AArch64 optimized implementation for X25519

Emil Lenngren

August 2019

Abstract

Over the years since Curve25519 was introduced, many assembly optimized implementations have been written for different 32-bit ARM processors and Intel processors, but so far, there has been no attempt of writing an optimized version for AArch64. This paper describes how such an implementation was created that performs three field multiplications in parallel utilizing both A64 and SIMD pipelines simultaneously for a Cortex-A53 processor. The results are incredible; one scalar multiplication for a public point is performed in only 145k cycles. This is on par with the Intel x64 implementations and around three times faster than the fastest 32-bit ARM implementations.

1 Introduction

This is an implementation of Curve25519 scalar multiplication [3] for the AArch64 architecture. It is optimized with the ARM Cortex-A53 processor in mind. It should however work fast on most other AArch64 processors as well having the NEON floating point processor.

As is common practice today, the implementation is constructed to be side-channel resistant; either by using code that runs in constant time and has no branches or memory access pattern depending on secret data, or by using blinding.

This document describes some thoughts and comments on the implementation. It is assumed that the reader is already familiar with ECC.

2 Performance

The following cycle counts were obtained on a Cortex-A53 processor.

Operation	Cycles	Code size	Stack needed
Scalar multiplication	145k	5840 bytes	416 bytes

3 Previous work

Daniel J. Bernstein and Peter Schwabe presented an implementation [4] for 32-bit ARM processors utilizing the NEON floating point unit, performing operations in radix-$2^{25.5}$. Adam Langley has written a C implementation called curve25519-donna [2] using the same radix. Tung Chou wrote a parallel implementation for the Intel x64 architecture [5], also using the same radix. This implementation follows the same ideas, but is optimized for the AArch64 architecture.

4 The AArch64 + SIMD instruction set and Cortex-A53

There are many similarities between the 32-bit ARM architecture and the AArch64 architecture. Both are RISC architectures, where each instruction is usually of the form "`op dst, src1, src2`", and many instructions are the same. The syntax is also the same in many cases. There are however important differences. The 32-bit ARM instruction set has 16 general purpose registers (including the program counter and stack pointer). AArch64 has 31 general purpose registers. Program counter, stack pointer and the special "zero register" are separate. AArch64 registers are called `r0` to `r30` in general, but when referenced from code they are called `x0` to `x30` if we want to access all 64 bits and `w0` to `w30` if we want to access only the lower 32 bits. The register `r18` is reserved as a platform register, so this implementation never uses it. Otherwise the other 30 registers are free to use. Note that upon function entry and exit, `r29` and `r30` are used as frame pointer and link register so they must be spilled to the stack if we want to use them.

The SIMD engine in ARM (called NEON) has also changed in AArch64. In 32-bit ARM, there are 32 64-bit SIMD registers named `d0` to `d31`. They can also be viewed as 16 128-bit registers, named `q0` to `q15`, where `q0` is another way of viewing the pair `d0` (lower bits) and `d1` (upper bits), `q1` represents `d2`, `d3` and so on. In the same way the 32-bit registers `s0` to `s31` are another way of viewing the 16 first 64-bit registers `d0` to `d15`. Two of the most important instruction are `vmull.s32` and `vmlal.s32`, which each takes two 64-bit source registers and one 128-bit destination register as operands.

The lower 32 bits in each source register are multiplied and the result is placed in the lower 64 bits of the destination register. The higher 32 bits are multiplied from the source registers and the result is placed in the higher 32 bits of the destination register. The difference between `vmull` and `vmlal` is that the latter adds the result to the destination register rather than just overwriting it.

AArch64's SIMD implementation is similar but also different. There are 32 128-bit registers, named `v0` to `v31`. Each register can be viewed as an array of "lanes", where a lane is either 8-bit, 16-bit, 32-bit or 64-bit. The registers get the suffix `.b`, `.h`, `.s` or `.d`, respectively and are then indexed by `[` and `]`. For example, the upper 32 bits in `v1` are referenced as `v1.s[3]`. In instructions that operate on vectors of lanes, we insert the number of elements after the "`.`", so a half of `v0` can be referenced as `v0.8b`, `v0.4h` or `v0.2s`. The whole `v0` can be referenced as `v0.16b`, `v0.8h`, `v0.4s` or `v0.2d`. All SIMD instructions operate on either the whole vector (128 bits) or a half vector (64 bits). For example, we have the instruction "`umull v0.2d, v1.2s, v2.2s`" which multiplies `v1.s[0]` by `v2.s[0]` and `v1.s[1]` by `v2.s[1]` and places the results in `v0.d[0]` and `v0.d[1]`. The instruction "`umull2 v0.2d, v1.2s, v2.2s`" on the other hand multiplies `v1.s[2]` by `v2.s[2]` and `v1.s[3]` by `v2.s[3]` and places the results in `v0.d[0]` and `v0.d[1]`. There is unfortunately no instruction to multiply `v1.s[0]` by `v2.s[2]` and `v1.s[1]` by `v2.s[3]`, so in a way the new SIMD instruction set for AArch64 can be seen more restrictive than the 32-bit ARM version. Anyway, we also have the umlal instruction which just as in the 32-bit ARM version adds the results to the destination vector rather than overwriting it.

