h e p i a
département I T I
ingénierie des technologies
de l'information

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Highly parallel algorithms on GPU with Futhark

Practical case with block ciphers

Bachelor thesis defended by:

Michaël El Kharroubi

for the title : Bachelor of Science HES-SO in

Information technologies engineering with a specialisation in Softwares and complex systems

August 2020

Referent HES teachers

Pr. Foukia Noria & Dr. Malaspinas Orestis

Legend and source of the cover picture : Elder Futhark alphabet https://commons.wikimedia.org/wiki/File:Runic_letters_elder_futhark.svg

Contents

To Severino Zanchettin

Acknowledgments

I want to thank all the people who helped and followed me during this project.

Namely :

- Dr. Orestis Malaspinas, who took this project, was available, helped, and encouraged me to do a bit more every time.
- Pr. Noria Foukia, for joining the project at the last moment, and for the help she provided during this project.
- Dr. Troels Henriksen, who generously answered my questions, and for his insights.
- Théo Pirkl for his template of document, which I used to make this bachelor thesis.

Abstract

The steadily growing need for heavy graphical computations, such as video games or video encoding, requires Graphics Processing Units (GPUs) in almost all high-end laptops and nearly all desktop computers. Currently, in General-Purpose computing on GPU (GPGPU), we either use frameworks for specific applications like TensorFlow for Artificial Intelligence (AI) or Application Programming Interfaces (APIs) like Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) for custom ones. The two laters require a specialization high-level and advanced knowledge of the targeted hardware. In this project, we work with Futhark, an experimental functional language that aims to be parallelism-efficient, while being hardware-agnostic. We implement two cellular automata in Futhark to compare performances against traditional CUDA and OpenCL. We find out that our Futhark code reaches satisfying performances without finely optimizing the code. Indeed, with no Futhark advanced knowledge, our code provide results comparable with CUDA and OpenCL. Then, we develop a cryptographic library prototype implementing AES and Camellia. During this process, we conceive a small library and make it available for the community. It reduces transfer times between the main memory and the GPU up to 40%, provided that we compensate for the extended allocation time with the quantity transferred. Finally, to evaluate our prototype, we make three encryption tool versions for Linux and measure the fastest one against OpenSSL. In this benchmark, we reach satisfying achievements; we obtain comparable performances with AES hardware implementation. Our tool is up to four times faster than OpenSSL implementation of Camellia.

Examinee :
El Kharroubi Michaël
Field of study : Information technologies

Referent teachers :
Pr. Foukia Noria
Dr. Malaspinas Orestis

Glossary

Advanced Encryption Standard Originally Rijndael, it is the National Institute of Standards and Technology (NIST)'s recommended 128-bits block cipher.

AES Advanced Encryption Standard.

AI Artificial Intelligence.

anonymous function A function that does not have an identifier.

API Application Programming Interface.

AVX Advanced Vector Extensions.

backend Target languages/API for Futhark, for instance, CUDA or OpenCL.

binary function A function that takes two parameters.

block cipher Cipher, which operates on a fixed-length of bits.

Camellia A block cipher with 128-bits blocks, made by NTT and Mitsubishi.

CBC Cipher Block Chaining.

Cipher Block Chaining Mode of operation that chains each encrypted block with the previous one, using an exclusive or.

Compute Unified Device Architecture GPGPU API made by NVIDIA, for NVIDIA's products.

CounTeR Mode of operation that, for each block, increments the counter, encrypts the value plus the nonce, and applies an exclusive or with the block.

CPU Central Processing Unit.

CTR CounTeR.

CUDA Compute Unified Device Architecture.

ECB Electronic Code Book.

Electronic Code Book Mode of operation that encrypts each block individually.

FHA Futhark Host Allocation.

FIPS Federal Information Processing Standards.

General-Purpose computing on GPU Use of the GPU to make computation, that are usually done on the Central Processing Unit (CPU).

GFLOPS Giga FLoating-point OPerations per Second.

GPGPU General-Purpose computing on GPU.

GPU Graphics Processing Unit.

higher-order function A function whose arguments and/or return type is a function.

HTTPS HyperText Transfer Protocol Secure.

I/O Input/Output.

MIT Massachusetts Institute of Technology.

mode of operation It is an algorithm used to deal with multiple blocks in block ciphers.

mutex mutual exclusion.

NIST National Institute of Standards and Technology.

NTT Nippon Telegraph and Telephone.

OOP Object-Oriented Programming.

Open Computing Language GPGPU framework/API made by Apple and maintained by the Khronos Group, for multiple platforms.

OpenCL Open Computing Language.

OS Operating System.

PKCS7 Public Key Cryptography Standards #7.

POSIX Portable Operating System Interface uniX.

predicate A unary function that returns a boolean value.

REPL Read Eval Print Loop.

SDL2 Simple Directmedia Layer 2.

Second-Order Array Combinator is designed to resemble familiar higher-order function from other functional languages; it has some restrictions to enable efficient parallel execution.

sequence An ordered list of elements.

SIMD Single Instructions Multiple Data.

SOAC Second-Order Array Combinator.

SSL Secure Sockets Layer.

unary function A function that takes one parameter.

List of illustrations

URL references

URL01 https://github.com/karlrupp/microprocessor-trend-data
URL02 https://github.com/karlrupp/cpu-gpu-mic-comparison
URL03 https://commons.wikimedia.org/wiki/File:No_ecb_mode_picture.png
URL04 https://commons.wikimedia.org/wiki/File:Ecb_mode_pic.png
URL05 https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc

List of tables

Introduction

Today, our processing units tend to be increasingly parallel. Despite that, we still have numerous entirely sequential programs. That is why we decided to evaluate Futhark in a practical case. It is a top-level language designed for efficient parallelism. It offers multiple backends, like for instance, CUDA or OpenCL. Currently, to take full advantage of the hardware, you have to use specific frameworks like TensorFlow for AI, or you have to be a specialist in low-level APIs like CUDA or OpenCL. Sometimes specialists use optimization technics relevant for a family of GPUs, making it a waste of time as soon as a new generation is released. Futhark tries to change that by abstracting some of the hardware specificities, allowing us to focus on the algorithmics. We aim to demonstrate that Futhark provides results comparable with state of the art implementation.

We choose block ciphers encryption as our practical case. The most used mode of operation is Cipher Block Chaining (CBC). We can only use it to sequentially encrypt, but we can concurrently decrypt. CBC is not the only secure mode of operation; some modes are fully concurrent, thus motivating our choice.

To achieve that, we introduce the interest of parallelization by reviewing the trends in current processing units. Then we decide to implement two cellular automata with CUDA, OpenCL, and Futhark. We evaluate the performances one could get with these three tools on a textbook case. Later, we proceed with AES and Camellia. During this process, we create a Futhark short library to minimize transfer times with the GPU. Finally, to ensure our work quality, we validate the implementation using official test vectors published by the NIST for AES and Nippon Telegraph and Telephone (NTT) for Camellia.

The leading resources we used to carry out the first part of this project were Futhark user guide and CUDA documentation. We also exchanged repeatedly with Futhark creator Troels Henriksen. Then for the block cipher implementation, we mostly used Christof Paar's courses and OpenSSL open-source code. As for Camellia specifically, we used the algorithm specifications and the implementation guide published by NTT.

