispatched every two cycles. On the 680 and 780Ti, one multiply and 5 add/subtracts can be dispatched

Model	Exponentiations Per Second (thousands) / Utilization					
	9800	260	580	680	780Ti	750Ti
Sampled S=20	47.15 49.3%	96.70 68.2%	137.04 –	104.52 –	160.13 –	34.80 –
Sampled S=21	51.60 50.2%	106.51 70.0%	148.81 –	113.79 –	176.57 –	37.48 –
Distributed T=1	4.50[3] 9.0%	41.0[3] 55.2%	562.21 95.3%	409.30 96.5%	719.73 96.0%	147.57 72.8%
Distributed T=2	30.4[3] 61.1%	44.7[3] 61.5%	547.20 94.2%	395.94 94.8%	663.39 89.9%	132.87 67.5%
Distributed T=4	27.0[3] 56.3%	40.2[3] 56.6%	393.53 69.8%	381.45 94.1%	581.69 81.2%	116.67 62.8%
Local	14.89 24.2%	44.17 48.6%	715.58 96.4%	513.91 97.2%	894.99 94.9%	199.21 78.5%

Table 6.4. Impact of algorithms on Performance - 512 bits with $w = 5$ and a launch geometry of 128 threads per block

per cycle. On the 580, the fastest is KM=1, KS=1. The jump to KS=2 shifts the workload enough that other instructions start to interfere with dispatching multiplies (this can be seen by the drop in utilization). On the 680 and 780Ti, the jump to KS=2 continues to improve performance, although the returns are diminishing.

The results for the 750Ti are a surprise. Given that it takes 4 instructions for madc.lo and 6 instructions for a madc.hi, one would expect using Karatsuba would be profitable, as is the case on the 9800 and 260. However, from our results, we see that it is faster to not use Karatsuba multiplication or squaring. The decreasing utilization rates suggest that perhaps the control instructions used in the Karatsuba implementation are substantially slower on Maxwell than on the other architectures.

In Table 6.4 we look at the Sampled, Distributed and Local models. The table is for 512 bit exponentiation with a window size of 32 and 128 threads per block.

The Sampled model is designed to take advantage of the native 24-bit multiplication instructions and shines on the 9800 and 260 cards, achieving rates of 51.60K and 106.51K modular exponentiations per second respectively, as compared to 29.40K and 47.85K for

[3]results from our prior work [41], compiled under CUDA 4.2

Model	Exps per Second (thousands) / Utilization / Warps / Blocks					
	LG=32	LG=64	LG=128	LG=192	LG=256	LG=512
Three N	414.25	714.323	713.91	**729.14**	709.21	708.63
no kar	55.9%	96.3%	96.3%	**98.3%**	95.6%	95.6%
warps	8	16	16	**18**	16	16
blocks	8	8	4	**3**	2	1

Table 6.5. Impact of Launch Geometry on Performance - GTX 580, 512 bits with $w = 5$

the Three N model. For the other cards where 24-bit multiplication is emulated in software, the performance is quite poor. As noted in Table 6.2 we have not determined the number of instructions required for each 24-bit multiplication on each card, so we are unable to calculate the utilization. However, it's not a serious omisssion because the Sampled model is so much slower than the Three N model on these cards.

For the Distributed model, we show three sets of results for T=1, T=2 and T=4, where T is the number of threads per exponentiation instance. Comparing the results for T=1 to the Three N model with no Karatsuba, we see that the Distributed model is roughly 20% slower for equivalent configurations. This is due to the inter-thread communication overhead and the lack of a fast square operation. For the 9800 and 260, we ran into what appears to be compiler errors under CUDA 6.5, so we report our results from our prior work [41] which was compiled under CUDA 4.2.

The Local model uses a two pass approach to multiplication and squaring, allowing it to efficiently run with 2 MP values in registers instead of 3. As we can see, the 512-bit performance is roughly equivalent to the Three N model without Karatsuba.

In Table 6.5 we look at the impact of launch geometry on the performance of 512 bit modular exponentiation on the GTX 580 using the Three N model, with no Karatsuba.

The modular exponentiation kernel compiled for the GTX 580 uses 54 registers. The 580 supports a maximum of 8 blocks and 32K registers. With a launch geometry of 32 threads per block, the GTX 580 will run 8 warps per SM. With a launch geometries of 64, 128, 256 and 512 threads per block, there will be 16 resident warps per SM. With a launch geometry of 192, there will be 18 resident warps. From Table 6.5 we see that performance

Model	Exps per Second (thousands) / Utilization / Warps / Blocks					
	RM=54	RM=50	RM=44	RM=42	RM=36	RM=32
Three N	713.91	**725.07**	720.00	722.80	713.34	509.79
no kar	96.3%	**97.8%**	97.1%	97.5%	96.2%	68.7%
warps	16	**20**	20	24	28	28
blocks	4	**5**	5	6	7	8

Table 6.6. Impact of Register Max allocation on Performance - GTX 580, 512 bits with LG=128, $w = 5$

on the GTX 580 is maximized by maximizing the number of resident warps and secondarily by maximizing the number of blocks.

The CUDA ptxas assembler supports an option *-maxrregcount* to specify the maximum number of registers that can be used by the kernel. The fewer registers the more often the kernel will spill to local memory, thus making the kernel run slower. However, the fewer registers used, the more warps that can be resident, which increases parallelism and increases throughput. We explore these trade-offs in Table 6.6.

Without specifying a maximum number of registers (RM), ptxas allocates 54 registers for the modular exponentiation kernel. This produces the nominal 713.91K mod exps per second seen in the prior tables. As we reduce the number of registers available for allocation, the performance decreases smoothly until we reach 50 registers at which point, the number of resident warps on the GTX 580 jumps to 20 and the performance increases to 725.07K mod exps per second. From 50 to 44 registers the performance again decreases smoothly, and at 42 registers the GTX 580 can fit a 6th block and the performance increases slightly to 722.80K. But at some point, the number of register spills overwelms the benefits of the added parallelism. For this kernel it starts at 34 registers and by 32 registers, the performance has dropped to 509.79K mod exps per second.

6.8 Comparison to Prior Work

In Table 6.7 we present our best results for each of the six architectures and for four problem sizes. The top section gives the peak throughput achieved (in thousands where

	256 bits			512 bits			1024 bits			2048 bits		
Card	Model	Exps/s (1000s)	Utiliz.	Model	Exps/s (1000s)	Utiliz.	Model	Exps/s (1000s)	Utiliz.	Model	Exps/s	Utiliz.
9800	Sampled	545.61	77.3%	Sampled	51.63	50.3%	Distrib	3.67	57.0%	Distrib	478	58.0%
260	Sampled	822.42	78.7%	Sampled	118.72	78.0%	Distrib	5.68	59.5%	Distrib	643	52.7%
580	Three N	5806.28	95.1%	Three N	765.94	94.7%	Local	85.58	88.6%	Distrib	9163	95.6%
680	Three N	3913.66	89.9%	Three N	566.97	94.5%	Local	59.72	86.8%	Distrib	6476	95.3%
780Ti	Three N	6753.25	87.1%	Three N	998.81	93.4%	Three N	127.85	95.1%	Distrib	10778	89.4%
750Ti	Three N	1736.34	90.2%	Three N	204.49	80.6%	Three N	22.75	69.1%	Distrib	2052	63.2%

Corresponding Model Parameters

	256 bits, W=4 Model / Parameters	512 bits, W=5 Model / Parameters	1024 bits, W=6 Model / Parameters	2048 bits, W=6 or 7 Model / Parameters
9800	Sampled: LG=192, S=22	Sampled: LG=128, S=21, RM=60	Distrib: LG=64, T=2	Distrib: LG=64, T=4, W=6
260	Sampled: LG=128, S=22	Sampled: LG=192, S=21	Distrib: LG=64, T=2	Distrib: LG=64, T=4, W=6
580	Three N: LG=128, KS=1, KM=1	Three N: LG=128, KS=1, KM=1, RM=50	Local: LG=128	Distrib: LG=128, T=8, W=6
680	Three N: LG=128, KS=1, KM=1	Three N: LG=128, KS=2, KM=1	Local: LG=128	Distrib: LG=128, T=8, W=6
780Ti	Three N: LG=128, KS=1, KM=1	Three N: LG=128, KS=2, KM=1, RM=63	Three N: LG=128, KS=1, KM=1, RM=126	Distrib: LG=128, T=8, W=7
750Ti	Three N: LG=128, KS=0, KM=1	Three N: LG=512, KS=0, KM=0	Three N: LG=128, KS=0, KM=0, RM=126	Distrib: LG=128, T=8, W=7

TABLE KEY: LG=Launch Geometry KS=Karatsuba Squaring levels KM=Karatsuba Multiplication Levels S=Bits per Sample T=Threads per exponentiation instance RM=Register max (specified via ptxas option -maxrregcount) W=Window Size

Table 6.7. Best Performing Model by Size and Card

noted). The bottom section has the same organization but shows the search parameters corresponding to each of those results.

We compare our results to the two papers with the fastest results we are aware of. The first is by Jang et al [64]. They report RSA decryptions per second of 322K at 512 bits, 75K at 1024 bits, 12K at 2048 bits and 1.7K at 4096 bits on a GTX 580. Each RSA decrypt requires two half-size modular exponentiations and a CRT reconstruction. To scale our results to compare with Jang we must divide our throughput numbers by approximately 2.05 (the 0.05 is to make allowance for the CRT reconstruction). On the GTX 580, for 512-bit RSA decryption, our peak throughput is 2.83M per second, 8.8 times faster than Jang et al. At 1024 bits, our peak performance is 374K, 5 times faster. At 2048 bits, our peak is 41.7K, or 3.5 times faster and at 4096 bits, our peak is 4.5K, slightly more than 2.6 times faster. Compared with the 780Ti, our performance is 10.2, 6.5, 5.2, and 3.1 times faster, respectively, for the same problem sizes.