4.1 Dispatching, pipelining and instructions per cycle

The Cortex-A53 can dispatch up to two instructions per cycle. That means, if the combination of two consecutive instructions supports "dual issuing" in a compatible way, we achieve a rate of two instructions per cycle. If there is a data dependency between two consecutive instructions, they can never be dispatched in the same cycle. Compared to more high-end processors, Cortex-A53 does not implement instruction reordering and it is therefore crucial to order the instructions in a way to make sure that two instructions are issued every cycle as much as possible. Fortunately, ARM has released a Software Optimization Guide for Cortex-A55 at [1] which documents all instructions and how different combinations support dual issuing. The Cortex-A55 pipelines are mostly similar to Cortex-A53's, where the main difference is that the single load/store pipeline in Cortex-A53 is replaced by two independent pipelines in Cortex-A55.

After an instruction has been issued, it is fed into a pipeline which performs the operation. There are for example two ALUs, which means that two simple additions can always be performed in parallel during the same cycle. There is however only one MAC (multiply accumulate) pipeline. Two normal (non-SIMD) multiplication instructions can never be dual issued. Some instructions also stall the pipeline, meaning the pipeline gets a throughput of less than one instruction per cycle, like the Multiply High (`umulh`) instruction, which performs a 64$\times$64-bit $\rightarrow$ 128-bit multiplication, discards the lower 64 bits and has a throughput of 1/3 instructions per cycle (3 cycles per instruction). We can also see in the table that the normal 64$\times$64-bit $\rightarrow$ 64-bit multiplication takes a variable number of cycles (2 or 3) depending on the data. Instructions taking variable number of cycles are usually banned for cryptographic calculations, since they can introduce side channels to access the secret keys. That means normal C code that is otherwise written to be constant-time that performs 64-bit multiplications and are compiled to AArch64 will in many cases not be constant-time. Calculating full 64$\times$32-bit $\rightarrow$ 128-bit multiplications with non-SIMD instructions therefore has a throughput of 1/5 or 1/6 instructions per cycle. It is easy to see that performing four 32$\times$32-bit $\rightarrow$ 64-bit multiplications instead and combine the result somehow will lead to higher throughput since it can sustain one 32$\times$32-bit $\rightarrow$ 64-bit multiplication per cycle.

The SIMD pipeline can usually perform only one instruction at a time, if the instruction operate on full size 128-bit vectors, including `umull` and `umlal` multiplication. More simple instructions (such as addition) operating on only half of vectors can usually be performed two in parallel.

The key observation is that a normal A64 (non-SIMD) 32$\times$32-bit $\rightarrow$ 64-bit multiplication can run in parallel with a SIMD `umull` or `umlal` instruction. This means we can perform three 32$\times$32-bit $\rightarrow$ 64-bit multiplication-accumulate instructions per cycle! The latency of the A64 version (`umaddl`) is 3 and the latency of the SIMD version (`umlal`) is 4. However, if the result is only used as the input to the accumulator in the next multiply-accumulate instruction, the latency is 1. According to experimenting, this is true for the SIMD version if the next instruction can be performed in the next cycle (so one A64 instruction can be inserted in between). However, for the A64 version (`umaddl`), the following `umaddl` must be the immediately following instruction (no SIMD instruction in between). If the dedicated forwarding path is missed, the full latency of 3 or 4 cycles will be observed.

Since almost no A64 instruction interferes with the SIMD instruction in terms of dual issuing, the goal is two write two independent program sequences, one A64 part and one SIMD part that can perform independent calculations in the Curve25519 main loop. These two are then being interleaved to make sure two instructions can be performed every cycle.

