Radboud University Nijmegen

Master's Thesis

Solving prime-field ECDLPs on GPUs with OpenCL

Author:
Erik Boss

Supervisor:
dr. Peter Schwabe

Second Reader:
dr. Lejla Batina

August 17, 2015

Abstract

The intractability of the ECDLP is part of what makes many cryptographic application work. As such, viewing this problem from as many angles as possible is worthwhile. In this thesis, we explore the angle of creating a GPU ECDLP solver using OpenCL. In the process, we discuss the many issues, limitations and solutions we encounter. The main result is that, on our testing setup, we can construct a solver that computes a 112-bit ECDLP in 18.5 years but which can also scale up to 114-bit ECDLPs without a relative loss of performance, i.e., we solve a 114-bit ECDLP in approximately 37 years. This nature of the problem is such that, when we scale over many GPUs, we can decrease this number linearly by number of GPUs available.

Acknowledgements

I would like to use this small portion of this thesis to acknowledge the help and support of all the people, tangentially or otherwise, involved in this project. First and foremost, to Peter Schwabe, who supervised this work, for usually knowing exactly what the cause of a given problem was as well as being an overall dedicated and supportive supervisor. Also, to the friends and family who supported me these last few months for always providing a sounding board for my ideas, frustrations and jubilations. In particular, my thanks go out to the people of `#RU`, who more than once provided new ideas and solutions. Lastly, I would like to pay respect to my Lord and Master, the black god known as coffee, for keeping me somewhat alert during the both the less eventful *and* the more stressful times.

Contents

1 Introduction

The intractability of solving discrete logarithms in general, and on elliptic curves specifically, is one of the most important assumptions for practical cryptographic applications today. It keeps us safe. Therefore, we must continuously try to find the limits of what we *can* actually break. This informs our decisions, e. g., as to practical key sizes. Also, this problem needs to be studied from many angles in terms of architecture, tools used, etc. Issues, ideas and solutions that apply to one platform do not necessarily apply to others. Even if they do, they may not apply in the same way. Thus, differentiating the platform allows for more views on the same problem.

The goal of this thesis is simple. Create a solver for ECDLPs (elliptic curve discrete logarithms) for curves over a 116-bit prime $2^{116} - 3$. This allows for some leeway in terms of problem size. Specifically, we are concerned with 112-bit till 114-bit ECDLPs. This solver is intended to run on many GPUs at once. For the purposes of this work, we do not want to bound by any given GPU vendor and thus we use OpenCL [14] to implement this. In the process of doing so, we will learn about the advantages, problems and solutions related to creating such a ECDLP solver in this particular context.

There is a significant amount of related work on other platforms. For example, it is well known how to create such a solver for the Cell processor [4, 5]. Also, recent work, but applied on binary curves, showed good results using FPGAs for 113-bit problems [27]. However, the particular combination of prime-field ECDLPs, GPUs and OpenCL is not something that has been done to the best of our knowledge.

1.1 Structure

This thesis is structured as follows. In chapter 2 we give an overview of all the background necessary for understanding this thesis. Besides some elementary mathematical notation we try not to assume too much in the way of prior knowledge. We do skip over many details hen they are not important to the rest of the thesis. If the reader is well-versed in the domain of elliptic curve cryptography, we advise to skip most of this. It may be beneficial to skim section 2.5, even if only because of the terminology that comes with OpenCL.

In chapter 3 we discuss the results we obtained. Specifically, section 3.2 and section 3.3 detail the results of the implementation phase: what choices and trade-offs occur when implementing an ECDLP solver in a OpenCL context. In section 3.4 we show the performance obtained by our software in both a statistical sense and in absolute performance benchmarks. It contains the primary result, namely that our software should solve 112-bit ECDLPs in approximately 18.5 years but at the same time can linearly scale up to 114-bit ECDLPs. In terms of iterations, this equates to approximately $109.12 \cdot 10^6$ iterations per second.

Finally, in chapter 4 we make some final remarks as to how the results compare to previous work in this area as well as conclude with a view on how to progress from this point.

2 Background

This chapter will provide the background necessary for an understanding of the results and discussions presented in chapter 3. We do not mean for this chapter to be comprehensive, but to be sufficient. It serves to provide intuition and terminology more than anything else.

2.1 Finite Fields

This section is mostly adapted from [19, Ch 3], and as per its definitions, a finite field is considered to be a field with containing a finite number of elements. The number of elements is called the *order* of the field. For this thesis we are only interested in fields of prime order p [1]. We will denote a field of order p as $\mathbb{F}_p$. For these particular prime fields it holds that $\mathbb{F}_p = \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$. We often consider the *multiplicative group* of $\mathbb{F}_p$, denoted as $\mathbb{F}_p^*$, which contains all the non-zero elements of $\mathbb{F}_p$. As such, the order of $\mathbb{F}_p^*$ is $p-1$.

Considering the fact that $\mathbb{F}_p = \mathbb{Z}_p$, for primes p, we know how to do basic arithmetic operations.

Definition 2.1. *Basic operations*: given $a, b \in \mathbb{F}_p$ and for all $\oplus \in \{+, \times, -\}$, it holds that $a \oplus b \equiv c \pmod{p}$.

Exponentiation easily follows from multiplication. Usually, this is performed by *exponentiation by squaring*, which entails repeatedly squaring and multiplying on the basis of the binary representation of the exponent. A more detailed explanation of how to do exponentiation is provided in section 2.4 and subsection 3.2.4, especially as it pertains to exponentiation with a fixed exponent.

Furthermore, we need to define the inverse in $\mathbb{F}_p$.

Definition 2.2. *The multiplicative inverse*: $b \in \mathbb{F}_p$ is the multiplicative inverse of $a \in \mathbb{F}_p$ iff $ab \equiv 1 \pmod{p}$.

Finding the inverse can be done in several ways, e.g., extended Euclidean, but a straightforward way in terms of computation is to compute the $p-2$-th power of a. This finding can be easily derived from the fact that $a^p \equiv p \pmod{p}$ (Fermat's little theorem).

To explain discrete logarithms, we only need the following additional definition of a generator of $\mathbb{F}_p^*$:

[1] For cryptographic purposes binary fields $\mathbb{F}_{2^n}$ are also used, but we will not consider them in this thesis.

Definition 2.3. *The generator*: let $g \in \mathbb{F}_p^*$, g is a generator of $\mathbb{F}_p^*$ if $\{g^i \mid 0 \leq i \leq p-1\} = \mathbb{F}_p^*$.

In other words, repeated multiplication of g yields the entire group $\mathbb{F}_p^*$. The group generated by g is often denoted as $\langle g \rangle$.

2.1.1 Discrete Logarithm Problem

The discrete logarithm problem (DLP) is the basis of some of the most common cryptographic techniques used in practice, the most well-known of which is probably Diffie-Hellman [9].

Definition 2.4. *The discrete logarithm problem*: given $\mathbb{F}_p^*$, a generator g of $\mathbb{F}_p^*$, and $h \in \mathbb{F}_p^*$, find the x, $0 \leq x \leq p-2$, such that $g^x \equiv h \pmod{p}$.

The assumption that this problem is intractable is the reason why it can be used in cryptography.

While this a DLP defined over $\mathbb{F}_p$, in a more general sense, the DLP is defined over finite cyclic groups:

Definition 2.5. *The generalized discrete logarithm problem*: given a finite cyclic group G of order n, a generator g and an element $h \in \langle g \rangle$, find the x, $0 \leq x \leq n-1$, such that $g^x = h$.

We mention this for its similarity to the definition for elliptic curves in section 2.3. As a final note, we will sometimes say that, for these definitions, x is the discrete logarithm of h to the base g.

2.2 Elliptic Curves

The idea of using elliptic curves for cryptographic purposes has been around since the late 1980's [16, 20]. In general, points (x, y) on an elliptic curve can be defined as solutions of the long Weierstrass equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

Additionally, there is a point at infinity $\mathcal{O}$ which acts like the identity element. Mostly we do not concern ourselves with the full equation but we consider solutions to the short form of this equation

$$y^2 = x^3 + a_4x + a_6.$$

For this thesis we are concerned with cases where all a_i, x, y are elements of $\mathbb{F}_p$. In this case, we say that a curve E is defined over $\mathbb{F}_p$.

Given a point $P = (x, y)$, the points generated by repeated point addition of P act like a finite cyclic group. P is called the base point of such a group $G = \langle P \rangle$. The concept of a base point is analogous to the concept of a generator. Similarly, the order of a point P is the number of elements in $\langle P \rangle$.

Point negation in Weierstrass curves is simply $-P = (x, -y)$, making it a very fast operation. Addition in (short) Weierstrass curves of two points P, Q is somewhat more involved and is done as follows:

- If $Q = \mathcal{O}$, then $P + Q = P$;
- If $P = -Q$, then $P + Q = \mathcal{O}$;
- If $P \neq Q$, then $(x_3, y_3) = P + Q$ for $P = (x_1, y_1), Q = (x_2, y_2)$ is computed by:

$$\begin{aligned} x_3 &= \frac{(y_2 - y_2)^2}{(x_2 - x_1)^2} - x_1 - x_2, \\ y_3 &= \frac{((2x_1 + x_2)(y_2 - y_2)}{x_2 - x_1} - \frac{(y_2 - y_1)^3}{x_2 - x_1)^3} - y_1. \end{aligned}$$

- If $P = Q$, then we get a slightly different formula:

$$\begin{aligned} x_3 &= \frac{(3x_1^2 + a_4)^2}{(2y_1)^2} - x_1 - x_1, \\ y_3 &= \frac{((2x_1 + x_1)(3x_1^2 + a_4)}{2y_1} - \frac{(3x_1^2 + a_4)^3}{(2y_1)^3} - y_1. \end{aligned}$$

More simply, the addition and doubling formulates can be computed as follows [4]. For the addition, compute $\lambda = (y_1 - y_2)/(x_1 - x_2)$, then simply compute (x_3, y_3) as

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2, \\ y_3 &= \lambda(x_1 - x_3) - y_1. \end{aligned}$$

For doubling simply compute $\lambda = 3(x_1^2 + a_4)/(2y_1)$ and use the same formulas for x_3, y_3. Note that the a_6 curve parameter does not occur in any of the addition formulas. Therefore, if we write a program implementing point addition for a given curve with a certain a_6, it will also be correct for a different curve with another a_6 (but with the same a_4). In particular, for a different a_6 we may very well get lower-order points, decreasing the security. Normally, this is a problem. For us, however, we can (ab)use it to test our software on lower-order ECDLPs without changing the implementation.

2.2.1 CFRG Curves

The CFRG IETF draft is standard describing an algorithm for deterministically generating high-security curves [17]. It is the standard we use to generate the curve(s) on which we perform all our experiments.

Note that the standard prescribes an algorithm that generates a Montgomery curve of the form

$$y^2 = x^3 + a_2x^2 + x.$$

Whenever we use the algorithm described in this draft, we simply map this curve to a short Weierstrass form, something which is always possible [24]. We do this primarily because it allows us to very easily change the a_6 parameter to get lower-order points, which aids in testing and development.

2.3 ECDLP

The elliptic curve discrete logarithm problem is, of course, very similar to the normal DLP on finite fields.

Definition 2.6. *The elliptic curve discrete logarithm problem*: given a base point P of order l on curve E and a point $Q \in \langle P \rangle$, find the n, $0 \leq x \leq l-1$, such that $nP = Q$.

Note that this definition is essentially identical to the previously mentioned DLPs, except that we write the problem additively instead of multiplicatively.

If the point P is of sufficiently large prime order l and E has no special properties, this problem is believed to be hard [5]. This is a somewhat generalized statement, as there are many ways in which a curve can be insecure, but we need not go into that. Due the intractability of ECDLP, several algorithms that previously relied on the normal DLP have been adapted to use elliptic curves and thus rely on ECDLP. Examples of such are ECDH [18] and ECDSA [13], for elliptic curve versions of Diffie-Hellman and DSA respectively.

2.3.1 Pollard's rho Algorithm

There exist several different ways to compute discrete logarithms but for the purposes of this thesis we will consider Pollard's rho method [25]. This method, or variants thereof, still appear to be the best solution for solving DLPs in generic groups [5].

