Elliptic Curve Factoring on the GPU

Saminul Haque

Abstract— **This paper analyzes some variations of the Elliptic Curve Factoring method on the GPU. Different scaling methods as well as different scaling methods on elliptic curves are experimented with. As well, the performance of Montgomery curves and Edwards curves is analyzed as well as various curve generation techniques for them. It is found that Montgomery curves are significantly more effective at finding a nontrivial factor and are faster to run than Edwards curves. It is also found that scaling by two prime factors at a time is preferable to just doing one at a time and that choosing the point scalar to be a least common multiple rather than a simple product is more effective.**

I. Introduction

A. The Problem

The main problem analyzed is the classic problem of factoring a number. That is, given a number N, find the prime factorization of this number. This is usually broken down to a simpler sub-problem, where given a number N, find a nontrivial factor of N. A nontrivial factor is simply one that is not 1 or N. This problem is one of the most important problems in computer science and it still remains difficult to factor large numbers efficiently. In fact, a large portion of online security relies on the fact that factoring is difficult. Thus, a lot of work has gone into finding better and better ways to factor numbers.

This paper focuses on the Elliptic Curve Method for factoring numbers and will discuss an implementation of the algorithm on GPU infrastructure. As well, it will explores a few variations and analyze how the factoring algorithm benefits from the different variations. Previous related works includes [1], where they implement the factoring algorithm on a specific type of elliptic curve.

B. Elliptic Curve Factoring Algorithm

The Elliptic Curve Method (ECM) starts by generating an arbitrary elliptic curve C modulo N and an arbitrary point P that lies on C. There are many different types of elliptic curves that C can be, and here the Montgomery and Edwards forms are analyzed. These different elliptic curves will have different identities and addition operations for their points.

Next, an integer k with many small factors is chosen and then kP (defined as $P + P + \cdots + P$ k-times) will be computed. k is generally chosen by first choosing a small bound B and taking $k = \text{lcm}(1, 2, \cdots, B)$. An alternative method explored will be k equals the product of all primes less than B. Since k is too large to be stored, kP is calculated by taking P and scaling it by each prime factor of k one at a time. A variation of this would be to multiply pairs of prime numbers and scale by each of the products rather than one prime at a time, so as to halve the number of scalings required.

To scale P by a number c, first the binary representation of c is considered. Then from most significant bit of c to least, the point is doubled if the bit is 0. If the bit is 1, it is doubled and the original point is added.

After kP is computed, $\text{gcd}(N,\ (kP)_x)$ is calculated. If it is 1 or N, then this whole process is repeated; otherwise, a nontrivial factor has been found and the method is complete.

Algorithm 1 Compute cP

```
$nP \leftarrow O$ //$O$ is the identity on the curve
**for** $b =$ msb of $c$ to lsb of $c$ **do**
    $nP \leftarrow double(nP)$
    **if** $b = 1$ **then**
        $nP \leftarrow add(nP, P)$
    **end if**
**end for**
**return** $nP$
```

C. GPU Architecture

The Graphical Processing Unit (GPU) is a device that specialized in parallel processing. Originally designed for the purposes of graphical purpose, its parallel computing capabilities are being used more and more recently for general purpose computing.

The system used here is Nvidia's CUDA infrastructure. On CUDA, a host program is run on the CPU, and when it reaches a point to run code on a GPU, called a kernel, the GPU will initialize "threads" and "thread blocks" to the quantity that was specified by the host program. A thread is one instance of the parallel process and it runs through the launched kernel once, while a thread block is a group of threads (whose size and dimensionality is determined by the host program) that run together with shared resources. While the exact numbers may vary between GPUs, one block may manage up to 512 or 1024 threads. The group of blocks together form a grid, which encompasses all threads running the kernel. Within each block, threads are in further subgroup of 32 threads called warps in which each thread must perform the exact same instruction at any given time [5].

An important aspect of GPUs is the memory management. The GPU has many types of memory that have different characteristics and are appropriate for different situations. The following are the most important ones for this project.

