University of Southern Denmark

Computer Science

BADM500 - Bachelorprojekt i datalogi

Karatsuba and Curve25519 on RISC-V

Author

Michael Saabye Salling
misal18
29-01-1994

External censor

Claudio Orlandi

Supervisor

Ruben Niederhagen

June 1, 2021

Contents

1 Abstract

I denne rapport vil jeg beskrive hvordan Karatsuba er blevet brugt til at optimere udregningerne brugt til Curve25519 som er blevet portet over til RISC-V ISA'en. RISC-V er en open source `open standard instruction set architecture` også kaldet ISA. Det er altså en mere åben tilgang til CPU design.
Karatsuba er en hurtig multiplikations algoritme som kan bruges til store integer værdier.
En reference implementation af Curve25519 er blevet ported til RISC-V og bliver simuleret på PQVexRiscV CPU'en.
Der er kodet nogle test scripts til host maskinen og den simulerede CPU som bruges til at måle antal cyklusser det tager at køre krypteringen. Udfra disse resultater kan der laves en evaluering.
Karatsuba implementationen er 0.676 megacycles hurtigere end reference implementationen. Selvom en optimering på 0.676 megacycles er minimal, er det stadig hurtigere end den originale implementation, samt lægger et godt grundlag til at fortsætte med nogle andre optimeringer.

2 Introduction

A essential part of computing is having a secure cryptographic algorithm. Theese algorithm's has become a crucial part of messaging, and network systems, hence the focus on making these algorithms faster. The goal of this project is to port Curve25519 to RISC-V, and optimize it for the VexRiscv platform.

Curve25519 is a elliptic cure algorithm released by Daniel J. Bernstein in 2005. The algorithm was originally defined as a Diffie-Hellman function. ECC uses smaller keys than RSA while providing the same level of security, with faster key generation. A ECC public key contains a pair of integer coordinates *(x, y)* on the curve. A elliptic point can be compressed to a single X coordinate, which is used by Curve25519.

The Elliptic-curve Diffie-Hellman is a key agreement scheme, is a classical use case which allows two parties to establish a shared secret with 2 different public/private key sets. Eliptic-curve Diffie-Hellman (ECDH) is very similar to a classical Diffie Hellman, the difference is that ECDH uses *ECC point multiplication* where the classical approach uses *modular exponentiation*.

Karatsuba is an multiplication algorithm which is faster than typical schoolbook multiplication.

The ISA RISV-V has been increasing in popularity as it offers a modular and expandable design in a open-source form. The RISC-V implementation VexRiscv is easy to expand, quite modular and optimized for FPGA's.

The report and project files is publicly available on the git repository.
https://github.com/Clonex/RiscV-ECC

3 Background

3.1 Scheme

Curve25519 is an elliptic curve, which is a modern approach of public-key cryptography. It is based on elliptic curves over finite fields. The curves is defined in a Montgomery curve form, which is different from the simple Weierstras form which is typically used in elliptic curves.
A Montgomery form is defined as [6]:

$$(B * y^2) = x^3 + (A * x^2) + x$$

Where Curve25519 is given by:

$$y^2 = x^3 + (486662 * x^2) + x$$

Over the prime field

$$p = 2255 - 19$$

Elliptic-curve Diffie-Hellman (ECDH) is a key agreement scheme, which allows for 2 parties to agree on a shared secret over a insecure channel. ECDH is similar to the classical Diffie Hellman key exchange algorithm, but uses point multiplication instead of modular exponentiation.

$$(a * G) * b = (b * G) * a = secret$$

Both **(a * G)** and **(b * G)** is the public keys of the 2 parties, **a** and **b** being their private keys. Therefore a shared secret can be calculated with only the other parties public key, and its own private key.

3.2 RISC-V

The RISC-V ISA is a open standard instruction set which is free and uses a open license. The license allows for development of closed and open-source implementations for commercial and non-commercial use.
The standard is based on reduced instruction set computer (RISC) principles, which is defined as:

- **Single-cycle execution**
  RISC focuses on single-cycle execution.
- **Hard-wired control, little or no microcode**
  Microcode can add unnecessary overhead to a instruction. It can make a single instruction require multiple cycles.
- **Simple instructions, few addressing modes**
  It avoids complicated addressing modes and instructions involving microcode.
- **Load and store, register-register design**
  It only loads and stores access memory. Every other operation is performed register-to-register.
- **Efficient, deep pipe-lining**
  One method to achieve hardware parallelism it to use pipe-lining. A pipeline keeps n instructions active at the same time, and loads the next when its finished.