Neves and Araujo report a throughput of 41.4K 512-bit modular exponentiations per second on a GTX 260. Our result for the same computation on the same processor is

118.72K, so we are roughly 2.9 times faster. Our result for 512-bit exponentiation on the 780Ti is 998.81K, or 24 times faster.

Some interesting points to note in our results are that the GTX 680 and 750Ti both deliver lower performance than their immediate predecessors and in fact the 750Ti is only about twice as fast as the GTX 260. In the case of the 680, the architecture halved the number of SMs (and thus the number of integer multipliers), doubled the throughput of the multipliers, and decreased the clock speed by nearly 30%. Although the new SMs (called SMX) each have double the number of registers, the GTX 680 ends up with the same number of registers as the GTX 580. As we have noted, GPU performance on modular exponentiation depends on keeping values in registers, so the GTX 680 only matches the 580s multiplication capacity, but falls behind due to the decreased clock speed.

The 750Ti is an early release of the Maxwell architecture, which is redesigned for lower power. It has even fewer SMs than the GTX 680 and runs at essentially the same clock speed. It also returns to the practice of the earliest generations of requiring multiple instruction cycles for integer arithmetic. We found it particularly interesting that this architecture, unlike the Kepler generation, does better without Karatsuba squaring, and at all but the smallest problem size, it even needs to avoid the Karatsuba multiply.

One of the particularly interesting aspects of our results is that in the majority of cases, a unique combination of the search space parameters was needed to obtain best performance. Although we do not show the hundreds of weaker results from suboptimal combinations, in nearly every case, there was a large gap between the best and next best combination. Thus, the full search was indeed necessary. And at the same time, our high utilization figures indicate that our search space was sufficient for finding combinations that closely approach peak performance.

6.9 Conclusions

We have shown that GPUs are capable of performing multiprecision modular exponentiation at a very high level of throughput. For the most common size employed in cryptography, we report that a recent GPU can perform nearly a million exponentiations per second, and even for a GPU that is two generations old, a rate of 766 thousand per second is achievable. Our results are several times better than prior work, and in many cases are within ten percent of the theoretical peak utilization, leaving little room for additional improvement. We have applied our techniques across six generations of NVIDIA GPU, reaching as far back as Compute Capability 1.1. These levels of performance can provide a low cost means of offloading computationally intensive decryption work from server CPUs.

We found that it was very difficult to predict performance a priori for any given combination of algorithm and architecture. Many of our optimal combinations were counterintuitive and it took considerable effort to understand why they outperformed others that appeared to be better choices. In many cases we were only able to find the optima as a result of searching a multidimensional design space. After years of preliminary ad hoc exploration, it was the development of this comprehensive search space that enabled us to methodically find the best combinations. A key enabling element was the identification of four different storage models for registers and memory that enabled us to treat these resources as a dimension in the search, and that offer alternative virtual views of the storage resources for different algorithms and problem sizes.

A critical insight was that prior approaches implicitly assumed a small register file and fast cache, whereas maximizing parallelism on the GPU depends on keeping computations in registers as much as possible and minimizing the shuffling of data through memory. Although of secondary importance, the available operations also have a strong influence on optimization. Thus, attaining best performance depends on creating and searching a comprehensive design space that in part involves defining a set of virtual views that manage

storage resources in different arrangements to which specific algorithms more naturally map.

Our work was partly motivated by inefficiencies we observed in traditional register allocation as done in the CUDA compiler. In solving that problem, it became clear that the traditional approach used in CUDA and ptxas, where an intermediate form assumes infinite registers and a subsequent phase tries to allocate those to the actual resources through techniques such as dependence and live range analyses, is inadequate, at least for developing core libraries.

Instead, we would argue that a new paradigm is needed in which operations are implemented through multiple equivalent algorithms that can be described and parameterized at a higher level. Code generation then involves search over a well-defined space with dimensions such as parameterized algorithms, storage models, problem sizes, processor capabilities, launch geometry, etc. The virtual views presented by our models, in particular, provide a higher level organization for register allocation that avoids much of the waste we observed in the usual fine-grained view.

If we can find ways to apply such a paradigm more broadly, then obtaining best performance from GPU-like architectures, especially for core libraries, will no longer have to depend on heroic manual efforts that can nonetheless fail to obtain optimality and which can be made obsolete with even a minor change in the next generation of hardware.

CHAPTER 7

MODULAR EXPONENTIATION USING DOUBLE PRECISION FLOATING POINT ARITHMETIC

In this chapter we examine a new approach to multiple precision (MP) modular exponentiation on the GPU using double precision floating point arithmetic. There have been a number of papers that have used floating point arithmetic as the basis for modular multiplication and exponentiation algorithms. Moss, Page and Smart [84] use an RNS approach with a vector of co-prime moduli, where each modulus is 12-bits in length and compute $a_i \cdot b_i$ fmod m_i using single precision floating point arithmetic. Fleissner [47] implements 192-bit modular exponentiation with six 24-bit values. The 24-bit values are further sampled into bytes and the computations on the bytes are done using single precision floating point arithmetic. In [11] Bernstein et al. implement a 280-bit modular multiplication based on a traditional FRNS with 28 limbs of 10-bit samples and a three-product Montgomery reduction (see Section 2.2.6 for a full explanation of the Montgomery reduction). They use a complicated scheme to distribute each multiplication across 28 threads in a warp. But in essence, each column sum can be thought of as a dot product, $\sum_{i,j} a_i \cdot b_j$ where A and B are the MP values to be multiplied and $i + j$ equals the column number. The a_i and b_i samples and all of the computations are done using single precision floating point arithmetic. But we note that $28 \cdot (2^{10} - 1) \cdot (2^{10} - 1)$ can exceed 2^{24} which can result in rounding errors. To work around this problem, Bernstein et al. compute two partial sums of at most 14 terms, which guarantees there won't be any rounding. Bernstein et al. have carefully crafted a highly optimized implementation, but unfortunately, 10-bit samples are just too small to be efficient. In their subsequent paper [10], Bernstein et al. perform all computations in the integer domain, which proved to be much faster. Zheng et al. in [112]

and a follow-on paper by Dong, Zheng, et al. [33] again use floating point arithmetic to implement modular multiplication, modular exponentiation, and RSA. They use an FRNS with either 22 or 23 bits per limb (depending on the size of the RSA) and a fast word-by-word Montgomery reduction. The samples and all computations are performed using double precision floating point arithmetic and by using 22 and 23 bit limbs, they can ensure that the column sum does not exceed 2^{53}, the largest integer that can be stored in a double without rounding. The latter work by Dong, Zheng, et al. is the first paper that shows that floating point implementations can outperform the best integer implementations, at least at large sizes. These papers all use *narrow samples*, i.e., the number of bits in each sample is always less than or equal to half the number of bits (width) available in the mantissa. In this chapter, we will develop techniques for a new approach using *wide samples* where each sample will have almost the same number of bits as the mantissa.

The rest of this chapter is organized as follows, Section 7.1 describes the wide samples approach with tricks and optimizations that allows it to be implemented efficiently. Section 7.2 estimates the performance of the wide samples approach as compared to the traditional integer approach across a range of NVIDIA micro-architectures. Section 7.3 covers our implementation of modular exponentiation and modular multiplication using wide samples. In Section 7.4 we present our experiments and results, and we close with conclusions and future work in Section 7.5.

7.1 New Approach Using Wide Samples

The idea behind our approach is straightforward. We will use wide samples with 52-bits per sample and will compute the products with double precision arithmetic. We will require the ability to compute the high and low products of two samples a_i and b_i. Computing these products is a well known problem with solutions that date back to the 1970s. Dekker [31] solved the problem by splitting a_i and b_i into upper and lower halves and computing the products from the half precision values. Dekker's approach is quite slow, but with the

advent of hardware based fused-multiply-and-add (FMA), which is present on the GPU, there is a fast algorithm as follows, where *__fma_rz* is the CUDA built-in function for a double precision FMA with rounding towards zero:

```
full_product(double a_sample, double b_sample)
  p_hi = __fma_rz(a_sample, b_sample, 0.0);
  p_lo = __fma_rz(a_sample, b_sample, -p_hi);
  return (p_hi, p_lo);
}
```

Figure 7.1. Compute the high and low product of two samples using fused multiply and add

this approach is discussed in [70] and [59]. However, the resulting products must be normalized to align the decimal points before they can be added to a column sum. To illustrate, consider the following example: suppose a_0 and b_0 are both 2^{50} and a_1 and b_1 are both 1. Using the full product routine gives us $p_0 = (2^{100}, 0)$, $p_1 = (1, 0)$. With component-wise addition $p_0 + p_1$ yields $(2^{100}, 0)$ because of round-off and alignment problems, not the desired sum of $(2^{100}, 1)$.

The double precision format in the IEEE-754 standard uses a 64-bit representation, where the most significant bit is the sign, the next 11 bits are the exponent (with a bias of 1023), and for normal floating point values, the remaining 52 bits form a 53-bit mantissa, where the most significant bit of the mantissa is always implicitly set to one. Implicit meaning the most significant bit of the mantissa is not stored in the IEEE-754 representation, which saves a bit.

We can take advantage of this representation as follows. Since the product of two samples is guaranteed to be less than 2^{104}, if we compute $p\_hi' = a_i \cdot b_i + 2^{104}$ as a floating point value, then 2^{104} will be the most significant bit of the result and will become the implicit bit in the representation. The next 52 bits (the explicit bits) of the mantissa will exactly match the 52-bit high product that would have resulted from the same computation in the integer domain. The same trick can be applied in the computation of the low product, $p\_lo' = a_i \cdot b_i - (p\_hi' - 2^{104}) + 2^{52}$. Computing the low and high products in this way

```
full_product(double a_sample, double b_sample)
  double   c1=2^104, c2=2^104 + 2^52, sub;
  uint64_t mask=2^52 - 1;

  p_hi = __fma_rz(a_sample, b_sample, c1);
  sub = c2 - p_hi;
  p_lo = __fma_rz(a_sample, b_sample, sub);
  return (to_u64(p_hi) & mask, to_u64(p_lo) & mask);
}
```

Figure 7.2. Normalized high and low products. Note, this algorithm **requires** rounding towards zero. Rounding towards nearest will produce incorrect results.

forces the alignment of the decimal points and can be efficiently computed using only three double precision instructions. Once we have the low and high products, we have to accumulate them into column sums. If we try to compute the column sums in the floating point domain, we will need more precision than a double since our high and low products already have 53-bit values. It makes more sense to compute the column sums using 64-bit unsigned integer arithmetic. To do so, we can simply convert the p_hi' and p_lo' values to their IEEE-754 bit representation and mask off the top 12 bits. The bottom 52 bits are exactly the integer value that needs to added to the column sum. This leads to the full product algorithm shown in Figure 7.2 which returns a pair of unsigned 64-bit integers representing the high and low products. The routine employs a function *to_u64* to convert double precision values to their bit representation. The function can be implemented with an inline PTX assembly or with C unions. On the GPU, the floating point values use the same registers as integer values, so the conversion is essentially free.