1) Global Memory: The global memory is the memory on the GPU that every thread has access to. If there is to be any data transfer to the CPU, it will have to be done through the global memory. Thus, it is the largest memory section on the GPU. As it is accessible to all threads, even those across SMs, it is by far the slowest memory to access [5].

2) Local Memory: Every thread running has local memory, which is its own private memory in the global memory. Since it is on the global memory, it is also very slow. While the global memory endures after a kernel has run, local memory will be destroy once the kernel has finished [5].

3) Register Memory: This is a small amount of memory that resides in the SM. Every thread has its own private portion of it, and this is the fastest memory access that the thread has. Like the local memory, this memory will be destroyed when the kernel finishes [5].

4) Shared Memory: Shared memory is a larger memory in the SM that is divided amongst thread blocks. All threads of a block can access the block's shared memory. This memory is also fast, but not as fast as the register. But like the register, its contents are destroyed when the kernel finishes. This shared memory is a big reason that blocks are important, and will be key to how one organizes their threads and blocks, so as to maximize shared over global memory usage [5].

D. GPU Optimization Techniques

1) Choosing Memory Types: As previously discussed, register memory is the fastest, then comes shared memory, and lastly comes global memory. While the register is ideally used for local information of a thread, there can be a lot of optimization to be done by using shared memory over global memory. If threads need to share which memory to access with other threads, those threads can be organized to be together. That way, instead of both threads accessing global memory for the shared memory accesses, the data can be stored in shared memory [5].

2) Coalescing Memory Accesses: When a unit of global memory is accessed by a thread, not only is the single unit provided by the GPU, but so is a whole contiguous chunk of memory. If threads access memory in a manner where no accessed memory is close to each other, the GPU will have to create chunks for every access call, which is inefficient. If, however, the threads all access memory nearby to each other, the GPU can reuse previously made chunks for other memory. The reorganization of the accessed memory to be contiguous to each other is called coalescing. If the accessed memory appears in regular intervals instead, the memory is called strided [5].

3) Avoiding Bank Conflicts: In shared memory, the memory is stored in groups called chunks. And unlike with coalesced memory accesses, if different memory within the same bank is requested at the same time, they need to processed in serial, which is very inefficient. So to avoid bank conflicts, memory needs to be accessed such that each thread requests memory from different banks or the same memory address within the same bank.

4) Minimizing Code Divergence: Recall that all the threads in a thread warp execute the same instruction. This system is called SIMD, Single Instruction Multiple Data. However, this poses an issue when code branches. A branch is a divergence in code, be it through a conditional statement or loops. When threads in a warp branch out, the warp will go through each branch one by one and any thread not in the current branch of the warp is deactivated. So if there are many branches, there will be a lot of inactive threads, which means lost efficiency. To minimize thread divergence, the code should be rewritten such that continuous groups of 32 threads (the size of the warp) do not have drastically different code paths [5].

II. Experimented Variations of ECM

A. Types of Curves

This variation is of which kind of curve the point will be scaled on. The different curve types have different forms and different laws for adding points. As such, one can expect different behaviour and rates of finding non-trivial factors. This will largely depend on the methods used to generate the curves.

1) Montgomery Curves: Montgomery curves are expressed in the form:

$$By^2z \equiv x(x^2 + Axz + z^2) \pmod{N},$$
$$B \neq 0 \text{ and } A \neq \pm 2$$

The operations on the points turn out to be independent of the point's y coordinate, so only the x and z coordinates will be considered. It follows from this that the B parameter for the curve is not necessary, and only A decides the curve.
For points P, Q, the addition formula for $P + Q = S$ on a Montgomery curve is:

$$S_x = D_z(U + V)^2$$
$$S_z = D_x(U - V)^2$$

for $D = P - Q$, $U = (P_x - P_z)(Q_x + Q_z)$, and $V = (P_x + P_z)(Q_x - Q_z)$.
For a point P, doubling formula for $T = 2P$ on a Montgomery curve is:

$$T_x = (P_x + P_z)^2(P_x - P_z)^2$$
$$T_z = U((P_x - P_z)^2 + CU)$$

for $U = (P_x + P_z)^2 - (P_x - P_z)^2$, and $C = \frac{A+2}{4}$. Notice C is a constant with respect to the point.
As outlined above, these doubling and adding formulas are used to scale a point P by scalar

c. However, notice that addition requires the difference of the two points to add them, so a different algorithm from Algorithm 1 had to be used (Algorithm 2).