Instruction set architecture (ISA) defines the the instructions which the architecture must support. A ISA also specifies instructions for handling data and memory operations, and how data is encoded in the registers.

The RISC-V ISA has modular design with different ISA bases which defines instructions for integers and weak memory ordering, and a number of extensions developed in a collective efforts with researchers and companies. A modular approach allows the implementors to disable extension which is not needed, and save money on space and power used [2].
Extensions are actively being developed, with some being in a `frozen` state. Which is defined as not expected to change during the ratification process [9].

A company might not find the extensions they need for their product. In that situation a

custom ISA extension can be used, to add their special instructions. This doesn't break the compliance with the main specification, which means that software made for the main specification will still work on the extended ISA.

3.3 Multiplication

Different multiplication approaches has existed since the ancient Egyptians [1]. Lattice multiplication is another method of multiplication, which uses a lattice to multiply two numbers, This method has been taught sine medieval times and is still being taught today. In more recent years efficient algorithms which can be used on computers is more of a focus. Efficient algorithms have been created depending on the size of the numbers, one such algorithm is the Karatsuba algorithm,

4 Optimization

4.1 Karatsuba

as multiplication is used intensely during the curve's algorithm steps, this is an interesting place to focus on optimizing. The reference implementation of the curve uses simple schoolbook multiplication.

In this method every single digit is multiplied with the corresponding digit in the other number, and a carry is carried on to the next iteration when the result is bigger than the radix. This carry will then be multiplied with the next set of digits.
For example, if we wanted to multiply $23 \cdot 35$.

$$a = 29$$

$$b = 35$$

Then it would find the product of every digit in `a` and `b`, with its correct decimal position. Then find the sum of all the products.

$$\begin{array}{r} (30 \cdot 20) \\ (30 \cdot 9) \\ (5 \cdot 20) \\ \underline{+\ (5 \cdot 9)} \\ = 1015 \end{array}$$

The schoolbook approach requires n^2 single-digit products. This approach is the approach used in the reference implementation, which can be replaced with a more optimal algorithm.

The Karatsuba algorithm is used to perform multiplication of large numbers with fewer operations, than the school-book multiplication approach. This method can be performed in $O(n^{\log_2 3}) \approx O(n^{1.58})$.
Using schoolbook multiplication can be represented as such, where $a_0 + a_1 x$ and $b_0 + b_1 x$ are each a polynomial.
$(a_0 + a_1 x)(b_0 + b_1 x) = a_0 b_0 + (a_0 b_1 + a_1 b_0)x + a_1 b_1 x^2$
This computation needs 4 products. The Karatsuba approach can do this multiplication with only 3 products.
$(a_0 + a_1 x)(b_0 + b_1 x) = a_0 b_0 + 2^m((a_0 + a_1))(b_0 + b_1) - a_0 b_0 - a_1 b_1) + 2^{2m} a_1 b_1)$
This approach requires 5 products, with 2 of them being the same multiplications. For this reason 2 of them can be stored and them re-used without any calculations.

An example would be as if the target was to multiply 2973 with 3572

$$a = 2973$$

$$b = 3572$$

Then the Karatsuba steps would be as following:

1. Compute $a_0 \cdot y_0$ call the result L.
   $L = 29 \cdot 35 = 1015$

2. Compute $a_1 \cdot y_1$ call the result H.
   $H = 73 \cdot 72 = 5256$

3. Compute $(a_0 + x_1) \cdot (y_0 + y_1)$ call the result MM.
   $MM = (29 + 73) \cdot (35 + 72) = 10914$

4. Compute $MM - L - H$ call the result M.
   $MM - L - H = 4643$

5. Compute $L * M^2 + M * M + L$, where M is the digit numbers.
   $10150000 + 5256 + 464300 = 10619556$

The Karatsuba algorithm uses 3 multiplications, 4 additions and 2 subtractions compared to the schoolbook approach which uses 4 multiplications and 3 additions. So the algorithm saves multiplications at the cost of additions and subtractions. Thus is Karatsuba faster if

multiplication is more expensive than additions and subtractions. The recursive implementation stops at a threshold and uses schoolbook multiplication instead.