With one more small trick, we can eliminate the masking operations by noticing that the sign bit of p_hi' is always 0 and the exponent is always 104 plus 1023 (the bias), thus the top 12 bits of p_hi' are always 0x467, and the top 12 bits of p_lo' are always 0x433 (52 plus 1023). Instead of the masking operations, we can add the top 12 bits into the column sums and then by tracking how many high and low products have been summed into each column, we can cancel off their total with a single subtraction operation. We can even save the final subtraction by initializing the column sums with the correct magic values. The algorithm in Figure 7.3 computes an 8-sample (420 bits) by 8-sample multiple precision

```
#define N 8
void sampled_product(uint64_t *column_sums, double *a_samples, *double b_samples)
  double   c1=2^104, c2=2^104 + 2^52, sub;
  uint64_t mask=2^52 - 1;
  int      index;

  for(index=0;index<N;index++) {
    column_sums[index]=magic(index, index+1);
    column_sums[2*N-1-index]=magic(index+1, index);
  }

  for(i=0;i<N;i++) {
    for(j=0;j<N;j++) {
      p_hi = __fma_rz(a_samples[i], b_samples[j], c1);
      sub = c2 - p_hi;
      p_lo = __fma_rz(a_samples[i], b_samples[j], sub);
      column_sums[i+j+1] += to_u64(p_hi);
      column_sums[i+j] += to_u64(p_lo);
    }
  }
}
```

Figure 7.3. MP Sampled Product Algorithm

full product returning the 16 column sums of the result. The algorithm uses a routine called *make_initial* (shown in Figure 7.4) which takes two arguments, the high product count and low product count, and generates the correct initial value for the column sum that will cancel the 0x467s and 0x433s.

If the loops of the algorithm in Figure 7.3 are completely unrolled and we make some reasonable assumptions about the code generated by the CUDA tool chain, then an N-sample by N-sample product will require $3N^2$ double precision floating point operations and $4N^2$ 32-bit additions (two 32-bit addition instructions per 64-bit addition in the algorithm). This can be quite efficient for GPU cards with high double precision throughput.

```
uint64_t make_initial(int high_count, int low_count) {
  uint64_t value=high_count*0x467 + low_count*0x433;

  return -((value & 0xFFF)<<52);
}
```

Figure 7.4. MP Sampled Product Algorithm

7.2 Performance Estimates for Various Cards

In the previous section, we established that a 52-bit full product can be computed with 3 double precision instructions and 4 32-bit integer add operations. In this section we look at the NVIDIA micro-architectures where a sampled approach could potentially outperform the traditional 32-bit integer approach to MP multiplication and modular reduction. The basic operation needed to implement the classic $O(N^2)$ algorithms is the ability to compute a full product and accumulate it with the column sum(s), henceforth called FPACS. Table 7.1 shows the number of instruction dispatch cycles to implement FPACS across a warp of 32 threads on a single streaming multiprocessor (SM) of a particular generation using the 52-bit sampled approach and a traditional 32-bit integer approach. On the GPU, the number of dispatch cycles is the most important metric in determining the throughput of an operation, assuming there are sufficient ready warps to saturate the ALU pipelines. Compute capability 1.x devices do not support double precision arithmetic, so Table 7.1 begins with Fermi, compute capability 2.x. The first row of the table is directly calculated from the tables in Section 5.4 of the CUDA Programming Guide [28] on the throughput of various instructions, for example, Kepler cards supporting fast double precision floating point arithmetic can dispatch 64 double precision operations per clock cycle and 160 32-bit add operations per clock cycle. Thus the 52-bit sampled FPACS represents 1.5 cycles of floating point dispatch and 0.8 cycles of 32-bit addition, for a total of 2.3 cycles. Fractional cycles might seem like a strange notion, but keep in mind that a streaming multiprocessor can dispatch multiple instructions to multiple warps simultaneously. We also note that all

	Fermi 2.x	Kepler 3.x	Maxwell 5.x	Pascal 6.x	Volta 7.x
52-bit sampled FPACS	10.0	2.3	25.0	5.0	5.0
32-bit integer FPACS	4.0	2.0	1.2	2.4	1.1
Estimated speedup, if faster	6%	130%	—	27%	—

Table 7.1. Cycles required to dispatch a 52-bit sampled or 32-bit integer FPACS to a warp of 32 threads

Maxwell cards have very poor double precision performance, hence the 25 cycles for the sampled FPACS method. The second row of the table represents the number of cycles to dispatch a 32-bit integer FPACS and is slightly more complicated. The Fermi and Kepler values are derived from the same tables in [28]. Maxwell has a 16-bit multiplier in hardware, which leads to complicated alignment problems. It takes 10 instruction dispatches to implement a single 32-bit integer FPACS, however, Maxwell can dispatch four instructions to different warps simultaneously, thus for throughput measurements it's 2.5 cycles per 32-bit integer FPACS for a single column sum. For MP multiply and reduce, there are some clever algorithms that can significantly reduce the alignment problems which brings the cycles down to something in the neighborhood of 1.2 cycles per 32-bit integer FPACS on a 256-bit multiply and reduce (see for example [37]). Pascal is quite similar to Maxwell, with a 16-bit hardware multiplier. The difference is that it has half as many cores per SM, but it has more than twice as many SMs. Thus the cycles per 32-bit FPACS doubles (compared to Maxwell), but the overall throughput per GPU increases. Unlike Maxwell, some Pascal GPUs do have good double precision performance. Finally, there is Volta. Volta has a full 32-bit multiplier in hardware and can dispatch a 32-bit times 32-bit value plus a 64-bit value in a single cycle. However, it is a new architecture and the compiler is not yet generating very good code on the MP routines. Our best guess is that when everything settles down it'll be between 1.0 and 1.1 cycles per 32-bit integer FPACS.

From the 52-bit sampled FPACS cycles (which we'll call s) and the 32-bit integer FPACS cycles (i) we can calculate an estimate for the speed up of the sampled approach over an integer approach as:

$$\text{estimated speedup} = \frac{i \cdot 52^2}{s \cdot 32^2}$$

The bit sizes (52 and 32) are squared because the classic multiply and reduce algorithms are $O(N^2)$. The calculated estimated speedup is shown in the third row of the table. We note there are many factors that impact performance that are absent from this analysis including register pressure, detailed instruction pairing rules on each generation, compiler code

generation, sampling overhead, memory access patterns, shared memory block conflicts, and finally, sometimes the hardware just doesn't behave the way you expect it to. Thus, while this is a useful analysis, it's important to understand its limitations. However, we can safely conclude that Kepler will, by a large margin, be the best target for the wide samples approach.

7.3 Implementation of Modular Exponentiation using Wide Samples

We implement modular exponentiation using the fixed window algorithm of Section 2.2.9. As in the previous chapter, the fixed window exponentiation routine uses an abstract interface to decouple it from the implementation details of the modular multiplier. The interface between the two classes is similar to the one described in Figure 6.3, but in this case, we assume that there are three MP values, A, B and M stored in sampled form in registers. The interface relies on two routines, the *multiply* routine computes $A \cdot B \bmod M$ using a Montgomery reduction and *square* computes $A \cdot A \bmod M$, also using a Montgomery reduction. Since we wish to use the same code to support sizes up to 2048 bits, we employ a distributed model where 2, 4, or 8 threads are assigned to each instance. The limbs of the A, B, and, M values and the column sums are divided into contiguous chunks which are distributed to the group of threads responsible for the instance. This is analogous to the *slices* of Chapter 4. The modular multiplier is split up in three routines, as follows. The first is shown in Figure 7.5, whereby each thread computes a product of a single 52-bit

```
template<int limbs>
void rowmul(uint64_t sums[limbs+1], double term, double v[limbs]) {
  double hi, temp, lo, c1=2^104, c2=2^104+2^52;

  for(int index=0;index<limbs;index++) {
    hi=__fma_rz(term, v[index], c1);
    temp=hi-c2;
    lo=__fma_rz(term, v[index], temp);
    sums[index+1] += to_u64(hi);
    sums[index] += to_u64(lo);
  }
}
```

Figure 7.5. 52-bit sampled row multiplier

```
template<int n, int threads, int limbs>
void modmul(uint64_t sums[limbs+1], double a[limbs], double b[limbs],
            double m[limbs], uint64_t np0) {
  uint32_t groupBase=threadIdx.x & ~(threads-1);
  uint64_t send64, mask52=0xFFFFFFFFFFFFFull;
  double   term;

  // initialize the low limbs
  for(int word=0;word<limbs;word++)
    sums[word]=make_initial(2*word, 2*word+2);

  for(int index=0;index<n;index++) {
    // initialize the high limb
    if(index<n-1-limbs)
      sums[limbs]=make_initial(2*limbs, 2*limbs);
    else if(index<n-1)
      sums[limbs]=make_initial(2*(n-1-index), 2*(n-1-index-1));

    // accumulate b_i * A, followed by q_i * M
    term=__shfl(b[index % limbs], index/limbs + groupBase);
    rowmul<limbs>(sums, term, a);

    term=(double)__shfl(sums[0]*np0 & mask52, groupBase);
    rowmul<limbs>(sums, term, m);

    // because of the Montgomery property, the least significant 52 bits of the
    // least significant thread will always be zero.  So we can use wrap around
    // without any additional zeroing.
    send64=__shfl(sums[0] & mask52, threadIdx.x+1, threads);
    sums[0]=sums[1] + (sums[0])>>52);

    // all other limbs shift right by one column
    for(int word=1;word<limbs-1;word++)
      sums[word]=sums[word+1];
    sums[limbs-1]=sums[limbs] + send64;
  }
  // the high limb is empty, zero it
  sums[limbs]=0;
}
```

Figure 7.6. 52-bit sampled modular multiplication based on a CIOS Montgomery product

sample (called *term*) by its slice of the MP value and accumulates the results into its slice of the column sums. The routine uses a single template argument to specify the number of 52-bit limbs in the slice.