5 High level algorithm

Algorithm 1 Scalar multiplication of a variable point

```
    INPUT: A scalar $k \in [0, 2^{256} - 1]$ and a point $X1$.
    OUTPUT: $kP$.
 1: $k = k$ AND $2^{255} - 8$
 2: $k = k$ OR $2^{254}$
 3: $X2 = 1$
 4: $Z2 = 0$
 5: $X3 = X1$
 6: $Z3 = 1$
 7: Treat bits in $k$ as $b_0, b_1, \cdots, b_{254}$ from LSB to MSB
 8: $lastbit = 0$
 9: **for** $i = 254$ to 0, step $-1$ **do**
10:     $bit = b_i$
11:     $B = X2 - Z2$
12:     $D = X3 - Z3$
13:     $A = X2 + Z2$
14:     $C = X3 + Z3$
15:     **if** $bit \neq lastbit$ **then**
16:         $F, G = C, D$
17:     **else**
18:         $F, G = A, B$
19:     **end if**
20:     $lastbit = bit$
21:     // In parallel A64 and SIMD:
22:     $AA = F^2$ (A64)
23:     $BB = G^2$ (A64)
24:     $E = AA - BB$ (A64)
25:     $Z4 = E(BB + 121666 \cdot E)$ (A64)
26:     $DA = D \cdot A, CB = C \cdot B$ (SIMD)
27:     $T1 = DA + CB$ (SIMD)
28:     $T2 = DA - CB$ (SIMD)
29:     $X5 = T1^2, T3 = T2^2$ (SIMD)
30:     $X4 = AA \cdot BB, Z5 = X1 \cdot T3$ (SIMD)
31:     $Z2 = Z4$
32:     $X2 = X4$
33:     $Z3 = Z5$
34:     $X3 = X5$
35: **end for**
36: **return** $X2 \cdot inv(Z2)$
```

An important optimization here is that instead of performing conditional swaps (as many implementations do), we perform conditional moves at another step. Line 11, 12, 13, 14, 26, 27, 28, 29, 30 (second half) implement the addition formula, calculating $(X2, Z2) + (X3, Z3)$. Since the order of the two terms is irrelevant, the calculation is not dependent on the scalar bit, and therefore there is no need to perform any conditional swaps. Each loop iteration also implements the doubling formula, calculating $2(X2, Z2)$ or $2(X3, Z3)$, so here we need to conditionally select the input. Note that lines 11-14 are all used for the addition formula, but either 11-12 or 13-14 are needed for the doubling formula. The most efficient way turns out to first calculate A, B, C, D and then select either A, B or C, D as F and G, since F and G are two values. Otherwise we need to select four values $(X2, Z2), (X3, Z3)$ or $(X3, Z3), (X2, Z2)$ in order to calculate all four different lines 11-14.

There are no final conditional swaps or moves since the last bit in the scalar is fixed to be 0. Most of the main loop is also split into two parallel paths, one calculated by SIMD instructions and the other one with simple A64 instructions. These parts are interleaved in order to be dual issued by the CPU.

6 High level overview of field operations

The field used for the Curve25519 curve is $\mathbb{F}_p$, where $p = 2^{255} - 19$. We need to be able to perform fast modular multiplication, squaring, addition, subtraction and inversion. The operations that are of particular interest are multiplication and squaring.

We represent all 255-bit integers in radix $2^{25.5}$, or really alternating 2^{26} and 2^{25}. An integer f modulo $2^{255}-19$ is represented as $f_0 + 2^{26}f_1 + 2^{51}f_2 + 2^{77}f_3 + 2^{102}f_4 + 2^{128}f_5 + 2^{153}f_6 + 2^{179}f_7 + 2^{204}f_8 + 2^{230}f_9$. We use unsigned integers f_i rather than signed, which leads to fast reduction but means slightly slower subtraction. Each integer f_i is stored in a 32-bit value. The benefit of having a few extra bits is to be able to sum more values before the need to perform a carry propagation.

6.1 Adding and subtracting

To add two 255-bit integers f and g, we simply add each f_i with each g_i. To subtract two 255-bit integers f and g, we cannot simply subtract each g_i from f_i since that could result in underflow. Instead, we calculate $(4p-g)+f$, where $4p$ is represented as $\text{0x7ffffb4} + \text{0x7fffffe} \cdot 2^{26} + \text{0x7fffffc} \cdot 2^{51} + \text{0x7fffffe} \cdot 2^{77} + \text{0x7fffffc} \cdot 2^{102} + \text{0x7fffffe} \cdot 2^{128} + \text{0x7fffffc} \cdot 2^{153} + \text{0x7fffffe} \cdot 2^{179} + \text{0x7fffffc} \cdot 2^{204} + \text{0x7fffffe} \cdot 2^{230}$. We can then safely subtract g from $4p$, assuming each limb in g has at most 26 bits.