Chapter 1:

The growing interest of parallelization

The growing interest in terms of parallelization, leads us to first define. There are two distinguishable concepts: concurrency and parallelism. According to Oracle's multithreading programming guide(1), concurrency is defined as a state where each task is executed independently; whereas parallelism means that both or more tasks are making progress simultaneously. Note that parallelism implies concurrency while the reverse is not true; for instance, it is possible to execute tasks concurrently with time-slicing. It is a technic used mostly by the Operating System (OS). It gives an illusion of parallelism by regularly switching between tasks, but by definition, it is not parallelism.

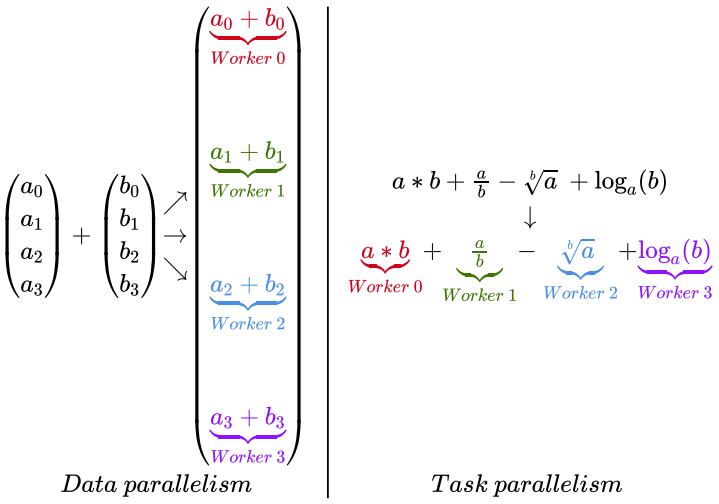


Figure 1.1: An example of task and data parallelism

Source: Made by El Kharroubi Michaël with mathcha.io

A closer look at parallelism, as defined by Michael Hicks(2) shows two sub-types: data-parallelism and task-parallelism. As you can see in figure 1.1, data-parallelism allows tasks to run on different data in parallel, whereas task-parallelism makes tasks run on the same data. For illustration, if we want to watch a video, we can load the file on one side

and displaying the already loaded parts on the other. However, when enlarging a picture, each part is individually enlarged with different tasks, which is data-parallelism.

The Parallelism performance improvement is usually quite significant, but concurrency does not always imply a performance gain. In case of a chain of operations each intermediary output is the next link input; therefore we could not start a new operation without finishing the last one. Thus, we would be back at a sequential execution. A good analogy is a one-kilometer race; if we have ten runners running a hundred meters each, we would be ten times faster, in case of a relay race, there would be no gain. The cost for starting a task and getting the result back once it is finished, needs to be added. We could have the same performance as sequential execution. However, in practice, parallelism and more broadly concurrency imply an overhead. Nonetheless, it is often possible to pipeline the tasks. For instance, when you want to make a massive computation and then send the result over the network, you are not bound to wait for it. Even if those tasks seem mutually dependent, it is possible to start the computation and simultaneously prepare the connection.

1.1. The current situation

As you can see in figure 1.2, even though CPUs keep getting faster at single-thread performances(3), we can see in figure 1.3, that they cannot beat GPUs at massive raw computations(4).

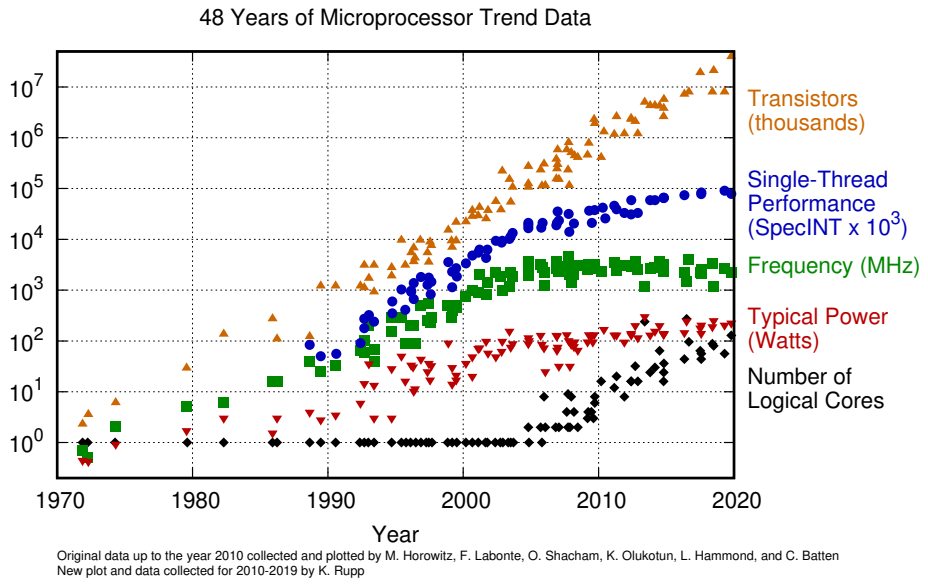


Figure 1.2: Log scale, the evolution of processors specs over 48 years

Source: Generated from Kurt Rupp's data, ref. URL01

In figure 1.2, the blue dots show that single-thread performances keep getting better over time, but at a slower rate. However, we can notice with the black diamonds, that the

number of logical cores keeps augmenting. Through this means, constructors expand the concurrency capacity consequently the total computing capacity.

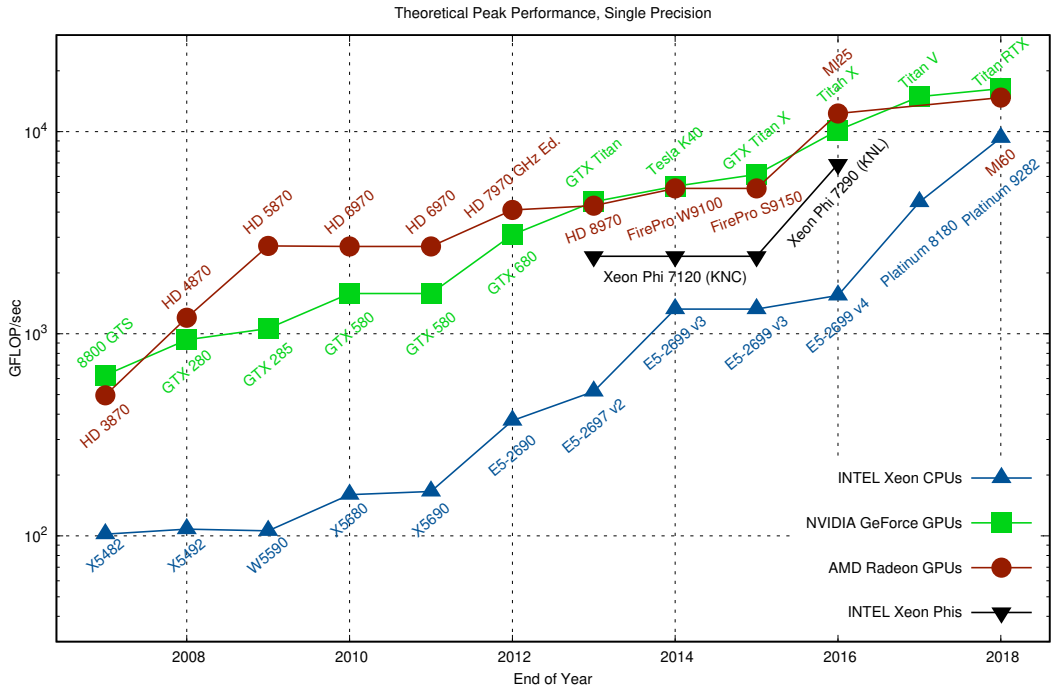


Figure 1.3: Log scale, the evolution of single-precision performances over 11 years.

