Parallel Windowed Method for Scalar Multiplication in Elliptic Curve Cryptography

Parallel Windowed Method for Scalar Multiplication in Elliptic Curve Cryptography

By
Tanya Bouman B.A.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE(2020) McMaster University
COMPUTER SCIENCE Hamilton, Ontario

TITLE: Parallel Windowed Method for Scalar Multiplication in Elliptic Curve Cryptography

AUTHOR: Tanya Bouman B.A.Sc. (McMaster University)

SUPERVISOR: Dr. Christopher K. Anand & Dr. Wolfram Kahl

NUMBER OF PAGES: ix, 28

LEGAL DISCLAIMER: This is an academic research report. I, my supervisor, defence committee, and university, make no claim as to the fitness for any purpose, and accept no direct or indirect liability for the use of algorithms, findings, or recommendations in this thesis.

Abstract

Commercial applications, including Blockchain, require large numbers of cryptographic signing and verification operations, increasingly using Elliptic Curve Cryptography. This uses a group operation (called point addition) in the set of points on an elliptic curve over a prime field. Scalar multiplication of the repeated addition of a fixed point, P, in the curve. Along with the infinity point, which serves as the identity of addition and the zero of scalar multiplication, this forms a vector space over the prime field. The scalar multiplication can be accelerated by decomposing the number of additions into nibbles or other digits, and using a pre-computed table of values P, $2P$, $3P, \ldots$ This is called a windowed method. To avoid side-channel attacks, implementations must ensure that the time and power used do not depend on the scalar. Avoiding conditional execution ensures constant-time and constant-power execution.

This thesis presents a theoretical reduction in latency for the windowed method by introducing parallelism. Using three cores can achieve an improvement of 42% in the latency versus a single-threaded computation.

Acknowledgments

Thank you to Christopher Anand for kindness and guidance, beginning in my undergrad, and through my Masters. I appreciate the many wonderful opportunities. Thanks also to Wolfram Kahl for additional guidance. Thanks to my parents, Glenn and Jorina, and my siblings, Henry, Mark, and Janae for your love and encouragement. Thanks to my colleagues in ITB 229 for camaraderie and friendship, especially Musa Al-hassy for the helpful feedback on this thesis. And I owe everything to the Lord, who gives strength sufficient for every day.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Scalar multiplication is a fundamental operation in the various elliptic curve cryptographic scheme, such as the Elliptic Curve Digital Signing Algorithm (ECDSA). ECDSA is the most frequently used signing algorithm for public blockchains [WSL+19]. In this thesis we present a method for improving the performance of this fundamental operation through parallelism. This is an expanded version of a previous published paper [BIYA20].

1.1 Motivation

ECDSA requires key generation, signature generation, and signature verification. Key generation includes a private and public key generation. The private key, d, is a random integer bounded by the prime, p, and the public key, Q, is computed by the scalar multiplication of d and G. Both parties agree in advance on G, which is one of the points in an elliptic curve. The signature is generated based on the private key and some hash[1] H of the message, and then verified based on the public key, and hash of the message. Algorithms 1 and 2 show signature generation and verification, which both include scalar multiplications. In both algorithms, H is a cryptographic hashing function.

When the messages to be signed are small enough that computing the hash of the message does not dominate the computation time, scalar multiplication dominates. Thus any improvement to the scalar multiplication significantly benefits the overall cost of the signature algorithms.

1.2 Objective

There exist numerous methods [HV04] for computing the scalar multiplication, such as the binary method, windowed method and the Montgomery ladder. Building on work of previous authors, we show that the windowed method for scalar multiplication can be parallelized, first to generate the lookup table, and then to perform the rest of the computation. In particular, we work out a method which uses 3 cores and could achieve a speedup of 42% compared to a serial windowed method.

[1] A specific hashing method is not defined in the protocol.

Algorithm 1 ECDSA signature generation [HV04]

Input: Private key d, message m, hash H, curve parameters, including base point G and prime p.
Output: Signature (r, s).

1: Randomly choose $k \in [1, p-1]$.
2: Compute $(x_1, y_1) = kP$.
3: Compute $r = x_1 \mod p$.
4: **if** $r = 0$ **then**
5: Go to step 1.
6: **end if**
7: Compute $e = H(m)$.
8: Compute $s = k^{-1}(e + dr) \mod p$.
9: **if** $s = 0$ **then**
10: Go to step 1.
11: **end if**
12: **return** (r, s).

Algorithm 2 ECDSA signature verification [HV04]

Input: Public key Q, message m, hash H, signature (r, s), curve parameters, including base point G and prime p.
Output: Acceptance or rejection of the signature.

1: **if** $r \notin [1, p-1]$ or $s \notin [1, p-1]$ **then**
2: **return** "Reject the signature".
3: **end if**
4: Compute $e = H(m)$.
5: Compute $w = s^{-1} \mod p$.
6: Compute $u_1 = ew \mod p$ and $u_2 = rw \mod p$.
7: Compute $(x_1, y_1) = u_1 P + u_2 Q$.
8: **if** $(x_1, y_1) = \infty$ **then**
9: **return** "Reject the signature"
10: **end if**
11: Compute $v = x_1 \mod p$.
12: **if** $v = r$ **then**
13: **return** "Accept the signature"
14: **else**
15: **return** "Reject the signature"
16: **end if**

Chapter 2

Background

In the rest of this section, we discuss elliptic curve cryptography, focusing on scalar multiplication, and examine existing methods for performing those scalar multiplies.

2.1 Elliptic Curve Cryptography

The performance estimates in this thesis and implementations use the NIST P-256 parameters. NIST P-256 uses an elliptic curve defined on an xy-plane in short Weierstrass form.

$$y^2 = x^3 + ax + b$$

Given two points, P and Q, on that elliptic curve, an addition operation to calculate a third point, R, can be defined by taking the line through P and Q and extending it to find the third point on the curve where the line intersects, and negating the third point. Then R is the negation of that third point. In the case that $P = Q$, the tangent to the curve at the point is used. Figure 2.1 illustrates both cases. The identity of addition on an elliptic curve is the point at infinity, O, i.e. $P + O = P = O + P$. If there is no third point where the line intersects the elliptic curve, R is O. This happens, for example, when adding P to $-P$.

For the purposes of cryptographical schemes, the elliptic curve is defined over a prime field. The prime field $\mathbb{F}_p$ of order p consists of the numbers $0, 1, \ldots, p-1$, and addition and multiplication as the usual integer operations, performed modulo p.

There are two representations for points on the curve used in computation. The usual representation of a point in (x, y) coordinates is called *affine coordinates*. This representation is compact, but cannot represent the identity point (infinity). Since not all points (x, y) are points on the curve, a point not on the curve could be used to represent the identity.

Projective coordinates represent the same (x, y) point with (X, Y, Z). The conversion is $(x, y) = (X/Z^c, Y/Z^d)$, where c and d are positive integers. Different types of projective coordinates are defined by different values of c and d. *Jacobian coordinates* are one example of projective coordinates, where $c = 2$, and $d = 3$. Any point with $Z = 0$ cannot be converted to the affine representation, so this is a good representation of the infinity point. Jacobian coordinates are useful for calculations on elliptic curves because they avoid expensive modular inverses that are necessary to compute in affine coordinates [HV04]. Unlike affine coordinates, an inconvenience of any projective coordinates is that due the ratio caused by division by a power of Z, they are not unique representations of a point.

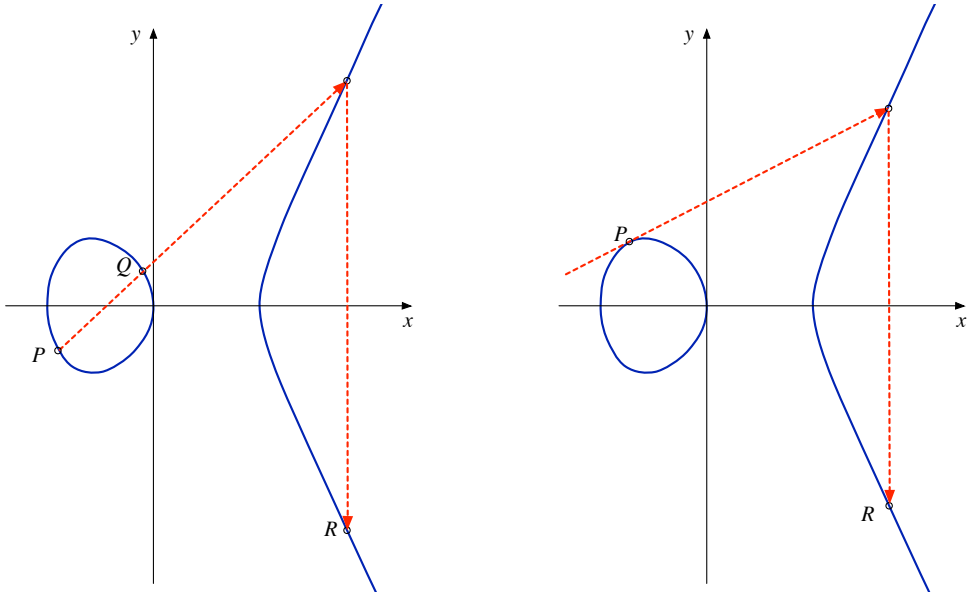

Figure 2.1: Addition and doubling of points on an elliptic curve over $\mathbb{R}$ [HV04]