Generally, Pollard rho is defined as follows [4]: consider a generator P of a finite cyclic group G of order l, $Q \in \langle P \rangle$ and a scalar n such that $nP = Q$. Pollard rho finds n given P, Q by finding a collision in the map

$$(a, b) \mapsto aP + bQ \quad \text{for } a, b \in \mathbb{Z}.$$

Finding such a collision (a, a', b, b') where $aP + bQ = a'P + b'Q$ has a high probability of revealing n by computing

$$n \equiv (a - a')/(b' - b) \pmod{l} \qquad \text{if } \gcd(b - b', l) = 1.$$

To find the aforementioned collision, we define an iteration function f that *walks* over the points in $\langle P \rangle$ such that $W_{i+1} = f(W_i) = a(W_i)P + b(W_i)Q$ where $a, b : \langle P \rangle \mapsto \mathbb{Z}$, starting from an initial combination $W_0 = a_0P + b_0Q$. Now we iterate f until we find

$$\begin{tikzcd}
W_i \arrow[r, "f"] & W_{i+1} \arrow[r, "f"] & W_{i+2} \arrow[r, dashed] & W_j \arrow[dr, bend left, "f"] & \\
 & & W_k \arrow[ur, bend left, "f"] & & W_{j+1} \arrow[dl, bend left, "f"] \\
 & & & W_{j+2} \arrow[ul, dashed, bend left] &
\end{tikzcd}$$

Figure 2.1: Graphical representation of Pollard's rho algorithm. A collision is found if there is a cycle $W_j = W_{k+1}$ for some j, k such that $j \neq k + 1$.

a collision between two distinct points of the walk W_i and W_j, i.e., $W_i = W_j$ and $i \neq j$. Note that if $W_i = W_j$, then also $W_{i+1} = W_{j+1}, W_{i+2} = W_{i+2}, \ldots$, etc. In other words, we enter a *cycle*. See Figure 2.1 for a graphical representation. This cycle can be detected efficiently, and without extraneous memory consumption, by cycle detection methods, the simplest of which is probably Floyd's cycle-finding method [15].

If the aforementioned maps $a(R), b(R)$ are random, the iteration performs a *random walk*. For such a random walk, we know that the average number of iterations required until a collision occurs is approximately $\sqrt{\pi l/2}$, as per the birthday paradox.

Several variations of f are, of course, possible. Note that the iteration function as it is mentioned in the above paragraph requires two scalar multiplications for each iteration. In fact, it is much more convenient to perform an *additive walk*. Let $W_0 = bQ$ be the starting point, $R_i = c_iP + d_iQ$ for $0 \leq i \leq r - 1$ a precomputed table of P, Q-combinations and have $h : \langle P \rangle \mapsto \mathbb{Z}$. The iteration function is now defined as

$$W_{i+1} = W_i + R_{h(W_i)}.$$

To further simplify this, we can set d_j to 0, and just have R be a table of P-multiples. In fact, this is (almost) precisely the type of walk we will use for the iteration function in this thesis.

For small values of r, this type of walk is noticeably non-random [4], which in turn increases the expected number of iterations. If r is large enough, however, the walk approaches a random walk and a similar number of expected iterations as for the random walk applies.

2.3.2 Parallel Pollard rho

Intuitively, the Pollard rho method is trivial to parallelize. In theory, one can just have m different machines start with a different multiple of Q and have each perform the normal Pollard rho algorithm and then we just wait for one to finish, giving us the

$$\begin{tikzcd}
W_i^0 \arrow[r, "f"] & W_{i+1}^0 \arrow[r, dashed] & W_j^0 \arrow[dddr, bend left, "f"] & \\
W_i^1 \arrow[r, "f"] & W_{i+1}^1 \arrow[r, dashed] & W_k^1 & \\
\vdots & & & \\
W_i^c \arrow[r, "f"] & W_{i+1}^c \arrow[rr, dashed] & & (W_j^0 = W_l^c) \arrow[d, "f"] \\
 & & & (W_{j+1}^0 = W_{l+1}^c)
\end{tikzcd}$$

Figure 2.2: Graphical representation of parallel Pollard rho. A collision occurs when two walks converge.

answer we need. However, the expected speedup of such a solution is only $\sqrt{m}$ [5]. As such, this is *not* how parallel Pollard rho is done.

Instead, as proposed in [26] and implemented in most practical ECDLP solvers (like [5, 4]), we build a parallel Pollard rho that achieves a linear speedup, i. e., for m processors we get an m-times speedup.

First, consider the fact that our goal is not to find a cycle per se, but to find a collision. Given this fact, the idea is simply to start m separate walks in parallel, each with their own starting point b_iQ for $0 \leq i \leq m-1$ but with identical precomputation tables R and identical indexing functions h. Instead of doing cycle-detection, we simply send the W_i's of all m walks to a central server. This server then checks all the incoming points for a collision. See Figure 2.2 for a graphical explanation of this method.

Obviously, it would be highly unfeasible to send and store all points of every given walk, especially in light of the large number of iterations required for even a relatively small problem (and thus the large number of points generated). Instead, as per [26], we selectively send points to the server. The selector should be a simple-to-compute function over the point, e.g., call a point *distinguished* if the amount of trailing zeroes in the binary representation of the point is more than some bound. These points are called *distinguished points.* Other examples of such functions include computing leading zeros, hamming weights, and so on. It is helpful if the function can be parameterized in such a way that we can tweak the probability of finding a distinguished point, as is obviously the case for the examples we have given.

Note that once two walks converge, every subsequent point on that walk is also a collision. If we assume that a distinguished point exists in the subsequent points of that walk before it enters a cycle, we will find a collision on said distinguished point. The odds of a walk entering a cycle before we find a distinguished point are slim if we choose the distinguished-point property correctly. Thus, while we introduce a slight delay in collision detection, we reduce the storage and communication overhead significantly, while not missing any collisions.

This way of parallelizing Pollard rho lends itself very well to very distributed setups. A given machine may have a number of walks in parallel, but the setup parallelizes just as well over several machines. The only requirement is that they communicate to the same server.

2.3.3 Negation Map

Consider the case where, in the normal Pollard rho setting, where we choose the maps $a(R)$ and $b(R)$ in f such that $f(W_i) = f(-W_i)$. For such an iteration function f, we essentially halve the number of points to consider. As per our normal birthday-paradox calculation, this leads to an average number of iterations $\sqrt{\pi * l/4}$, a speedup factor of $\sqrt{2}$ [4].

However, as per [4], this type of *negating generic walk* is still not as fast as a normal *additive walk*. Instead, we would like to use a *negating additive walk*. This combines the low number of expected iterations from the negating walk and the performance of the iteration function of the additive walk.

In fact, it is rather easy to construct a function f such that $f(W_i) = f(-W_i)$ in the form of an additive walk. Specifically, we just need a way make a consistent choice $|W_i|$ between W_i and $-W_i$ given just W_i. For instance, we can compute both W_i and $-W_i$, the latter is fast since negation is fast. Subsequently, we simply choose the lexicographic minimum $|W_i| = min(W_i, -W_i)$. Given such an efficient way to choose $|W_i|$, we simply define the iteration function as

$$W_{i+1} = |W_i + R_{h(W_i)}|.$$

2.3.4 Fruitless Cycles

The negating additive walk mentioned above is nice, but has a certain disadvantage: *fruitless cycles* [11]. Consider the situation where $|W_{i+1}| = -Wi + 1$ and $h(W_{i+1}) = h(w_i)$, then

$$W_{i+2} = -W_{i+1} + R_{h(W_i)} = -(|W_i| + R_{h(W_i)}) + R_{h(W_i)} = -|W_i|.$$

If $W_{i+2} = -|W_i|$, then $|W_{i+2}| = |-|W_i|| = |W_i|$. In other words, we enter a cycle of length 2. Similar arguments can be made for the existence of, increasingly rare, cycles of length 4, 6, 8, etc. See [10] for a more thorough discussion.

A fruitless cycle of length $2c$ appear with an approximate probability of $1/r^c$, where r is the size of R [4]. As such, increasing the size of R decreases the probability of such a cycle appearing. However, it turns out that for practical values of r, cycles do appear and do need be considered in any implementation of the negation map.

The literature contains many examples of how to deal with such cycles, some more effective then others. One example, as applied successfully for solving `ECC2K-95` [12], is to use a multiplicative (negating) walk. Here, we map W_i to a small set of scalars S, e.g., $2, 3, 5, 7, 11, 13, 17, 19$ and define the iteration function as

$$W_{i+1} = h(W_i)W_i,$$

where h maps W_i to an element in S.

This has a negligible chance of entering a cycle and is compatible with the negation map [4]. However, for general curves this is still slower than normal additive walks, because a multiplication of a point with a given scalar needs several point additions, whereas an additive walk only needs one. As such, this solution slows the computation down for general curves. Still, the solution in [12] used a Koblitz curve, for which multiplications by certain scalars are as efficient as point addition.

While the above solution avoids cycles, most techniques revolve around detecting and subsequently escaping from cycles. Another, fairly simple, solution has to do with incrementing the index gotten from $h(W_i)$ if $h(W_i) = h(W_{i+1})$ until this condition no longer holds [30]. This prevents 2-cycles, but does nothing for larger cycles. As noted in [4], cycles of lengths greater than 2 are rather rare and thus aborting walks after a number of iterations has been reached will clean up all the cycling walks. However, this technique requires a fairly high distinguished-point probability, which increases the communication and storage overhead.

Another simple idea, due to [11], is to find a cycle of a length c by saving a point W_i every c iterations and comparing W_i with W_{i+j}, where $j \in \{0, 2, \ldots, c\}$. To escape a cycle is then done by applying a "modified iteration" to the lexicographic minimum of all points in a cycle.

The method used in [4] is very much a combination of the ideas in [11] and the idea that doubling is useful to escape cycles. This method entails occasionally modifying the walk to check of cycles of length c in a similar way as [11], while maintaining a record of the minimum W_{min} of all W_{i+j} in the possible cycle. If we encounter a cycle, i.e., $W_i = W_{i+c}$, we modify the iteration to

$$W_{i+1} = |2W_{min}|.$$

2.4 Arithmetic in Representations of $\mathbb{F}_p$

The numbers we use in practical cryptographic computations often exceed the limits of standard datatypes and word sizes. In particular, the 116-bit prime over which we define our curves is much larger than the 64 bits of a `long` type. Therefore, it is useful to consider, in the general sense, how to represent these larger integers and then to define the ways in which we can perform arithmetic operations on such integers. For this section, we consider such operations on big integers.

2.4.1 Representation

Representing an integer $x \in \mathbb{F}_p$, in general terms, is done by splitting x into n *limbs* of radices b^{r_i}, for $0 \leq i \leq n-1$. A given x is then represented in polynomial form with coefficients x_i as

$$x = \sum_{0 \leq i \leq n-1} x_i b^{r_i}.$$

For example, a 128-bit integer can be represented as 2 limbs of 64-bits, i.e., $n = 2, b = 2, r_0 = 0, r_1 = 64$. So for a given 128-bit integer this gives us $x = x_0 2^0 + x_1 2^{64}$.

For the purposes of this thesis, we only consider cases where $b = 2$ and where $r_i = iw$, for some width w. In other words, a w-bit representation with evenly-spaced limbs. Strictly speaking, it is sometimes useful to *not* evenly space the limbs, as is done [4], where the width is 12.8 and $r_i = \lceil i \cdot 12.8 \rceil$. For simplicity, we will not consider such cases here. Also, we assume that, unless otherwise specified, we are dealing with *signed* representations for the limbs.

Often, since the values for n and r_i are fixed, we will often write an integer in this polynomial representation as merely the sequence of the coefficients $(x_0, \ldots, x_{n-1})$.

2.4.2 Addition & Subtraction

Adding or subtracting two integers $x, y \in \mathbb{F}_p$ is rather trivial, as we discussed in section 2.1. It is really no different in their respective polynomial representations. The most simple way of doing addition and subtraction is denoted in algorithms 2.1 and 2.2, as per [19, Ch. 14]. Note that we explicitly apply a simply carry algorithm.

However, consider the circumstance where we have enough space in our datatype to represent both the result and the carry. In that case we can delay computing the carries until we are done with the addition. In that case we can simply do

$$x \oplus y = (x_0 \oplus y_0, \ldots, x_{n-1} \oplus y_{n-1}) \quad \text{for } \oplus \in \{+, -\}.$$

And *then* perform a carry algorithm. In fact, in the case of addition we also want to reduce the $n+1$ limbs back to n limbs. An example of this can be found in subsection 3.2.3.

2.4.3 Multiple-precision Multiplication

Note that multiplication of x, y in their polynomial representations yields twice as many possible exponents and thus limbs, just like in normal schoolbook polynomial multiplication. Specifically, a multiplication between two n-limbed integers yields $2n$ limbs. If we include carries we can compute such a multiplication like in 2.3, as adapted from [19, Ch. 14]. Note that if we exclude carries, we get $2n-1$ limbs.