4.2 Future optimizations

Karatsuba was the only algorithm that was implemented, and tested due to time constraints, however other areas could have been interesting to look into. The Toom–Cook multiplication algorithm is another approach for large integer multiplication, which could have been interesting to experiment with. As squaring is also a significant part of the process, a speedup of squaring could have been interesting as well.

5 Implementation

This section will describe how the reference implementation from Bernstein was ported, and how the Karatsuba algorithm is implemented in practice.

5.1 Reference implementation

The reference implementation was straight forward to port, as the arithmetic itself could be used directly without any changes. This add confidence that the algorithm is still correct.

The `checksum_compute` function used the `random` function given in the C standard library to fill `p` and `n` with random values. In the main implementation `randombytes` is used to fill `p` and `n` with random values, as `randombytes` is supplied with the implementation from the rainbowgit implementation[4], As the C standard library is not available, an alternative to the `random` function is used. Part of the library rainbowgit[4] is used for the `randombytes` implementation, this implementation uses `AES` to give a pseudo-random output. As *AES* isn't supplied either, the libcrypto library[8] is used for its *AES* implementation. It can be found in *src/libcrypto*. It is maintained by the official RISC-V organization, and is actively being developed [8].

5.2 Karatsuba algorithm

To easily allow to enable and disable Karatsuba a C preprocessor definition `ENABLE_KARAT` is used. The Karatsuba algorithm will only be included when the definition is present, otherwise it will use the unmodified reference implementation.

The implementation of the algorithm which is described more in theory in subsection 4.1, and will follow a step by step, similar to the one given in that section. In this example the values

2973 and 3572 is multiplied.

$$x = 2973 = \underset{a}{\boxed{29}}\underset{b}{\boxed{73}}$$

$$y = 3572 = \underset{c}{\boxed{35}}\underset{d}{\boxed{72}}$$

Step 1 and 2, is to compute $a * c$ which is stored in LOW and $b * d$ which is stored in $HIGH$. The recursive step in the code will continue to recursively call itself until the operand length is smaller than the defined `KARAT_L`, and the base case is reached and schoolbook multiplication will be used instead.

```
if(length > KARAT_L)
{
  karat(LOW, &x[0], &y[0], SHIFTED_L);
  karat(HIGH, &x[SHIFTED_L], &y[SHIFTED_L], SHIFTED_L);
}else{
  multSimple(LOW, &x[0], &y[0], SHIFTED_L);
  multSimple(HIGH, &x[SHIFTED_L], &y[SHIFTED_L], SHIFTED_L);
}
```

Step 3 is to compute $(a + b) * (c + d)$ which is stored in MM.

```
addSimple(M0, &x[0], &x[SHIFTED_L], SHIFTED_L);
addSimple(M1, &y[0], &y[SHIFTED_L], SHIFTED_L);

multSimple(MM, M0, M1, SHIFTED_L + 1);
```

Step 4 subtracts the values that was calculated in the last steps. Specifically $MM - LOW - HIGH$, which is stored in MED.

```
subSimple(M0, MM, LOW, length + 1);
subSimple(MED, M0, HIGH, length + 1);
```

Step 5 we simply add LOW, MED and $HIGH$, where the numbers have been shifted correctly.

```
addSimple(out, out, LOW, length);
addSimple(&out[SHIFTED_L], &out[SHIFTED_L], MED, length + 1);
addSimple(&out[length], &out[length], HIGH, length);
```

5.3 Testing

I wanted to be able to test the code with different values, without having to recompile and restart the emulation between every tests. This could be achieved using the serial-functions, supplied by the PQVexRiscV[5] template, which allows the code to communicate with the host.