Operation	Cost (modular multiplications)
$2 * J$	8
$J + A$	12
$J - A$	12
$J + J$	16
$J - J$	16

Table 2.1: Cost of point operations on affine (A) and Jacobian (J) coordinates

Since good performance is our goal, we use Jacobian coordinates in our calculations. We use existing algorithms for point addition and point doubling (see [BL]). These algorithms depend on modular addition and multiplications on the underlying prime field. The modular additions consist of an addition, and if the result is larger than p, a subtraction to take care of the modulo. In comparison, modular multiplications are much more expensive, with many additions to calculate the initial multiplication, and a full modulo operation at the end. For performance estimates, we count the number of modular multiplies performed in each calculation, since other operations such as modular addition have negligible cost in comparison. See Table 2.1.

Given these addition and doubling operations, it is possible to define a scalar multiplication, sometimes known as point multiplication, dP for an elliptic curve point P and $d \in \mathbb{N}$. In elliptic curve cryptography, the secret key d is the scalar in the multiplication and the product dP is the public key. [HV04].

2.2 Double and Add

The simplest method of calculating the scalar multiplication is to take the binary representation of the scalar and go through each bit of the scalar, d_i, doubling to calculate $2^i P$ and adding the power of 2 scaling to an accumulator when d_i is 1 [Gor98]. This is the same concept as methods used to calculate binary multiplication using addition and shifting [Rei57], and similar methods can be used to improve performance. More sophisticated methods of doubling and adding attempt to use shorter addition chains than the one given by the binary representation of the scalar [BC90]. Since subtraction of an elliptic curve point is simply addition of the negative, and the negative of an elliptic curve point is formed by taking the negative of the second coordinate, subtraction costs about the same as addition and addition-subtraction chains further improve performance. Double and add also uses the same concept as multiply and squaring methods used in exponentiation, for example in other cryptography contexts, such as RSA. One notable difference when multiplying and squaring integers is that division is much more expensive that multiplication, so including it in the cases when subtraction improves performance is not helpful [MO90].

For any of these approaches, the value of the scalar affects the number of operations needed to compute the product, and thus the amount of computational time and energy spent. This makes it possible for an attacker to discover information about the secret key by monitoring *side channel* information such as the time elapsed or the energy expended, in a timing attack or Simple Power Attack (SPA). One example takes an elliptic curve defined over a prime field, where the prime is 256 bits long. Within 200 signatures, an attacker may be able to reconstruct a secret key through side channel information from cache hits and misses. Reconstructing the key without the side channel information would be computationally impossible [BvdPSY14]. One way to avoid this problem [Koc96] is to perform all of the possible doublings and additions needed for the scalar multiplication, but discard unnecessary intermediate results in the final product [Cor99]. The Montgomery ladder multiplication method is an efficient way to do that [Mon87].

A more sophisticated side-channel attack, such as Differential Power Analysis (DPA), can extract side-channel information even if the sequence of operations does not depend on the scalar. Countermeasures include randomization of the private exponent, blinding the point P, and randomization of projective coordinates [Cor99]. We do not discuss these measures any further, but the techniques apply to the proposed parallel algorithm.

2.3 Windowed Method

The *windowed* method improves performance by calculating multiples of P in advance, i.e.,

$$1P, 2P, \ldots, (2^w - 2)P, (2^w - 1)P,$$

where w is the width, in bits, of the window. To compute the scalar multiple dP, d is decomposed into "digits"

$$d = d_{D-1} d_{D-2} \cdots d_1 d_0$$

each with w bits, and containing a value between 0 and $2^w - 1$, where D is $\left\lceil \frac{l}{w} \right\rceil$, and l is the number of bits in d. The smallest digit, d_0 could have fewer bits w_s than the others, if the total number of bits does not equally divide by w. In the first step, a table lookup finds the corresponding $d_{D-1}P$ to be used as the accumulator. In subsequent steps, the accumulator is doubled w times before adding

the next $d_i P$. So if the running total started as

$$d_{D-1}P = \sum_{i=D-1}^{D-1} d_i 2^{(i-(D-1))w} P,$$

after w doublings, it becomes

$$2^w(d_{D-1}P) = \sum_{i=D-1}^{D-1} d_i 2^{(i-(D-1)+1)w} P.$$

To this we add the value $d_{D-2}P$ from the table to obtain

$$d_{D-2}P + 2^w(d_{D-1}P) = \sum_{i=D-2}^{D-1} d_i 2^{(i-(D-2))w} P.$$

After $D-1$ steps, we obtain

$$\sum_{i=1}^{D-1} d_i 2^{(i-1)w} P,$$

and finally w_s doublings and adding $d_0 P$ gets

$$d_0 P + \sum_{i=1}^{D-1} 2^{w_s+(i-1)w} d_i P = dP$$

When the window size equally divides the number of bits, $w = w_s$ and it can be simplified to

$$\sum_{i=0}^{D-1} d_i 2^{iw} P = dP.$$

Note that when $w = 1$, this is the same as the double and add method, since the table only contains 0 and $1P$.

Other methods exist which improve upon the windowed method by skipping computation for bits which have a zero value, and using a non-adjacent form (NAF) [Rei60] to increase the number of zero bits in the representation. Skipping the zero bits means that the performance is no longer in constant time, and those methods vulnerable to side-channel attacks like SPA [HV04]. Finally, there are other methods such as applying a Frobenius map [MS93] or other endomorphisms [GLV01], but these only apply to specific types of curves.

2.4 Scheduling

This section discusses a performance estimate of a straightforward, serial windowed method, first for table pre-computation, and then for main computation.

2.4.1 Table Generation

To generate the table for the windowed method, we calculate all of the points between $2P$ and $2^w P - 1$, starting with $1P$ as the input value in affine coordinates. The number of operations

	Op.	Cost (mod. mults.)
$2 * nP$	$2 * P$	8
$(2n-1)P + 1P$	$P + 1$	12
$(2n-i)P + iP$ where $2 \leq i < n$	$P + Q$	16

Table 2.2: Ways to calculate $(2n)P$, including costs in number of modular multiplications.

	Op.	Cost (mod. mults.)
$(2n)P + 1P$	$P + 1$	12
$(2n-i+1)P + iP$ where $2 \leq i \leq n$	$P + Q$	16

Table 2.3: Ways to calculate $(2n+1)P$, including costs in number of modular multiplications.

necessary to calculate each point does not matter; only the total number of operations to calculate the entire table matters. The cost for a certain point is the number of operations used to calculate that point, given all the previously calculated points. This varies depending on the cost of the point doubling and point addition algorithm. The addition of a point in affine coordinates to a point in Jacobian coordinates costs less than the addition of two Jacobian points, and the only point available in affine coordinates is P itself, so adding $1P$ is preferred over other additions. [1]

Starting with $1P$, the only possible way to get $2P$ is by doubling $1P$, which costs 8 modular multiplies. Then, if all the points from $1P$ to $(2n-1)P$ are already calculated, there are several ways of calculating $(2n)P$, and then $(2n+1)P$. See Tables 2.2 and 2.3. For $(2n)P$, doubling nP is the cheapest option. This costs 8 modular multiplies. For $(2n+1)P$, the cheapest option is adding $1P$ to $(2n)P$, which costs 12 modular multiplies. Therefore, if all the table pre-computation occurs serially, the best solution is that the even numbers be computed by point doubling and the odd numbers by adding P to the even number immediately preceding it, and this has a cost of

$$(2^{w-1} - 1)(c_J + c_{JA})$$

where c_J is the cost of doubling a Jacobian and c_{JA} is the cost of adding a Jacobian to an affine. These costs are 8 and 12 modular multiplies, respectively.