However, if we can avoid carrying until we are doing multiplying, we like to think of it as this: given $x = (x_0, \ldots, x_{n-1})$, $y = (y_0, \ldots, y_{n-1})$, we first compute the intermediate coefficients $c = (c_0, \ldots, c_{2n-2})$ as

Algorithm 2.1 Base-b addition. Note that we end up with $n+1$ limbs.

```
Input: $x = (x_0, \ldots, x_{n-1})_b, y = (y_0, \ldots, y_{n-1})_b$, for base $b = 2^r$
Output: $z = x + y = (w_0, \ldots, w_n)_b$
 1: $c \leftarrow 0$
 2: for $i \leftarrow 0, n-1$ do
 3:     $z_i \leftarrow (x_i + y_i + c) \mod b$
 4:     if $(x_i + y_i + c) < b$ then
 5:         $c \leftarrow 0$
 6:     else
 7:         $c \leftarrow 1$
 8:     end if
 9: end for
10: $w_n \leftarrow c$
11: return $(w_0, \ldots, w_n)_b$
```

Algorithm 2.2 Base-b subtraction.

```
Input: $x = (x_0, \ldots, x_{n-1})_b, y = (y_0, \ldots, y_{n-1})_b$, for base $b = 2^r$
Output: $z = x - y = (z_0, \ldots, z_{n-1})_b$
 1: $c \leftarrow 0$
 2: for $i \leftarrow 0, n-1$ do
 3:     $z_i \leftarrow (x_i - y_i + c) \mod b$
 4:     if $(x_i - y_i + c) \geq 0$ then
 5:         $c \leftarrow 0$
 6:     else
 7:         $c \leftarrow -1$
 8:     end if
 9: end for
10: return $(z_0, \ldots, z_{n-1})_b$
```

$$c_k = \sum_{i=0}^{k} \sum_{j=k}^{0} x_i y_j \qquad \text{for } 0 \leq k \leq n-1,$$

$$c_k = \sum_{i=k-(n-1)}^{n-1} \sum_{j=n-1}^{k-(n-1)} x_i y_j \qquad \text{for } n \leq k \leq 2n-2.$$

The above holds for general multiple-precision multiplication. However, for the purposes of this thesis we are interested in *modular* multiplication. Note that c has a different polynomial representation from x and y, c has a larger degree. As such, we want to reduce c back to a n-limbed representation. Because we are working in $\mathbb{F}_p$ we have a relatively easy way to do just that.

For this, consider the fact that $2^{nw} > p$. As such, we can “split” 2^{nw} into $r = 2^{nw}$

Algorithm 2.3 Base-b multiple-precision multiplication.

```
Input: $x = (x_0, \ldots, x_{n-1})_b, y = (y_0, \ldots, y_{n-1})_b$, for base $b = 2^r$
Output: $z = xy = (z_0, \ldots, z_{2n-1})_b$
 1: for $i \leftarrow 0, 2n-2$ do
 2:     $z_i \leftarrow 0$
 3: end for
 4: for $i \leftarrow 0, n-1$ do
 5:     $c \leftarrow 0$
 6:     for $j \leftarrow 0, n-1$ do
 7:         $(u, v)_b \leftarrow z_{i+j} + x_j y_i + c$
 8:         $z_{i+j} \leftarrow v$
 9:         $c \leftarrow u$
10:     end for
11:     $z_{i+n} \leftarrow u$
12: end for
13: return $(z_0, \ldots, z_{2n-1})_b$
```

(mod p) and 2^0. Multiplying c_n by r, both of which belong to the 0-th limb, essentially merges c_0 and c_n. The same can be done for c_{n+i} for $1 \leq i \leq n-2$. So the final result $z = (z_0, \ldots, z_{n-1})$ is

$$z = xy = (c_0 + rc_n, c_1 + rc_{n+1}, \ldots, c_{n-2} + rc_{2n-2}, c_{n-1}) .$$

Note that the last limb c_{n-1} is unchanged because there is no other limb to match it with.

2.4.4 Exponentiation

There are a number of ways to perform exponentiation. They generally entail repeated squaring and multiplication, the most common of which is probably *binary exponentiation*, i.e., perform multiplications and/or squarings based on the binary representation of the exponent. The simplest way to do this is by doing either right-to-left (Algorithm 2.4) or left-to-right binary exponentiation (Algorithm 2.5).

Note, however, that we can also process more than one bit at a time. This is basic idea behind the windowing methods of exponentiation. Consider the fixed window, or $k - ary$, exponentiation algorithm (Algorithm 2.6). Notice that we essentially replace some of the multiplications in the inner loop of Algorithm 2.5 with precomputed multiples of g. This technique is particularly useful when we have an exponent that has a nice binary representation such that we do not need as many precomputations.

For cryptography it is not uncommon to have an exponentiation with a fixed exponent. For those, we can often create faster specific exponentiation algorithms. Specifically, we will discuss the concept of an *addition chain*.

An addition chain S of length s for a positive exponent e is a sequence $u_0, u_1, \ldots, u_s$ and an associated sequence $w_1, \ldots, w_s$ of pairs $w_i = (i_1, i_2)$, $0 \leq i_1, i_2 < i$ for which

Algorithm 2.4 Right-to-left binary exponentiation [19].

```
Input: $e > 1, g \in \mathbb{F}_p$
Output: $h = g^e$
 1: $h \leftarrow 1, s \leftarrow g$
 2: while $e \neq 0$ do
 3:     if $e \equiv 1 \pmod 2$ then
 4:         $h \leftarrow h \cdot s$
 5:     end if
 6:     $e \leftarrow \lfloor e/2 \rfloor$
 7:     if $e \neq 0$ then
 8:         $s \leftarrow s \cdot s$
 9:     end if
10: end while
11: return $h$
```

Algorithm 2.5 Left-to-right binary exponentiation [19].

```
Input: $e = (e_l, e_{l-1}, e_0)_2, g \in \mathbb{F}_p$
Output: $h = g^e$
1: $h \leftarrow 1$
2: for $i \leftarrow t, 0$ do
3:     $h \leftarrow h \cdot h$
4:     if $e_i = 1$ then
5:         $h \leftarrow h \cdot g$
6:     end if
7: end for
8: return $h$
```

Algorithm 2.6 Left-to-right k-ary binary exponentiation [19].

```
Input: $g \in \mathbb{F}_p, e = (e_l, e_{l-1}, e_0)_b, b = 2^k$ for $k \geq 1$
Output: $h = g^e$
 1: $g_0 \leftarrow 1$
 2: for $i \leftarrow 1, 2^k - 1$ do
 3:     $g_i \leftarrow g_{i-1} \cdot g$
 4: end for
 5: $h \leftarrow 1$
 6: for $i \leftarrow t, 0$ do
 7:     $h \leftarrow h^{2^k} \cdot h$
 8:     if $e_i = 1$ then
 9:         $h \leftarrow h \cdot g_{e_i}$
10:     end if
11: end for
12: return $h$
```

the following holds [19]:

- $u_0 = 1, u_s = e$
- $u_i = u_{i_1} + u_{i_2}$ for each $u_i, 1 \leq i \leq s$.

In other words, each term in S is the sum of two previous terms. For example, $S = \{1, 2, 3, 6, 12, 24, 30, 31\}$ is an addition chain for $e = 31$, where the indices w_i are $\{(0,0), (0,1), (2,2), (3,3), (4,4), (3,5), (6,0)\}$. Given such a chain, exponentiation is very simple, as can be seen from algorithm 2.7. Finding the smallest of such chains, however, is NP-hard.

Algorithm 2.7 Addition chain exponentiation [19].

```
Input: $g \in \mathbb{F}_p, S = (u_0, u_1, \ldots, u_s), w_1, \ldots, w_s$, where $w_i = (i_1, i_2)$
Output: $g^e$
1: $g_0 \leftarrow g$
2: for $i \leftarrow 1, s$ do
3:     $g_i \leftarrow g_{i_1} \cdot g_{i_2}$
4: end for
5: return $g_s$
```

2.4.5 Inversion

In section 2.1 we mentioned that computing the inverse of x in $\mathbb{F}_p$ is easily achieved by computing $x^{-1} = x^{p-2}$. This is a case of exponentiation with a fixed exponent and is thus very efficiently computed by the method detailed in subsection 2.4.4.

Inversion of some x in $\mathbb{F}_p$ is relatively slow, even if we use the most efficient methods for computing it. However, if we need to compute many different x_i^{-1}, there are some tricks we can apply to make things faster. Specifically, the technique for simultaneous inversion colloquially know as *Montgomery's trick* [22];

Consider the simply fact that for $x, y \in \mathbb{F}_p$, $x^{-1}y^{-1} = (xy)^{-1}$. Therefore, given $(xy)^{-1}$, we can compute x^{-1} and y^{-1} by computing $x^{-1} = (xy)^{-1}y, y^{-1} = (xy)^{-1}x$. For the computation of these two inverses, we replace two inversions with one inversion and three multiplications. Similar arguments hold for any batch of n inversions we may want to do [5]. For batches of 3, for instance, we get $(xyz)^{-1}$ and we first get $z^{-1} = (xyz)^{-1}xy, y^{-1} = (xyz)^{-1}xz$ and $z^{-1} = (xyz)^{-1}yz$. See algorithm 2.8 for the general algorithm for batches of $n \geq 2$. In effect, this trick transforms n inversions into $3(n-1)$ multiplications and 1 inversion.

2.5 GPGPU & OpenCL

GPGPU programming, or general purpose GPU programming, relates to the times where we leverage the parallel processing power of the GPU for speeding up algorithms. If a problem can be parallelized in a SIMD-fashion, solving it on a GPU may be

Algorithm 2.8 Montgomery's simultaneous inversion trick.

```
Input: $z_i \in \mathbb{F}_p$, for $0 \leq i \leq n-1$
Output: $r_i = z_i^{-1}$, for $0 \leq i \leq n-1$
 1: $w_0 \leftarrow 1$
 2: for $i \leftarrow 1, n-1$ do
 3:     $w_i \leftarrow z_i w_{i-1}$
 4: end for
 5: $w_{inv} \leftarrow w_{n-1}^{-1}$
 6: for $i \leftarrow n-1, 1$ do
 7:     $r_i \leftarrow w_{i-1} w_{inv}$
 8:     $w_{inv} \leftarrow z_i w_{inv}$
 9: end for
10: return $(r_0, r_1, \ldots, r_{n-1})$
```

beneficial. Intuitively, SIMD (Single Instruction, Multiple Data) simply means that we execute the same instruction of many different inputs at once. This is what a GPU excels at.

OpenCL is the first open standard for doing such parallel programming on a wide variety of devices, including GPUs [14]. Its primary competitor is NVIDIA's CUDA [23] platform. As we mention in the introduction, however, we are looking for a solution that is independent of the GPU vendor and as such OpenCL is a natural choice.

Let us introduce the relevant terminology. In OpenCL we have the notion of a *compute kernel*, which is a small unit of execution that can be executed in parallel on a multiple independent input streams [8]. These kernels are written in a C-derivative[2] and are compiled separately from the rest of the program. To execute a kernel, we *enqueue* it to the device. The device has several compute units, each with their own memory and processing capabilities.

A given instance of a kernel executing on a compute unit is called a *work item* and multiple work items are bundled in *work groups*. All work items in the same group execute on the same compute unit. In OpenCL, we configure the total number of work items as well as the size of an individual work group. In effect, every work item has its own input stream. Furthermore, we have the notion of a *wavefront*, which the smallest scheduling unit. It consists of of a number of work items which execute their instructions in lockstep. Specifically, we have 32 work items per wavefront for NVIDIA GPUs and 64 work items is typical for recent AMD GPUs. For each work group, the GPU spawns the required number of wavefronts for each compute unit, which are then scheduled.

If two work items within a wavefront execute separate branches within the code, we get diverging instructions that need to be executed. In practice, this will likely mean we execute both branches on every work item all the time and thus is a massive drag on performance. There are always ways to work around branches, however. Best practices

[2]OpenCL 2.0+ is more like C++, but our GPUs support only up to 1.1/1.2.

in most of the vendor guides [7] indicate that branching, if possible, should be avoided. Note that if a given wavefront always executes the same branch, this problem should not appear.

OpenCL's memory layout entails three (or four, depending on your definition) types of memory.

- Private memory is the fastest memory. It is local to each work-item and consists of the scalar and vector registers. This fast memory is rather scarce and often limits the amount of work items which can run on a given compute unit at the same time.

- Local memory is memory shared within a work group. As such, it is commonly used for values that we need to share between work items of a given group. Also, the contents are cached.

- Global memory is memory accessible to every work item. It is located the furthest from the actual computation units and is the slowest. There is, however, a lot of space for global memory on your typical GPU. Also, it is the only way to get data to-and-from the host program. As such, they often act as input and output buffers. Due to its performance characteristics, it often best to avoid making more use of global memory than strictly necessary.

3 Results

This chapter serves as a collection of all results, conclusions and theories that this thesis has spawned. In particular, we first discuss the general architecture of the ECDLP solver that we created. Subsequently, we follow with a discussion on the inner workings of, in succession, the arithmetic operations and the iteration kernel in general.