In the current implementation 6 Addition/Subtraction operations and 6 Multiplication operations to add two points, while doubling requires 4 Addition/Subtraction operations and 5 Multiplication operations. And as shown in Algorithm 2, both an addition and a doubling is required per bit of the scalar. Thus, every bit of a scalar requires 10 Addition/Subtraction operations and 11 Multiplication operations.

Algorithm 2 Compute cP for Montgomery Curves

```
$nP \leftarrow O$ //Assign the identity
$np1P \leftarrow P$ //Assign the initial point
**for** $b =$ msb of $c$ to lsb of $c$ **do**
    $sum \leftarrow add(nP, np1P)$ //Difference between the points is always $P$
    **if** $b = 0$ **then**
        $np \leftarrow double(nP)$
        $np1p \leftarrow sum$
    **else**
        $np \leftarrow sum$
        $np1p \leftarrow double(np1p)$
    **end if**
**end for**
**return** $nP$
```

2) Edwards Curves: Edwards curves are expressed in the form:

$$x^2 + y^2 = 1 + dx^2y^2,\ d \neq 0, 1$$

To circumvent the issue of fractional values, the inverted coordinate system is used, where the tuples (x, y, z) corresponds to the point with coordinates $(x/z, y/z)$. This makes the equation into:

$$(x^2 + y^2)z^2 = z^4 + dx^2y^2,\ d \neq 0, 1$$

For points $P,\ Q$, the addition formula for $P + Q = S$ on an Edwards curve is [2]:

$$S_x = U(P_z^2Q_z^2 - dP_xQ_xP_yQ_y)(P_xQ_y + P_yQ_x)$$
$$S_y = U(P_z^2Q_z^2 + dP_xQ_xP_yQ_y)(P_yQ_y - P_xP_y)$$
$$S_z = P_z^4Q_z^4 - d^2P_x^2Q_x^2P_y^2Q_y^2$$

for $U = P_zQ_z$.

For a point P, doubling formula for $T = 2P$ on an Edwards curve is [2]:

$$T_x = 2P_xP_y$$
$$T_y = P_x^4 - P_y^4$$
$$T_z = (P_x^2 + P_y^2)(P_x^2 + P_y^2 - 2P_z^2)$$

Unlike Montgomery curves, there are no idiosyncrasies with these formulas, so Algorithm 1 can be used.

In the current implementation, 6 Addition/Subtraction operations and 12 Multiplication operations to add two points, while doubling requires 6 Addition/Subtraction operations and 6 Multiplication operations. And as shown in Algorithm 1, doubling occurs every bit while addition occurs only on a 1. If there is an even distribution of 1's and 0's, then the total number of operations per bit is 9 Addition/Subtraction operations and 12 Multiplication operations. This is similar to, but slightly worse than, the Montgomery operation since multiplication is more expensive than operation.

3) Montgomery Curve Generation: Suyama's Parametrization [6] was used to generate Montgomery curves. In this parametrization, an arbitrary value for σ is chosen that is greater than 5, the following values are computed:

$$\begin{aligned} u &= \sigma^2 - 5 \\ v &= 4\sigma \\ x &= u^3 \\ z &= v^3 \\ A &= \frac{(v-u)^3(3u+v)}{4u^3v} - 2 \end{aligned}$$

where (x, y, z) is the starting point. However, a slight change was made where instead of computing A, C from the addition formula (the only place where the value of A is used) was computed as $C = \frac{(v-u)^3(3u+v)}{16u^3v}$. Notice, this division is modulo N, so it involves calculating $(16u^3v)^{-1} \pmod{N}$.
Using this system, every thread was given a σ based on its thread index relative to the grid and this parametrization was computed, thus giving each thread a unique curve.