2.4.2 Main Computation

A straightforward serial implementation does the main computation of the windowed method as:

$$\begin{aligned} dP &= d_0P + \sum_{i=1}^{D-1} 2^{w_s+(i-1)w} d_iP \\ &= d_0P + 2^{w_s}(\ldots(d_{D-2}P + 2^w(d_{D-1}P))\ldots) \end{aligned} \tag{2.1}$$

where w_s (the width of the short digit at the end) is $l - (D-1)w$. In the equation above, each operation must occur in order, leaving no room for parallelism. The cost to perform all of these

[1] When adding affine to Jacobian, the algorithm is the same as adding two Jacobians, except that we know that one of the Z-coordinates is 1, and this makes 4 of the modular multiplications unnecessary.

operations in serial is

$$16 \cdot (D-1) + 8 \cdot (l-w).$$

Every digit needs to be added to the total, so there are $\lceil \frac{l}{w} \rceil - 1$ or $D-1$ additions. Then the repeated doubling in 2^w happens for all but one of the digits, for a total of $l-w$ doubles.

Putting together the table and main computations, this results in a total estimated cost of 3164 modular multiplies for $w = 4$, and 6120 for $w = 1$, which is essentially the double and add method.

Chapter 3

Parallel Scheduling

This chapter discusses the scheduling methodology used. First, we make a parallel pre-computed table, which can scale up to many cores. Second, we examine several possibilities for a parallel main section, including a method which only works on two cores, and two methods which scale to many cores.

3.1 Table Generation

3.1.1 Greedy Scheduling for Infinitely Many Cores

When multiple cores can work in parallel to compute the table, it becomes possible that the best schedule performs different calculations than the optimized ones discussed in section 2.4.1. Figure 3.1 shows a possible schedule for the initial calculations of a table. It starts out the same way as the table in serial. However, the order begins to differ at the computation of $8P$, as it could begin before the computation of $6P$ and $7P$. After a while, things get more complicated. Notice, for example, that at time 24, the computation $10P$ is not yet started, even though it could be calculated from $8P$ + $2P$. This is because at the next step we find out that it will actually be faster to compute $10P$ by doubling $5P$, if we just wait one more step until it is completed. So $10P$ was removed from that step and moved to a later step.

This scheduling is greedy in that it does as many computations as possible at the earliest time that they can possibly end.

Since we have no theoretical limit for simultaneous calculations, the completion time of each point can be minimized using the same methods as double-and-add calculations. The time to calculate the whole table is equivalent to the longest time required to calculate any single given point. In the case of the table of width 3, the point $7P$ is the last to finish, so that whole table requires time equivalent to 36 modular multiplies to calculate, compared to 60 modular multiplies if the table was calculated serially. However, the total amount of calculation increases to 64, because the calculation of $7P$ from the addition of $3P$ and $4P$ costs more than the addition of $1P$ to $6P$, and that adds two Jacobians together, rather than an affine point to a Jacobian point.

One of the relevant improvements to the double-and-add methods is that rather than limiting the chain of calculation to addition operations, we can include subtractions in the chain, and obtain a better result. The smallest point which benefits from a mixed addition subtraction chain is $15P$. Figure 3.2 demonstrates the improvement, and the benefits continue for larger multiples of P.

Time (mod. mults.)						
0	$2P_{=2*P}$					
4						
8	$3P_{=P+2P}$	$4P_{=2*2P}$				
12						
16		$5P_{=P+4P}$	$8P_{=2*4P}$			
20	$6P_{=2*3P}$			$7P_{=3P+4P}$		
24			$9P_{=8P+P}$		$11P_{=8P+3P}$	$16P_{=2*8P}$

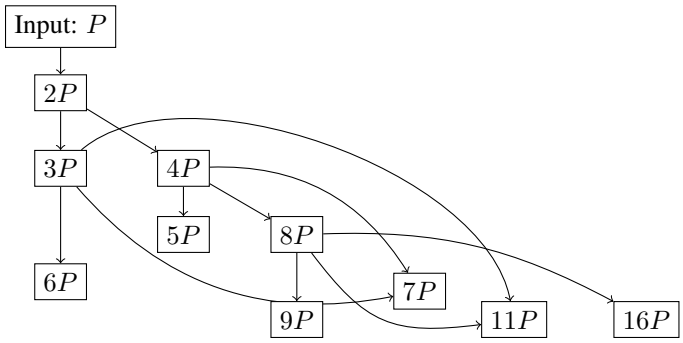

Figure 3.1: Given unlimited cores, this is the greedy schedule. The subscripts indicate how each value is calculated. The second part of this figure shows the dependencies for each value calculated.

Time	Without Subtraction		With Subtraction
0	$2P_{=2*P}$		$2P_{=2*P}$
4			
8	$3P_{=P+2P}$	$4P_{=2*2P}$	$4P_{=2*2P}$
12			
16		$8P_{=2*4P}$	$8P_{=2*4P}$
20	$7P_{=3P+4P}$		
24			$16P_{=2*8P}$
28			
32			$15P_{=16P-P}$
36	$15P_{=7P+8P}$		
40			
44			
48			
52			

Figure 3.2: Comparison of calculating $15P$ with and without subtraction.

3.1.2 Generate and Prune Scheduler for Finite Cores

The above algorithm schedules as many computations as possible at a time. Realistically, we only have a limited number of cores available. When more computations are ready than cores avail-

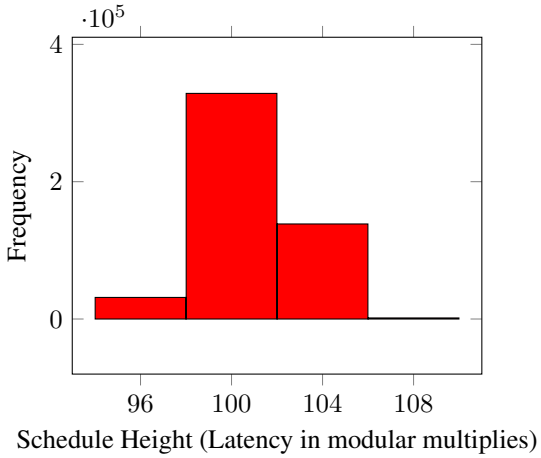

Figure 3.3: A histogram of schedule heights generated by depth-first search shows that most schedules have similar heights.

able, we must decide which computations to allocate and which computations to delay for future allocation. Given all of the computations possible at a certain point, we come up with all of the possibilities for which computations to allocate, and which computations to delay until there is a core available. Since we have no obvious way to know which choice of delays will produce the shortest schedule, we try them all out. However, there might be schedules whose partial heights are longer than whatever schedules were already generated, so these are pruned away to avoid any further scheduling on them.

The results given here are an estimate based on the number of modular multiplications that an operation takes. This gives us a cost of 8 for point doubling, 12 for addition of the base point to another point and 16 for any other addition. While there are other operations involved, their cost is negligible in comparison, so for the purposes of this estimate, they are not considered.

The histogram in Figure 3.3 shows the distribution of schedule lengths produced by the brute force scheduling of a table with width 5 on 4 cores. Since the brute force scheduler produces many possible schedules, we select only the first 500 000 to show in the graph. While most of the schedules in the graph do not have the optimal height of 96 modular multiplications, none of them have a latency longer than 108, and the majority of the latencies are between 96 and 108.

Figure 3.5 shows how much parallelism is available in the window computation by plotting the latency of table generation for different bit widths versus the number of cores. For example, at bit width 2 with a table size of 4, there is enough parallelism to reduce latency by 30% with two cores, but not enough parallelism to exploit more than two cores. At bit width 4, two cores brings a 40% reduction in latency, four cores a 48% reduction, and there are no further gains. Unsurprisingly, it is easy to exploit parallelism across a small number of cores. But for bit widths $w \leq 5$, there is no advantage to using more than 2^{w-1} cores.

Figure 3.6 shows the latency of table generation compared to the width. Plotting the latency of window computation as a function of the window width in bits shows that the computation scales exponentially with a single core, but has sub-exponential scaling with larger numbers of cores. For a single core, the cost depends on the width as $10(2^w) - 20$. For two cores, it is $5(2^w) - 4$, when $w > 2$.

```haskell
tablesScheduleNarrowedBruteForce ::
        -> Int -- ^ number of parallel cores
        -> Int -- ^ table width, in bits
        -> [Schedule] -- ^ schedule of when and how to calculate
         ↪  the multiples
tablesScheduleNarrowedBruteForce cs w =
  let
    -- initialize schedule, initially running operations, and
     ↪  completed computations
    initSchedule = [(0,Double 1)]
    initRunning = (Double 1, 8)
    initComplete = [1]
    addToSchedule time complete schedule running delayed =
      let
        -- update the variables tracking which operations have
         ↪  completed
        (justCompleted, stillRunning) = partition isDone running
        newComplete = ops justCompleted ++ complete

        -- figure out which calculations are possible to be
         ↪  scheduled
        newPossibleOps = getNewOps delayed justFinished
         ↪  newComplete stillRunning w

        -- come up with all possible further schedules
        (newSchedules, newRunnings) = allAssignments schedule
         ↪  stillRunning cs newPossibleOps
      in
        if allEmpty newRunnings
        then newSchedules
        else
          concat $ zipWith3 (addToSchedule (time+gcd opCosts)
           ↪  newComplete) newSchedules (map updateRunTime
           ↪  newRunnings) newDelayeds
  in
    addToSchedule 0 initComplete initSchedule initRunning []
```