Source : Generated from Kurt Rupp's data, ref. URL02

As shown in figure 1.3, we remark that GPUs stay in the lead. However, the CPUs seem to close in. Before making any assumption, it is relevant to understand that the GPU leader in 2018 is the Nvidia Titan RTX which costs about 2499$[1], while the CPU leader, the Intel Platinium 9282 would cost between 25'000$ and 50'000$(5), which is 10 to 20 times the Titan price. It gives a cost per Giga FLoating-point OPerations per Second (GFLOPS) of $0,15$$ for the GPU and between $2,68$$ and $6.52$$ for the CPU.

This GFLOPS by dollar difference between the Titan and the Intel Platinium, is due to the fundamental design difference between GPU and CPU. CPUs are a general-purpose computing unit; they are the main computer component. They need to compute any algorithm in decent laps of time, even if it is not parallelizable. GPUs on the other hand, as their name suggests, were initially designed for graphical computation and massively parallelizable algorithms. Since GPUs have a more specific purpose, they are better at it, which justifies the GFLOPS by dollar gap between the Titan and the Intel Platinium.

When you want to access the same data from two different tasks or if there is a dependency, you have to use synchronization mechanisms. Synchronization mechanisms are a common cause of bugs in concurrent programs. The most common ones are deadlocks. They occur

[1] As of July 2020

when at least two concurrent processes wait for one another to release access to the same resource. For example, let us take a resource sharing issue. Beforehand we will define a lock that can only be locked by one person at the time and can not be locked again, as long as the same person has not unlocked it. We will call it mutual exclusion (mutex). Let tell that person A wants to cut an apple wedge with a knife. He first locks the apple's mutex, and he takes the apple. Simultaneously, person B has to do the same thing, with the same resources, but in the inverse order. Firstly, he locks the knife's mutex and tries locking the apple's mutex. That is our deadlock. Person A can not lock the knife's mutex, and person B can not lock the apple's mutex; hence, they will wait indefinitely. Apart from the bugs, it can also cause a drop in performance. Synchronization mechanisms have a cost; that why we should avoid using them as much as possible. For simplicity, we will not present all the synchronization mechanisms here, but the interested reader will find complements in the lecture notes of Jaswinder P. Singh(6).

1.2. Functional programming

Our previous project(7) largely inspired this section. Functional programming is a declarative programming paradigm that became quite popular over time since the late 1960s. As its name indicates, it uses functions, but not just any kind of functions; we will address here "pure functions" and only those. This notion of pure function will be detailed later on.

The essential principle in terms of functional programming is the referential transparency. Considering that we choose Jean-Luc Falcone's definition in his functional programming introduction course : "*An expression is said to be referentially transparent if one can replace each of its occurrences with the result of its evaluation without changing the program's behavior*"(8). To put it in another way, the expression evaluation does not change the system state.

A function that is referentially transparent for given arguments, is a pure function. Such a function is also said to contain no side effects. For illustration, let us take a function that adds two numbers, for two given inputs, the result and the system state never change no matter the state of the function call in the program. On the contrary, a function writing to the disk drive changes the system state and depends on it. Suppose you want to write into a file, the system state change. If there were another function that depended on its content, the program behavior would change. If the file was deleted or the access rights were modified between two calls, there is no guarantee regarding the result's invariability. Therefore writing on the disk can not be referentially transparent, implying that such a function is impure. We thus understand that we cannot make a purely functional practical program; it would have no inputs nor outputs.

This programming paradigm is free of loops such as `while` and `for`; it rathers uses higher-order functions instead. Jean-Luc Falcone defines a higher-order function, in his functional programming introductory class, as "*a function whose arguments and/or return type is a function.*"(9).

The essential higher-order functions in functional programming, are map, reduce, and filter; those are pure functions as long as their arguments are pure functions. We will define those functions guided by the Massachusetts Institute of Technology (MIT)'s software construction course(10). The function map takes two parameters. Firstly, a unary function, that is a one parameter function, and secondly, a sequence, that is an ordered list of elements. Map applies this unary function on each element of the sequence and returns a new sequence with the result in the same order as the input sequence. The filter function also takes two parameters, a unary function that returns a boolean value called a predicate, and a sequence. This higher-order function returns a sequence composed of the elements that are validated by the given predicate. Lastly, the reduce function takes three parameters, a binary function, a function that takes two parameters, an initial value for the accumulator, and a sequence. The reduce function combines the sequence elements by pairs; hence the binary function takes two elements as inputs and returns one element. The reduce function returns a single element. The reduction uses an accumulator. To start this accumulation process, we need an initial value, which is the initial value given as a parameter. These functions are potent, but the reduce one is the most remarkable because it can also replace the filter and map functions.

When we satisfy these requirements and use these functions, we can write a faster, shorter, more understandable code, that approaches the mathematical notation(11). The strong connection between mathematics and functional programming is an asset since it advantages the use of specific mathematical tools, such as factorization, simplification, or composition. One can read functional code as a series of actions. Thus obtaining a more intelligible code, therefore, simpler to correct or modify in the absence of side effects. Such as an evaluation that would modify a variable outside its scope. As a result, functional code is renowned for its robustness and maintainability.

1.3. Functional programming and parallelization

As earlier mentionned, a pure function is referentially transparent and without side-effects. The absence of side effects forbid us to mutate any values outside the function. This eliminates the concurrent modification access risks. Eliminating such risks limits the need for synchronization during the parallel execution. For instance, if we take the map function, we can safely use it for data-parallelism. As a reminder, the map function applies a given foo function to each element of a sequence.

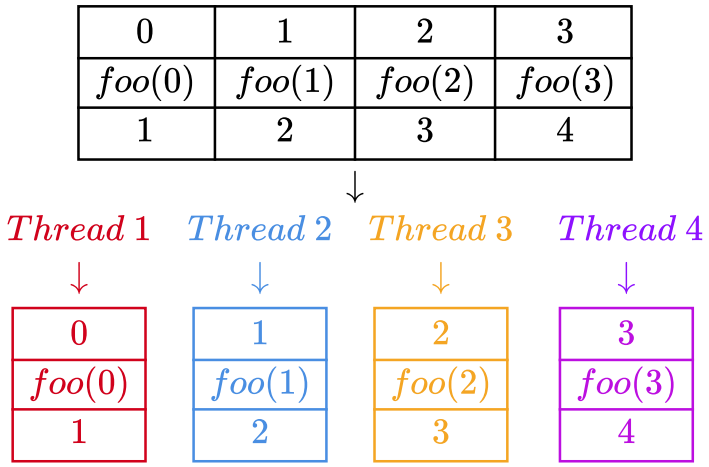


Figure 1.4: An example of data-parallelism with the map function

Source: Made by El Kharroubi Michaël with mathcha.io

There are some catches with this paradigm. For instance, let us take a massive collection with billions of numbers in which we want to double the first. If we follow the referential transparency principles, we can not update the first element alone; it would mean mutating the input, which is a side-effect by definition. In theory, we should make a copy of the collection, which would be a significant drop in performance. In practice, if it is the last usage of the original collection, the compiler can optimize the code which will do an in-place update instead. By fixing such an issue at compile-time, we keep functional programming benefits with the performances of an imperative paradigm.