During this chapter we will sometimes refer to our testing setup, especially in cases where the is some uncertainty as to the cause of certain findings. Unless otherwise specified we are speaking of running our software, often on a smaller problem and on maybe a limited number of iterations, on the following machine:

Operating System Arch Linux (up-to-date as of 2015-08-04)

GPU AMD Radeon HD7850

CPU Intel Core i5 4670K @ 3.4Ghz

RAM 8GB

Compiler clang 3.6.1

In particular, this means that most of the results related to OpenCL are obtained when using the AMD OpenCL compiler. However, at a relatively late stage of the research we also got access to a machine with an NVIDIA GPU.

Operating System Ubuntu 14.04 (LTS)

GPU NVIDIA GeForce GTX 780 (2x, but we only use one)

CPU AMD FX 8350 @ 4.0 GHz

RAM 32GB

Compiler clang 3.4.2

For generating the ECDLPs, we use the CFRG method described in subsection 2.2.1. We used the prime $p = 2^{116} - 3$ to generate a nice Montgomery curve. For cofactor 8 we got the following curve

$$y^2 = x^3 + 89546x^2 + x.$$

Which gives us base points of 113-bit order.

Mapping this to (short) Weierstrass form gives us

$$y^2 = x^3 + 83076749736557242056484477281520717x + 2481488722968349824.$$

Which is a fairly unwieldy value for a_4. However, this form does allow us to change the a_6 to generate base points of lower order, as we will see in subsection 3.4.1. Also, not unimportant, we know how to do efficient arithmetic using such short Weierstrass forms.

3.1 General Architecture

In this section, we will put together the overall picture of how we built our ECDLP solver. More detailed descriptions follow later.

Our software essentially consists of the following components:

- A program that builds a size-r table R of random P-multiples such that $R_i = a_i P$ for $0 \leq i \leq r-1$ and a random $2 \leq a_i l - 1$ where l is the order of P;
- A program that continuously outputs random starting points $W_0 = bQ$;
- A program that takes starting points and outputs distinguished points by computing succesive iterations of said starting points;
- A program that stores dinstinguished points and outputs a collision whenever such is found;
- A program that, given such a collision, computes the answer to the ECDLP.

As to how they fit together, see Figure 3.1. Note that the architecture works for however many clients are needed, distributed over however many different machines on various different networks. The use of OpenCL makes sure that clients can have different hardware and everything will still work.

As an aside, we say that we output random points for table and point generation. However, in practice it is useful to have predictable points that are randomly distributed, especially for purposes of testing and development.

As such, here is how we generate multiples of P and Q, for table and point generation respectively: to generate the i-th point that is a multiple of P, take the first 128 bits of `sha1`(i) as a seed. Create a scalar n where we compute $n' = \texttt{AES}_{\text{seed}}(ptext)$, i.e., encrypt under the seed a given 128-bit plaintext. Compute n from n' by taking the first l bits from n' where l is the bit-order of P. This has the additional advantage of being able to just change the plaintext for each different machine in order to get different points for each machine. Note that there will be identical seeds over the course of a long ECDLP computation, but the collision detection software should not consider two identical seeds as a valid collision. In fact, it should only store one of them.

To reduce storage overhead, the actual output of the iteration program does not include the scalars a, b that make up any given $W_i = aP + bQ$. Instead, as per [1], we

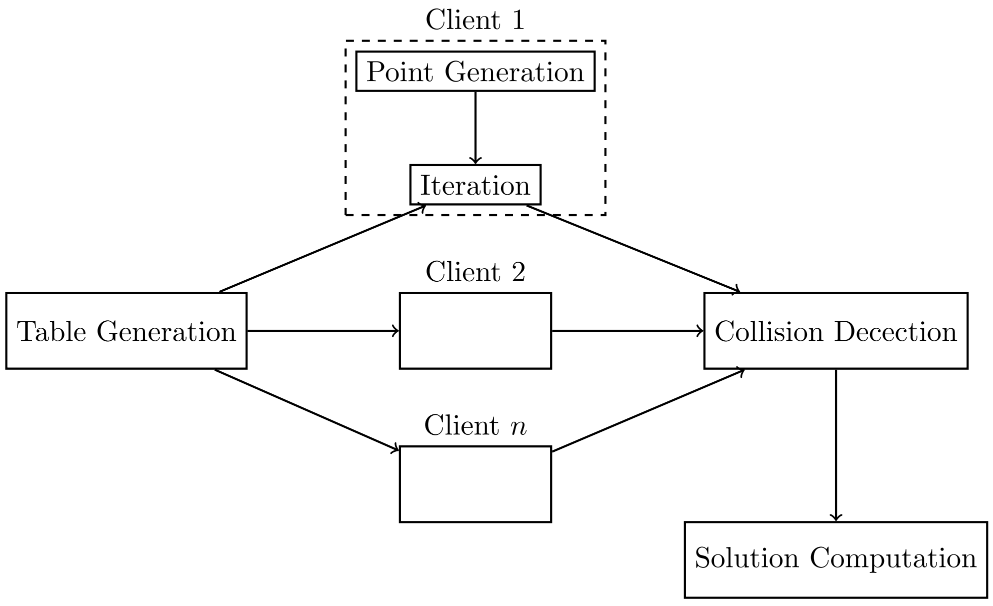

Figure 3.1: General architecture. Every client receives the same table but generates different points. There can be as many different clients as needed.

simply store the seed for a given index i and output this alongside any distinguished point we find at i. Using the aforementioned procedure, we can simply use the seed to recompute the initial scalars used for the starting point of this particular walk.

To recompute the solution from a collision then requires us to redo these two walks from their starting points while we do, in fact, keep track of the scalars. In fact, for this we implemented a separate program that implements the same iteration semantics but on a CPU. This program is especially fast at executing a single walk until we find the collision point. This program *does* keep track of the scalars and it is the scalars this program outputs for both of the collision points that we use for computing our final solution.

3.1.1 Iteration Design Considerations

In terms of performance, only the iteration program is really important. This is where you can optimize and really impact the running time. As such, we will discuss the design of this component in more detail. While we will discuss the specifics of implementing the actual iteration function in an OpenCL kernel in section 3.3, we still need to consider the overall design of the host program.

This particular component receives a sufficiently large list of starting points W_0 and a precomputed table of P-multiples R. The size of the list of starting points be large enough that we fully utilize the GPU. Given these inputs, the device executes the OpenCL kernel implementing the iteration function and outputs the found distinguished points to some other component for collision detection. Most of the boilerplate for setting up the inputs and the kernel execution is fairly straightforward. However, the way we actually enqueue these iteration kernels is important for performance considerations. In particular, we are concerned with the amount of successive iterations per kernel, the amount of successive kernels and the amount of kernels needed for a single iteration. In more detail:

- Having multiple kernels per iteration entails splitting computation in separate computation units. This has a particular advantage in the sense that every one of these kernels get relatively more resources, especially in terms of vector registers (VGPRs). Also, it allows for different work dimensions per kernel. Concluding, splitting a kernel can have a positive effect on VGPR usage and register spilling. However, it precludes the ability to have a single kernel execution compute successive iterations.

- On the other hand, having successive iterations in a single kernel decreases overhead significantly at the cost of significantly less resources per work item/work group. Furthermore, performing successive iterations within a kernel allows for less global memory writes since we can delay writing results to global memory until all iterations are done. In theory, the disadvantage of having less resources should not even exist because we do not, in theory, need significantly more registers to perform more than one iteration in sequence. In practice, the relevant compilers do allocate more resources in these situations and as such, do have this disadvantage. As mentioned, this precludes having multiple kernels per execution.

- Enqueueing multiple kernels in succession can be combined with either of the other two options. It reduces processing overhead due to the fact that we only process the results on the host-side until all kernels are finished executing. However, it does not reduce the amount of global memory operations because for a second kernel to use the first kernel's results, these results need to be written to global memory.

For us, it turned out that for the standard iteration kernel, the second option is fastest *if* we combine it with methods for reducing VGPR usage. As we will see later, we do get some minor register issues when we choose this option and without these VGPR reductions it would not have been feasible to do so.

3.2 Implementing Arithmetic in $\mathbb{F}_p$

In this section we discuss the way we have implemented the relevant arithmetic operations in the ECDLP software. In order to have efficient point addition, we first need efficient arithmetic in general.

3.2.1 Representation

For arithmetic in $\mathbb{F}_p$ where $p = 2^{116} - 3$, we must decide on a representation to use. Representing an element x in $\mathbb{F}_p$ needs 116 bits. We choose to represent x using the following polynomial 4-limbed, base 2^{29} representation:

$$\sum_{0 \le i \le 3} x_i \cdot 2^{29i}$$

Every limb x_i is stored in a *signed* 32-bit integer, yielding 128 bits per element. We could also have chosen a different representation. For instance, the unevenly spaced 10-limbed base $2^{\lceil 11.6i \rceil}$ representation will also work. This has an advantage in the sense that additions and multiplications are done on 16-bit and 32-bit integers respectively, which is faster than the 32-bit and 64-bit operations of the base 2^{29} representation. However, note that a 10-limbed representation needs more operations. Initial experiments showed that the 4-limbed base 2^{29} representation *is* faster, at least on the AMD setup. We theorize that this may be, at least in part, due to the fact that our GPU natively has 256 32-bit registers [7], and thus the 128-bits of the 4-limbed representation fits more neatly than the 160-bits of the 10-limbed representation.

Note that this representation has a number of redundant bits. Only 29 of the 32 bits are strictly speaking necessary for representing such points. This extra space allows us to delay the carrying operations, which simplifies the computations tremendously. Addition and subtraction then become simple component-wise addition (or subtraction) of two vectors.

In OpenCL-specific terms, this means we can store each 116-bit integer in a `int4` datatype. Addition (without carrying) can then be denoted simply as:

```
int4 x = ...;
int4 y = ...;
int4 z = x + y;
```

Subtraction is similar. We discuss the carry operations in subsection 3.2.3.

3.2.2 Multiplication

Multiplication is just a simple application of the technique mentioned in subsection 2.4.3. To make it less abstract, though, we can write out all the computations we need to do. Note that, once again, we delay the carry step until it is needed. Note that every limb is 32-bit and as such the result of a multiplication needs to be stored in a 64-bit integer. Specifically, and this is expanded upon in subsection 3.2.3, a 64-bit signed integer allows for the storage of any given limb calculation if every limb $x_i < 2^{29}$. In OpenCL, this means we need a `long4` datatype for all the intermediate results during multiplication.

We described in subsection 2.4.3 how to compute multiplications generally. Following that, let us first compute the $(2n-1)$-limbed intermediate representation c as

$$\begin{aligned}
c_0 &= x_0y_0,\\
c_1 &= x_0y_1 + x_1y_0,\\
c_2 &= x_0y_2 + x_1y_1 + x_2y_0,\\
c_3 &= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0,\\
c_4 &= x_1y_3 + x_2y_2 + x_3y_1,\\
c_5 &= x_2y_3 + x_3y_2,\\
c_6 &= x_3y_3.
\end{aligned}$$

Subsequently, we reduce to the n-limbed representation. To reiterate subsection 2.4.3, we use the fact that $2^{116} \equiv 3 \pmod p$ to merge the i-th and the $n+i$-th limb, resulting in

$$\begin{aligned}
r_0 &= c_0 + 3c_4 = x_0y_0 + 3(x_1y_3 + x_2y_2 + x_3y_1),\\
r_1 &= c_1 + 3c_5 = x_0y_1 + x_1y_0 + 3(x_2y_3 + x_3y_2),\\
r_2 &= c_2 + 3c_6 = x_0y_2 + x_1y_1 + x_2y_0 + 3(x_3y_3),\\
r_3 &= c_3 = x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0.
\end{aligned}$$

Note that the resulting integer r consists of 4 64-bit limbs, unlike the normal 4-limbed 32-bit representation. So, for actual storage, we need to apply an algorithm to get back to a 4-limbed base 2^{29} representation. This is where the carry step comes in.

Squaring

Squaring in $\mathbb{F}_p$ is slightly easier than a normal multiplication. For optimization, we can use the above $r = (r_0, \dots, r_3)$ calculation and subsequently reuse some of the intermediate results. For example, note that in the calculation of r_0 there exists a computation $x_1y_3 + x_3y_1$, which, if $x_1 = y_1, x_3 = y_3$, simply equates to $2x_1x_3$. Similar considerations apply to the other limbs as well.