4) Edwards Curve Generation: Initially, a crude method to generate curves was used. This method is similar to that used for Weierstrass Curves was experimented with [3]. It involved assigning all but one variables at random and then solving for the remaining variable. Of the variables d, x, y, it was chosen that x and y would be arbitrarily generated since d is the simplest to isolate and an isolation is guaranteed. For x and y however, a modular square root is involved, which is not always possible.
With this system, the threads generate an x and y as a function of the thread index and then isolate for d.

Another method used were parameterizations to generate a point and curve pair that is more likely to generate nontrivial factors [1].
The first one takes a number $\sigma \neq 0, \pm 1$ and then generates the variables as:

$$\begin{aligned} x &= \frac{\sigma^2 - 1}{\sigma^2 + 1} \\ y &= -\frac{(\sigma - 1)^2}{\sigma^2 + 1} \\ d &= \frac{(\sigma^2+1)^3(\sigma^2 - 4\sigma + 1)}{(\sigma - 1)^6(\sigma + 1)^2} \end{aligned}$$

B. Values of k

Recall, k is the value by which the point on the elliptic curve is scaled. This value needs to be highly-composite in order to maximize the chances of finding a nontrivial factor on a curve. To achieve this, a number B is arbitrarily chosen and then k is calculated as either $k_{lcm} = \text{lcm}(1, ..., B)$, which is the standard, or as k_{primes} = product of the primes up to B. Notice that for a given B, both k values have the same prime factors, only that k_{primes} raises them all to the power of 1 while the power varies for k_{lcm}. k_{lcm} is more effective since it has more factors, but for that reason, k_{primes} is computationally quicker.

C. Scaling Values

Since the B value 10,000, k will be very large regardless of which k is used. Thus, instead of scaling by k, the point is scaled successively by the prime factors of k. The "one-prime" scaling method simply scales by one prime factor at a time while the "two-prime" scaling method would scale by the product of two primes. No higher amounts are tested as this would overflow the memory on a 32-bit integer. While the two-prime method does half the scaling operations, each scaling operation is linear on the number of *bits* of the scalar. And the number of bits of a product is very close the sum of the bits of the multiplicands. So while two-primes should be faster, a major time difference is not to be expected.

III. Implementation

A. Montgomery Arithmetic

Since CUDA has no library for arbitrary-precision arithmetic, it was necessary to implement for ECM. All numbers are stored in arrays of 32-bit unsigned integers. 32-bit was chosen instead of 64-bit so that intermediate calculations can be stored in 64-bit integers so as to avoid memory overflows. Since nearly all of the arithmetic in ECM is all modulo N, Montgomery arithmetic was used, which is an arithmetic system designed for faster modular arithmetic.

Montgomery arithmetic first requires the numbers be converted to its Montgomery representation. The Montgomery representation of a number x is simply $xR \pmod{N}$ for some pre-chosen value of R. For computers, the ideal choice of R is 2^{wb} where w is the word size (which is chosen to be 32 bits) and b is the number of words to store N. This is so multiplication and division by R is relatively simple – multiplication is simply left shifts while division is right shifts when there are trailing zeros.

It can easily be seen that addition and subtraction in Montgomery form is the same as the standard addition and subtraction. For multiplication, a more complex algorithm is required. Let $x_R,\ y_R$ by the Montgomery representations of x and y, respectively. Then the method of getting $(xy)R \pmod{N}$ is to compute $x_R y_R R^{-1} \pmod{N}$. This can be done by first computing $x_R y_R$ and then multiplying R^{-1}; however, the CIOS method [4] was chosen to be used (Algorithm 4), which interweaves the R^{-1} computation within the computation of $x_R y_R$.

Another aspect to implement is the conversion of a number x to Montgomery form. Algorithm 3 outlines the method used to accomplish this. However, this is a fairly expensive method to convert a number. To minimize the usage of this method, this method is called twice in the beginning of the algorithm to convert 1 to $R \pmod{N}$ then to $R^2 \pmod{N}$. Then this value is stored and whenever a number is required to be converted, Montgomery multiplication with R^2 is instead performed. That way, the result of the multiplication is $x(R^2)R^{-1} = xR \pmod{N}$, which is the desired output of the conversion process.