5.3.1 Device

The embedded device is waiting for the host to send a command, which it acts on. It has 5 different modes that the host put it in.

```
#define MODE_BLANK 0
#define MODE_HASH 1
#define MODE_SEED 2
#define MODE_PING 3
#define MODE_SET_KARAT 4
```

MODE_BLANK is the default mode. When the device is in *MODE_BLANK* it waits for a new mode from the host.

```
      if(mode == MODE_BLANK)
      {
        mode = hal_getc();
      }
```

MODE_HASH initializes the m and n memory with random values, and measures the amount of cycles it takes to run *crypto_scalarmult_base*. Then sends the *checksum* and *cycles* back to the host.

```
        send_start();
        send_string("log", "Hashing..");
        send_stop();

        randombytes(m, CRYPTO_SCALARBYTES);
        randombytes(n, CRYPTO_SCALARBYTES);

        int start = hal_get_time();

        crypto_scalarmult(p, m, base);
        int cycles = hal_get_time() - start;
        checksum_compute();

        send_start();
        send_string("checksum", checksum);
        send_unsigned("cycles", cycles, 10);
        send_stop();
```

MODE_SEED waits for 48 integers from the host, which is then put into the seed memory space, and the seed is set with *randombytes_init*.

```
        for(int i = 0; i < 48; i++)
        {
          int curr = hal_getc();
```

```
            seed[i] = curr;
        }
         randombytes_init(seed, NULL, 256);
```

MODE_PING is used as part of the initialization of the connection with the test-script on the host. When the host connects to the serial port it sends a *MODE_PING* command and waits for a reply to ensure the device is listening.

```
        send_start();
        send_string("ping", "Hello");
        send_stop();
```

MODE_SET_KARAT sets *KARAT_L* which defines the length of the numbers when the Karatsuba algorithm should use schoolbook multiplication.

```
        KARAT_L = hal_getc();
```

5.3.2 Host

On the host a test script to connect to the device was written. The script is made in NodeJS[7], and uses the library SerialPort[3] to establish a connection and communicate with the device. A wrapper function is written to convert the function *port.write(msg, callback)* into a *async* function.

```
function write(message)
{
    return new Promise(r => port.write(message, r));
}
```

When the connection is opened, the *MODE_PING* command is sent to the device.

```
port.on('open', async () => {
    console.log("Opened connection..");
    await write([MODES.PING]);
```

As the data from the embedded device is sent in chunks the script needs to be able to detect when a message has been fully received. As a message always ends with **}** and then **,** the script will look for thoose two characters to detect when a message has been fully received.

```
const disallowed = [32, 10];
let temp = "";
let lastChar = false;
export function dataParser(data)
{
    let done = false;
    for(let i = 0; i < data.length; i++)
    {
        const d = data[i];
        if(d === 44 && lastChar === 125)
        {
            done = true;
        }else if(!disallowed.includes(d))
```

```
        {
            temp = temp + String.fromCharCode(d);
        }
        lastChar = d;
    }

    if(done)
    {
        let msg = temp;
        temp = "";
        lastChar = false;
        return JSON.parse(msg);
    }

    return false;
}
```

The *dataParser* function copies every part of a string into a temporary string, and checks the condition described above. When the message has message has been received it returns a *JSON* object, otherwise it will return *false*.

The function `dataParser` is called when data is received, and only acted upon when the full message has been recieved, and it returns a object.

```
port.on('data', async (data) => {
    let message = dataParser(data);
    if(message)
    {
```

The exsistance of keys on the message object, is used to check what kind of message it is.
A **ping** message is received in response to the **MODE_PING** sent when the connection is opened. This automatically sets the seed and starts a hash.

```
            console.log("Setting seed..");
            await write([MODES.SEED]);
            for(let i = 0; i < 48; i++)
            {
                await write([7]);
            }

            console.log("Starting hash..");
            await write([MODES.HASH]);
```

The **checksum** message is received when a hash iteration is successful, which just prints out the message and replies with the **MODE_HASH** mode.
If the variable *karaI* is set to a number, then it will collect all cycle-counts up to the operand length of 64. This changes at what operand length the Karatsuba should use schoolbook mulplication instead. It does this by waiting for **ping** and **checksum** messages, and trying every value up until 64, and saving the cycles result for every value.

```
            if(message.checksum)
            {
```

```
                karaData[karaI] = message.cycles;
                if(karaI >= 64)
                {
                    console.log("Got results", karaData);
                    return;
                }
                console.log("Got results for", karaI, message);
            }else{
                console.log("Starting..");
            }
            await write([MODES.SET_KARAT]);
            await write([karaI++]);

            await write([MODES.SEED]);
            for(let i = 0; i < 48; i++)
            {
                await write([7]);
            }

            await write([MODES.HASH]);
```

6 Evaluation

An evaluation can me made using the data gotten using the test scripts described in the implementation section 5. All measurements is measured by counting the cycles it takes to run the code.