6.2 Multiplication

The product of $f_0 + 2^{26}f_1 + 2^{51}f_2 + \cdots$ and $g_0 + 2^{26}g_1 + 2^{51}g_2 + \cdots$ is $h_0 + 2^{26}h_1 + 2^{51}h_2 + \cdots$ modulo p where

$$\begin{array}{rrrrrrrrrrr}
h_0 = & f_0g_0+ & 38f_1g_9+ & 19f_2g_8+ & 38f_3g_7+ & 19f_4g_6+ & 38f_5g_5+ & 19f_6g_4+ & 38f_7g_3+ & 19f_8g_2+ & 38f_9g_1 \\
h_1 = & f_0g_1+ & f_1g_0+ & 19f_2g_9+ & 19f_3g_8+ & 19f_4g_7+ & 19f_5g_6+ & 19f_6g_5+ & 19f_7g_4+ & 19f_8g_3+ & 19f_9g_2 \\
h_2 = & f_0g_2+ & 2f_1g_1+ & f_2g_0+ & 38f_3g_9+ & 19f_4g_8+ & 38f_5g_7+ & 19f_6g_6+ & 38f_7g_5+ & 19f_8g_4+ & 38f_9g_3 \\
h_3 = & f_0g_3+ & f_1g_2+ & f_2g_1+ & f_3g_0+ & 19f_4g_9+ & 19f_5g_8+ & 19f_6g_7+ & 19f_7g_6+ & 19f_8g_5+ & 19f_9g_4 \\
h_4 = & f_0g_4+ & 2f_1g_3+ & f_2g_2+ & 2f_3g_1+ & f_4g_0+ & 38f_5g_9+ & 19f_6g_8+ & 38f_7g_7+ & 19f_8g_6+ & 38f_9g_5 \\
h_5 = & f_0g_5+ & f_1g_4+ & f_2g_3+ & f_3g_2+ & f_4g_1+ & f_5g_0+ & 19f_6g_9+ & 19f_7g_8+ & 19f_8g_7+ & 19f_9g_6 \\
h_6 = & f_0g_6+ & 2f_1g_5+ & f_2g_4+ & 2f_3g_3+ & f_4g_2+ & 2f_5g_1+ & f_6g_0+ & 38f_7g_9+ & 19f_8g_8+ & 38f_9g_7 \\
h_7 = & f_0g_7+ & f_1g_6+ & f_2g_5+ & f_3g_4+ & f_4g_3+ & f_5g_2+ & f_6g_1+ & f_7g_0+ & 19f_8g_9+ & 19f_9g_8 \\
h_8 = & f_0g_8+ & 2f_1g_7+ & f_2g_6+ & 2f_3g_5+ & f_4g_4+ & 2f_5g_3+ & f_6g_2+ & 2f_7g_1+ & f_8g_0+ & 38f_9g_9 \\
h_9 = & f_0g_9+ & f_1g_8+ & f_2g_7+ & f_3g_6+ & f_4g_5+ & f_5g_4+ & f_6g_3+ & f_7g_2+ & f_8g_1+ & f_9g_0
\end{array}$$

The 10 32×32-bit $\rightarrow$ 64-bit multiplications in each row are summed to produce h_i. Since h_i can be up to 64 bits (assuming the worst case, where both f and g are results of a subtraction as calculated in the previous section where 0 has been subtracted from a 26-bit value, h_0 could result in 64 significant bits), we need to reduce each h_i to 25 or 26 bits. This is done by simply carrying over excessive bits from h_i to h_{i+1}, followed by h_{i+1} to h_{i+2} etc. Each such operation is performed using a shifted add instruction to add the excessive bits to h_{i+1} followed by an and instruction that clears the excessive bits in h_i. When carrying over from h_9 to h_0, we multiply the excessive bits by 19 in order to reduce modulo p. If we start at any even i, and after this operation has been done 11 times, h_{i+1} will be less than 2^{26}. All other h_j where j is odd will be less than 2^{25} and all h_j where j is even will less than 2^{26}. This will be enough for subsequent operations.