Algorithm 3 Convert x to $xR \pmod{N}$

```
//Assumed that $x < N$ and that $R = 2^{wb}$
$newX \leftarrow x$
**for** $i = 1$ to $wb$ **do**
  $newX \leftarrow leftShiftByOne(newX)$
  **if** $newX \geq N$ **then**
    $newX \leftarrow newX - N$
  **end if**
**end for**
**return** $newX$
```

B. Parallelization

Since ECM requires running many curves to see what yields a nontrivial factor, it is natural the ECM is parallelized by running many curves at once. For this, there are a couple of options: either have the calculations for a curve be run on a single thread in serial, or split the load amongst multiple curves. Ultimately, it was decided that only one thread would handle a curve (if one wanted one thread to handle multiple curves, it can be simply achieved by looking the single curve case as many times as desired with different start conditions, i.e. different σ values). This is because splitting a curve amongst threads would ideally only speed up the runtime of the algorithm by a factor of the number of threads running on each curve. Thus, in the ideal case, splitting amongst threads would not increase throughput. So to avoid dealing with any sort of conflicts, each

Algorithm 4 CIOS Montgomery Multiplication to calculate xyR^{-1} (mod N)

```
//$x, y, N$ are all stored in $b$ element arrays of $w$-bit unsigned integers
//$nInv$ is predefined such that $N \cdot nInv \equiv -1$ (mod $2^w$)
//$hi$ and $lo$ return the first and last $w$-bits of a $2w$-bit number, respectively
$p \leftarrow b$-element array of $w$-bit unsigned integers
$overflow \leftarrow 0$
$overflow2 \leftarrow 0$
for $i$ from 0 to $b-1$ do
    $C \leftarrow 0$
    for $j$ from 0 to $b-1$ do
        $temp \leftarrow p[j] + x[j] * y[i] + C$
        $p[j] \leftarrow lo(temp)$
        $C \leftarrow hi(temp)$
    end for
    $temp \leftarrow overflow + C$
    $overflow \leftarrow lo(temp)$
    $overflow2 \leftarrow hi(temp)$
    $C \leftarrow 0$
    $m \leftarrow (p[0] \cdot nInv)$ mod $2^w$
    $C \leftarrow hi(p[0] + m \cdot N[0])$
    for $j$ from 0 to $b-1$ do
        $temp \leftarrow p[j] + m * N[j] + C$
        $p[j-1] \leftarrow lo(temp)$
        $C \leftarrow hi(temp)$
    end for
    $temp \leftarrow overflow + C$
    $p[b-1] \leftarrow lo(temp)$
    $overflow \leftarrow overflow2 + hi(temp)$
end for
return $p$
```

curve is handled by one thread. Furthermore, while addition and subtraction could easily be handled in parallel, the CIOS method implemented for Montgomery multiplication is not easily parallelizable.

Recall that to scale by k, the scaling is done by each prime factor. These prime numbers need to be stored in an array. Instead of all the threads accessing the prime numbers array from global memory, the threads first load the prime numbers array into shared memory and then access the shared memory copy of the array. This gave a small speedup of around 1%.

C. Optimization

The first optimization was improving the memory locations. Initially, the threads stored their memory in the local-global memory, but changing it to the registers improved the speed around 1000 times.

In the implementation of the basic curve and arithmetic operations, there are very few conditional statements, thus minimizing the potential for code divergence. And the potentially divergent while-loops were only used for modular inversions, which are very rarely called. Thus, running all the threads on the same starting conditions versus on different ones showed little to no difference in runtime, so code divergence was not a problem.

Another optimization was using shared memory to store the primes; albeit, the speedup was not too major. However, it was made sure that every thread scale by the same prime at any given moment, so all bank conflicts were avoided. In fact, striding the accesses made a marked negative impact on the runtime.

IV. Analysis

Experimentation was done to determine what was the optimal number of blocks and threads per block to yield the highest curves per second. This experimentation was done a semiprime that is the product of a 10-digit and a

20-digit prime. For ECM, despite the maximum number threads per block in theory is 1024, the memory usage of this implementation only allowed up to 512 threads in a block.