Chapter 2:

Futhark

Futhark is a purely functional, high-level, statically typed, hardware-agnostic language. It aims to generate an efficient data-parallelis code. Presently, it is at an experimental stage, but it is already capable of compiling complex programs. It frees the developer from synchronization mechanisms. It uses SOACs instead. In Futhark's documentation, they are defined as: "*the basic parallel building blocks of Futhark programming. While they are designed to resemble familiar higher-order functions from other functional languages, they have some restrictions to enable efficient parallel execution.*(12). A SOAC example is map, a fundamental functional programming tool. I will present SOACs in detail later in this document. Futhark is a high-level language with functionalities like function as a variable or anonymous functions, also called lambda expressions, which are functions without an identifier. It works with array transformations; it offers safety mechanisms for array access with bound checks and immutability. It proposes attributes to control the parallelism or bound checks like `#[sequential]`or `#[unsafe]`. It offers an interpreter similar to JShell in Java, called Futhark Read Eval Print Loop (REPL), to experiment with it. Futhark itself does not provide I/O, which can sometimes be challenging. It is possible to compile an executable directly from source code. When possible, it should only be used for testing purposes. A Futhark program should be trans-compiled to a library(13).

Futhark's code should be trans-compiled into another language that we call backend. Futhark possesses multiple backends; currently, the release version[1] proposes five backends. The first backend is sequential C. The second is CUDA. CUDA is the GPGPU API proposed by Nvidia for their GPUs. Third there are OpenCL with the C and Python versions. It is a parallel framework that aims to be hardware-agnostic, OpenCL's code can be compiled for CPUs, GPUs, and other types of processing units(14). The last backend is sequential Python. Since the code is trans-compiled, one has to compile it

[1] As of August 2020, Futhark version: 16.2.

afterward or interpret it in the case of Python. This two-step compilation process can be problematic for the reproducibility(13). For instance, with the C backend, the Futhark trans-compilator generates the same C code[2], but if later compiled with GCC 5.4 and CLang 3.2, it will most likely not be the same executable.

2.1. Relevant examples

This sub-section does not aim to give a complete introduction to Futhark, but instead to highlight some pertinent examples. The first one is a simple code for a function that adds one unit to a given input and returns it.

```
entry add_one (input: i32) : i32 = input + 1
```

Let us take this expression apart; first, we have the keyword `entry`, meaning that we can call this function outside the Futhark code. Then, we have the function name, `add_one`. Following, we have `(input: i32) : i32`. On the left side of the colon, we have the function arguments; here it is a 32-bits integer named "input". On the right side, we have the function return type, which is also a 32-bits integer. In Futhark, a function is declared the same way as a variable declaration with the equal sign in the fourth position. Following, the function body: `input + 1`.

This function is a single line function, but multiple line functions also exist. For instance, to first add one to the argument and then add the argument bitwise inverse to the intermediary result, we would have :

```
let add_one_and_inverse (input:i32):i32 =
    let intermediary = input + 1
    in intermediary + !input -- ! -> bitwise inverse
```

At then end, we have to use the keyword `in` to return the value since there are at least two lines. On single-line functions, it is implicit. The world `in ...` should be understood as *in the expression*. Notice that the keyword `let` is used twice in this snippet. The first one indicates a function declaration, whereas the second one is a variable with `input + 1` value. The keywords `let` and `in` allow us to read the code like a sentence. If I take my function `add_one_and_inverse`, we could read it as: "*Let the function add_one_and_inverse, which takes a 32-bit integer called input, and returns a 32-bit integer, be equal to …*". We have said earlier that a functional language should approach the mathematical notation. In a mathematical statement, we could say: "Let a be equal to b plus one*", which is translated in Futhark as `let a = b+1`.

[2]If the code is trans-compiled with the same version of the Futhark trans-compilator.

Futhark does not currently support recursivity. The documentation indicates: "*Top-level definitions are declared in order, and a definition may refer only to those names that have been defined before it occurs. This means that circular and recursive definitions are not permitted.*". Nevertheless, it is possible to replace recursive functions with loops. For instance, if we take the factorial function, the hypothetical recursive code in Futhark should look like the following :

```
let factorial (n:u32) : u32 = match n
    case 0 -> 1
    case 1 -> 1
    case _ -> n*(factorial (n-1))
```

Good practice avoids simple recursion and favors tail-recursion. Tail-recursion does not overflow the stack; instead of stacking function calls, it uses an accumulator. Our hypothetical tail-recursion function would look like this :

```
let factorial (n:u32) : u32 =
    let tail_factorial (accumulator:i32) (n:i32) : i32 = match n
        case 0 -> accumulator
        case 1 -> accumulator
        case _ -> tail_factorial (accumulator*n) (n-1)
    in tail_factorial 1 n
```

Furthermore, we can see that in our hypothetical tail recursion, we do not have to wait for each recursive call as we had to with our simple recursive function. Instead, we can directly compute the arguments passed to the recursive call, since neither argument depends on the recursive call. Considering that we cannot make any recursion, we use a loop instead, which makes sense, in the absence of tail-recursion. Most modern compilers optimize tail-recursion as a loop equivalent, thus avoiding the overhead of a function call and the risk of stack-overflow. The loop version of our recursion is:

```
let factorial (n:u32) : u32 = match n
    case 0 -> 1
    case 1 -> 1
    case _ -> loop accumulator=1 for x in 2...n do accumulator*x
```

Let us decompose this loop expression. In the first part, `loop accumulator=1` set the accumulator initial value at one. Then we have `for x in 2...n`, which means that the x variable takes values from two to n included. The last part, `do accumulator*x`, means that we update the accumulator value sequentially with every x value multiplied by the accumulator current value. Notice that a loop expression returns the last value of the

accumulator, as we did earlier with our hypothetical tail-recursion.

a) C backend example

To continue, let us take a small function summing the elements of an array with a loop.

```
entry my_sum (array: []i32) : i32 =
    loop accumulator = 0 for x in array do accumulator + x
```

When we trans-compile this code into a C library, we get two files a C file with the generated code and a header file containing the exposed functions signatures. If we take a closer look at this header, firstly, we get the context initialization functions.

```
struct futhark_context_config *futhark_context_config_new(void);
void futhark_context_config_free(struct futhark_context_config *cfg);
...
struct futhark_context *futhark_context_new(struct futhark_context_config
    *cfg);
void futhark_context_free(struct futhark_context *ctx);
```

Later, we have the array related functions:

```
struct futhark_i32_1d *futhark_new_i32_1d(struct futhark_context *ctx,
                                          int32_t *data, int64_t dim0);
...
int futhark_free_i32_1d(struct futhark_context *ctx,
                        struct futhark_i32_1d *arr);
int futhark_values_i32_1d(struct futhark_context *ctx,
                          struct futhark_i32_1d *arr, int32_t *data);
...
```

Finally, we have our entry function, which could be run asynchronously by Futhark's generated code and a function to wait for the result.

```
int futhark_entry_my_sum(struct futhark_context *ctx, int32_t *out0, const
                         struct futhark_i32_1d *in0);
...
int futhark_context_sync(struct futhark_context *ctx);
```