The second algorithm, shown in Figure 7.6, implements a word-by-word CIOS Montgomery product. It uses three template arguments, where *n* specifies the total number of limbs in an instance, *threads* is the number of threads assigned to each instance and *limbs* is the number of 52-bit limbs in each slice. Note, in some cases, *n* might be less than $threads \cdot limbs$. The algorithm first initializes the column sums. Then it iterates through the limbs of B, accumulating $A \cdot b_i$ followed by $M \cdot q_i$ into the column sums. Since the

column sums are 64 bits per limb, rather than the 52 bits of the sample, the column sums form a redundant representation where the upper 12 bits of each column overlap with the next (more significant) column sum. At the end of each iteration, we must shift the column sums one sample, or 52 bits to the right. To implement this, each thread splits the least significant column sum into an upper 12 bits and a lower 52 bits. The upper 12 bits gets added to its next most significant column sum. The lower 52 bits gets sent, using a *__shfl*, to the next lower thread for inclusion in its most significant column sum. One clever trick we use is that the shuffle is circular, i.e., the lowest thread in the group sends its value around to the highest thread. This only works because we know that the lowest thread's value will be zero (recall that a word-by-word Montgomery product always zeros out the least significant limb). Once we have iterated through all n of the b_i terms, the column sums will represent Montgomery product result. However, the column sums are still in an overlapped form, and these carries must be resolved and the column sums reduced down to 52 bits before the next multiplication can be performed.

Since we are using a sampled approach, for the most common modular exponentiation sizes of 1024 bits, 1536 bits, and 2048 bits, we will have at least 16 extra bits available in at the top of each MP value. We can take advantage of this to improve performance. First, as per Orup [87] and Walter's [107] work, we can allow the Montgomery results to range from 0 to $2M - 1$ and thus eliminate the need for the correction step. Second, even after allowing for the greater range, the top 15 bits of the MP value will always be zero and we can use that to shave off a few instruction here and there in the carry resolution routine.

The third algorithm, shown in Figure 7.7, is used to resolve the carries between column sums (within a thread) and between threads. It first sends the upper 12 bits the most significant column sum to the next higher thread. This is done in a circular fashion where the highest thread sends its 12 bits, which will be zero, to the lowest thread in the group. The received values are added in to the least significant column sum in the thread. Next, we iterate from least significant column sum to the most significant, masking off the top 12

```
template<int threads, int limbs>
void resolve(uint64_t *sums) {
  uint32_t send32, generate, propagate, lane=1<<(threadIdx.x & 0x1F), carry, nextCarry;
  uint64_t mask52=0xFFFFFFFFFFFFFull, testCritical;

  // use a circular shift, we assume the top 12 bits of the highest thread are zero
  send32=sums[limbs-1]>>52;
  send32=__shfl(send32, threadIdx.x-1, threads);
  sums[limbs-1] = sums[limbs-1] & mask52;

  // shift upper 12 bits to next limb
  sums[0] += send32;
  for(int index=1;index<limbs;index++) {
    sums[index] += sums[index-1]>>52;
    sums[index-1] = sums[index-1] & mask52;
  }

  testCritical=sums[0];
  for(int index=1;index<limbs;index++)
    testCritical=testCritical & sums[index];

  // Construct the generate and propagate masks.
  // For 1024, 1536, and 2048 bits, no interference between instances is possible
  generate=__ballot(sums[limbs-1]>mask52);
  propagate=__ballot(critical==mask52);

  // if carry in, increment the critical samples
  carry=((generate + generate + propagate) ^ propagate) & lane;
  for(int index=0;index<limbs;index++) {
    nextCarry=carry!=0 && sums[index]==mask52;
    if(carry)
      sums[index]=(sums[index]+1) & mask52;
    carry=nextCarry;
  }
}
```

Figure 7.7. 52-bit sampled carry resolution routine

bits which are added to the next column up. At the end of this iteration, all columns will be less than 2^{52} with the exception of the last column, which will be less than $2^{52} + 2^{12}$. The next step is to resolve the carries across threads. There are two approaches which were described in Section 2.3.1. Here we use a generate/propagate scheme implemented with *__ballot* instructions to resolve all the carry propagation without looping. This a nice approach because it runs in constant time and is therefore resistant to side channel attacks. Further, we note that the top 15 bits in the most significant column sum of the highest thread will always be zero. Therefore, the highest thread can neither be critical nor can it generate a carry out. Thus, we don't require any special handling to prevent carries from propagating from one instance to another within the same warp. All carries and carry chains will be stopped at the highest thread in each instance.

These three routines serve as the implementation of the modular multiplier. The other routines required to implement fixed window exponentiation are straightforward, requiring only data loading, data storing, and some sampling routines that convert a standard 32-bit FRNS to and from a 52-bit sampled representation.

7.4 Experimental Setup and Results

To evaluate the wide samples approach to modular exponentiation, we ran experiments on a GeForce GTX Titan Black card, which uses a Kepler micro-architecture with 15 SMs with a nominal clock rate of 889 MHz and a maximum clock rate of 1280 MHz. The Titan Black card supports high throughput double precision floating point operations. The card is hosted a 64-bit Ubuntu Server (version 16.04 LTS) machine, with an MSI Z270M motherboard and an Intel Core i5-7400 running at 3.0 GHz with 16 GB of main memory. We have installed CUDA version 8.0 and NVIDIA driver version 375.26 and use GMP version 6.1.1 to verify the modular exponentiation results generated on the GPU. For all of our experiments, we use *ndivia-smi* to configure several settings on the driver/card:

Setting	Value
GPU Operating Mode (-gom)	1 (Compute)
Persistence Mode (-pm)	1 (Enabled)
Application Clocks (-ac)	3500,1280

Table 7.2. NVIDIA Driver / GPU card settings

The GPU Operating Mode enables high performance double precision floating point on the card. The Persistence Mode and Application Clocks are to turn off *Turbo Boost* mode and lock the GPU to its highest available memory and compute ALU clock rates of 3500 MHz and 1280 MHz respectively.

To test the performance for each size n, (1024, 1536, or, 2048 bits), we generate the number of random instances in accordance with Table 7.3. Each random instance consists of three randomly generated values A, K, and M, each of which is n bits in length. There

Bits	Threads / Instance	Samples / Thread	Threads / Block	Max Registers	Instances	Blocks
1024	4	5	768	80	2880	15
1536	8	4	768	80	1440	15
2048	8	5	768	80	1440	15

Table 7.3. Parameter that deliver the best performane on 1024, 1536 and 2048 bit modular exponentiation

are no restrictions on A or K, but M must be odd as required for use in Montgomery reductions. The test procedure is straightforward. Generate the random instances on the CPU and for each instance pre-compute two terms that are dependent on M: $R \bmod M$ and $R^2 \bmod M$, where R is $2^{52 \cdot \lceil n/52 \rceil}$. Then copy the five n-bit values for each instance to the GPU. On the GPU we first launch warm up kernels, followed by the timing runs. Once all the timing runs are complete, we copy the results back to the CPU where they are checked for correctness using GMP. For the timing runs, we launch the kernel and measure the execution time. The kernel loads the instance data from GPU global memory computes $A^K \bmod M$ and writes the result back to global memory. The timing runs do not include any of the CPU processing time such as generating the random instance data, the pre-computations, copying the instances to and from the GPU or verifying the result.

The modular exponentiation kernel can be compiled with a number of different parameters, such as threads per instance, samples per thread, threads per block, maximum number of registers to use per thread, etc. For each size n, we test a few of the possibilities and pick the set of parameters that produces the best performance. The best values are summarized in Table 7.3. One result that is surprising is that we achieve the best performance with a single large block per SM rather multiple smaller blocks. With other kernels it's common to see multiply smaller blocks outperform a single large block.

In Table 7.4 we gives the throughput (modular exponentiations per second) and latency of the wide samples approach on a Titan Black card. As can be seen from the table, a single warm up run followed by small number of timing runs gives the best performance. As we

Bits	Warm Up Count	Timing Run Count	Average Throughput	Average Latency
1024	1	4	159.4K	18.0 ms
	1	10	158.4K	18.2 ms
	1	50	137.1K	21.0 ms
	1	100	134.4K	21.4 ms
	100	200	131.8K	21.8 ms
	100	500	131.9K	21.8 ms
1536	1	4	44.0K	32.7 ms
	1	10	40.2K	35.8 ms
	1	50	37.3K	38.7 ms
	1	100	36.9K	39.0 ms
	100	200	36.5K	39.4 ms
	100	500	36.3K	39.6 ms
2048	1	4	18.9K	76.1 ms
	1	10	17.9K	80.6 ms
	1	50	17.6K	82.0 ms
	1	100	17.4K	82.9 ms
	100	200	16.8K	85.5 ms
	100	500	16.9K	85.5 ms

Table 7.4. Performance results for three sizes and different warm up counts and timing run counts

increase the number of timing runs, the performance drops, but then levels off. In the last two rows for each size, we do 100 warm up runs and then either 200 or 500 timing runs. The throughput and latency of these last two rows are similar and represents the steady state for the size. The *nvidia-smi* can be used to monitor the current state of the GPU, including temperature, power consumption, clock rates, etc. What we see is that at the start of a set of runs, the GPU core clock rate is 1280 MHz and the power consumption of the GPU hits the maximum allowed, which is 250 watts. The GPU responds by ratcheting down the core clock rate. At the end of the first few runs it has dropped down to 1030 MHz, and by about the 100th run, the clock rate settles between 862 MHz and 875 MHz and the power consumption hovers at around 245 watts. We can conclude that the kernel is not compute bound or memory bound, it's actually limited by power consumption.