Figure 3.4: Outline of Haskell function for Brute Force Scheduling of Parallel Table Pre-Computation

When the windowed method is used without parallelization, the normal recommendation is to use a width of size 4, due to the trade-off between the amount of pre-computation and the main calculation itself. However, if the pre-computation portion can be performed in parallel, the latency is significantly reduced, meaning that larger size widths can be considered. When we calculate the total latency including both the window pre-computation and calculation using the window, we find a more modest expected speedup of 20% when comparing the best window size for single-core execution with the best window size for 32-core execution. This is because the window-using

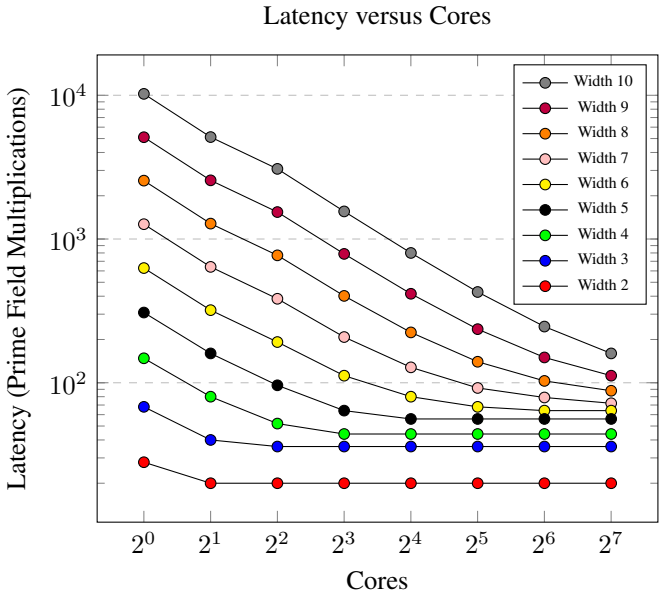

Figure 3.5: Table cost for different numbers of cores and widths.

computation is still serial. It is interesting to note that even without parallelizing that part of the computation, the best window size increases with the number of cores. For a key (scalar) size of 256 bits, Figure 3.7 shows the total cost for the pre-computation and the main computation together, illustrating the trade-off that occurs for various different widths. Thus for 32 cores, it would be better to use a table of bit width 7 or 8, rather than 4. Figure 3.8 shows the advantage from a parallel table when the key size is 512 bits.

One danger with a larger table is if the table is larger than the cache and some of the loads from the table miss the cache, it would be possible for an attacker to get side-channel information from those cache misses. For example, while the Montgomery ladder performs the same amount of computation regardless of the input, it is still possible on certain processors to perform a side-channel attack using information from the cache, since the memory lookups are not the same [YB14]. Another consideration with large tables is the comb method, which would normally be used for situations where the input point P is fixed across many calculations. It calculates a different table in advance, but that table is large enough that performance improvements only occur after several multiplications. With the capacity to calculate large tables efficiently, the comb method might possibly do better than the windowed method [HV04].

Basu calculates a table for a non-constant windowed method in parallel, but his estimate has doubles and add with all the same cost, making performance estimation much simpler. The table only requires $2P$ and odd multiples of P up to $2^w - 3P$, but he includes even multiples up to $2nP$, where n is the number of cores, so that parallelism is possible. The calculation of $2P$ to $2nP$ happens on one core, and the calculations from $(2n+1)P$ to $(2^w - 3)P$ spread out nicely on n cores. For his setup, the cost is $2n - 1 + \lceil \frac{2^{w-1}-n-1}{n} \rceil$ adds or doubles, but after putting it through the brute force schedule, we notice that $2P$ to $2nP$ can be partly parallelized for a total cost of $\lceil \log_2(2n-1) + \frac{2^{w-1}-n-1}{n} \rceil$ adds or doubles [Bas12].

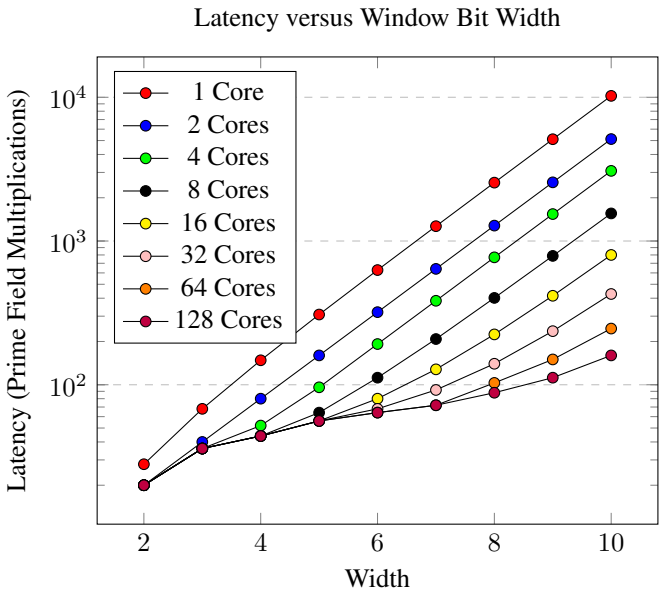

Figure 3.6: Table cost for different widths and numbers of cores.

3.2 Main Computation

In order to allow parallelism in the main computation, Equation 2.1, shown below for the reader's convenience, must be modified to a different form that still calculates the same dP.

$$\begin{aligned} dP &= d_0 P + \sum_{i=1}^{D-1} 2^{w_s+(i-1)w} d_i P \\ &= d_0 P + 2^{w_s}(\ldots(d_{D-2}P + 2^w(d_{D-1}P))\ldots) \end{aligned} \tag{2.1}$$

These modifications may do redundant or duplicate operations in order to improve parallel performance. We present three methods of introducing parallelism, and combine them to produce the final speedup.

3.2.1 Symmetric table

Since we can very cheaply negate a point by negating its y-coordinate, calculating a table of 0 to $2^w P$ costs almost exactly as much as calculating a table of $-2^w P$ to $2^w P$. Methods that use the NAF or other heuristics to maximize the number of 0-bits or 0-digits also take advantage of a symmetric table to look up the resulting negative digits [HV04], but those methods are not constant-time, so we cannot use them. Instead, the advantage here is that the secret key d can be divided into larger digits, reducing the overall number of digits. This means that the width used in calculating the table is 1 less than the width of the digits, so we use w_d and w_t to distinguish them, where $w_d = w_t + 1$. For any digit d, we first subtract 2^{w_t}, then look it up in the table, and finally add $2^{w_t} P$. The cost of a single addition is 16 modular multiplications with the cost of the subtraction and table lookup

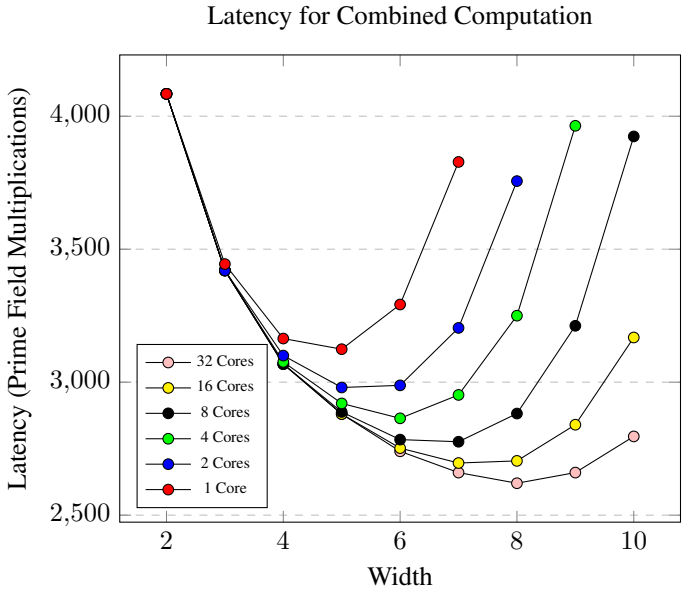

Figure 3.7: Cost of table computation for different widths and numbers of cores, and main computation on one core.

considered negligible.

$$dP = (d_0 - 2^{w_t})P + 2^{w_t}P + \sum_{i=1}^{D-1} 2^{w_s+(i-1)w_d}((d_i - 2^{w_t})P + 2^{w_t}P) \tag{3.1}$$