First, we have to initialize a config struct and then initialize the array argument of our Futhark function. We then call our function and synchronize the execution context before using the returned value. Finally, we have to free everything. Which translate into the following code:

```c
//The context is not re-initialized at each function call usually,
//this is a simplified example.
int my_sum_wrapper(int size, int *array)
{
    int result;

    // Initializing a config struct.
    struct futhark_context_config *config = futhark_context_config_new();
    // Initializing a context struct with the config.
    struct futhark_context *context = futhark_context_new(config);

    // Initializing the input array with our arguments.
    struct futhark_i32_1d *input_array = futhark_new_i32_1d(context,
        array, size);

    // Calling our futhark function.
    // We pass the address of the return variable as a parameters.
    futhark_entry_my_sum(context, &result, input_array);

    //Synchronization in case of concurrency.
    futhark_context_sync(context);

    //Cleaning up
    futhark_free_i32_1d(context, input_array);
    futhark_context_free(context);
    futhark_context_config_free(config);

    return result;
}
```

2.2. Parallelization with SOACs

As we have said earlier, Futhark's parallelization rests on SOACs, which are a variant of higher-order functions with some constraints to guarantee efficient parallelism. The main SOACs are map, filter, reduce, all, any, and partition. We could also include scatter, which is close to a SOACs but is not a variant of a higher-order function because it does not take a function as an argument, nor does it returns one.

```
let double (array : []i32):[]i32 = map (*2) array
```

This is an example of Futhark map function. In this case, the function takes an array of integer, denoted `[]i32`, and applies the anonymous function `(*2)`, that multiplies an element by two. This kind of function is called a curried function. The full function would be `(*)`, which is the binary operator multiply. Instead of passing two arguments, we simply create a new unary operator from `(*)` by specifying one of the arguments. Our map's result is a new array where each element is multiplied by two. Furthermore, there are some variants of map, namely, map2, map3, map4, and map5. Those functions take respectively 2,3,4, and 5 arrays and the argument functions take 2,3,4 and 5 arguments. For instance, the following shows how to sum two arrays elementwise: `map2 (+) array1 array2`.

```
let first_five_even : []i32 = filter (\x -> (x%2)==0) (iota 10)
```

Above, we have the filter function. We try to get the first five even numbers and store the result in a homonym variable. To that end, we pass an anonymous function called a predicate. This predicate: `(\x -> x%2==0)`, is true if x is even. In this case, we do not specify an array directly. Instead, we call the `iota` function as an argument of the filter function, and it generates an array from 0 to n excluded.

We define the return of the parity function as true if the number of `true` values is odd. We can compute it with the SOACs reduce as follows:

```
let parity (array : []bool) : bool = reduce (!=) false array
```

To achieve this, we define the anonymous function `!=`, which is a binary operator that means not equal. We pass our accumulator initial value, and then we give the input array as the last argument. This function, unlike the others, has some specific restrictions. The function passed as an argument to reduce must be associative, and have a neutral element. In abstract algebra, it is called a monoid. The accumulator's initial value will be used as the neutral element for the given function(15). If we do not satisfy these restrictions, we will have an undetermined result. Moreover, there are some variants of the reduce function, `reduce_comm` and `reduce_by_index`. The first one adds a restriction; the function given as an argument must be commutative. However, `reduce_comm` is potentially faster(15). The `reduce_index` function behaves almost like the `map2` function, but it also takes an index array to reorganize the first array. It has the same restriction as the `reduce_comm` function.

```
reduce_by_index [5,6,7,8] (+) 0 [3,1,2,0] [10,11,12,13]
```

Above, the first array `[5,6,7,8]` is the source array, `(+)` is the anonymous function, `0` is

the neutral element. Then we have the index array, and the last array contains the values to be reduced with the given function. The expression above does something equivalent to:

```
[
    0 + 8 + 10, -- 8 is the element at index 3
    0 + 6 + 11, -- 6 is the element at index 1
    0 + 7 + 12, -- 7 is the element at index 2
    0 + 5 + 13, -- 5 is the element at index 0
]
```

Those are the principal SOACs. As for the last third main SOACs, `all` and `any` are the optimized version of reductions that we could write as follows:

```
-- && : logical and
let all (input: []bool):bool = reduce (&&) true input
-- || : logical or
let any (input: []bool):bool = reduce (||) false input
```

Concerning the `partition` function, it takes the same arguments as the `filter` function. However, it returns two arrays, one that contains the element corresponding to the predicate and one with the remaining elements. Finally, the `scatter` function is parallelized like a SOAC, but it is not a higher-order function. This function reorganizes a given array using a given index array. For instance, `scatter [0,0] [1,0] [2,1]` returns the array `[1,2]`. The first argument is the destination array. Then we have the index array and finally the array to be reorganized.

2.3. Comparison between CUDA, OpenCL & Futhark with cellular automata

To learn Futhark and evaluate its performance, we took simple algorithms and compared performances against pure CUDA or OpenCL. Constructors made GPUs for data parallelism, they have a lot of small specialized cores. If we want to take full advantage of GPUs capacities, we should divide the work into similar small tasks. For that reason, we had to choose an algorithm that would match these criteria.

Cellular automata are the perfect test case. They consist of a small set of rules that we apply on each cell of a given matrix. We can easily arrange the computational charge by fixing the matrix's number of elements. We choose two cellular automata, *cyclic* and *parity*.

$$\text{neighbors}(c_{i,j}) \;=\; \{c_{i-1 \bmod m,\; j}, c_{i,\; j-1 \bmod n}, c_{i+1 \bmod \;\; m,\; j}, c_{i,\; j+1 \bmod n}\} \qquad \text{with } C_{m\times n}$$

$$parity(c_{i,j}) = \bigoplus\nolimits_{k\in\text{neighbors}(c_{i,j})} k \qquad \text{with } C_{m\times n} = (c_{i,j}) \;\in \{0,1\}^{m\times n}$$

$$cyclic(c_{i,j}) = \begin{cases} (c_{i,j}+1) \mod \mu & \text{if } \exists k \in \text{neighbors}(c_{i,j}) \mid k = (c_{i,j}+1) \bmod \mu \\ c_{i,j} & \text{otherwise} \end{cases}$$

$$\text{with } \mu \in \mathbb{N}^*,\; C_{m\times n} = (c_{i,j}) \in \{a \mod \mu \mid a \in \mathbb{Z}\}^{m\times n}$$

The first one is the parity automaton. The matrix $C_{m\times n}$ has binary values, and the next state of a given cell is an exclusive or of its neighbors. The second one is the cyclic automaton. The matrix contains integer values modulo μ a parameter that we can fix, and the next state of a given cell, in a state $c_{i,j}$, is $(c_{i,j}+1) \bmod \mu$, if at least one of the neighbors has the value $(c_{i,j}+1) \bmod \mu$. The complexity of those cellular automata is $O(n)$, where n is the number of matrix elements.