We can compare our results to those of Dong, Zheng, et al. in [33] who used double precision floating point and 22/23 bit limbs on a Titan GPU card. They achieved the equiv-

alent of 84.4K modexp ops/sec at 1024 bits, 24.3K modexp ops/sec at 1536 bits and 11.6K modexp ops/sec at 2048 bits. Comparing to our steady state results, we find we are 56% faster at 1024 bits, 49% faster at 1536 and 45% faster at 2048 bits. The Titan GPU has one less SM than the Titan Black GPU and has a nominal clock rate of 837 MHz vs. 889 MHz. However, since our software is hitting the power cap, it's likely that our results wouldn't change by more than 5%-10% on the older Titan GPU.

The Titan Black card is very similar to the GTX 780Ti and we can compare these results against the integer implementations in Chapter 6. Both GPUs are based on the Kepler micro-architecture, have 2880 cores, and have the same power limit of 250 watts. The main differences are that the Titan Black card supports high performance double precision and has a slightly higher nominal clock rate of 889 MHz vs. 875 MHz. At 1024 bits, the wide samples approach achieves a steady state performance of 131.9K modular exponentiations per second, versus the 127.9K reported in Chapter 6. At 2048 bits the wide samples approach achieves 16.9K ops/sec, versus 10.8K ops/sec reported in Chapter 6. At 1024 bits, the integer implementation is using the Three N model with an instance per thread, which tends to be much more efficient (because it's possible to implement fast squaring and Karatsuba), but at the expense of higher latency and a more complex code base. At 2048 bits where the integer and wide samples implementations are both using a distributed model, we see that the wide samples implementation significantly outperforms the integer one.

7.5 Conclusions and Future Work

The important contribution of this chapter was the combination of three tricks that allowed efficient use of wide samples. The first trick was the injection of 2^{104} and 2^{52} into the floating point computations to solve the normalization/alignment problems. The second was taking advantage of the IEEE 754 representation, whereby the least significant 52 bits of the double precision value exactly matches the integer high and low products,

and performing the column sums with unsigned 64-bit integers. The final trick was to take advantage of the fact that upper 12 bits of a high product is always 0x467 and for a low product is always 0x433. Rather than repeatedly masking off the top bits, we can include them in the column sums and cancels them off with a single subtraction or alternatively by initializing the column sum with the right magic value.

The combination of these tricks allowed our wide samples approach to outperform both narrow samples and integer implementations. However, there is still significant work to be done. We could implement a wider range of models, including the Three N model and Two N plus Local model using wide samples and double precision arithmetic. This would significantly improve the performance at smaller sizes. It would also be interesting to look at more GPU cards, such as Pascal and Volta. According to Table 7.1 the wide samples approach should be faster than an integer approach on Pascal, but slower on Volta. It would be nice to confirm this conjecture experimentally.

In Chapter 6 we built utilization metrics for all of the models. These metrics gave us insight into the number of dispatch cycles that were being wasted. However, in this case, the performance is not limited by wasted dispatch cycles, but rather that we're hitting the power cap for the card. Wasting fewer dispatch cycles would only serve to increase the power requirements of the computation and the card would probably respond by further slowing the clock rate. Thus, the software as it stands is probably close to its peak potential performance.

Finally, we believe that it might be possible to improve the performance of libraries such as QD and CAMPARY using the same tricks explored in this chapter.

CHAPTER 8

CONCLUSION

We return to the hypothesis of this dissertation — an order of magnitude improvement in three important criteria could be achieved by moving multiple precision operations from the CPU to the GPU. The criteria are operations per second per socket, operations per watt, and operations per second per dollar. We test the codes developed in the Chapters 4, 6, and 7 across four generations of GPU and compare these results against two CPUs, a consumer grade Core i5-7400 and a server class Xeon E5-2690v3 with 12 cores running CPU equivalents (GMP and MPFR using multiple threads to saturate the cores). The speedups achieved are presented in Tables 8.1 and 8.2. The raw data for the tables is quite lengthy and can be found in Appendix A.

	Xeon E5-2690 v3	Core i5-7400	GTX Titan Black	GTX 980	Pascal P100	Volta V100
set_{ui,si,f,d}	1	0.3	6.9	5.9	14.2	19.6
sgn	1	0.6	43.9	43.9	52.8	47.7
cmp	1	0.5	10.8	13.4	24.1	39.5
neg	1	0.3	7.5	5.9	15.8	23.9
add	1	0.3	6.3	5.2	13.3	21.4
sub	1	0.3	6.3	5.2	13.3	21.4
mul	1	0.5	2.7	2.6	4.9	15.9
div	1	0.5	1.4	1.3	2.6	8.1[1]
sqrt	1	0.4	1.1	1.0	1.9	6.0[1]
256-bit powm	1	0.4	7.6	7.8	11.6	47.6
512-bit powm	1	0.5	6.2	5.1	9.9	32.9
1024-bit powm	1	0.6	6.1	3.4	15.6	37.0
1536-bit powm	1	0.5	4.7	2.9	3.2	29.8
2048-bit powm	1	0.5	4.9	2.7	14.5	29.8

Table 8.1. Speedup table: Throughput / Xeon E5-2690 Throughput

	Core i5-7400	Xeon E5-2690 v3	GTX Titan Black	GTX 980	Pascal P100	Volta V100
set_{ui,si,f,d}	1	3.2	22.4	18.9	45.2	62.2
sgn	1	1.8	71.4	71.4	91.2	83.8
cmp	1	2.1	20.0	23.7	45.2	81.4
neg	1	3.7	27.8	22.0	58.4	88.4
add	1	3.7	23.5	19.5	49.7	80.0
sub	1	3.7	23.5	19.5	49.7	80.1
mul	1	1.9	5.2	4.9	9.4	30.4
div	1	2.2	3.1	2.9	5.7	15.5[1]
sqrt	1	2.3	3.4	2.2	4.3	11.6[1]
256-bit powm	1	2.4	18.1	18.5	27.4	112.8
512-bit powm	1	1.9	11.7	9.8	18.9	62.6
1024-bit powm	1	1.8	10.7	5.9	32.5	65.6
1536-bit powm	1	2.0	9.4	5.9	26.4	59.6
2048-bit powm	1	2.0	9.8	5.3	29.1	58.4

Table 8.2. Speedup table: Throughput / Core i5-7400 Throughput

Operations per Second per Socket: the Xeon E5-2690v3 is a server class processor with excellent throughput on a per socket basis. From Table 8.1, we see a Volta V100 card has over 20x the performance on most APIs and between 6x and 15x the performance for MP floating point multiplication, division and square root. It's worth noting that the MP floating point library has not been carefully optimized or tuned for the Volta architecture and there are likely significant gains still to come. Intel does offer larger server CPUs, in particular the 22 core E5-2699v4 processor at 2.2 GHz. If we scale the E5-2690v3 by number of cores and clock rate, we estimate the E5-2699v4 should be about 1.6 times faster than the E5-2690v3. Volta would still be at least an order of magnitude faster on all operations except MP floating point division and square root. It's also worth noting that division and square root represent only a small fraction of the floating point operations in a typical application.

[1] Division and square root on Volta are produce incorrect results due to a compiler bug. These table entries represent our estimate for the performance of the current code once the compiler is fixed.

Operations per Watt: first, we note that we do not have the instrumentation to measure the actual power consumption of the GPU or CPU. For this analysis, we will assume that the CPUs and GPUs are all running at their full Thermal Design Power (TDP). Operations per Watt is a tricky metric because performance tends to drop linearly with clock speed, whereas power usage quadratically. Thus low end processors with slow clocks, such as the NVIDIA Jetson SoC cards have the best performance per watt. Here we restrict our comparison to the cards and processors that we have data for. The Xeon E5-2690v3 has better performance per watt than the Core i5-7400. Thus we compare it against the Volta card which has the best performance per watt of the GPUs. The V100 uses 300 watts vs 135 for the Xeon, i.e., 2.2 times the power. APIs that achieve the order of magnitude improvement must have speedup of at least 22 in the last column of Table 8.1. As we can see, most APIs meet this requirement except for the three compute heavy floating point operations. As noted above, there is probably considerable headroom for improvement in these three APIs.

Operations per Second per Dollar: we see from Table 8.2 that a Xeon E5-2690v3 is somewhere between 2 and 3 times the performance of a Core i5-7400. However, as a server chip, it's much more than 3 times the cost. Therefore the consumer grade Core i5 represents a better performance per dollar. If we look at shipping GPUs, there is a GTX 1080 card and we estimate it has 85% of the performance of a P100. The GTX 1080 costs 2.9 times as much as a i5-7400 ($549 vs $189), thus to achieve an order of magnitude improvement in price performance we would need to see greater than 34 (2.9 divided by 0.85) in the 5^{th} column of table 8.2. A few of the APIs achieve this level of performance but not the important compute bound ones. This is largely a result of NVIDIA going with a 16-bit multiplier at the hardware level for the Maxwell and Pascal micro-architectures. In the next few months, NVIDIA is likely to release a consumer grade Volta graphics card and if the performance is proportionate at the same price point, then we will easily meet the order of magnitude improvement in performancce per dollar.

In conclusion, in this thesis and related publications, we have shown that it is possible to obtain significant gains in operations per second per socket, operations per watt, and operations per second per dollar by moving multiple precision operations from the CPU to the GPU and we have made significant strides towards producing libraries that enable multiple precision arithmetic on the GPU, however further work needs to be done to produce production quality libraries for release to a broader audience.