In serial, the cost of this calculation is

$$16 \cdot (2 \cdot \left\lceil \frac{l}{w_d} \right\rceil - 1) + 8 \cdot (l - w_d),$$

and it needs to include another 8 modular multiplications, because the table needs to include the calculation of $2^w P$. The extra cost compared to the shortest serial version is

$$16 \cdot (2 \cdot \left\lceil \frac{l}{w_d} \right\rceil - 1) + 8 \cdot (l - w_d) + 8 - 16 \cdot (\left\lceil \frac{l}{w_t} \right\rceil - 1) - 8 \cdot (l - w_t)$$

which simplifies to

$$16 \cdot (2 \cdot \left\lceil \frac{l}{w_d} \right\rceil - \left\lceil \frac{l}{w_t} \right\rceil).$$

Thus, this scheme requires more computation. However, the extra addition of $2^{w_t}P$ can be put in parallel with the rest of the computation, and the timing can be improved over the initial serial version. The amount of computation that can be put onto the second core is

$$16 \cdot (\left\lceil \frac{l}{w_d} \right\rceil - 1).$$

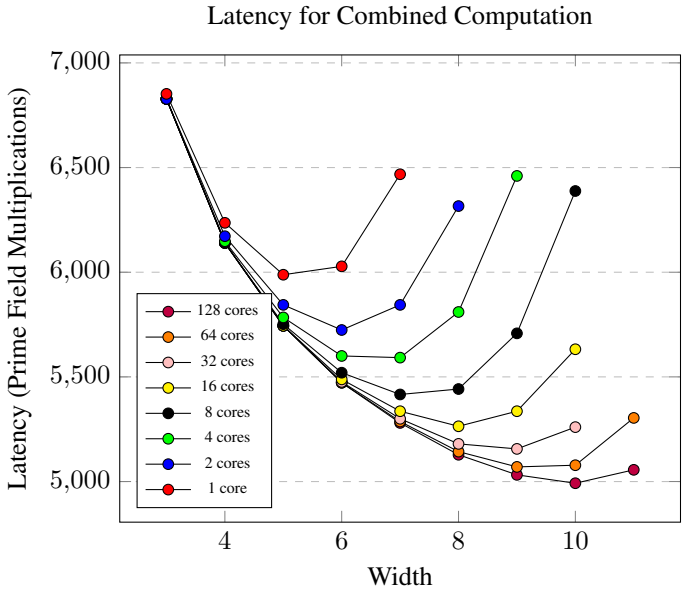

Figure 3.8: Cost of table computation for different widths and numbers of cores, and main computation on one core. This is for a key of size 512 bits.

So the total amount of time saved is

$$16 \cdot (\left\lceil \frac{l}{w_d} \right\rceil - 1) - 16 \cdot (2 \cdot \left\lceil \frac{l}{w_d} \right\rceil - \left\lceil \frac{l}{w_t} \right\rceil),$$

simplified as

$$16 \cdot (\left\lceil \frac{l}{w_t} \right\rceil - 1 - \left\lceil \frac{l}{w_d} \right\rceil).$$

Figure 3.9 compares the costs of a serial main calculation (Postive table: 2 core table, 1 core main; Positive table: 1 core table, 1 core main), with the extra cost of using the symmetric table (Symmetric table: 1 core table, 1 core main), and finally the parallel version with the symmetric table (Symmetric table: 2 core table, 2 core main).

For the best width, 5, the cost reduces from 3140 to 2848, an improvement of 10%. The first core is busy for the entire time of the main computation, while the second core is only busy part of the time. For width 5, this is 2688 and 672 modular multiplies, respectively. The balance is much better for the table, where the cost of 160 represents an almost even split with a gap of 8 at the beginning, and 4 at the end, so that the second core has 148 modular multiplies. In total, 29% of the runtime has 2 cores running simultaneously, while the other 71% is on one core only.

3.2.2 Multiple tables

To introduce more parallelism, we split the summation formula from Equation 2.1 into 2 parallel pieces.

$$dP = d_0 P + \sum_{i=1}^{D-1} 2^{w_s+(i-1)w} d_i P \tag{2.1}$$

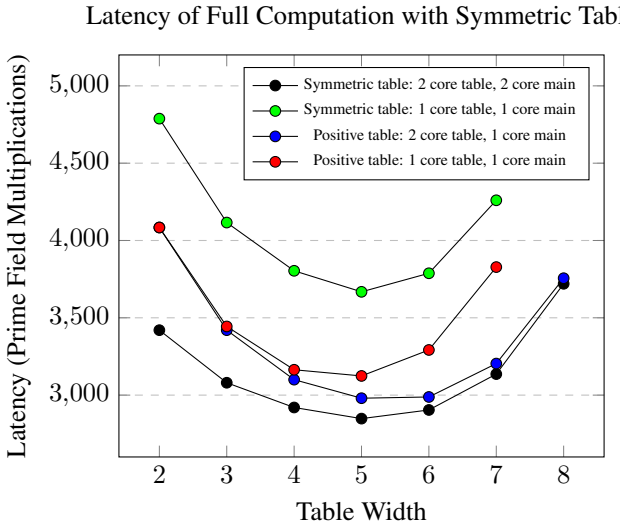

Figure 3.9: By expanding the table to include the negatives, we attain a slight performance improvement.

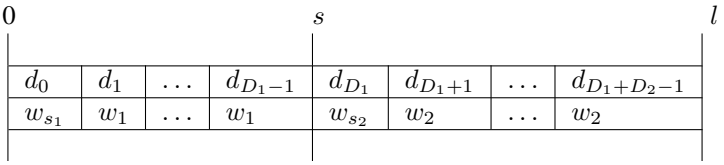

Figure 3.10: Division of the scalar into digits

We split the secret key at some point s. There are s bits of the scalar on the lower side, and $l - s$ on the upper side. The bits of the scalar are laid out into digits of sizes w_1,w_2, etc. as shown in Figure 3.10.

$$dP = \left(d_0 P + \sum_{i=1}^{D_1-1} 2^{w_{s_1}+(i-1)w_1}(d_i P)\right) + \left(d_{D_1} 2^s P + \sum_{i=1}^{D_2-1} 2^{w_{s_2}+(i-1)w_2}(d_{D_1+i} 2^s P)\right) \quad (3.2)$$

where

$$\begin{aligned} D_1 &= \left\lceil \frac{s}{w_1} \right\rceil, \\ D_2 &= \left\lceil \frac{l-s}{w_2} \right\rceil, \\ w_{s_1} &= s - (D_1 - 1)w_1, \\ w_{s_2} &= (l-s) - (D_2 - 1)w_2. \end{aligned}$$

Rather than calculating $d_i 2^s P$ at every digit in the second half, we use a separate table to look up each digit. This table's values are $0, 2^s P, \ldots, (2^{w_2} - 1)2^s P$ (or $-2^{w_2}2^s P, (-2^{w_2} + 1)2^s P, \ldots, 2^{w_2}2^s P$ if it includes negatives.) The extra computation here is the calculation of $2^s P$ (which is s doubles or $8s$), plus the cost of the extra table calculation, which is $12 \cdot (2^{w_2} - 2)$. Note that since the initial point of the second table is not the input point, we do not have it in affine coordinates, and this table costs slightly more. The total amount of computation done is

$$10 \cdot (2^{w_1} - 2) + 12 \cdot (2^{w_2} - 2) + 16 \cdot (D_1 + D_2 - 1) + 8 \cdot (l + s - w_1 - w_2).$$

Put in parallel, the latency is

$$16 + \max\left\{10 \cdot (2^{w_1} - 2) + 16 \cdot (D_1 - 1) + 8 \cdot (s - w_1), 12 \cdot (2^{w_2} - 2) + 16 \cdot (D_2 - 1) + 8 \cdot (l - w_2)\right\}.$$