a) The implementation

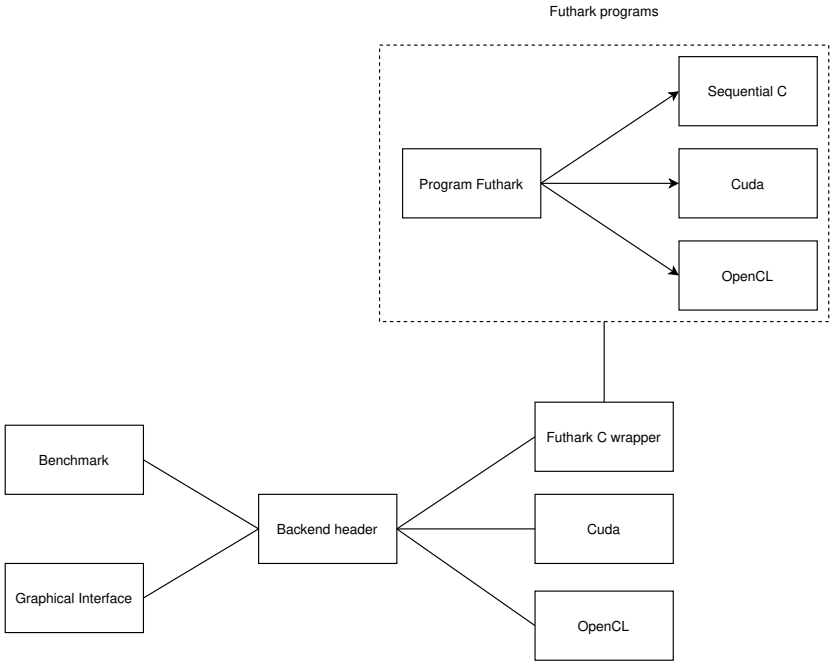


Figure 2.1: Components of the implementation

Source: Made by El Kharroubi Michaël with draw.io

As you can see in figure 2.1, we designed the implementation to be as modular as possible. We wanted to avoid redundant code and stay as clear as possible. The critical part

that supports this design is the backend header. It defines the behavior of the different backends libraries, and its role is similar to interfaces in Object-Oriented Programming (OOP). Furthermore, we have two programs that will use the backends libraries. A first program is a benchmark tool that we can use to evaluate each backend performance. Then, we have a visualization program using the graphical library Simple Directmedia Layer 2 (SDL2).

CUDA implementation is straightforward. It uses a small variant of C++; the same source code file contains the kernels and the host functions. On GPGPU, we distinguish between a function executed on the device (GPU) called a kernel and one executed on the host (CPU) simply called a function. To illustrate a kernel written for CUDA, let us take the parity automaton.

```
__global__ void parity_automaton(uint* src, uint*dst, int width, int length){
    // Recover the index
    int index = get_index();

    // get the direct neighbors of the pixel
    int4 neighbors = get_neighborhood(index, width, length);

    dst[index] = src[neighbors.x] ^ src[neighbors.y]
        ^ src[neighbors.z] ^ src[neighbors.w];
}
```

The `__global__` qualifier indicates that this kernel can be called from the host. On the opposite side, there is the `__device__` qualifier; it indicates that the function will only be called from the device. The rest is standard C++, even C in this case, except `int4`, which is a vector type composed of four signed-integers.

The OpenCL version is purely C; the main difference regarding the device-side code is that we have to store it in an array of characters, also called a string. It can also be stored in a file and loaded at runtime into an array of characters, which can be convenient.

```
__kernel void parity_automaton(__global uint *src, __global uint *dst,
    int width, int length) {
    // Recover the index
    int index = get_global_id(0);

    // get the direct neighbors of the pixel
    int4 neighbors = get_neighborhood(index, width, length);
```

```
    // update pixel in destination matrix
    dst[index] = src[neighbors.x] ^ src[neighbors.y]
        ^ src[neighbors.z] ^ src[neighbors.w];
}
```

The only difference here is the qualifiers. `__kernel` is the OpenCL equivalent of CUDA's `__global__` qualifier. The `__global` before the argument indicates that this pointer is accessible globally, which means both from the device and the host.

The last version is the Futhark one. Futhark does not ask the developers to make the difference between device and host code.

```
let parity_automaton [n] (src: [n]u32) (width: i32) : [n]u32 =
    map (\idx ->
        vec4.reduce (^) 0 (get_neighborhood src idx width)
    ) (iota n)
```

Depending on the backend, the SOAC, namely `map`, in the code extract above, will be trans-compiled into a kernel. Futhark's version is smaller and more intelligible. We can read the behavior, which is expected from a declarative paradigm. The `map` function will visit each cell, then the `vec4.reduce` function will apply an exclusive or between the neighbors from the function `get_neighborhood` and returns us the next value of the cell.

There is a specificity in the above code extract, `vec4.reduce` is a SOAC executed sequentially. The vector type is a module made by Futhark's creator; the purpose of this module is to complement arrays. Its objective is to replace small arrays; for example, we just need the four neighbors in our case. When asked about this, Troels Henriksen the creator of Futhark told us: "*The problem with small arrays is using them in bulk operations, since the compiler does not fully understand that some arrays are small, and may perform (relatively) high overhead operations that are only worthwhile for big arrays.*"[3].

The vector module is an additional package that encapsulates tuples. Tuples are a type of collection with a small fixed-sized; the most common type of tuple is a pair. In Futhark, an instance of tuple would be `let t : (i32, i32) = (0,1)`. Unlike simple tuples, vectors offer the main SOACs, hence mimicking an array behavior. If we look at the generated code, a vector, or more generally a tuple, is not stored in a memory chunk like an array but rather in registers. Registers are approximately a hundred times faster at I/O than the global memory. This difference partially explains the tuple's performances for small collections.

[3]Written exchange on Futhark's Gitter.

The existence of additional packages implies the existence of packages management. Adding a package to a Futhark project is trivial; first, we have to make a file named *futhark.pkg*. For instance, with the vector's package, we can either use Futhark's tool like this `futhark init github.com/athas/vector` or append a line in the Futhark's package file.

In both case the package file will look like this :

```
require {
  github.com/athas/vector 0.6.0
#36473256e4834ac9ff31243cceb439c1f79bf1a4
}
```

To download the package in our project, we can simply do `futhark pkg sync`.

b) Results

Firstly, we can observe the visible result of our automata.