Appendices

APPENDIX A

PERFORMANCE ACROSS A RANGE OF CPUS AND GPUS

In table A.1 through A.6 we present the performance of multiple precision floating point arithmetic implemented with a multithreaded (OpenMP) testing tool based on MPFR for the CPU and our floating point library from Chapter 4, for the GPU. Volta was recently released and there are some compiler issues that are causing the floating point library to return incorrect results. These entries are marked with an X. We have reported the bugs to NVIDIA and they have acknowledged the problem. In tables A.7 through A.9 we present the performance of multiple precision modular exponentiation. For the CPU, we use a multithreaded (OpenMP) testing tool based on GMP's *mpz_powm* function. For the GPU we use the modular exponentiation routines as implemented in Chapters 6 & 7. For the Three N Model, we generate simple kernels with KM=0 and KS=0. For the Wide Samples approach we always use 768 threads per block and 4 threads if the size is 1024, otherwise 8 threads. The Distributed Model does not produce correct results on Volta (which could be because of the forementioned compiler bugs), thus we mark those entries in the table with an X. The Wide Samples approach only supports three sizes: 1024 bits, 1536 bits and 2048 bits. All performance testing is done in accordance with the procedure described in the corresponding chapters.

Core i5-7400 @ 3.00 GHz (4 cores)								
bits:	1024	2048	3072	4096	5120	6144	7168	8192
instances:	1M	500K	500K	500K	100K	100K	100K	100K
set_ui	27.85	21.29	29.82	38.29	10.93	11.02	12.70	14.33
set_si	25.18	21.27	29.68	38.25	9.39	11.05	12.71	14.32
set_float	26.00	21.34	29.83	38.29	9.87	11.09	12.81	14.47
set_double	24.44	20.47	28.44	36.34	8.92	10.52	12.14	13.64
sgn	2.97	1.68	1.67	1.66	0.62	0.62	0.62	0.62
cmp	5.11	2.68	2.80	2.78	0.84	0.84	0.84	0.84
neg	36.77	31.77	44.86	57.91	14.45	17.12	19.73	22.37
add	48.78	41.96	59.07	76.18	19.00	22.35	25.72	29.08
sub	48.82	42.02	59.19	76.17	18.98	22.34	25.73	29.07
mul	50.04	64.82	110.92	189.15	52.64	69.60	87.85	105.71
div	105.15	115.04	203.53	297.24	85.87	111.19	142.66	178.44
sqrt	147.41	123.84	176.59	256.31	66.02	84.44	100.81	123.25

Table A.1. Parallel MPFR on a Core i5-7400 (running time in milliseconds)

Xeon E5-2690v3 @ 2.60 GHz (12 cores/24 hyperthreads)								
bits:	1024	2048	3072	4096	5120	6144	7168	8192
instances:	1M	500K	500K	500K	100K	100K	100K	100K
set_ui	7.40	6.40	8.84	11.29	3.11	3.62	4.13	4.62
set_si	7.42	6.40	8.85	11.29	3.11	3.62	4.12	4.62
set_float	7.51	6.42	8.85	11.29	3.12	3.62	4.12	4.61
set_double	7.32	6.21	8.47	10.79	2.99	3.45	3.92	4.38
sgn	0.94	0.71	0.71	0.72	0.55	0.55	0.55	0.55
cmp	2.98	1.43	1.43	1.43	0.74	0.74	0.74	0.76
neg	9.73	8.47	11.78	15.10	4.12	4.77	5.44	6.12
add	12.75	11.05	15.44	19.82	5.27	6.14	7.01	7.89
sub	12.80	11.05	15.46	19.81	5.27	6.14	7.02	7.89
mul	28.90	34.07	60.26	95.85	26.56	34.76	44.48	53.06
div	42.99	52.61	90.70	138.18	39.35	52.20	66.34	81.65
sqrt	59.28	52.09	76.67	109.77	29.61	38.03	47.82	58.79

Table A.2. Parallel MPFR on a Xeon E5-2690v3 (running time in milliseconds)

GTX Titan Black (Kepler)								
bits:	1024	2048	3072	4096	5120	6144	7168	8192
instances:	1M	500K	500K	500K	100K	100K	100K	100K
set_ui	1.75	1.17	1.22	1.33	0.44	0.47	0.53	0.56
set_si	1.76	1.17	1.23	1.33	0.43	0.47	0.51	0.56
set_float	1.87	1.23	1.28	1.26	0.42	0.46	0.50	0.54
set_double	2.23	1.43	1.46	1.43	0.43	0.45	0.49	0.54
sgn	0.04	0.02	0.02	0.02	0.01	0.01	0.01	0.01
cmp	0.28	0.15	0.14	0.14	0.04	0.04	0.04	0.04
neg	1.61	1.19	1.56	1.99	0.50	0.59	0.69	0.78
add	2.83	1.71	2.44	2.97	0.77	0.93	1.09	1.21
sub	2.85	1.72	2.51	2.96	0.77	0.95	1.06	1.18
mul	6.75	9.95	20.86	35.96	11.08	15.48	21.25	26.96
div	63.00	64.83	75.23	98.67	22.63	29.69	36.40	44.94
sqrt	53.93	77.47	90.78	108.39	26.11	32.99	38.69	45.96

Table A.3. Our FP library on a GTX Titan Black (running time in milliseconds)

GTX 980 (Maxwell)								
bits:	1024	2048	3072	4096	5120	6144	7168	8192
instances:	1M	500K	500K	500K	100K	100K	100K	100K
set_ui	1.79	1.14	1.36	1.69	0.51	0.58	0.64	0.70
set_si	1.72	1.18	1.36	1.69	0.51	0.57	0.64	0.70
set_float	1.90	1.29	1.35	1.51	0.50	0.56	0.62	0.69
set_double	3.44	1.81	1.84	1.82	0.49	0.54	0.60	0.66
sgn	0.04	0.02	0.02	0.02	0.01	0.01	0.01	0.01
cmp	0.27	0.14	0.14	0.14	0.03	0.03	0.03	0.03
neg	1.54	1.35	1.97	2.64	0.67	0.81	0.96	1.10
add	2.69	1.94	2.91	3.88	0.97	1.16	1.36	1.55
sub	2.73	1.94	2.91	3.88	0.97	1.16	1.36	1.55
mul	7.11	11.83	23.82	38.70	11.31	15.80	20.95	27.02
div	64.24	63.72	81.00	102.40	25.41	31.90	38.72	47.83
sqrt	57.91	76.63	95.86	118.65	29.51	35.80	43.05	51.48

Table A.4. Our FP library on a GTX 980 (running time in milliseconds)

P100 (Pascal)								
bits:	1024	2048	3072	4096	5120	6144	7168	8192
instances:	1M	500K	500K	500K	100K	100K	100K	100K
set_ui	1.12	0.99	1.01	1.06	0.15	0.17	0.20	0.22
set_si	1.15	1.00	1.02	1.05	0.15	0.17	0.20	0.22
set_float	1.39	1.12	1.15	1.12	0.16	0.17	0.19	0.21
set_double	1.44	1.13	1.16	1.13	0.16	0.17	0.19	0.21
sgn	0.02	0.02	0.02	0.02	0.01	0.01	0.01	0.01
cmp	0.11	0.06	0.06	0.06	0.02	0.02	0.02	0.02
neg	0.70	0.52	0.73	0.95	0.25	0.30	0.34	0.39
add	1.38	0.85	1.10	1.43	0.36	0.43	0.50	0.57
sub	1.38	0.85	1.11	1.43	0.36	0.43	0.50	0.57
mul	3.72	6.13	12.48	19.88	5.89	8.20	10.86	14.05
div	30.43	30.89	40.68	51.12	12.96	16.17	19.97	24.41
sqrt	29.21	38.66	48.27	59.19	15.00	18.56	22.39	26.95

Table A.5. Our FP library on a P100 card (running time in milliseconds)

V100 (Volta)								
bits:	1024	2048	3072	4096	5120	6144	7168	8192
instances:	1M	500K	500K	500K	100K	100K	100K	100K
set_ui	1.21	0.79	0.79	0.86	0.11	0.13	0.14	0.16
set_si	0.98	0.75	0.78	0.86	0.11	0.13	0.14	0.16
set_float	0.95	0.73	0.76	0.84	0.11	0.12	0.14	0.15
set_double	0.97	0.75	0.76	0.84	0.11	0.12	0.13	0.15
sgn	0.02	0.02	0.02	0.02	0.01	0.01	0.01	0.01
cmp	0.05	0.03	0.03	0.03	0.01	0.01	0.01	0.02
neg	0.42	0.34	0.49	0.64	0.17	0.20	0.23	0.26
add	0.63	0.49	0.70	0.93	0.24	0.29	0.33	0.38
sub	0.64	0.49	0.70	0.93	0.24	0.29	0.33	0.38
mul	1.32	2.01	3.72	6.02	1.74	2.35	3.39	4.07
div	X	X	X	X	X	X	X	X
sqrt	X	X	X	X	X	X	X	X

Table A.6. Our FP library on a V100 card (running time in milliseconds)

CPU modular exponentiation					
bits:	256 bits	512 bits	1024 bits	1536 bits	2048 bits
instances:	10M	1M	100K	100K	100K
Core i5-7400	357.3K	75.3K	12.3K	3.86K	1.72K
Xeon E5-2690v3	847.4K	143.0K	21.8K	7.73K	3.44K

Table A.7. Parallel GMP *mpz_powm* throughput (operations per second)

GPU modular exponentiation (using the Three N and Distributed models)					
card	256 bits	512 bits	1024 bits	1536 bits	2048 bits
GTX Titan Black (Kepler)	6471.7K	882.4K	118.5K	27.0K	10.0K
GTX 980 (Maxwell)	6629.8K	736.2K	73.1K	22.7K	9.2K
P100 (Pascal)	9788.8K	1421.9K	160.6K	44.3K	17.8K
V100 (Volta)	40306.1K	4711.5K	611.9K	X	X

Table A.8. GPU *modexp* throughput (ops/sec) using the code generation approach (see Chapter 6)

GPU modular exponentiation (using Wide Samples)					
card	256 bits	512 bits	1024 bits	1536 bits	2048 bits
GTX Titan Black (Kepler)	X	X	131.9K	36.3K	16.9K
GTX 980 (Maxwell)	X	X	24.2K	6.7K	3.1K
P100 (Pascal)	X	X	339.8K	101.9K	50.0K
V100 (Volta)	X	X	806.4K	230.1K	100.4K

Table A.9. GPU *modexp* throughput (ops/sec) using wide samples (see Chapter 7)

BIBLIOGRAPHY

[1] Antão, S., Bajard, J. C., and Sousa, L. Elliptic curve point multiplication on GPUs. In *Application-Specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on* (Rennes, France, July 2010), IEEE, pp. 192–199.