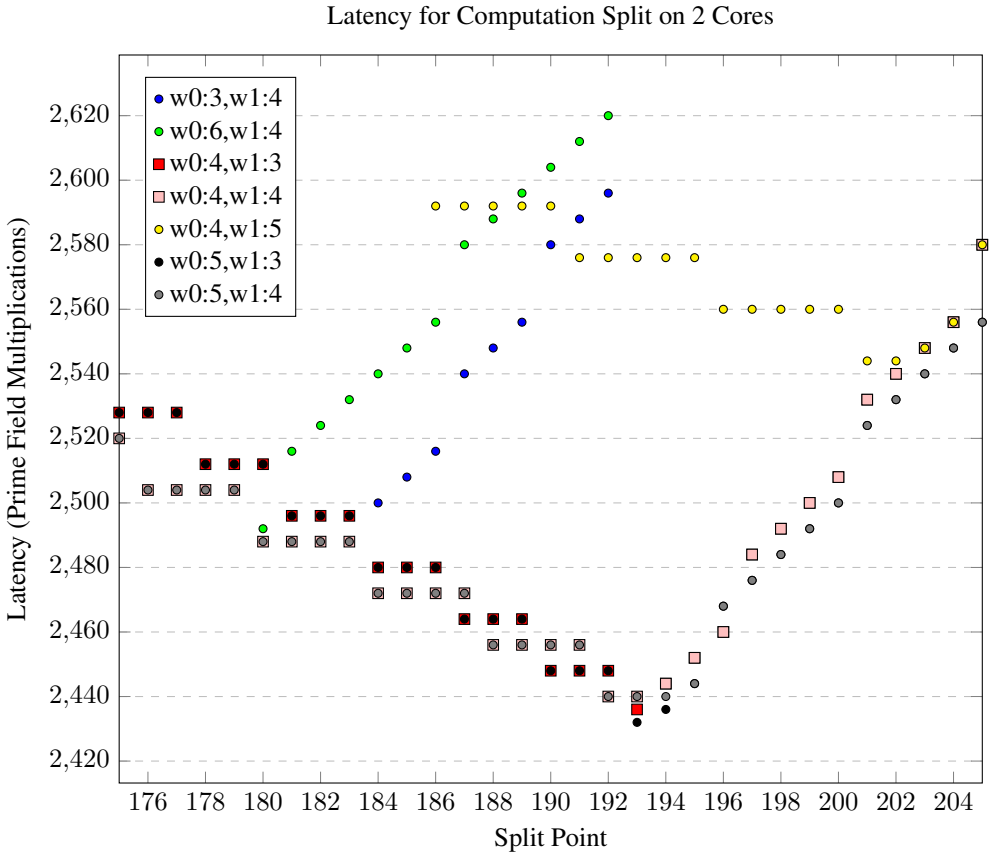

Figure 3.11: By splitting the main computation across two cores, the performance improves, depending on the split point and table widths.

Figure 3.11 compares the costs of choosing different widths for the two tables and split points. The best performing schedule is 2432 modular multiplies, when the first width is 5, the second width is 3, and the split point is 193. The total amount of computation is 4824 modular

<table>
<tr><td>Core 1</td><td>Core 2</td><td>Core 3</td></tr>
<tr><td rowspan="2">Table 2,3:
$8 \cdot s_1$</td><td colspan="2">Table 1:
$5 \cdot 2^{w_1} - 4$</td></tr>
<tr><td>idle</td><td rowspan="3">Main 1:
$16 \cdot (D_1 - 1) +$
$8 \cdot (s_1 - w_1)$</td></tr>
<tr><td>Table 3:
$8 \cdot (s_2 - s_1) + 12 \cdot (2^{w_3} - 2)$</td><td>Table 2:
$12 \cdot (2^{w_2} - 2)$</td></tr>
<tr><td rowspan="2">Main 3:
$16 \cdot (D_3 - 1) + 8 \cdot (l - s_2 - w_3)$</td><td>Main 2:
$16 \cdot (D_2 - 1) + 8 \cdot (s_2 - s_1 - w_2)$</td></tr>
<tr><td colspan="3">Addition: $16 \cdot 2$
The first addition can occur while one of the cores is still running.</td></tr>
</table>

Figure 3.12: Work assignment with three tables.

multiplications, and is split almost evenly across the two cores, with only four modular multiples more than the final point addition not happening in parallel. According to this estimate, there are a few other possible schedule parameters that would result in similar performance, so a good implementation would try out several of these options.

Using the same concept, we can use 2 split points, s_1 and s_2 to expand to 3 tables. The calculations are laid out as in Figure 3.12. The total amount of computation done is

$$10 \cdot (2^{w_1} - 2) + 12 \cdot (2^{w_2} - 2 + 2^{w_3} - 2) + 16 \cdot (D_1 + D_2 + D_3 - 1) + 8 \cdot (l + s_2 - w_1 - w_2 - w_3).$$

Using the fact that the order of the two final additions can vary, there are 3 possibilities for the latency, and we choose the minimum.

$$total = 32 + \min_{i \in 0..2} \left(\max_{j \in 0..2} (c_j - 16 \cdot \delta_{ij}) \right) \tag{3.3}$$

where

$$\begin{aligned}
c_0 &= 12 \cdot 2^{w_3} - 2 + 16 \cdot (D_3 - 1) + 8 \cdot (l - w_3), \\
c_1 &= 12 \cdot 2^{w_2} - 2 + 16 \cdot (D_2 - 1) + 8 \cdot (s_2 - w_2), \\
c_2 &= 5 \cdot 2^{w_1} - 4 + 16 \cdot (D_1 - 1) + 8 \cdot (s_1 - w_1), \\
D_1 &= \left\lceil \frac{s_1}{w_1} \right\rceil, \\
D_2 &= \left\lceil \frac{s_2 - s_1}{w_2} \right\rceil, \\
D_3 &= \left\lceil \frac{l - s_2}{w_3} \right\rceil, \\
\delta_{ij} &= \text{if } i = j \text{ then } 1 \text{ else } 0, \\
w_1 &> 2,\ 8 \cdot s_1 > 5 \cdot 2^{w_1} - 4.
\end{aligned}$$

By brute force searching through all the possibilities, we find that the best parameters are $w_1 = 5, w_2 = 3, w_3 = 3, s_1 = 184$ or $185, s_2 = 235$. Both require 5244 modular multiplications

cycle	Core 1	Core 2	Core 3
0	$2^s P$	Make Table 1	
152		idle	Use Table 1
1488	Make Table 2		
1584			
2192	Use Table 2		
2216	Addition	idle	idle
2232	Done		

Figure 3.13: Work assignment in blocks to three cores.

and have a latency of 2216 modular multiplications. Compared to the serial version of 3164, this is 1.7x the amount of computation for a 41% improvement in the latency.

Putting the method with two tables together with the symmetric table method uses a total of 4 cores. However, the 2 cores which add the extra $2^{w_d} P$ at each digit are not that busy, because there is only one addition that needs to happen at the same time that the addition and w_d doubles occur. So these 2 cores can be merged together for a computation that runs on 3 cores. The latency of that method is

$$16 + \max\{10 \cdot 2^{w_{2_t}} - 12 + 16 \cdot (D_2 - 1) + 8 \cdot (l - w_{2_d}), 5 \cdot 2^{w_{1_t}} + 16 \cdot (D_1 - 1) + 8 \cdot (s - w_{1_d})\}$$

where $w_{1_t} > 1$. The best case occurs with parameters $w_{1_d} = 6, w_{2_d} = 4, s = 204$, with 2308 modular multiplies.

Due to the amount of time spent on calculating $2^{sw_d} P$, there is also room on the 3 cores to split one of the main computations again, further improving the performance.

$$\Big(\dots d_0 + 2^{w_d}(d_1 P + 2^{w_d}(\dots)) \Big) + 2^{sw_d} \Big(\dots d_{D-1} P + 2^{w_d}(d_D P) \dots \Big)$$

Figure 3.13 shows how the calculations are laid out to achieve a performance of 2232. The total amount of computation done is 5220 modular multiplications, which is split across the cores as 2216, 2184, and 820, respectively. Comparing this to the fastest serial computation, of 3164 modular multiplications, it takes 30% less time by doing 1.6x more computation.

Figure 3.14 shows how the performance is affected by the choice of split point 1 and 2. To find the best one, we try out different values.

3.2.3 Split With One Table

Similar to the multiple table method, we split the summation formula into 2 parallel pieces. We again split it at some point s, which is the split point. However, this does not require a second table, because 2^s happens once at the end of the second core portion of the calculation. We use the same digit layout as shown in Figure 3.10, except that since there is only one table, there is only one width, $w_1 = w_2$.

$$dP = \Big(d_0 P + \sum_{i=1}^{D_1 - 1} 2^{w_{s_1} + (i-1)w} (d_i P) \Big) + 2^s \Big(d_{D_1} P + \sum_{i=1}^{D_2 - 1} 2^{w_{s_2} + (i-1)w} (d_{D_1 + i} P) \Big) \qquad (3.4)$$

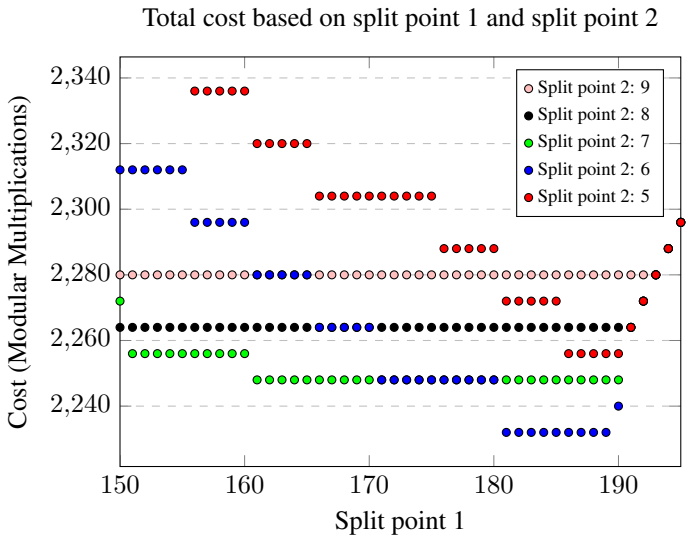

Figure 3.14: Finding the optimal split point 1 and 2 by trying different values.