3.2.3 Carrying

Consider the fact that the resulting limbs after multiplication should fit in a 64-bit (signed) integer. We can find a reasonable upper bound x_i^{max} on the limb's values after multiplication by filling in the maximum value for a given limb, i.e. $2^{29} - 1$, which gives us

$$x_i^{max} = 4 \cdot 3 \cdot (2^{29} - 1)^2 = 3458764513820540928 = 0.75 \cdot 2^{62}.$$

In this formula, the 4 arises from the 4 multiplications per limb computation and the 3 from the (possible) multiplication with 3 of some of the results. We could specify a more strict bound, but this is sufficient. Note that the result easily fits into a 64-bit (signed) integer. Similarly, we can also define a lower bound, but we omit this for now.

In a similar fashion, we can compute a maximum reasonable factor s for which $x_i^{max} \leq 2^{63} - 1$ as

$$s = \sqrt{(2^{63} - 1)/4/3}/(2^{29} - 1) \approx 1.63.$$

So, if we can reduce and carry properly, we can use a given $x = (x_0, x_1, x_2, x_3)$ in a multiplication if every $x_i < 1.63 \cdot (2^{29} - 1)$. Note that due to this fact, we cannot actually simply use the addition of arbitrary $z = x + y$ in a multiplication in this particular 4-limbed representation. This is due to the fact that this addition can be worst-case represented by the case $s = 2$. Therefore, we get

$$z_i^{max} = 4 \cdot 3 \cdot ((2^{29} - 1) \cdot 2)^2 \approx 1.5 \cdot 2^{63}.$$

Which will, in fact, fit into an unsigned 64-bit integer; but not into a signed 64-bit integer (which is what we actually use). However, this has very little impact on the actual computation, since this only occurs when computing a point doubling, which is something we do relatively rarely. In this rare case, we simply perform the carry algorithm after an addition to fix the problem.

In order for the result of any multiplication to be used in another multiplication we need to reduce the 64-bit limbs back to 32-bit limbs, where every limb is sufficiently small, i.e., every limb $x_i < 1.63 \cdot 2^{29}$.

Consider $x_{max} = (x_0^{max}, \ldots, x_3^{max})$. To reduce this back to a nice representation, as per [4], we perform a sequence of *carry* operations. To carry from x_i to x_j we compute the following step

$$c = x_i \gg 29; x_i = x_i - (c \ll 29); x_j = x_j + c.$$

Except for the carry step from x_3 to x_0, which is

$$c = x_3 \gg 29; x_3 = x_3 - (c \ll 29); x_0 = x_0 + 3c.$$

The 3 in $3c$ is due to the fact that $2^{116} \equiv 3 \pmod{2^{116} - 3}$, similar to the reduction from $2n-1$ limbs to n limbs in subsection 2.4.3. Note that the carry value c for a x_3 is, in theory, the value to be added to a virtual extra limb x_4. However, we only have four limbs and thus we apply the same technique to merge x_4 and x_0, i.e., multiplication by 3.

In order to find out how long this sequence of carry operations needs to be, we trace the bounds of every limb after every operation. When the bounds on every limb are such that we can use the result in another multiplication, we stop.

For simplicity, consider the initial state x after multiplication where each limb $x_i \in \left[-2^{62}, 2^{62}\right]$. We can do this because $x_i^{max} < 2^{62}$ and as such, the length of the carry chain should be at least as long as the case where $x_i = x_i^{max}$. For similar reasons, we simply write 2^{29} instead of the more precise $2^{29} - 1$. Now we trace the bounds after successive carry operations:

$$\begin{aligned}
x_0 \to x_1 : &\quad x_0 \leftarrow \left[-2^{29}, 2^{29}\right], x_1 \leftarrow \left[-2^{62} - 2^{33}, 2^{62} + 2^{33}\right], \\
x_1 \to x_2 : &\quad x_1 \leftarrow \left[-2^{29}, 2^{29}\right], x_2 \leftarrow \left[-2^{62} - 2^{34}, 2^{62} + 2^{34}\right], \\
x_2 \to x_3 : &\quad x_2 \leftarrow \left[-2^{29}, 2^{29}\right], x_3 \leftarrow \left[-2^{62} - 2^{34}, 2^{62} + 2^{34}\right], \\
x_3 \to x_0 : &\quad x_3 \leftarrow \left[-2^{29}, 2^{29}\right], x_0 \leftarrow \left[-2^{29} - 2^{36}, 2^{29} + 2^{36}\right], \\
x_0 \to x_1 : &\quad x_0 \leftarrow \left[-2^{29}, 2^{29}\right], x_1 \leftarrow \left[-2^{29} - 2^{8}, 2^{29} + 2^{8}\right].
\end{aligned}$$

Note that, if we apply apply the carry operation to a 62-bit integer, we essentially split the integer into 29-bit and 33-bit parts. The latter 33-bit integer is added to the next limb in the carry chain. This causes the 2^{33} in the first part of the trace. Note that, for the second and third parts of the chain, we split into a 29-bit and a *34-bit integer*. For the fourth carry operation, we have a multiplication by 3 of the 34-bit integer, which gives us a 36-bit integer. Hence the 2^{36}.

After the last step, x_0, x_2 and x_3 are all 2^{29}, whereas x_1 is slightly larger, i.e., $2^{29}+2^8$. Even so, this is good enough for our purposes since

$$4 \cdot 3 \cdot (2^{29} + 2^8)^2 \approx 0.75 \cdot 2^{62}$$

is smaller than 2^{62}, which is the starting point of the above carry chain. Therefore, this carry chain is long enough to successfully reduce the result of a multiplication in such a way that we can use the result in yet another multiplication.

3.2.4 Inversion

A point addition (or doubling) entails, as part of the computation, one inversion in $\mathbb{F}_p$. As we mentioned in subsection 2.4.5, we can simply compute $x^{-1} \equiv x^{p-2} \pmod{p}$ to find an inverse.

Consider the binary expansion of $p-2 = 2^{116}-5$, it consists of a sequence of $1^{113}011$. That is, 113 1's follows by a 0 followed by two 1's. A slightly naive implementation using simple exponentiation-by-squaring (2.5) will solve this easily by doing the following: given $x \in \mathbb{F}_p$ and $h = 1$,

1. Compute 113 times $h = h^2 \cdot x$ (113 squarings, 113 multiplications);
2. Compute one simple squaring $h = h^2$ (1 squaring);
3. Compute $h = h^2 \cdot x$ twice more (2 squarings, 2 multiplications).

This has a total cost of 116 squarings and 115 multiplications. We can do much better. In particular, as per [19, Ch. 14], the minimum length s addition chain for $p-2$ is bounded by $s \geq \log(p) - \log(HW(p) - 2.13) \approx 120$. We know from 2.4.4 that exponentiation for an addition chain of length s costs s multiplications.

Finding the shortest such chain is hard, but we can rather easily do better than the above $116 + 115 \approx 231$ multiplications. Consider a k-ary exponentiation for $k = 5$. This represents $p - 2$ as $(1)(31)^{22}(27)$. In other words, we can compute $h = x^{p-2}$ given $x \in \mathbb{F}_p$ by:

1. Set $h = x$, this handles the first (1);
2. Compute $h = h^{2^5} \cdot x^{31}$ 22 times, for handling the $(31)^{22}$;
3. Compute $h = h^{2^5} \cdot x^{27}$ once, for the last (27).

Note that this algorithm, as written here, takes $23 \cdot 5 = 115$ squarings and 23 multiplications. That said, it does require the precomputation of two values: x^{31} and x^{27}. Naive k-ary exponentiation just precomputes all x^i for $0 \leq i \leq 2^k - 1$, but we know which exponents we need and thus we can be a bit faster. We essentially build an addition chain up to x^{31} that includes computing the value for x^{27}. Consider the following sequence of operations:

$$\begin{aligned} v_2 &= x^2, \\ v_3 &= v_2 x, \\ v_4 &= (v_2)^2, \\ v_9 &= (v_4)^2 \cdot x, \\ v_{18} &= (v_9)^2, \\ v_{27} &= v_{18} v_9, \\ v_{31} &= v_{27} v_4. \end{aligned}$$

Of all precomputed values v_i, we only need v_{27}, v_{31}. Too limit the precomputed values stored, we only need to store three values which at the end contain v_{27}, v_{31} and some other temporary value v_9 that we do not care about once v_{27} and v_{31} are computed.

$$\begin{aligned} v_9 &= x, \\ v_{31} &= (v_9)^2, \\ v_{31} &= (v_{31})^2, \\ v_{27} &= (v_{27})^2 \cdot x, \\ v_9 &= (v_{27})^2, \\ v_{27} &= v_{27} v_9, \\ v_{31} &= v_{31} v_{27}. \end{aligned}$$

This adds another 4 squarings and 3 multiplications giving us a total of 119 squarings and 26 multiplications.

3.2.5 Simultaneous Inversion

Inversions, as we know from subsection 2.4.5, can be batched efficiently. In our OpenCL GPGPU setting there are essentially two ways of batching inversions whilst in the process of doing a point addition or doubling. For the moment, it is enough to consider that we usually need to do some work *before* we invert and some work *after* we invert.

Consider two work items (or threads) processing n points that at some point need to compute an inverse. The two primary ways of utilizing the threads for this computation are:

- Perform some preliminary work. Store the to-be-inverted n values in shared local memory; have one of the work items perform the inversion and write the results back to local memory. See Figure 3.3.
- Perform the preliminary work. Each work item simply computes the inversion over its n values. See Figure 3.2.

Theoretically, the first option is able to compute an inversion over many more points than the option, i.e., for a work group with w work item, the first option will compute nw points simultaneously whereas the second only computes n points. However, note that the inter-work-item option requires synchronization of threads before *and* after the inversion. Also, there is a relatively high amount of reading/writing to/from local memory whereas the intra-work-item option can utilize fast private memory much better. Moreover, because only one work item is computing the inversion, the other work items with a group are pretty much idle. This can be remedied somewhat by applying a technique such as proposed in [21], but this requires even more synchronization.

As to which is the superior option, for our platform and our setting, the intra-work-item solution is much better. Experimental results during development showed that the inter-work-item option is even slower than the naive solution of having one input per work item and simply computing the inversion for that one element. We theorize that this is due to the synchronization overhead, which does not seem to be enough to offset the performance gained by batching more inversions.

The intra-work-item solution has one more issue in the sense that it tends towards utilizing a very high amount of private memory if implemented in the fastest way possible. That is, during inversion we need to store at least the n input elements and another n temporary elements. While this is fast, increasing n means we quickly run out of registers and get register spilling, meaning that we allocate those registers on slow global memory. This is a large performance issue. To alleviate the issue, we ended up storing the n inputs in local memory. This puts a rather significant limit on how large n can grow, depending on work-group size. However, it is still better than the alternative.

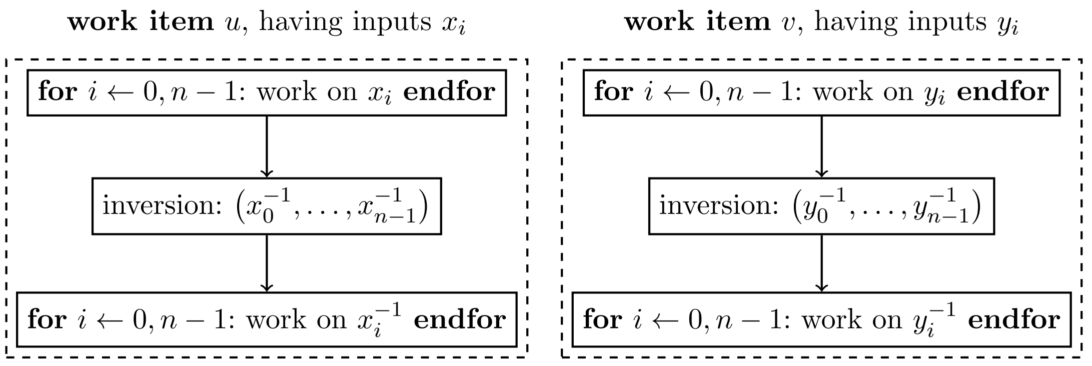
work item u, having inputs x_i
for $i \leftarrow 0, n-1$: work on x_i **endfor**
inversion: $(x_0^{-1}, \ldots, x_{n-1}^{-1})$
for $i \leftarrow 0, n-1$: work on x_i^{-1} **endfor**
work item v, having inputs y_i
for $i \leftarrow 0, n-1$: work on y_i **endfor**
inversion: $(y_0^{-1}, \ldots, y_{n-1}^{-1})$
for $i \leftarrow 0, n-1$: work on y_i^{-1} **endfor**

Figure 3.2: Batching inversions within a work item showing one iteration within kernel. Within a work item, every bit of work is done *sequentially*. Note that the two work items do not interact.