A. Runtime

The runtime data for Montgomery curves (Table I), shows that for a fixed number of total threads, the thread distribution that runs the fastest is that with 256 threads per block. However, this speed-up is minimal; only faster by 6 seconds on 2^{18} threads. The data for Edwards curves shows that instead of fixing the number of threads per block to 256, it is optimal to minimize the number of threads of block. And unlike the minimal difference for Montgomery curves, Edwards curves experience a massive speed-up from this optimization.

Figure 1 displays the optimal runtime for both curves on a given total thread count. It shows how the runtime scales exponentially with $\log_2$ of the number of threads; thus, it is a linear relationship between threads and runtime. However, the ideal is for the runtime to not increase at all sincethe work per thread does not change.

An interesting observation is how the Edwards curves are much slower than Montgomery curves – it takes almost double the time on the optimal configuration for both curves. The previous analysis had predicted that it would in fact run slower, but not to this scale. This means either the threads running Edwards curves encounter some sort of conflicts that they don't with Montgomery curves or the Edwards curves does the addition operation well over half the time.

For the other variations, it was found that k_{lcm} was only marginally slower to compute than k_{primes}, while there was approximately a 5% speed up by using the two-primes method.

B. Efficacy

After analyzing the runtime of ECM and the variations, the efficacy of the variations of ECM were tested. In particular, the rate of curves finding nontrivial relations was measured. Since scaling a point on an elliptic curve is associative, the two-primes variant will not differ from the one-primes variant, so this factor was not measured. Thus, the only variation measured was the different values for k. On the semi-prime used for runtime analysis, there was no difference for Montgomery curves. Both values of k yielded that 2^{12} out of 2^{17} curves found non-trivial factors. However, the Edwards curves had significantly less success. On 2^{18} curves, the curve generation method of isolating d only found a non-trivial factor with 1 curve. The first parameterization from [1] did much better, with 6 curves find a nontrivial factor, but this is still insignificant compared to the Montgomery curves. The second parameterization was the worst, with no curves finding a nontrivial factor.

To further test the efficacy of the different k values, a semi-prime that is the product of two 20-digit primes was analyzed. For k_{primes}, 28 out of 2^{15} curves found a non-trivial factor. On the other hand, for k_{lcm}, 72 out of 2^{15} curves find a non-trivial factor. While both these proportions are low compared to the previous test number, k_{lcm} has a significantly higher chance at finding a non-trivial factor once the prime factors become sufficiently large.

C. Memory Usage

Using the Nvidia profiler, information about the memory usage of the variants was obtained. The scaling variation and the k-value variation do not significantly affect memory usage. For the types of curves, the Montgomery curves took up 80 registers per thread while the Edwards curves required anywhere from 90 to 102 registers per thread. The variance

Number of Blocks	Threads Per Block	Total Threads	Montgomery Curves Runtime (s)	Edwards Curves Runtime (s)
1	64	2^6	0.25	2
64	512	2^{15}	7	51
128	256	2^{15}	6.5	36
256	128	2^{15}	7.5	22
512	64	2^{15}	7	14
1024	32	2^{15}	8.5	11
256	512	2^{17}	27	208
512	256	2^{17}	24	–
1024	128	2^{17}	25	86
4096	32	2^{17}	35	44
512	512	2^{18}	54	–
1024	256	2^{18}	48	–
4096	64	2^{18}	51	108
1024	512	2^{19}	108	–
2048	256	2^{19}	93	–
4096	256	2^{20}	190	–