where

$$D_1 = \left\lceil \frac{s}{w} \right\rceil,$$
$$D_2 = \left\lceil \frac{l-s}{w} \right\rceil,$$
$$w_{s_1} = s - (D_1 - 1)w,$$
$$w_{s_2} = (l-s) - (D_2 - 1)w.$$

Thus the total amount of calculation done is

$$10 \cdot (2^w) - 20 + 16 \cdot (D_1 + D_2 - 1) + 8 \cdot (l + s - w_1 - w_2).$$

and the latency is

$$5 \cdot 2^w - 4 + \max\{16 \cdot (D_1 - 1) + 8 \cdot (s - w_1), 16 \cdot (D_2 - 1) + 8 \cdot (l - w_2)\} + 16.$$

The table calculation happens on 2 cores, reducing its latency to $5 \cdot 2^w - 4$, for $w > 2$, and the two pieces of the main computation happen in parallel, with the latency being whichever piece takes longer.

Figure 3.15 compares the costs of choosing different widths and split points. From the graph, the best performance is 2348 and happens with a width of 4, and a split point 192. The total amount of computation is 4668 modular multiplications. The load balancing between the two cores is almost even, with a gap of 16 modular multiplies for the final sum at the end, and 12 modular multiplies in the table. This means that for 99% of the computation time, both cores are active. Compared to the best serial time, the performance improvement is 35%.

A similar method by Basu splits the secret evenly among the cores, and does the extra s doubles at the end on one core, along with the final adds [Bas12]. Equation 3.4 still describes this

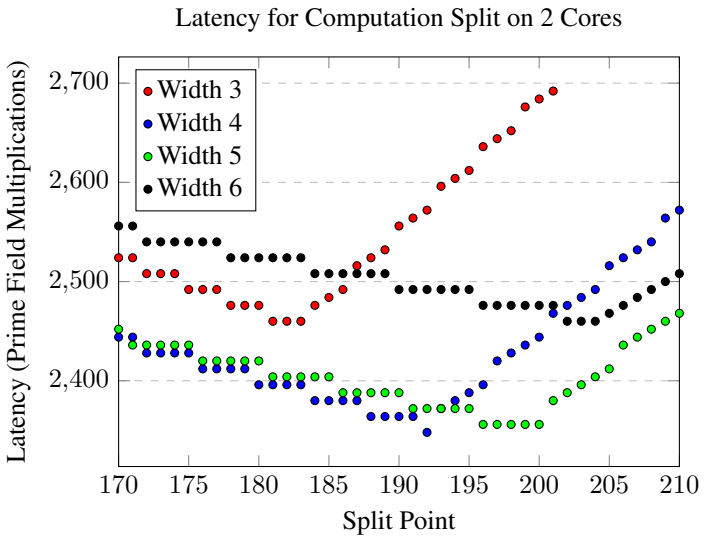

Figure 3.15: By splitting the main computation across two cores, the performance improves, depending on the split point and table width.

setup. Adapting his method for a comparison, we find the total amount of computation is

$$10 \cdot (2^w) - 20 + 16 \cdot (2 \cdot D - 1 + n - 1) + 8 \cdot (l + s - 2 \cdot w).$$

where

$$s = \left\lceil \frac{l}{n} \right\rceil,$$
$$D = \left\lceil \frac{s}{w} \right\rceil,$$

and the cost in parallel is

$$5 \cdot (2^w) - 4 + 16 \cdot (D - 1 + n - 1) + 8 \cdot (l - w)$$

for $w > 2$. For 2 cores, the best performing width is 5 and the total cost is 2580 modular multiplies. Due to the extra constraint that the split point be at 128, the load is not as well balanced between the two cores, and the performance is not quite as good.

Chapter 4

Implementation and Performance

4.1 Simulated Implementation in Haskell

This project was developed in Haskell to take advantage of an interpreter for elliptic curve operations and for ease of implementing heuristic scheduling with greedy phases. This made it easy to test the table computations and the applications of the tables in the multi-window computation. This allowed us to test for correctness of the underlying algorithm before attempting to schedule it.

The interpreter uses a class `Prime` to encode the information for specific elliptic curves. For the purpose of this project, we implement a `NISTP256` instance to store curve parameters. There is no way to construct an element of type `NISTP256`. Instead, the instance `Prime NISTP256` holds the information. Then a second class, `CryptType` indicates different repre-

```
data NISTP256
```

sentations of prime field and elliptic curve elements, including interpretation in Haskell and representations of code in lower-level languages. For the Haskell interpreter, the type is `CryptINTERP`. This instance uses the Haskell `Integer`s, which are arbitrary precision integers. Since `Integer` is flexible about the size of the number is represents, the number of leading 0 bits in 256-bit number affects the timing of operations on those `Integer`s [HI].

This already disqualifies the `CryptINTERP` interpreter from being a secure implementation, because timing information can leak out the size of the integers used in a computation. Nevertheless, the interpreter can give useful indicators about the timing, and test that we get correct answers from the implementation.

We implement most of the prime field operations in a fairly straightforward manner, making an instance of the `Num` class so that the regular arithmetic operators are available. One operation that does not fit into the `Num` class is the multiplicative inverse, so it is implemented separately.

These prime field operations make implementing the elliptic curve operations quite simple. The `EllipticCurve` class contains these operations, so that the type system can easily keep track of other implementations that might be necessary for other curves.

```
data CryptINTERP
```

```
instance (Prime p) => CryptType CryptINTERP p where
  data IDX CryptINTERP p = IDXI Int
    deriving (Eq, Ord, Show)
  data NUM CryptINTERP p = NUMI Integer
    deriving (Eq,Ord,Generic,NFData)
  data JAC CryptINTERP p = JACI (Integer, Integer, Integer)
    deriving (Eq,Generic,NFData)
  data AFF CryptINTERP p = AFFI (Integer, Integer)
    deriving (Eq,Generic,NFData)
```

```
instance forall p . (Prime p, CryptType CryptINTERP p) => Num (NUM
 ↪  CryptINTERP p) where
  (NUMI x) * (NUMI y) = NUMI $ (x * y) `mod` (thePrime (undefined
   ↪  :: p))
  (NUMI x) + (NUMI y) = NUMI $ (x + y) `mod` (thePrime (undefined
   ↪  :: p))
  (NUMI x) - (NUMI y) = NUMI $ (x - y) `mod` (thePrime (undefined
   ↪  :: p))
  negate (NUMI x) = NUMI $ thePrime (undefined :: p) - x
```

4.2 Parallel Implementation in Haskell

Given this model of the computation, we proceed to implement in parallel in Haskell, to do a preliminary test of the performance improvement. The improvement is in comparison to the standard, single core windowed method, also implemented in Haskell. Haskell provides `MVar` as the basic communication method among threads.

Given an implementation which performs a scalar multiplication on 3 threads, we check how the program actually uses the resources. With an AMD A10 Elite Quad-Core, there are cores available for all of the threads. ThreadScope is a program which analyzes parallel Haskell programs for their performance [TSW]. We used ThreadScope to help debug synchronization problems, and verify that computation was being distributed across cores. Unfortunately, it also shows a large amount of overhead on core 1, which is already scheduled to be the busiest core. Table 4.1 shows both real performance numbers and cost in terms of modular multiplies for the regular windowed method compared to running the 3-core parallel method on 1 core.

```
pointDoubleJJ :: forall ct p . (CryptType ct p, Num (NUM ct p),
 ↪  MkNum (NUM ct p))
                 => (NUM ct p, NUM ct p, NUM ct p) -- ^ The input
                  ↪  point in Jacobian Coordinates
                 -> (NUM ct p, NUM ct p, NUM ct p) -- ^ The doubled
                  ↪  point in Jacobian Coordinates
pointDoubleJJ (x1,y1,z1) =
  let
    delta = z1*z1 -- 0.67
    gamma = y1*y1 -- 0.67
    beta = x1*gamma  -- 1
    alpha = 3.*(x1-delta)*(x1+delta) --1
    x3 = (alpha*alpha)-8.*beta -- 0.67
    z3 = ((y1+z1)^(2::Int))-gamma-delta --0.67
    gamma2 = gamma*gamma --0.67
    y3 = alpha*(4.*beta-x3)-8.*(gamma2) --1
  in
    (x3,y3,z3)  -- total cost using squaring: 6.35
```

```
class EllipticCurve ec where
  pointDouble :: ec -> ec
  pointAdd :: ec -> ec -> ec
  pointNegate :: ec -> ec
  zero :: ec

instance (CryptType ct NISTP256,Eq (JAC ct NISTP256)
         ,NFData (JAC ct NISTP256),Eq (NUM ct NISTP256)
         ,Num (NUM ct NISTP256),MkNum (NUM ct NISTP256))
  => EllipticCurve (JAC ct NISTP256) where
  pointDouble a = (nums2jac . pointDoubleJJ . jac2nums) a
```

	estimate (mod. mults, ratio)	real time (s, ratio)
serial	3104, 1	11.680, 1
3 core parallel, run on one core	5244, 1.69	19.024, 1.63

Table 4.1: Performance results for the regular windowed method and 3 core version, all run on one core.