[2] Artzy, E., Hinds, J. A., and Saal, H. J. A fast division technique for constant divisors. *Communications of the ACM 19*, 2 (1976), 98–101.

[3] Bailey, D., Borwein, P., and Plouffe, S. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation of the American Mathematical Society 66*, 218 (1997), 903–913.

[4] Bailey, D. H. FFTs in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing* (1989), ACM, pp. 234–242.

[5] Barrett, P. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques* (1986), Springer, pp. 311–323.

[6] Beame, P. W., Cook, S. A., and Hoover, H. J. Log depth circuits for division and related problems. *SIAM Journal on Computing 15*, 4 (1986), 994–1003.

[7] Ben-Sasson, E., Hamilis, M., Silberstein, M., and Tromer, E. Fast multiplication in binary fields on GPUs via register cache. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey, June 2016), ACM.

[8] Bernstein, D., and Sorenson, J. Modular exponentiation via the explicit chinese remainder theorem. *Mathematics of Computation 76*, 257 (2007), 443–454.

[9] Bernstein, D. J., Chen, H., Cheng, C., Lange, T., Niederhagen, R., Schwabe, P., and Yang, B. ECC2K-130 on NVIDIA GPUs. In *International Conference on Cryptology in India* (Hyderabad, India, December 2010), vol. 6498 of *LNCS*, Springer, pp. 328–346.

[10] Bernstein, D. J., Chen, H. C., Chen, M. S., Cheng, C. M., Hsiao, C. H., Lange, T., Lin, Z. C., and Yang, B. Y. The billion-mulmod-per-second PC. In *Workshop Record of Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS)* (Lausanne, Switzerland, September 2009), vol. 9, pp. 131–144.

[11] Bernstein, D. J., Chen, T. R., heng, C. M., Lange, T., and Yang, B. Y. Ecm on graphics cards. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2009), Springer, pp. 483–501.

[12] Blelloch, G. E. Prefix sums and their applications. In *Synthesis of parallel algorithms*, J. H. Reif, Ed. Morgan Kaufmann Publishers Inc, 1993, ch. 1, pp. 35–60.

[13] Bodrato, M. and Zanoni, A. What about Toom-Cook matrices optimality? available at http://bodrato.it/papers/whataboutoomcookmatricesoptimality.pdf, 2006.

[14] Bodrato, M. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In *WAIFI 2007 Proceedings* (June 2007), C. Carlet and B. Sunar, Eds., vol. 4547 of *LNCS*, Springer, pp. 116–133.

[15] Bos, J. W. Low-latency elliptic curve scalar multiplication. *International Journal of Parallel Programming 40*, 5 (2012), 532–550.

[16] Brent, R. P., and Zimmermann, P. *Modern Computer Arithmetic*, vol. 18. Cambridge University Press, 2010.

[17] Burnikel, C., and Ziegler, J. Fast recursive division. Tech. Rep. MPI-I-98-1-022, Max-Planck-Institut fuer Informatik, 1998.

[18] Cesari, G., and Maeder, R. Performance analysis of the parallel Karatsuba multiplication algorithm for distributed memory architectures. *Journal of Symbolic Computation 21*, 4 (1996), 467–473.

[19] Char, B., Johnson, J., Saunders, D., and Wack, A. P. Some experiments with parallel bignum arithmetic. *Proceedings of the 1st International Symposium on Parallel Symbolic Computation* (1994), 94–103.

[20] Chiu, A., Davida, G., and Litow, B. Division in logspace-uniform NC. *RAIRO-Theoretical Informatics and Applications 35*, 3 (2001), 259–275.

[21] Chung, J., and Hasan, M. A. Asymmetric squaring formulae. In *18th IEEE Symposium on Computer Arithmetic (ARITH'07)* (2007), IEEE, pp. 113–122.

[22] Cohen, A. E., and Parhi, K. K. GPU accelerated elliptic curve cryptography in GF(2^m). In *2010 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)* (Seattle, USA, August 2010), IEEE, pp. 57–60.

[23] Cole, R., and Vishkin, U. Faster optimal parallel prefix sums and list ranking. *Information and Computation 81*, 3 (1989), 334–352.

[24] Cook, S. *On the minimum computation time of functions*. PhD thesis, Harvard University, Cambridge, MA, 1966.

[25] Cook, S., Dwork, C., and Reischuk, R. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing 15*, 1 (1986), 87–97.

[26] Cooley, J. W., and Tukey, J. W. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation 19*, 90 (1965), 297–301.

[27] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, vol. 6. MIT Press, Cambridge, MA, 2001.

[28] Corporation, NVIDIA. *CUDA C Programming Guide*, version 9.0.176 ed., 2017. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[29] Crandall, R., and Fagin, B. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation 62*, 205 (1994), 305–324.

[30] Davida, G., Litow, B., and Xu, G. Fast arithmetics using Chinese remaindering. *Information Processing Letters 109*, 13 (2009), 660–662.

[31] Dekker, T. J. A floating-point technique for extending the available precision. *Numerische Mathematik 18*, 3 (1971), 224–242.

[32] Diffie, W., and Hellman, M. New directions in cryptography. *IEEE Transactions on Information Theory 22*, 6 (1976), 644–654.

[33] Dong, J., Zheng, F., Pan, W., Lin, J., Jing, J., and Zhao, Y. Utilizing the double-precision floating-point computing power of GPUs for RSA acceleration. *Security and Communication Networks* (September 2017).

[34] Earle, J. G. Latched carry save adder circuit for multipliers, Sept. 5 1967. US Patent 3,340,388.

[35] Emeliyanenko, P. Efficient multiplication of polynomials on graphics hardware. In *International Workshop on Advanced Parallel Processing Technologies* (Rapperswil, Switzerland, August 2009), vol. 5737 of *LNCS*, Springer, pp. 134–149.

[36] Emmart, N., Chen, Y., and Weems, C. C. Computing the smallest eigenvalue of large ill-conditioned Hankel matrices. *Communications in Computational Physics 18*, 01 (2015), 104–124.

[37] Emmart, N., Luitjens, J., Weems, C., and Woolley, C. Optimizing modular multiplication for NVIDIA's Maxwell GPUs. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)* (Santa Clara, USA, July 2016), IEEE, pp. 47–54.

[38] Emmart, N., and Weems, C. High precision integer addition, subtraction and multiplication with a graphics processing unit. *Parallel Processing Letters 20*, 04 (2010), 293–306.

[39] Emmart, N., and Weems, C. High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes. *Parallel Processing Letters 21*, 03 (2011), 359–375.

[40] Emmart, N., and Weems, C. Parallel multiple precision division by a single precision divisor. In *2011 18th International Conference on High Performance Computing (HiPC)* (Bangalore, India, December 2011), IEEE, pp. 1–9.

[41] Emmart, N., and Weems, C. Search-based automatic code generation for multiprecision modular exponentiation on multiple generations of GPU. *Parallel Processing Letters 23*, 04 (2013).

[42] Emmart, N., and Weems, C. Pushing the performance envelope of modular exponentiation across multiple generations of GPUs. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Hyderabad, India, May 2015), IEEE, pp. 166–176.

[43] Emmart, N., and Weems, C. Asymptotic optimality of parallel short division. In *2016 IEEE International Parallel and Distributed Processing Symposium* (Chicago, USA, May 2016), IEEE, pp. 864–872.

[44] Fagin, B. S. Large integer multiplication on hypercubes. *Journal of Parallel and Distributed Computing 14*, 4 (1992), 426–430.

[45] Ferguson, H., Bailey, D., and Arno, S. Analysis of PSLQ, an integer relation finding algorithm. *Mathematics of Computation of the American Mathematical Society 68*, 225 (1999), 351–369.

[46] Fich, F. E. *The complexity of computation on the parallel random access machine*. Department of Computer Science, University of Toronto, 1993.

[47] Fleissner, S. GPU-accelerated Montgomery exponentiation. In *International Conference on Computational Science (ICCS)* (Beijing, China, May 2007), vol. 4487 of *LNCS*, Springer, pp. 213–220.

[48] Fousse, L., Hanrot, G., Lefévre, V., Pélissier, P., and Zimmermann, P. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw. 33*, 2 (June 2007).

[49] Fürer, M. Faster integer multiplication. *SIAM Journal on Computing 39*, 3 (2009), 979–1005.

[50] Gaudry, P., Kruppa, A., and Zimmermann, P. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation* (2007), ACM, pp. 167–174.

[51] Gentry, C. Fully homomorphic encryption using ideal lattices. In *41st ACM Symposium on Theory of Computing (STOC)* (Washington DC, USA, May 2009), vol. 9, ACM, pp. 169–178.

[52] Giorgi, P., Izard, T., and Tisserand, A. Comparison of modular arithmetic algorithms on GPUs. In *ParCo'09: International Conference on Parallel Computing* (Lyon, France, September 2009).

[53] Goldschmidt, R. E. Applications of division by convergence. Master's thesis, Massachusetts Institute of Technology, 1964.

[54] Granlund, Torbjörn, and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 ed., 2012. `http://gmplib.org/`.

[55] Harrison, O., and Waldron, J. Efficient acceleration of asymmetric cryptography on graphics hardware. In *International Conference on Cryptology in Africa* (Gammarth, Tunisia, June 2009), vol. 5580 of *LNCS*, Springer, pp. 350–367.