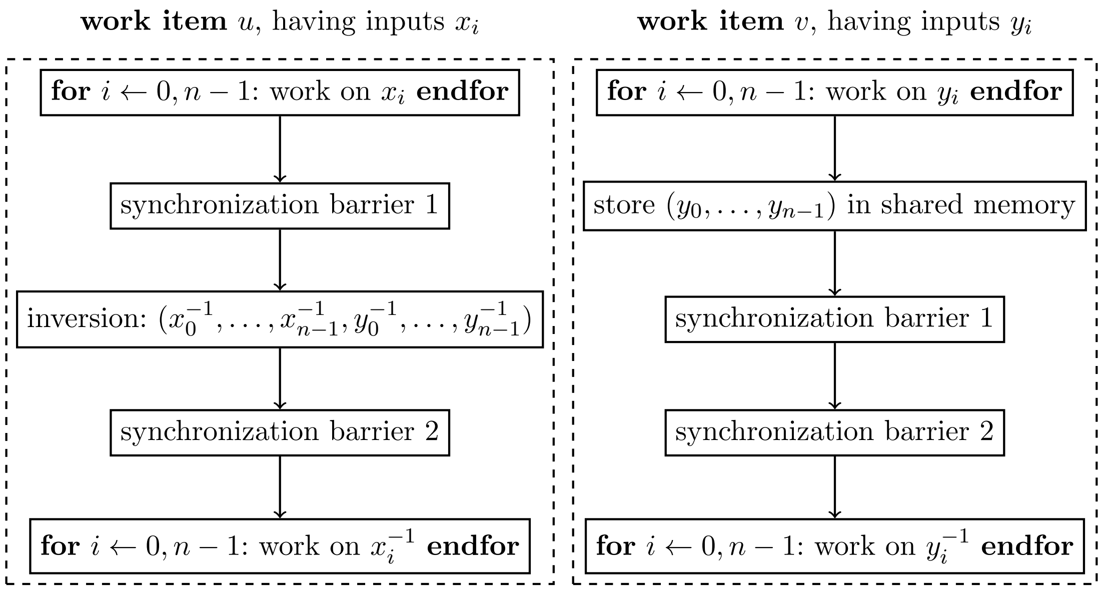
work item u, having inputs x_i
for $i \leftarrow 0, n-1$: work on x_i **endfor**
synchronization barrier 1
inversion: $(x_0^{-1}, \ldots, x_{n-1}^{-1}, y_0^{-1}, \ldots, y_{n-1}^{-1})$
synchronization barrier 2
for $i \leftarrow 0, n-1$: work on x_i^{-1} **endfor**
work item v, having inputs y_i
for $i \leftarrow 0, n-1$: work on y_i **endfor**
store $(y_0, \ldots, y_{n-1})$ in shared memory
synchronization barrier 1
synchronization barrier 2
for $i \leftarrow 0, n-1$: work on y_i^{-1} **endfor**

Figure 3.3: Batching inversions between work items for one iteration within kernel. Note how the two work items interact. This can be generalized to an arbitrary number of work items where one is designated to perform the inversion. The rest simply store their to-be-inverted inputs into shared memory.

3.3 Implementing the Iteration Function

Recall from subsection 2.3.1 that we have the following iteration function: given the ECDLP $nP = Q$, let $W_0 = bQ$ be the starting point of a walk, $R_i = a_iP$ for $0 \leq i \leq r-1$ the precomputation table of P-multiples and have $h : \langle P \rangle \mapsto \{0, 1, \ldots, r-1\}$ be an indexing function mapping a point to a table index. Given all this, the iteration function is simply computed as

$$W_{i+1} = W_i + R_{h(W_i)}.$$

Applying the negation map from subsection 2.3.3 gives us

$$W_{i+1} = |W_i + R_{h(W_i)}|,$$

where $|W_i|$ is defined as the lexicographic minimum of W_i and $-W_i$. In other words, this means that we replace $W_i = (x, y)$ with $-W_i$ iff y is odd. Because branching on a GPU platform is something we want to avoid, we can write this without branching as

$$|W_i| = |(x, y)| = (x, y + (y \bmod 2)(p - 2y)).$$

Note that we perform the reduction modulo p as well here.

In general, the complete iteration function can be thought of as consisting of a number of steps. Consider also that, for the simultaneous inversion to work, we want to compute a number of walks per work item at once. So, for the s walks in a given work item, where $0 \leq j \leq s-1$:

1. Find the point $S_j = R_{h(W_i^j)}$;
2. Compute the point addition $W_{i+1}^j = |W_i^j + S_j|$;
3. If W_{i+1}^j is a distinguished point, output W_{i+1}^j;
4. Repeat.

Computing $h(W_i)$ is simple. If we take $r = 2^d$ as some power of 2, computing h can be done by simply taking d bits from the x (or y) coordinate and treating it as an integer. Since the x coordinate is affected by the negation map, this may be a better choice. In particular, for some choices of h the negation map can exclude certain indices in R. For example, if we decide to compute h as taking the d leading bits as an index, and our y-coordinates are always even, we exclude half the points in R.

The point addition itself is pretty straightforward itself, except for the inversion. Mostly, it is just a matter of applying the formulas from section 2.2. As for the inversion, we have described how to do this in subsection 3.2.5. As mentioned there, we simply split the addition computation into the work prior to the inversion (this is one subtraction), the inversion and the computations after the inversion.

3.3.1 Point Representation

We have already mentioned how we represent elements in $\mathbb{F}_p$, i.e., we can store them as an `int4` vector. However, to store a point $P = (x, y)$ there are essentially two ways. We store the limbs of x and y *separately* or *successively*. Given $P = (x_1, y_1), Q = (x_1, y_1)$ that means:

Separate: Store as x_1, x_2, y_1, y_2, preferably as two separate buffers for the x_i and the y_i.

Successive: Store as x_1, y_1, x_2, y_2.

The first option has the notable advantage of *coalescing* memory access within work groups. That is, if work item i accesses the memory at x_i and work item $i+1$ accesses the memory right next to it, this makes such memory access faster [7].

Therefore, for all the points that we need to store, we allocate two buffers for the x and the y coordinates.

3.3.2 Distinguished Points

As described in subsection 2.3.2, we terminate a given walk after we reach a distinguished point. The distinguished-point property we will use is simple: a point $W_i = (x, y)$ is *distinguished* if the *leading* d bits of the first limb y_0 of y are all 0. Many variations on this same theme are possible, such as computing hamming weights, number of trailing zeros, and so on. However, OpenCL C has an inbuilt instruction of computing leading zeros `clz` and as such it is an easy decision to use this property.

The probability of finding a point is almost exactly 2^{-d} and as such, the parameter d can be easily fitted to the needs of the computation. That is, for such a distinguished-point property d, the expected number of points for solving a ECDLP is simply $\sqrt{\pi l/4}/2^d$, where l is size of the ECDLP. Therefore, if we have a sufficiently large l; increasing the parameter d gives us less points, resulting in less communication overhead *and* less storage required for finding a collision. One disadvantage is that once we find a collision, computing the answer from said collision takes longer. This is hardly a problem in practice. A more practical disadvantage is that when we need to restart the computation, for instance due to server reboots, we can only reasonably do so from the start of a given walk. If we have more common distinguished points we reduce the impact of having to recompute a walk.

In terms of computational cost, detecting a distinguished point for this is simply the cost of a comparison with d plus the cost of computing the inbuilt `clz` function, which for a 32-bit `int` input should be on the order of 10 shifts, 5 additions and 1 subtraction. Depending on the particular optimizations applied, this can be reduced even more. Even so, this indicates that this distinguished-point property can be efficiently computed.

Detection is relatively simple, but due to the nature of GPU programming, actually communicating distinguished points to the host is actually rather tricky and requires some consideration. First of all, we can only communicate via (slow) global memory buffers and thus we need, on the host's side, some processing to retrieve the distinguished points from said buffers. There is a distinct trade-off between the performance of the kernel, i.e., the performance of actually moving the relevant points to the result buffer, and the host-side processing overhead.

An easy way to perform this communication is by allocating a result buffer of the same size as the input buffer (the W_i's). When a distinguished point is found, simply write the point to the result buffer at the same index i as the walk. Once the iteration kernel is finished executing, simply go through the result buffer and output every point that this kernel execution found. Afterwards, reset the buffer. This works very well when we are running a relatively small amount of walks in parallel. However, as we will see later, the software can handle a very large amount of walks (in the order of 2^{21}) and in this case the host-side overhead is noticeable.

A possible solution for this is keep a global atomic counter i in memory. In OpenCL, the increment operation of such a counter actually returns the old value. Whenever we find a point in a particular work item, we do an atomic increment of $i' = i + 1$ and store the distinguished point at the i-th index in the result buffer. The host now only needs to go through the first few items of the result buffer to find all distinguished points. An issue with this is that a given point takes 256 bytes to store and thus we store a relatively large amount of data to global memory at an unpredictable index. This decreases performance of the actual write operation.

A third solution is more of a hybrid. Take two result buffers, one point result buffer res_p and one index result buffer res_i. As per the first solution, store a distinguished point p for walk i at the i-th index of res_p. Also, use the aforementioned counter j to, once again, count the distinguished points found and apply the same trick as above to store only the indices i into a separate buffer res_i. Now, on the host we still only need to go through all the indices in res_i, but we locate the points in res_p. This has the same host-side overhead as the second solution, but is more performant in terms of global-memory operations (at least, on our system). As such, this is what is used in our software.

As an aside, consider the fact that within one kernel execution, we do not only want to compute s different walks sequentially, but we only want to compute as many iterations sequentially as well. This reduces overhead as well, since we do not need as many global-memory operations, we limit the memory mapping between host and GPU and we reduce the costs of enqueueing new kernels. However, this means that we do not terminate a walk on a distinguished point immediately, only after all iterations are done. The first and third solutions mentioned above both have a problem in this context. If within t iterations (say, $t = 64$) we find, on the same walk, two or more distinguished points, we only store the last point due to the fact that we overwrite the result buffer at that index. We argue that this is not a big problem. In fact, given a distinguished-point property of $d = 20$, the chance of finding 2 points within 64 iterations is only $2^{-20} \cdot 64 \cdot 2^{-20} = 2^{-34}$. For an ECDLP of our size, this should only occur a few times over the course of the entire computation and thus is not much of a problem. Even so, we could always increase d to reduce the chance of this happening proportionally.

By its very nature, the act of having to communicate a point only if it is distinguished implies that we have a branching computation. That is, write to memory if we do, do not write to memory if we do not. Common wisdom for GPU programming indicates that we should avoid branches where possible as it leads to instruction divergence. Therefore, it follows that we should implement the storing of a distinguished point in global memory in the following way (adapted from the actual software):

```
__global int4 *result_x = ...; // result buffer for x coordinates
__global int4 *result_y = ...; // result buffer for y coordinates
size_t offset = ...; // global offset
int4 x = ...; // x coordinate
int4 y = ...; // y coordinate
// assume is_distinguished_point returns an int
int4 is_distinguished = is_distinguished_point(x, y);
result_x[offset] = select(result_x[offset], x, is_distinguished);
result_y[offset] = select(result_y[offset], x, is_distinguished);
```

However, the following code, which explicitly *includes* a branch, actually is more performant:

```
__global int4 *result_x = ..., *result_y = ...;
size_t offset = ...;
int4 x = .., y = ...;
bool is_distinguished = (bool)is_distinguished_point(x, y);
if (is_distinguished) {
    result_x[offset] = x;
    result_y[offset] = y;
}
```

We theorize this is due to the fact that in any given wavefront, it is actually unlikely for a work item to find a distinguished point at any point in time. Therefore, most wavefront will always take the exact same branch, leading to, apparently, no divergence of instructions and as such, no global memory operations taking place at all. It is hard to say which of these is more important and how much of this is due to specific device behaviour. All we can say is that branching in this setup under these circumstances is not always a bad thing.

3.3.3 Handling Fruitless Cycles

For dealing with fruitless cycles, we simply apply the technique mentioned in subsection 2.3.4. Every so often, we modify the iteration function; in other words, we execute a different OpenCL kernel. The modified kernel for checking cycles of length $2c$ works by keeping of running minimum W_{min} of the W_i during $2c$ iterations. If $W_0 = W_{2c-1}$ we enter a cycle of length $2c$. To escape, compute a new $W_{2c} = 2|W_{min}|$, i.e., double the minimum. To eliminate this branch, always compute the doubling and only copy the result to W_{2c} if a cycle is found using a `select` call such as in the previous section.

We know from [4] that fruitless cycles occur up to a length of $\log l / \log r$. For reasonable values for r, i.e., $2^{10}, 2^{11}, 2^{12}$ and so on, that means we need to check up to 12-cycles.

Also, we know that for a $2c$ cycle we need to check with a frequency proportional to $1/r^{c/2}$. This means that it is optimal for $r = 2048$ to check every 48 iterations for 2-cycles, every 2048 iterations for 4-cycles, and so on.