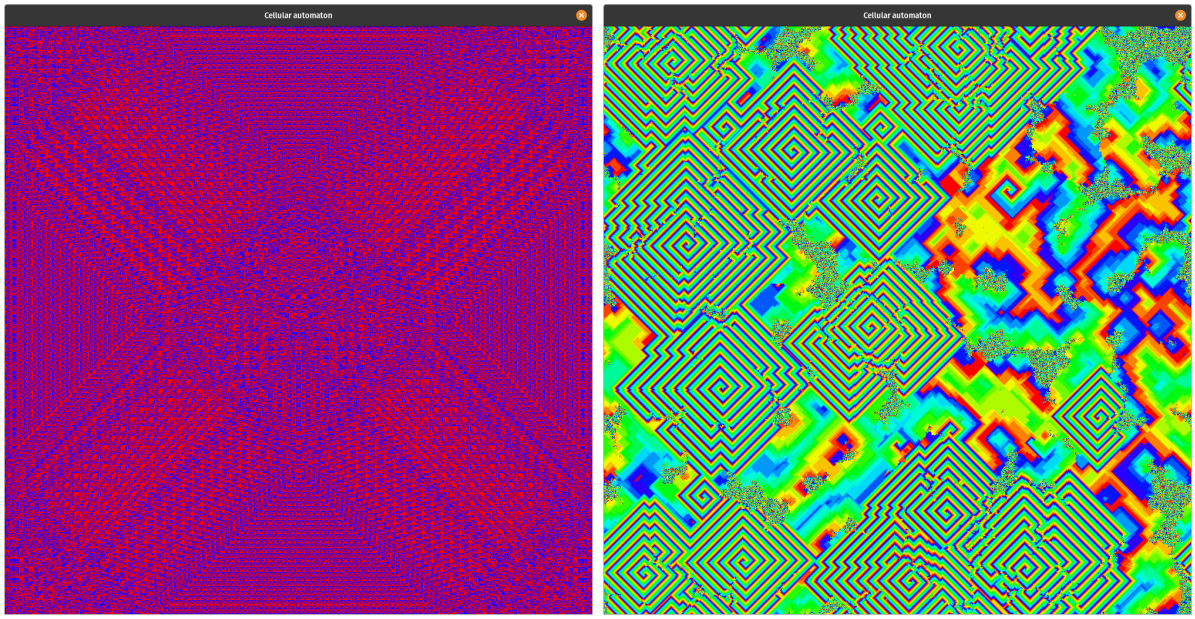


Figure 2.2: Our program displaying, on the left, a state of the parity automaton, and on the right, a state of the cyclic automaton.

Source: El Kharroubi Michaël took this screenshot

On figure 2.2, we see the representation of the computed matrix after n iterations. If we decide to show 24 images per second, we can perceive the matrix's evolution as an animation.

The second program is a benchmarking tool; we use it to compare performances between Futhark and pure CUDA or OpenCL. It served to post results on Reddit[4] to share them with the community.

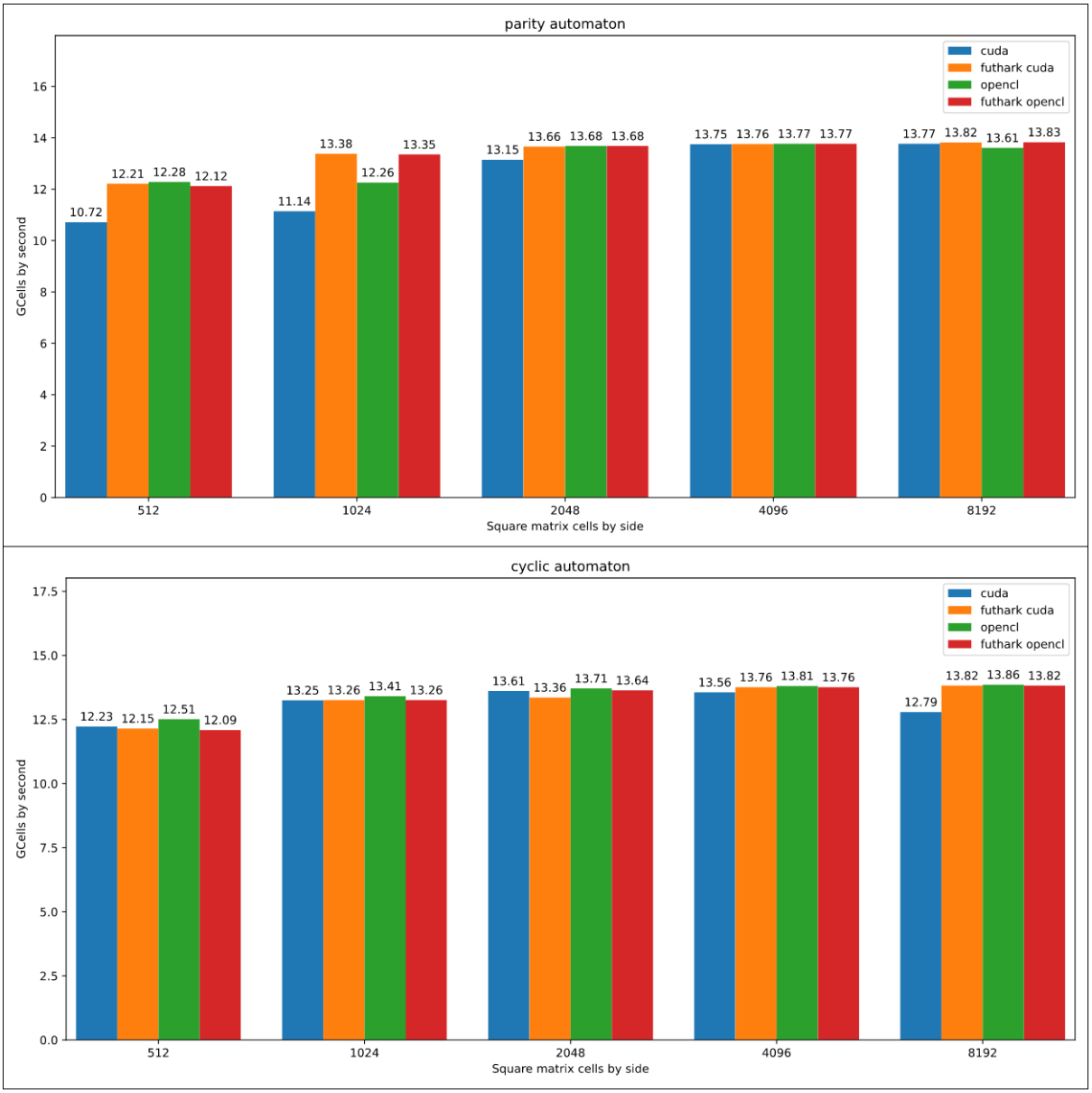


Figure 2.3: Time measured for 10'000 iterations, obtained on an Nvidia GTX 1650

Source: Measures made by El Kharroubi Michaël

The Nvidia GTX 1650 used in figure 2.3 is a laptop GPU. We can see that we achieve near peak performance with a small matrix of 512×512 cells. Furthermore, in this graphic, we can see that Futhark performs as well as CUDA or OpenCL.

[4]Accessible at the following URL: https://www.reddit.com/r/futhark/comments/gfvlkw/hpc_futhark_the_good_vs_cuda_the_bad_vs_opencl/