Chapter 5

Discussion

We have shown that there are many opportunities to reduce the latency of scalar multiplication in elliptic curves. We have used the number of multiplications in the Galois field as the primary unit of computation, ignoring other operations and overhead. Given that multiplication takes hundreds of cycles, including other computation would not change our recommendations. Overhead can significantly degrade performance, but in our case, we know the sequence of tasks required, which makes overhead easier to avoid.

There are a range of platforms which need to perform cryptographic operations, from high-throughput servers to desktops, laptops and mobile devices. On servers, we expect to have large numbers of cores and sophisticated thread scheduling in the operating system. In this case, using a thread pool and message queues with a fixed communication pattern would allow computation to be distributed across cores efficiently. Since the computation is not dependent on the data, we are reasonably confident that it is not subject to a side-channel attack like SPA, and a few extra measures, such as randomization of the private exponent can secure it against prevent DPA. If processor load varies significantly, the width of the algorithm could be tuned to the environment, from one computation to the next, trading latency off for efficiency as necessary. There is no need to dynamically schedule the computation. It would be relatively straightforward to implement a parallelization strategy on top of existing technology such as OpenMP, with either implicit or explicit synchronization, or Go using coroutines with communication via channels. On the smallest devices (which still have multiple threads), this support may be lacking, but the same structure as with coroutines could be implemented by using atomic memory accesses to communicate the availability of required inputs.

One side effect of using branch-free implementations to avoid side-channel attacks is that computation time is completely deterministic. Because multiplication in the Galois field dwarfs other computation at the same level of abstraction, the computation is also conveniently chunked into multiples of this time, which makes it easier to line up the computation to make spin locks efficient.

5.1 Related Work

Using addition chains or addition-subtraction chains has applications in other contexts. Another example in cryptography is using addition chains to calculate powers for RSA. In the case of calculating powers, it is not helpful to include subtraction in the chains, since division is significantly more expensive than multiplication [MO90, BC90]. Like many other methods, using chains of ad-

ditions and subtractions in an efficient manner dependent on the data leads to non-constant time calculation, although there is a proposal by Oswald to get around this with randomization [OA01].

Izu and Takagi parallelize Montgomery's scalar multiplication method by doing the add and double at each step in parallel [IT02]. This has a total cost of 1 double and 255 adds, or 4088 modular multiplies, across two cores. That still costs more than the serial windowed method with width 4, at 3164 modular multiplies. Instead, the advantage is that it does not require the extra memory to store the table for the windowed method. Basu parallelizes a windowed method that takes advantage of 0-bits. However, the idea of parallelism transfers to the constant-time windowed method, though it does not achieve as much speedup [Bas12]. Brickell, Gordon, McCurley and Wilson parallelize their fixed-point windowing method which performs well only when the point P is constant, and d is the only new input over multiple iterations. However, when P changes at every multiply, a fixed-base windowing method does not perform as well as the basic windowed method described in this thesis [BGMW95].

5.2 Future Work

The windowed method as already presented works well for scalar multiplication in the general case, with a new input point every time, and a single multiplication every at a time. However, ECDSA signature generation does not calculation the signature based on a new P every time. Instead, it calculates based on the agreed upon point. The windowed method, and our proposed parallel version of it do not take advantage of the fact that P is always the same.

ECDSA signature verification does two scalar multiplications instead of one, as in $u_1P + u_2Q$, and this also allows for simultaneous multiple point multiplication, also known as Shamir's trick [HV04]. The simultaneous multiple point multiplication depends on similar pre-computed tables and windows on the scalar, so future work could apply the same techniques of parallelism to that case, and probably see similar results.

While this thesis uses a short Weierstrass curve as an example, other curves can use the same method. For example, Koblitz curves, which can take advantage of the Frobenius endomorphism to improve performance, still end up performing a scalar multiplication that is well suited to windowed methods [GLV01], so future work could also investigate optimal parameters for that case.

Chapter 6

Conclusion

Initially, we showed that it is possible to parallelize the table pre-computation portion of the windowed method for elliptic curve scalar multiplication, and that it significantly improves the latency. Next, we considered block schedules, and found further expected efficiencies by scheduling computation using Table 1 in parallel with the calculation and use of Table 2. In future work, we hope to more optimally minimize idle blocks in our current schedules.

In addition to estimating performance, we have implemented a synchronization scheme using MVars in Haskell. This allows us to use the existing interpreter for elliptic curve operations. Results from the interpreter show that the schedule produces the correct answer. However, it does not give reliable performance information, probably due to overhead. This gives us confidence to implement the computation in Go using a similar mechanism.

Bibliography

[Bas12] Saikat Basu. A new parallel window-based implementation of the elliptic curve point multiplication in multi-core architectures. *Group*, 16(4a3):27b2, 2012. 13, 21, 27

[BC90] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 400–407, New York, NY, 1990. Springer New York. 5, 26

[BGMW95] Ernest F Brickell, Daniel M Gordon, Kevin S McCurley, and David B Wilson. Fast exponentiation with precomputation: algorithms and lower bounds. *Preprint, Mar*, 1995. 27

[BIYA20] Tanya Bouman, Yusra Irfan, James You, and Christopher K. Anand. Parallel windowed method for scalar multiplication in elliptic curve cryptography. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, CASCON '20, page 237–246, USA, 2020. IBM Corp. 1

[BL] Daniel J. Bernstein and Tanja Lange. Explicit formulas database - jacobian coordinates with a4=-3 for short weierstrass curves. Accessed 27 May 2020. 4

[BvdPSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 75–92, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 5

[Cor99] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *International workshop on cryptographic hardware and embedded systems*, pages 292–302. Springer, 1999. 5

[GLV01] Robert P Gallant, Robert J Lambert, and Scott A Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Annual International Cryptology Conference*, pages 190–200. Springer, 2001. 6, 27

[Gor98] Daniel M Gordon. A survey of fast exponentiation methods. *Journal of algorithms*, 27(1):129–146, 1998. 5

[HI] HI. Basic libraries. `https://hackage.haskell.org/package/base-4.14.0.0/docs/Prelude.html#t:Integer`. (Accessed on 09/18/2020). 23

[HV04] Alfred Menezes Hankerson, Darrel and SA (Scott Alexander) Vanstone. *Guide to elliptic curve cryptography*. New York: Springer, 2004. 1, 2, 3, 4, 6, 13, 14, 27

[IT02] Tetsuya Izu and Tsuyoshi Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography*, pages 280–296, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. 27

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. 5

[MO90] François Morain and Jorge Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *RAIRO-Theoretical Informatics and Applications*, 24(6):531–543, 1990. 5, 26

[Mon87] Peter L Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987. 5

[MS93] Willi Meier and Othmar Staffelbach. Efficient multiplication on certain nonsupersingular elliptic curves. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 333–344, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. 6

[OA01] Elisabeth Oswald and Manfred Aigner. Randomized addition-subtraction chains as a countermeasure against power attacks. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 39–50, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 27

[Rei57] George W Reitwiesner. Summary discussion on performing binary multiplication with the fewest possible additons. Technical report, ARMY BALLISTIC RESEARCH LAB ABERDEEN PROVING GROUND MD, 1957. 5

[Rei60] George W. Reitwiesner. Binary arithmetic. In Franz L. Alt, editor, *Advances in Computers*, volume 1, pages 231 – 308. Elsevier, 1960. 6

[TSW] Threadscope. `https://wiki.haskell.org/ThreadScope`. (Accessed on 05/04/2020). 24

[WSL+19] Licheng Wang, Xiaoying Shen, Jing Li, Jun Shao, and Yixian Yang. Cryptographic primitives in blockchains. *Journal of Network and Computer Applications*, 127:43 – 58, 2019. 1

[YB14] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014. 13