While it is straightforward to implement such a check every few iterations, we have to consider the effect on the other kernels. Because checking is a separate kernel iteration, we are bounded on the number of sequential iterations we can execute of the *standard* iteration function in the standard kernel. If we want optimal checking frequency, we cannot do more than the aforementioned 48 cycles for $r = 2048$. Luckily, we store R in constant (cached) global memory and as such we can get r rather high. We are limited in the sense that is useful if h only needs to work on one limb, but we can still get r to reasonably high values. In fact, most tests were performed at $r = 4096$ and 64 successive iterations per kernel. If kernels run too long, and have too many succesive iterations, it becomes increasingly unstable for purposes of development and testing.

Note that during this modified iteration function, we still do distinguished-point checking in the same way as normal. Also, the performance loss of the checking for cycles is very minor. As r increases, fruitless cycles become less frequent and checking becomes less expensive. Even at $r = 4096$, a modified checking kernel is only a little slower than a normal iteration function would be and this loss is only incurred every 64 iterations. The comparative loss of checking for, say, 10-cycles, a check which occurs very rarely, is even more negligible.

3.4 Experiments

This section details the experiments that we performed over the course of this work. Unless otherwise specified, we use a version of the software that is configured to use a distinguished-point property of $d = 20$, a work group size of 128 work items, 64 successive iterations per kernel and 16 simultaneous inversions per work item. Al

3.4.1 Distinguished Point Metrics

To give as an argument as to the validity of the software, we can verify whether it matches with established heuristics. Therefore, we ran a number of different experiments to check whether collisions occur at the expected rate of once every $\sqrt{\pi l/4}/2^d$ points, where d is the distinguished-point parameter. For these experiments, d is fixed to $d = 20$ and we use a table of size $r = 4096$.

Each experiment is performed as follows:

1. Generate a ECDLP of approximately the size wanted (45-bit, 50-bit, etc.). Note that, as per the explanation in section 2.2, this is rather easily done by varying

Experiment	a_6	l	Mean	Median	Standard Deviation
45-bit	14	$1.67 \cdot 2^{44}$	$5.66 \approx 1.48\sqrt{\pi l/4}/2^{20}$	5	2.54
50-bit	7	$1.41 \cdot 2^{49}$	$27.90 \approx 1.16\sqrt{\pi l/4}/2^{20}$	26	14.33
56-bit	21	$1.61 \cdot 2^{55}$	$212.41 \approx 1.04\sqrt{\pi l/4}/2^{20}$	200	110.49
60-bit	32	$1.05 \cdot 2^{59}$	$668.63 \approx 1.02\sqrt{\pi l/4}/2^{20}$	627	346.53

Table 3.1: Summary of the results obtained in the experiments. l is the size of the problem, a_6 is the curve parameter we change to get differently sized problems.

the a_6 parameter of the curve and then trying a few points.

2. Generate the relevant table and a sufficiently high number of input points.
3. Iterate over the input points as normal and output the distinguished points, but only output the first distinguished point at that index, not any of the later ones.
4. Keep iterating until we have a distinguished point for every index, barring a few walks that are aborted due to an overly long running time. We found that aborting after $20 \cdot 2^{20}$ iterations still completes well over 99% of all walks.
5. Sort the points into their original order, i.e, sort them on the index. This removes the relation between the order the points were found in and the iteration time of a given point.
6. Go through the sorted list until we find a collision; this is the first result. Discard all processed points from the list and find another collision. Repeat until the list is empty.
7. Optionally, compute and verify the solution to the ECDLP for the given collisions.

We ran the above experiment for 45, 50, 56 and 60-bit problems. The results are summarized in Table 3.1. You can find the distributions respectively in Figure 3.4, Figure 3.5, Figure 3.6 and Figure 3.7. In our analysis, these values are within expectations.

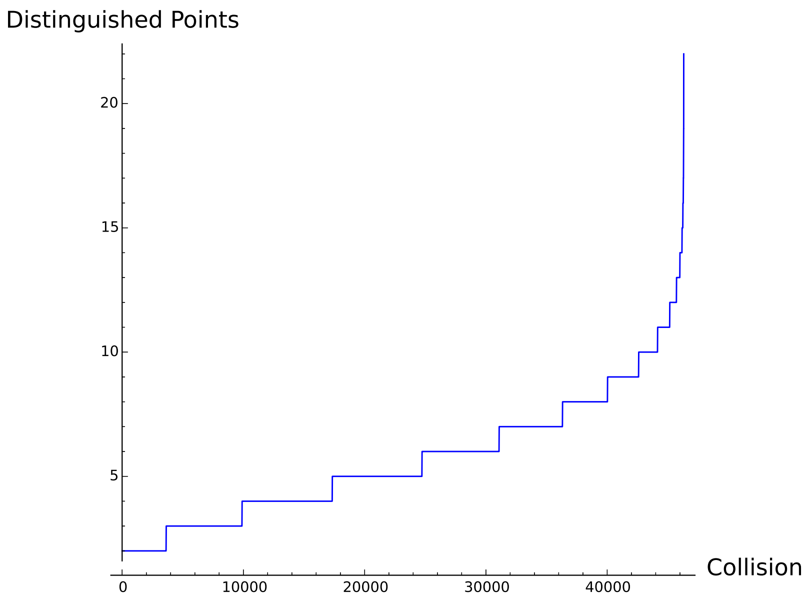

Figure 3.4: 45-bit experiment where $l \approx 1.67 \cdot 2^{44}$.

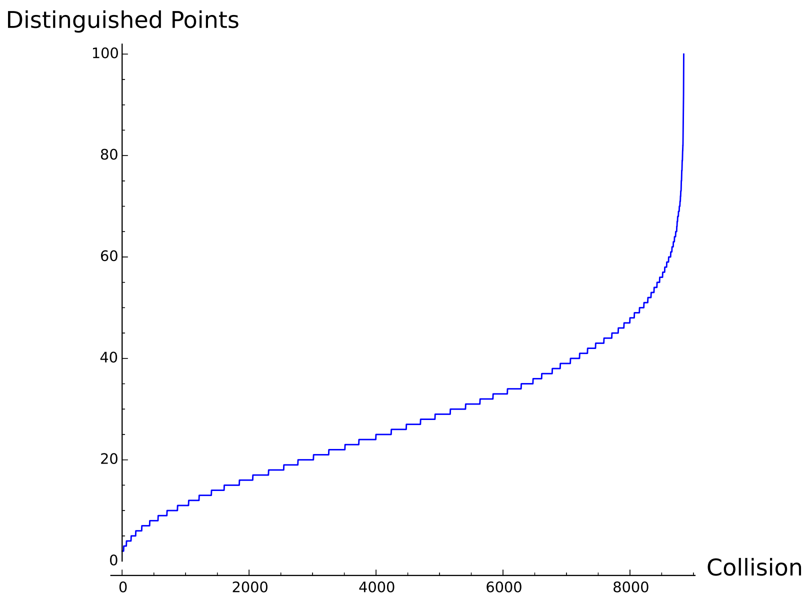

Figure 3.5: 50-bit experiment where $l \approx 1.41 \cdot 2^{49}$.

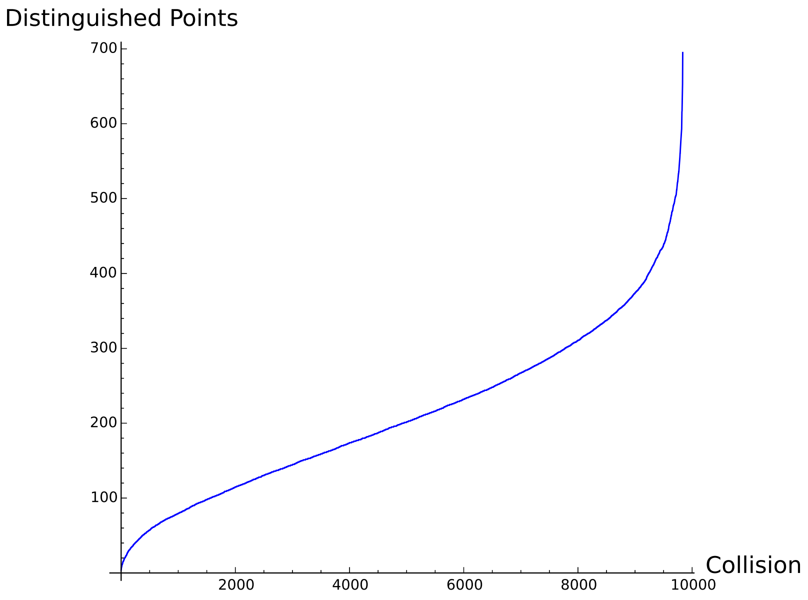

Figure 3.6: 56-bit experiment where $l \approx 1.61 \cdot 2^{55}$.

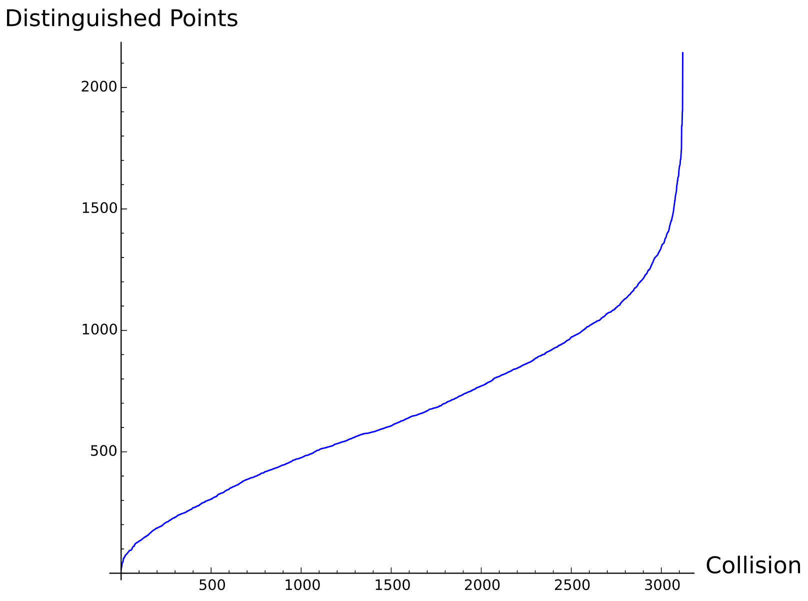

Figure 3.7: 60-bit experiment where $l \approx 1.05 \cdot 2^{59}$.

3.4.2 Performance

Inputs	Time (ns)	Inputs	Time (ns)
2^1	296636.2	2^{12}	576477.5
2^2	344908.5	2^{13}	564878.1
2^3	423303.8	2^{14}	594971.2
2^4	549546.8	2^{15}	593754.1
2^5	553007.0	2^{16}	641515.3
2^6	552566.5	2^{17}	$1.295711 \cdot 10^6$
2^7	552469.3	2^{18}	$2.527704 \cdot 10^6$
2^8	551614.9	2^{19}	$4.901548 \cdot 10^6$
2^9	553335.8	2^{20}	$9.793965 \cdot 10^6$
2^{10}	562527.9	2^{21}	$1.943748 \cdot 10^7$
2^{11}	566682.8	2^{22}	$3.843671 \cdot 10^7$

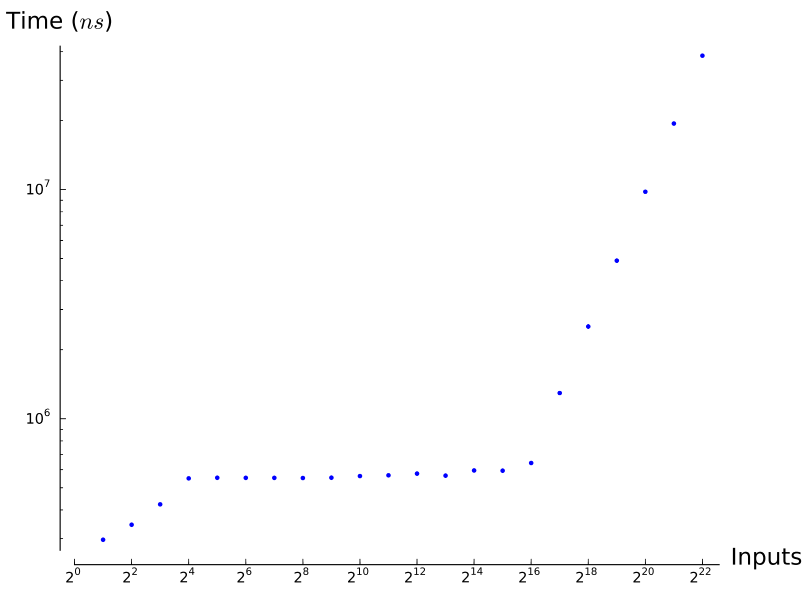

Table 3.2: Timing results of increasing input sizes on the AMD testing setup. Results are averaged over an average measured over 1000 iterations, ensuring a representative mix of both normal and cycle checking iterations. Time values indicate all iteration and processing time, not precomputation or setup.