6.1 Reference implementation

The reference implementation uses Montgomery ladder approach for point multiplication, school-book multiplication, and branch less fixed time approaches to the arithmetic in general. This makes it easier to measure the cycles as it they wont change depending on its input. The reference implementation takes 79.794 megacycles to run.

6.2 Karatsuba

The testing method described in 5.3.2 has been used to find the best operand length for the Karatsuba algorithm. The results of the test scripts has been plotted into a chart, and can be seen in figure 1. Based on those results the best operand length must be 32, as this results in the least cycles.

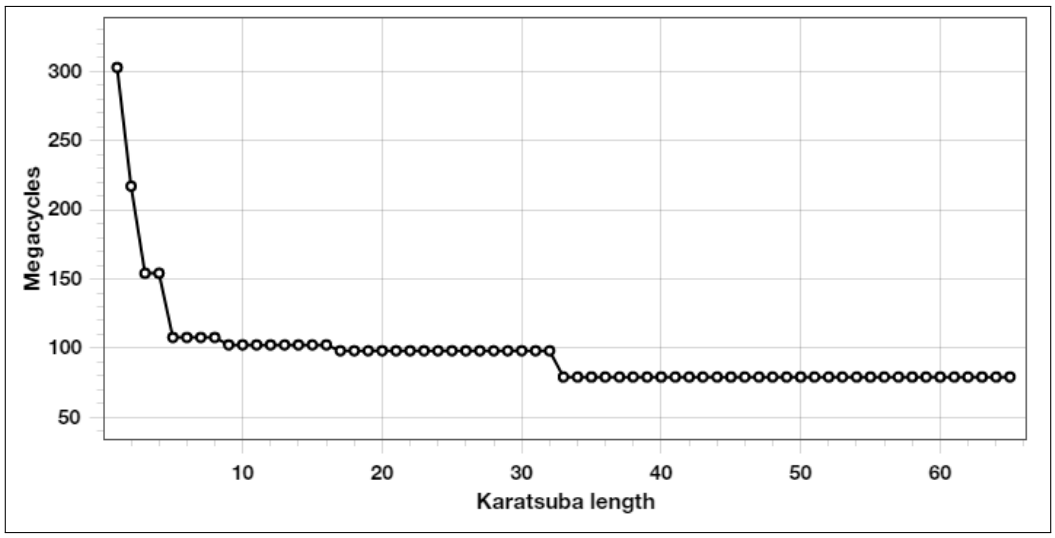

Figure 1: Karatsuba operand length - cycles

Using this operand length results in a measurement of 79.118 megacycles. To compare the Karatsuba cycles with the reference implementation both have been plotted into a chart in figure 6.2. This chart shows the difference between the two implementation is not magnificent, as the difference is 0.676 megacycles.