TABLE I: The Runtime of ECM with Montgomery and Edwards Curves

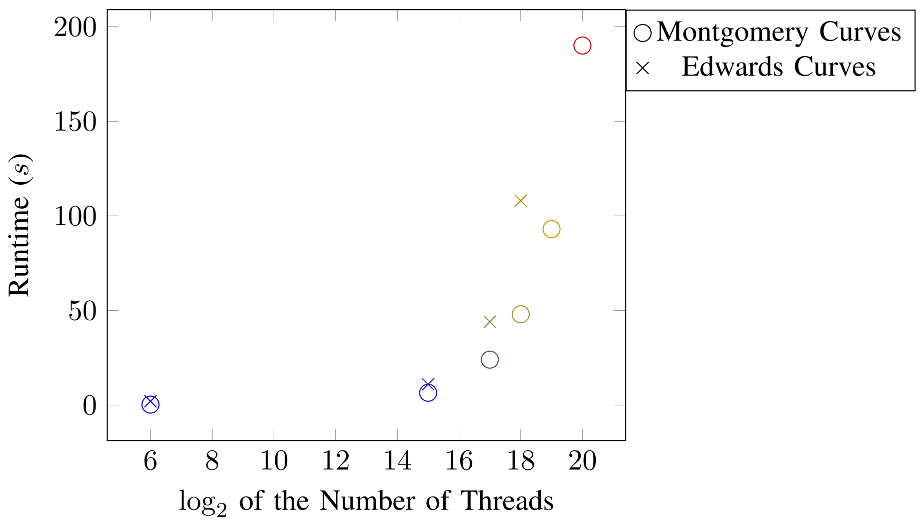

Fig. 1: Runtime of Optimal Thread Distribution for Montgomery and Edwards Curves

with the Edwards curves is due to the different curve generation methods, the isolation method requires 90 registers per thread while the two parameterizations require 96 and 102. This is because the the isolation method only requires modular inversion while the other two require multiple, and modular inversion is the most expensive in terms of time and space.

It is worthwhile to note that the total register space only 256kB. So with 80 registers per thread, where a register is a 32-bit integer, only at most 820 threads can run with these spatial restrictions. However, the tests were regularly being done with over 100,000 threads. It is currently hypothesized that this is the likely reason the runtime is linear with the number of threads. Only small batches of the total number of threads can run concurrently, so when a large number of threads run, it effectively runs in serial. This would be even worse with the Edwards curves, since they require more registers for a thread. This is a potential reason for why the Edwards curves run so much slower than the Montgomery curves, they must run in smaller batches so for a given number of threads, more batches need to run.

V. Conclusion & Future Work

This analysis shows that in this current implementation, Montgomery curves are much superior for elliptic curve factoring than Edwards curves. They run faster and they are significantly more effective at finding factors. Also, it is found that implementing the two-primes variant is worthwhile as there is a significant speedup by doing this. The last major finding is that it is preferable to use k_{lcm} over k_{primes}, because while it takes marginally longer, it is far more effective in the difficult cases for the algorithm.

In the future, it can be explored why the Edwards curves run slower than Montgomery curves and if the register usage can be reduced so as to run more threads in parallel. It would be ideal if the two curves ran at similar runtimes, as their computational complexities are very similar. Other discrepancies between the Edwards and Montgomery curves, such as why the optimal thread distributions between the curves is different, is also worth exploring.

Also, better curve generation techniques, especially for Edwards curves, can be worked on. This can help to try and extend the effective range of ECM (which is currently bounded at around 25 digit prime factors).

References

[1] Bernstein, Daniel J., et al. ”ECM on graphics cards.” Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 2009.

[2] Bernstein, Daniel J., and Tanja Lange. ”Faster addition and doubling on elliptic curves.” Asiacrypt. Vol. 4833. 2007.

[3] Lenstra Jr, Hendrik W. ”Factoring integers with elliptic curves.” Annals of mathematics (1987): 649-673.

[4] McLoone, Maire, Ciaran McIvor, and John V. McCanny. ”Coarsely integrated operand scanning (CIOS) architecture for high-speed Montgomery modular multiplication.” Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on. IEEE, 2004.

[5] Owens, John, and David Luebke. ”Intro to Parallel Programming.” Udacity.com. Udacity, 5 Aug. 2013. Web. 9 August 2017. ¡https://classroom.udacity.com/courses/cs344¿.

[6] Zimmermann, Paul, and Bruce Dodson. ”20 years of ECM.” Lecture Notes in Computer Science 4076 (2006): 525.