The results from Table 3.2 indicate an important property of our software and architecture. After around 2^{16} inputs, we have essentially saturated our resources. In theory, this is where we can stop. However, careful examination of the numbers actually shows performance increases beyond this boundary. Note that doubling the number of inputs results in (on average) *almost* doubling the time required. “Almost” being the operative word. This makes a kind of sense if you consider that this likely just means the GPU has many work groups compute *sequentially* as well as in parallel. However, there is likely some overlap between a given batch of work groups and the next batch. Also, increasing the inputs does not significantly increase processing overhead. The slowest part of processing is actually reading and outputting the distinguished points and subsequently writing a new input to the input buffer. The performance of this step depends on the number of distinguished points we find after a kernel execution is finished. On average, this number is very low; we usually find only one or two distinguished points each time. Increasing the inputs does increase the number of points we find. However, at these numbers, the absolute processing time is still very short and it would take a massive increase in inputs to make this noticeable.

It should be noted that precomputation time increases linearly with the number of inputs. In fact, it takes hours for our, pretty naive, point generation program to generate 2^{22} inputs. This input file is 1.5 GiB, which is not nothing. This is not a problem per se for large ECDLPs, where the few hours it takes for this to complete is worth it, but it is worth mentioning.

If we take these results and compute an expected 112-bit ECDLP time for them, we get

$$\frac{641515.3 \cdot \sqrt{\pi \cdot 2^{112}/4}}{10^9 \cdot 2^{16} \cdot 3600 \cdot 24 \cdot 365.25} \approx 19.81 \text{ years},$$

if we consider 2^{16} inputs. If we want, we can take more inputs (and thus, more precomputation time) and get down to approximately 18.54 years at 2^{22} inputs. More inputs can conceivably decrease this number even further by a small margin. However, we were not willing to increase our precomputation costs yet again.

Obviously, when we increase l from 2^{112} to a higher value, say 2^{114}, we get $\sqrt{2^2} = 2$ times the expected time. So 37.08 years for $l = 2^{114}$.

We reiterate that these results are obtained from the AMD GPU, which is the one we optimized the most heavily for.

3.4.3 NVIDIA versus AMD

The above results are, as we mentioned in the beginning of the chapter, from our AMD test and development platform. However, we also managed to run some of our experiments on the NVIDIA platform. Due to different natures of the devices different parameters have to be chosen to get the best performance out of the NVIDIA platform. Theoretically, the NVIDIA GPU is slightly more powerful than our AMD GPU. However, since development took place on the AMD setup it is no surprise that the software is more performant on the AMD device.

Specifically, the NVIDIA software could increase the number of work items per work group from 128 to 512; this however does lead to a reduction in the amount of simultaneous inversions from 16 to 4. Note that 4 simultaneous inversions are, for us, still faster than 2 or 1, but higher values decreased performance, partially due to the fact that we had to decrease the work-group size back to the original amounts.

Inputs	Time (ns)	Inputs	Time (ns)
2^1	29556.8	2^{12}	413245.2
2^2	32312.5	2^{13}	426019.8
2^3	45169.2	2^{14}	490683.0
2^4	26538.6	2^{15}	827074.1
2^5	33717.0	2^{16}	$1.297594 \cdot 10^6$
2^6	40879.5	2^{17}	$2.405491 \cdot 10^6$
2^7	18511.5	2^{18}	$4.513904 \cdot 10^6$
2^8	27215.4	2^{19}	$8.585982 \cdot 10^6$
2^9	37686.9	2^{20}	$1.683747 \cdot 10^7$
2^{10}	32476.0	2^{21}	$3.336085 \cdot 10^7$
2^{11}	408048.0	2^{22}	$6.584475 \cdot 10^7$

Table 3.3: NVIDIA timing results. The same procedure applies as in Table 3.2.

We repeated the experiment from subsection 3.4.2 on the (slightly) optimized NVIDIA setup. The results of which can be found in Table 3.3. The most interesting result is probably that for low input values, the pure performance of the NVIDIA GPU shines through. This trend continues up to around 2^{15} inputs. Afterwards, however, it tends to be approximately twice as slow. Because the feasibility of computing ECDLPs using our software relies so heavily on large numbers of inputs, the AMD platform has a far lower minimum time spent, the aforementioned 18.54 years, whereas the NVIDIA version has 31.77 years. Which, for a 112-bit problem is sufficient, but starts to get on the high side for larger problems.

4 Conclusions

We hope to have provided a sufficiently detailed report on this thesis' work. The overall goal was to get insight into a generalized GPU-based solution for ECDLP-solving which can run on whatever GPU supporting OpenCL. In particular, as it relates to the inherent limitations of such a platform. This setting has its own obstacles and we think that, within the scope of this research, we done an adequate job of addressing those.

We would have liked to have spent more time on a wider variety of systems, i.e., more than two (and one of those two in a limited capacity). In particular, the implementation does not perform as well as hoped on the NVIDIA platform. Also, someone with more experience in the respective fields of GPU programming and cryptographic implementation can likely find ways to make things even faster.

To summarize the results. We have created software that solves a 112-bit problem in approximately 18.5 years and thus an, also extrapolated, result of 26.2 years for a 113-bit problem. We compute 109.12 million iterations per second, if assuming a batch size of 2^{22} inputs. Furthermore, we have mentioned the insights and theories obtained from implementing such a generalized OpenCL-based ECDLP solver. In particular, note that in this setting, optimization is mostly limited to how one handles the memory allocation.

4.1 Comparison to Related Work

There have been, throughout the last two decades, a decent amount of projects with the specific aim of breaking ECDLPs. Most prevalent in these are the attempts at solving the ECC2K challenges by Certicom [6].

Notable for us, however, are the more recent results. To the best of our knowledge, the state-of-the-art in ECDLPs on prime fields, such as in this thesis, is still the extrapolated result in [4], where they solve a 112-bit ECDLP in 35.6 years on a Cell processor (PS3). This is especially notable in comparison to the 65.16 years for that same ECDLP from [5].

Our software, using our AMD HD7850, solves a ECDLP of similar (112-bit) size in 18.54 years, which is about two times faster. The GPU platform is theoretically much faster in the sense that it can run a great many more inputs in parallel. However, this is less opportunity for optimization if we are stuck in an OpenCL setting. In [1] there is very little difference between the Cell and GPU results, but this may very well be due to the constraints of the particular toolchain as well as the unwieldy bitsliced representation. Also, this work is done on ECC2K-130 and specifically targeted binary curves. That makes a comparison fairly hard.

There is some recent work done by Wenger and Wolfger [27, 28], which targets 113-bit binary curves, on FPGAs. This work manages 900 million iterations per second, which is about 8 times as much as our software. Note that the setting, i.e., binary curves, is different and this is on a vastly different platform.

4.2 Future Work, Hopes & Dreams

We feel that some of the limitations of the setting we were working with, i.e., OpenCL and GPUs in general, could be alleviated. For instance, we would love to see OpenCL and the respective GPU vendors adopt and support more low-level optimized operations on scalar types. In lieu of that, the existence of vendor-supported assemblers could fill much the same role. Moreover, the ability to have more fine-grained control over private memory allocation would be fantastic. Right now, the ability to constrain the compiler in assigning registers for critical functionality within kernels is severely limited. At this moment, you are more likely to make a random change and see if you "help" the register allocator.

Furthermore, we would like to see more work in the area of vendor-specific GPU ECDLP solving. This could be similar to work done in [3], especially considering the speed-ups found when using something like a `qhasm`. For AMD, there are some projects that use GCN ASM code [2, 29]. This would be an interesting direction, since this does give you the room for instruction-level optimization that OpenCL simply does not.

To a certain extent, previous work, such as the NVIDIA CUDA results in [1] could be improved by simply increasing batch sizes. In some cases, quite significantly larger batches. Especially on GPUs it often the case that the amount of *global* memory is not the problem. Note that we kept on seeing performances increases well after batch sizes of 2^{20} large.

As a last point, we are still planning on running the software to solve a 113-bit ECDLP. As such, we would really like to do this in the near future, if only to add some credence to the results in this thesis. Also, we would like to fully optimize the software in such a way that the NVIDIA performance is at least similar to the performance we managed on the AMD setup.

5 Bibliography

[1] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier Van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gürkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/514, 2009. `http://eprint.iacr.org/2009/541/`.

[2] Dániel Bali. AMD GCN ISA assembler. `https://github.com/balidani/gcnasm` [Accessed 2015-08-17].

[3] Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. ECC2K-130 on NVIDIA GPUs. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology – INDOCRYPT 2010*, volume 6498 of *Lecture Notes in Computer Science*, pages 328–346. Springer-Verlag Berlin Heidelberg, 2010. `http://cr.yp.to/papers.html#gpuecc2k`.

[4] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. On the correct use of the negation map in the Pollard rho method. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *Public Key Cryptography – PKC 2011*, volume 6571 of *Lecture Notes in Computer Science*, pages 128–146. Springer-Verlag Berlin Heidelberg, 2011. `http://cr.yp.to/papers.html#negation`.

[5] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *International Journal of Applied Cryptography*, 2(3):212–228, 2012. `http://joppebos.com/files/noan112.pdf`.

[6] Certicom. The Certicom ECC Challenge. `https://www.certicom.com/index.php/the-certicom-ecc-challenge`. [Accessed 2015-05-08].

[7] Advanced Micro Devices. OpenCL Optimization Guide. `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/`. [Accessed 2015-31-07].

[8] Advanced Micro Devices. AMD Accelerated Parallel Processing OpenCL Programming Guide. `http://developer.amd.com/wordpress/media/2013/07/`

AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf, 2013. [Accessed 2015-07-08].

[9] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[10] Iwan Duursma, Pierrick Gaudry, and François Morain. Speeding up the discrete log computation on curves with automorphisms. In *Advances in cryptology – Asiacrypt '99*, pages 103–121. Springer, 1999.

[11] Robert Gallant, Robert Lambert, and Scott Vanstone. Improving the parallelized pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.

[12] Robert J. Harley. Solution to Certicom's ECC2k-95 problem. http://cristal.inria.fr/~harley/ecdl5/ECC2K-95.submission.text, 1998.

[13] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.

[14] Khronos. OpenCL. https://www.khronos.org/opencl/. [Accessed 2015-07-08].

[15] Donald E. Knuth. Seminumerical algorithms, the art of computer programming, vol. 2, 1981.

[16] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[17] Adam Langley and Rich Salz. Elliptic curves for security. Internet-Draft draft-irtf-cfrg-curves-02, IETF Secretariat, March 2015. http://www.ietf.org/internet-drafts/draft-irtf-cfrg-curves-02.txt.

[18] Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. An efficient protocol for authenticated key agreement. In *Designs, Codes and Cryptography*, 1998.

[19] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography.* CRC press, 1996. http://cacr.uwaterloo.ca/hac/.

[20] Victor Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology – CRYPTO85 Proceedings*, pages 417–426. Springer, 1986.

[21] Pradeep Kumar Mishra and Palash Sarkar. Inversion of Several Field Elements: A New Parallel Algorithm. 2003. http://eprint.iacr.org/2003/264.

[22] Peter L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[23] NVIDIA. CUDA. `http://www.nvidia.com/object/cuda_home_new.html`. [Accessed 2015-07-08].

[24] Katsuyuki Okeya, Hiroyuki Kurumatani, and Kouichi Sakurai. Elliptic curves with the Montgomery-form and their cryptographic applications. In *Public Key Cryptography*, pages 238–257. Springer, 2000.

[25] John M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of computation*, 32(143):918–924, 1978.

[26] Paul C. Van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of cryptology*, 12(1):1–28, 1999.

[27] Erich Wenger and Paul Wolfger. Solving the discrete logarithm of a 113-bit koblitz curve with an fpga cluster. Cryptology ePrint Archive, Report 2014/368, 2014. `http://eprint.iacr.org/2014/368`.

[28] Erich Wenger and Paul Wolfger. Harder, better, faster, stronger - elliptic curve discrete logarithm computations on fpgas. Cryptology ePrint Archive, Report 2015/143, 2015. `http://eprint.iacr.org/2015/143`.

[29] Ryan S. White. GCN assembler for AMD GPUs. `http://www.codeproject.com/Articles/872477/Assembler-for-AMD-s-GCN-GPU` [Acessed 2015-08-17], 2015.

[30] Michael J. Wiener and Robert J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In *Selected areas in Cryptography*, pages 190–200. Springer, 1999.