6.2.1 SIMD multiplication

Since every `umull` and `umlal` instruction performs two calculations in parallel and the SIMD idea is to vectorize calculations, we calculate two full field multiplications in parallel rather than trying to parallelize within a field multiplication. This also means the algorithm does not get any different depending on how many field multiplications are done in parallel.

We are going to use the instructions `umull`, `umlal`, `mul`, `shl`, `usra`, `and` and `bic`. The `mul` instruction is used to perform 32-bit multiplication by 19. The `shl` instruction is used to shift left by 1. The `usra` instruction is used for right-shifted add. The `and` and `bic` instructions mask out bits.

The "multiplication by 19 and add" is done by first clearing the lower 25 bits in a copy of h_9 followed by three right-shifted adds ($16+2+1=19$). The reason for using a sum of shifted adds is due to a 64-bit multiply instruction is missing.

The first input operand's limbs are stored in registers `v0` to `v9`. The second input operand's limb are stored in registers `v10` to `v19`. The result limbs are stored in `v20` to `v29`. In `v30.d[0]` and `v30.d[1]` we store a mask of 26 bits set to 1 (bit 0 to 25).

In the "table" above that shows how each h_i is to be calculated, we have multiple multiplications by 2, 19 and 38. To save multiplications and instructions, the calculations by 19 are performed on the g limbs, overwriting g_i with $19g_i$. We therefore process the table "bottom $\rightarrow$ up" so that we do not overwrite values we need later. For even numbered rows, each odd column must be multiplied by 2 (or 38 which is $19 \cdot 2$). We therefore multiply and accumulate the odd columns first, shift the sum by 1 to the left and continue with the even columns. What is left now is to figure out how to avoid the pipeline stalls. For example, when we have processed the odd columns and need to shift the result before we continue, instead of waiting four cycles for the result we defer the shift and work on another row and go back when that new row's work needs to be deferred. We start the reduction from h_2 and interleave the rest of the reduction with the calculation of h_0 and h_1 in order to not stall the pipeline for the reduction (`usra` has latency 3). We cannot make the whole calculation pipeline stall free near the end, so we fill those holes with a head start of the following high level operation that does not depend on the result. In total 141 instructions are needed, plus 5 "holes" near the end that can be filled.

6.2.2 A64 multiplication

The A64 implementation needs to be a bit different than the SIMD version. While the instructions generally have lower latencies than their SIMD-variants (notably shifted add), there are fewer registers available. We also get latency 3 from the multiplication-accumulate instructions.

The idea is therefore two work with three or four rows at a time (h_9 to h_6, h_5 to h_3 and h_2 to h_0) and process them column by column. Odd columns are processed first so that the even rows can halfway through the calculation be multiplied by two.

The reduction is started at h_4 (not h_3 since we need to start at an even numbered row) and is performed without any interleaving of other operations down to h_9 thanks to the shifted add instruction has latency 2 and not 3 as in the SIMD case. During reduction, two consecutive result limbs are compressed into a single 64-bit register in order to save registers. Fortunately no extra instructions are needed for this, since this is done using the `bfi` instruction, which is used instead of the `and` instruction to mask the result. When we arrive to the reduction from h_9 to h_0, we leave the lowest 26 bits in h_9 (instead of 25) and the excessive bits need therefore be multiplied by 38 instead of 19 when adding to h_0. The trick here is to first multiply by 19, and accumulate this value together with the odd numbered columns in h_0, which is then multiplied by 2. This way we save a temporary register and one add instruction. After the rows h_0 to h_2 are completed, we can finally perform the rest of the reduction. The whole field multiplication operation is performed in 140 instructions.

6.3 Squaring

If f is equal to g in the multiplication operation, we only need to calculate 55 products instead of 100 since many products are duplicated. The calculations are shown in the table below.