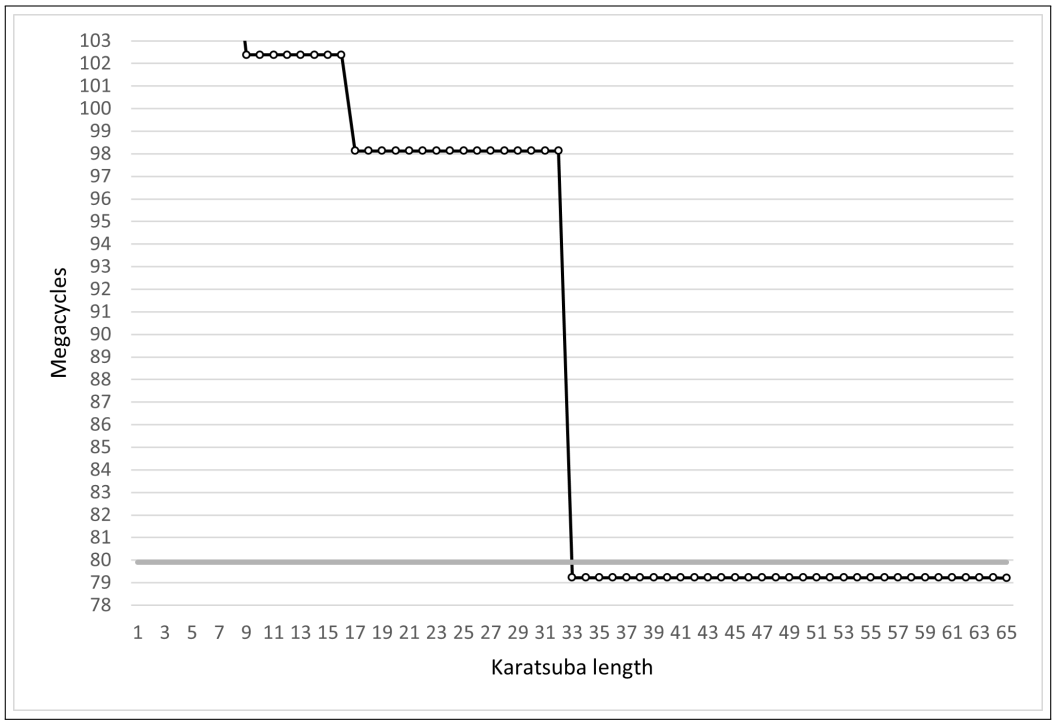

Figure 2: Karatsuba operand lengths (black) compared to reference implementation (grey)

6.3 Testing

Testing was done using the setup described in subsection 5.3. This enables testing of different seeds, Karatsuba operand lengths and multiple iterations of the algorithm.

The current setup only compares the reference implementation with Karatsuba on the emulated CPU. Other kind of tests should have been created as well, to cover more of the range of mistakes that could have happened.

6.3.1 Test with multiple different seeds

A script to automatically generate a bunch of input files, and output files should have been made. Thee files would contain the seed in the input, and n checksums in the output files. Which the test-script then could load in and automatically verify that all the input matched with the output files. This would help verify that the reference and modified implementations matched over multiple different seeds.

6.3.2 Comparing to the local reference implementation

The same input and output files could have been made to compare it with the local reference implementation located in `src/ref/`. In this case the input files would contain p and m used in *crypto_scalarmult* and the output files would contain the checksum.
The test script could then load in theese files just as described in subsection 6.3.1 This would help verify that the ported version matches the reference implementation.

6.3.3 CLI flags

Different CLI flags could have been made for the test-script. This would have allowed to switch between different tests easily, and without having to modify the code itself.

7 Conlusion

The goal of this project was to port was to setup a simulation of the RISC-V ISA using the platform VexRiscv. Then port a reference implementation of Curve22519 to this platform, and optimize and evaluate the optimizations.

A small optimization to the multiplication arithmetic was the only thing implemented due to time constraints. The Karatsuba algorithm implemented is only a nominally amount faster compared to the reference implementation. This could however be a good foundation for further optimizations, or other kinds of multiplication algorithms. A lot of time was used to understand what exactly was going on, and how everything works together in the reference implementation. It is difficult to determine the correctness of the project due to the scope of the project, but does seem like it works as intended from the testing and comparisons that has been done with the reference implementation.

The overall project was successful, even though it wasn't significant improvements in cycles.

References

[1] Alexander Bogomolny. *Egyptian Multiplication*. `https://www.cut-the-knot.org/Curriculum/Algebra/EgyptianMultiplication.shtml`.

[2] Codasip. "Extending RISC-V ISA With a Custom Instruction Set Extension". In: (June 2019).

[3] Francis Gulotta. *Serialport library*. `https://serialport.io/`.

[4] Fast Crypto Lab. *Rainbow-submission-round2*. `https://github.com/fast-crypto-lab/rainbow-submission-round2`.

[5] mupq. *PQVexRiscV*. `https://github.com/mupq/pqriscv-vexriscv`.

[6] Svetlin Nakov. *Practical Cryptography for Developers*. `https://cryptobook.nakov.com/`.

[7] *NodeJS*. `https://nodejs.org/`.

[8] RISC-V. *Riscv-crypto*. `https://github.com/riscv/riscv-crypto/tree/master/benchmarks`.

[9] "The RISC-V Instruction Set Manual". In: (May 2017).