[56] Harvey, D., van der Hoeven, J., and Lecerf, G. Even faster integer multiplication. *Journal of Complexity 36* (2016), 1–30.

[57] Henry, R., and Goldberg, I. Solving discrete logarithms in smooth-order groups with CUDA. In *Workshop Record of Special-Purpose Hardware for Attacking Cryptographic Systems (SHARCS)* (Washington, DC, USA, March 2012), pp. 101–118.

[58] Hesse, W., Allender, E., and Barrington, D. A. M. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences 65*, 4 (2002), 695–716.

[59] Hida, Y., Li, X. S., and Bailey, D. H. Algorithms for quad-double precision floating point arithmetic. In *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on* (2001), IEEE, pp. 155–162.

[60] Hoffstein, J., Pipher, J., and Silverman, J. H. *An Introduction to Mathematical Cryptography*, vol. 1. Springer, New York, USA, 2008.

[61] Honda, T., ITO, Y., and Nakano, K. GPU-Accelerated bulk execution of multiple-length multiplication with warp-synchronous programming technique. *IEICE TRANSACTIONS on Information and Systems 99*, 12 (2016), 3004–3012.

[62] Jacobsohn, D. H. A combinatoric division algorithm for fixed-integer divisors. *IEEE Transactions on Computers 100*, 6 (1973), 608–610.

[63] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., and Braynard, R. L. Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)* (Rome, Italy, December 2009), ACM, pp. 1–12.

[64] Jang, K., Han, S., Han, S., Moon, S. B., and Park, K. SSLShader: Cheap SSL acceleration with commodity processors. In *8th USENIX Conference on Networked Systems Design and Implementation (NSDI), Proceedings of the* (Boston, USA, March 2011), USENIX Association.

[65] Jebelean, T. An algorithm for exact division. *Journal of Symbolic Computation 15*, 2 (1993), 169–180.

[66] Jebelean, T. Using the parallel Karatsuba algorithm for long integer multiplication and division. In *European Conference on Parallel Processing* (1997), Springer, pp. 1169–1172.

[67] Johnson, D., Menezes, A., and Vanstone, S. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security 1*, 1 (2001), 36–63.

[68] Joldes, M., Muller, J. M., Popescu, V., and Tucker, W. CAMPARY: CUDA multiple precision arithmetic library and applications. In *International Congress on Mathematical Software* (Berlin, Germany, July 2016), Springer, pp. 232–240.

[69] Karatsuba, A., and Ofman, Y. Multiplication of multidigit numbers on automata. In *Soviet Physics Doklady* (1963), vol. 7, p. 595.

[70] Karp, A. H., and Markstein, P. High-precision division and square root. *ACM Transactions on Mathematical Software (TOMS) 23*, 4 (1997), 561–589.

[71] Karp, R. M., and Ramachandran, V. *A survey of parallel algorithms for shared-memory machines*. University of California at Berkeley, Berkeley CA, 1989.

[72] Kawamura, S., Koike, M., Sano, F., and Shimbo, A. Cox-Rower architecture for fast parallel Montgomery multiplication. In *International Conference on the Theory and Application of Cryptographic Techniques* (Bruges, Belgium, May 2000), vol. 1807 of *LNCS*, Springer, pp. 523–538.

[73] Knuth, D. E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3 ed., vol. 2. Addison-Wesley, Reading, MA, USA, 1997.

[74] Koç, Ç. K., Acar, T., and Kaliski, B. S. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro 16*, 3 (1996), 26–33.

[75] Koren, I. *Computer arithmetic algorithms*. Universities Press, 2002.

[76] Krandick, W., and Jebelean, T. Bidirectional exact integer division. *Journal of Symbolic Computation 21*, 4 (1996), 441–455.

[77] Kravitz, D. W. Digital signature algorithm, July 27 1993. US Patent 5,231,668A.

[78] Ladner, R. E., and Fischer, M. J. Parallel prefix computation. *Journal of the ACM (JACM) 27*, 4 (1980), 831–838.

[79] Law, L., Menezes, A., Qu, M., Solinas, J., and Vanstone, S. An efficient protocol for authenticated key agreement. *Designs, Codes and Cryptography 28*, 2 (2003), 119–134.

[80] Leboeuf, K., Muscedere, R., and Ahmadi, M. High performance prime field multiplication for GPU. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)* (Seoul, Korea, May 2012), IEEE, pp. 93–96.

[81] Leboeuf, K., Muscedere, R., and Ahmadi, M. A GPU implementation of the Montgomery multiplication algorithm for elliptic curve cryptography. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS)* (Beijing, China, May 2013), IEEE, pp. 2593–2596.

[82] Lu, M., He, B., and Luo, Q. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware* (Indianapolis, USA, June 2010), ACM, pp. 19–26.

[83] Montgomery, P. L. Modular multiplication without trial division. *Mathematics of Computation 44*, 170 (1985), 519–521.

[84] Moss, A., Page, D., and Smart, N. P. Toward acceleration of RSA using 3D graphics hardware. In *IMA International Conference on Cryptography and Coding* (Cirencester, UK, December 2007), vol. 4887 of *LNCS*, Springer-Verlog, pp. 364–383.

[85] Nakayama, T., and Takahashi, D. Implementation of multiple-precision floating-point arithmetic library for GPU computing. In *Proceedings of the 23rd IASTED International Conference on Parallel and Distributed Computing and Systems* (Dallas, USA, December 2011), ACTA Press, pp. 343–349.

[86] Neves, S., and Araujo, F. On the performance of GPU public-key cryptography. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on* (Santa Monica, USA, September 2011), IEEE, pp. 133–140.

[87] Orup, H. Simplifying quotient determination in high-radix modular multiplication. In *Computer Arithmetic, 1995., Proceedings of the 12th Symposium on* (1995), IEEE, pp. 193–199.

[88] P. B. McLaughlin, Jr. New frameworks for Montgomerys modular multiplication method. *Mathematics of Computation 73*, 246 (2004), 899–906.

[89] Parberry, I., and Yan, P. Improved upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing 20*, 1 (1991), 88–99.

[90] Quisquater, J. J., and Couvreur, C. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters 18*, 21 (1982), 905–907.

[91] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

[92] Robertson, J. E. A new class of digital division methods. *IRE Transactions on Electronic Computers*, 3 (1958), 218–222.

[93] Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2007.

[94] Schönhage, A., and Strassen, V. Schnelle multiplikation grosser zahlen. *Computing 7*, 3-4 (1971), 281–292.

[95] Solinas, J. A. Generalized Mersenne numbers. Tech. Rep. CORR-99-39, Center for Applied Cryptography Research, University of Waterloo, 1999.

[96] Sorenson, J. P. A randomized sublinear time parallel GCD algorithm for the EREW PRAM. *Information Processing Letters 110*, 5 (2010), 198–201.

[97] Srinivasan, P., and Petry, F. E. Constant-division algorithms. *IEE Proceedings-Computers and Digital Techniques 141*, 6 (1994), 334–340.

[98] Svoboda, A. An algorithm for division. *Information Processing Machines 9*, 25-34 (1963), 28.

[99] Szabo, N. S., and Tanaka, R. I. *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967.

[100] Szerwinski, R., and Güneysu, T. Exploiting the power of GPUs for asymmetric cryptography. In *International Workshop on Cryptographic Hardware and Embedded Systems* (Washington, DC, USA, August 2008), vol. 5154 of *LNCS*, Springer, pp. 79–99.

[101] Takahashi, D. A parallel algorithm for multiple-precision division by a single-precision integer. In *International Conference on Large-Scale Scientific Computing* (2007), Springer, pp. 729–736.

[102] Takahashi, D. Parallel implementation of multiple-precision arithmetic and 2,576,980,370,000 decimal digits of π calculation. *Parallel Computing 36*, 8 (2010), 439–448.

[103] Thall, A. Extended-precision floating-point numbers for GPU computation. In *ACM SIGGRAPH 2006 Research Posters* (Boston, USA, July 2006), ACM, p. 52.

[104] Tocher, K. D. Techniques of multiplication and division for automatic binary computers. *The Quarterly Journal of Mechanics and Applied Mathematics 11*, 3 (1958), 364–384.

[105] Toom, A. L. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady* (1963), vol. 4, pp. 714–716.

[106] Volkov, V., and Kazian, B. Fitting fft onto the g80 architecture, 2008. `https://people.eecs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf`.

[107] Walter, C. D. Montgomery exponentiation needs no final subtractions. *Electronics Letters 35*, 21 (1999), 1831–1832.

[108] Weber, K. An experiment in high-precision arithmetic on shared memory multiprocessors. *ACM Special Interest Group on Symbolic and Algebraic Manipulation 24*, 2 (1990), 22–40.

[109] Wegener, I. The critical complexity of all (monotone) Boolean functions and monotone graph properties. In *Fundamentals of Computation Theory* (1985), Springer, pp. 494–502.

[110] Yanik, T., Savas, E., and Koç, Ç. K. Incomplete reduction in modular arithmetic. *IEEE Proceedings-Computers and Digital Techniques 149*, 2 (2002), 46–52.

[111] Zhao, K., and Chu, X. GPUMP: A multiple-precision integer library for GPUs. In *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)* (Bradford, UK, June 2010), IEEE, pp. 1164–1168.

[112] Zheng, F., Pan, W., Lin, J., Jing, J., and Zhao, Y. Exploiting the floating-point computing power of GPUs for RSA. In *International Conference on Information Security* (Hong Kong, China, October 2014), vol. 8783 of *LNCS*, Springer, pp. 198–215.

[113] Zheng, F., Pan, W., Lin, J., Jing, J., and Zhao, Y. Exploiting the potential of GPUs for modular multiplication in ECC. In *International Workshop on Information Security Applications* (Jeju Island, Korea, August 2014), vol. 8909 of *LNCS*, Springer, pp. 295–306.

[114] Zuras, D. More on squaring and multiplying large integers. *IEEE Transactions on Computers 43*, 8 (1994), 899–908.