$h_0 =$	$f_0 f_0 +$					$19f_5 2f_5 +$	$19f_6 2f_4 +$	$19f_7 4f_3 +$	$19f_8 2f_2 +$	$19f_9 4f_1$
$h_1 =$	$f_0 2f_1 +$						$19f_6 2f_5 +$	$19f_7 2f_4 +$	$19f_8 2f_3 +$	$19f_9 2f_2$
$h_2 =$	$f_0 2f_2 +$	$f_1 2f_1 +$					$19f_6 f_6 +$	$19f_7 4f_5 +$	$19f_8 2f_4 +$	$19f_9 4f_3$
$h_3 =$	$f_0 2f_3 +$	$f_1 2f_2 +$						$19f_7 2f_6 +$	$19f_8 2f_5 +$	$19f_9 2f_4$
$h_4 =$	$f_0 2f_4 +$	$2f_1 2f_3 +$	$f_2 f_2 +$					$19f_7 2f_7 +$	$19f_8 2f_6 +$	$19f_9 4f_5$
$h_5 =$	$f_0 2f_5 +$	$f_1 2f_4 +$	$f_2 2f_3 +$						$19f_8 2f_7 +$	$19f_9 2f_6$
$h_6 =$	$f_0 2f_6 +$	$2f_1 2f_5 +$	$f_2 2f_4 +$	$f_3 2f_3 +$					$19f_8 f_8 +$	$19f_9 4f_7$
$h_7 =$	$f_0 2f_7 +$	$f_1 2f_6 +$	$f_2 2f_5 +$	$f_3 2f_4 +$						$19f_9 2f_8$
$h_8 =$	$f_0 2f_8 +$	$2f_1 2f_7 +$	$f_2 2f_6 +$	$2f_3 2f_5 +$	$f_4 f_4 +$					$19f_9 2f_9$
$h_9 =$	$f_0 2f_9 +$	$f_1 2f_8 +$	$f_2 2f_7 +$	$f_3 2f_6 +$	$f_4 2f_5$					

If we successively precalculate $2f_1$, $2f_2$, $2f_3$, $\cdots$, $2f_9$, $19f_9$, $19f_7$, $19f_5$, $19f_6$, $19f_8$, $4f_1$, $4f_3$, $4f_5$, $4f_7$, these values together with all f_i values are enough to compute all 55 products. Since there is only one input operand to the squaring operation, we have access to more temporary registers than when we performed field multiplication.

When we perform squaring with A64 instructions, we process the table, with a few exceptions, column by column (a column is processed from top to bottom) from left to right. This way a register containing $4f_i$ or $19f_i$ can overwrite register f_i since it is not needed anymore. In total, the A64 squaring operation requires 99 instructions.

When we perform squaring with SIMD instructions, we first process the first left five columns, but row by row, starting from the bottom, and $19f_9 2f_9$. The reduction is then started from h_8, interleaved with precalculating values for the right five columns. At last, these last five columns are calculated, row by row, from top to bottom, interleaved with the reduction. In total, the SIMD squaring operation requires 98 instructions.

6.4 Inversion

The last step is to perform field inversion in order to be able to calculate $X2/Z2$. To make it constant time, we use Fermat's little theorem, like most other Curve25519 implementations.

The inner loop here consists of repeated field squaring. There are few opportunities for parallelizing here. Calculating some of the 55 products with A64 instructions and some with SIMD instructions might first seem like a good idea, but the overhead passing the data between these two register sets turns out to be too big. In the end, an A64-only implementation turned out to be most efficient. Since the multiply pipeline can only sustain one operation per cycle, the best strategy is to perform one of the 55 multiplications every cycle, interleaving other work (multiplying by 2, 4 and 19, reduction, loop counting, branching) to be done to other pipelines. For example, multiplying by 19 is performed using shifted adds although it requires more instructions. After hard puzzling, an optimal strategy was finally found that causes no pipeline stalls, register spills or extra instructions. This means the squaring loop performs one squaring every 55 cycles.

7 Conclusion

This paper shows that it is possible to get a 3x speedup by using carefully written parallel assembler code, compared to using a relatively optimized C implementation with a good compiler. Due to the complexity of arranging vectorized SIMD instructions requires a thoroughly thought high-level structure of the whole operation, it is hard to see that general C compilers would be able to transform a generic Curve25519 into highly efficient SIMD code in the nearest future. Processors where parallelization is done automatically by executing multiple instructions in parallel that also implement instruction reordering fit the C language better since the compilers do not need to be as "smart" as when the parallelization strategy must be coded explicitly into the instruction as needed by SIMD instructions. This means we need assembler for many cryptographic operations for AArch64 when we need to maximize the performance.

References

[1] Arm® cortex®-a55 software optimization guide. `https://static.docs.arm.com/epm128372/20/arm_cortex_a55_software_optimization_guide_v2.pdf`.

[2] curve25519-donna. `https://github.com/agl/curve25519-donna`.

[3] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[4] Daniel J. Bernstein and Peter Schwabe. Neon crypto. In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'12, pages 320–339, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Tung Chou. Sandy2x: New curve25519 speed records. Cryptology ePrint Archive, Report 2015/943, 2015. `https://eprint.iacr.org/2015/943`.