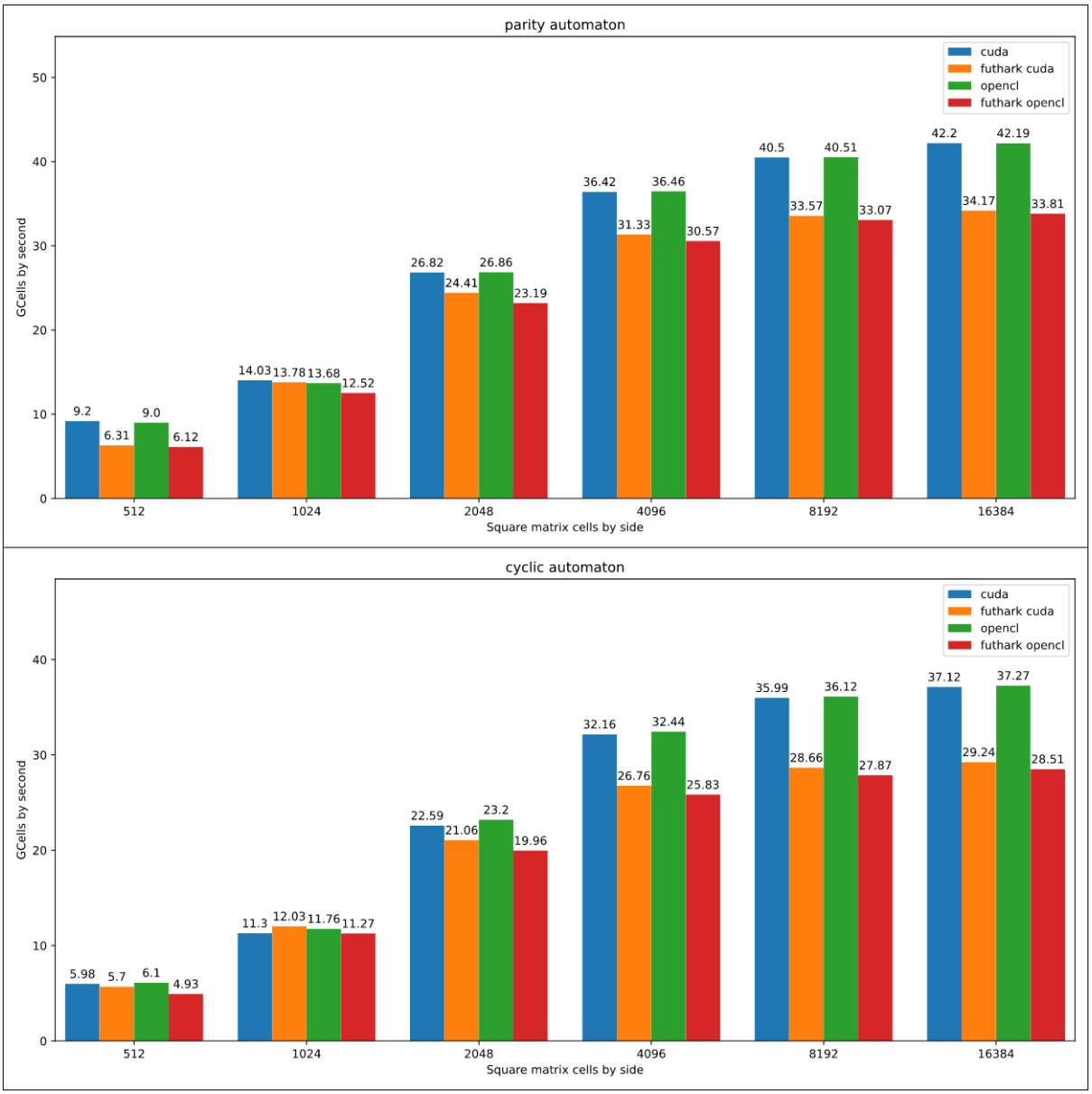


Figure 2.4: Time measured for 10'000 iterations, obtained obtained on the university cluster (Titan Xp or Tesla P100-PCI-E-12GB)

Source: Measures made by El Kharroubi Michaël

In figure 2.4, we can see that the university's cluster GPU is much more performant. We approach the peak performance with a matrix of $8'192 \times 8'192$ cells, which means 256 times more cells than with the laptop GPU. On those GPUs, at the peak, CUDA's and OpenCL's code seems to be 20% more performant. Without being a massive difference, it is still noticeable. The compilation of the kernels could cause those differences. Futhark compiles the kernels itself with specific options. In comparison, the pure CUDA and OpenCL codes were compiled without specifying options. A faulty choice of options on behalf of Futhark could explain this difference.

Although those graphics are promising, one should acknowledge that the measured pro-

grams where a simple version, it is undoubtedly possible to outperform our results with state of the art optimization and tuning for a targeted GPU. However, it strengthens our position, with Futhark, we can make decently fast programs with a high-level language. We can enjoy portability, taking months to make the perfect program for a target GPU is a tremendous loss of time. Suppose that tomorrow, Nvidia, for instance, loses its position as a leader in the GPU market, industry, and public sectors will have to spend considerable amounts of money and time, to adapt their code. Whereas with Futhark, if a new challenger emerges, people could develop a new targeted backend, and compile their assets.

When we consider the massive advantages of functional programming, small losses in performance seem acceptable.

Chapter 3:

Case study, block ciphers

The book *Cryptography engineering: design principles and practical applications* defines (16) block ciphers as encryption functions for fixed-size data blocks. If we imagine a block cipher with a two byte block size, we have to cut our plaintext into blocks of two bytes each and apply our encryption function, let us call it C, with a given key, let us call it K, on each of them. Finally, we concatenate the encrypted blocks, and we get the ciphertext. Like every cipher, our block cipher has a decryption function that reverses the process. Let us name our decryption function D. As before, we split the ciphertext into two byte blocks. As before, we apply D on each block. Finally, we concatenate the decrypted blocks to get the plaintext. We expressed it with a more illustrated approach in figure 3.1.

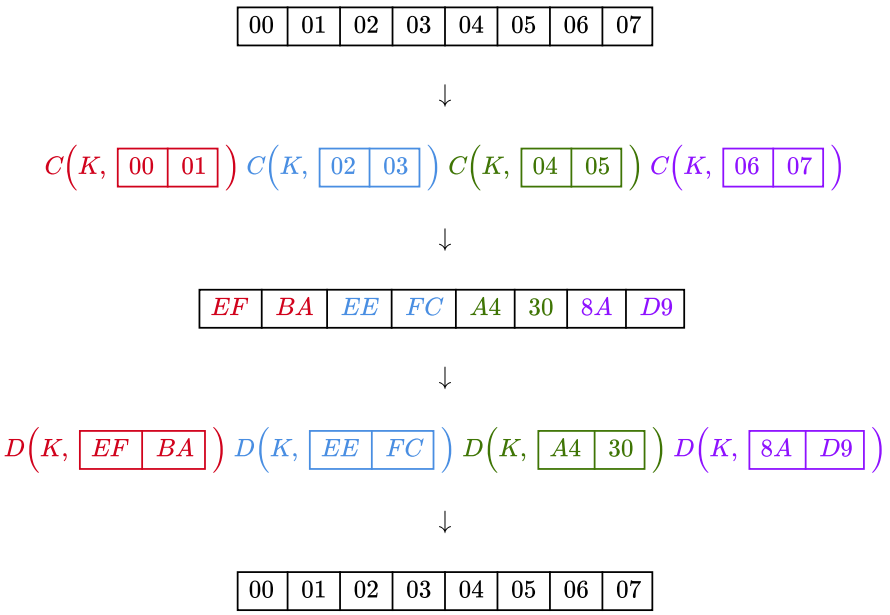


Figure 3.1: An example of block cipher-encryption

Source: Made by El Kharroubi Michaël with mathcha.io

3.1. Modes of operation

Observing how a block cipher works, the connection with data-parallelism and functional programming seems obvious. Once we split the plaintext into blocks, we could apply the SOAC map. Although block ciphers seem to be the appropriate test case for Futhark, in reality, it is a bit more complicated. Block ciphers have a significant flaw when directly used. With a block cipher, we need to choose a mode of operation, for instance, merely encrypting block after block is a mode of operation called ECB. This mode of operation does not hide patterns; it is then susceptible to various statistical attacks. One could shambles with the order of the encrypted blocks, which is a substitution attack. In his book(17), Christof Paar presents an example of block swapping with financial transactions. One could swap the transaction order, delete a transaction, or even duplicate one.

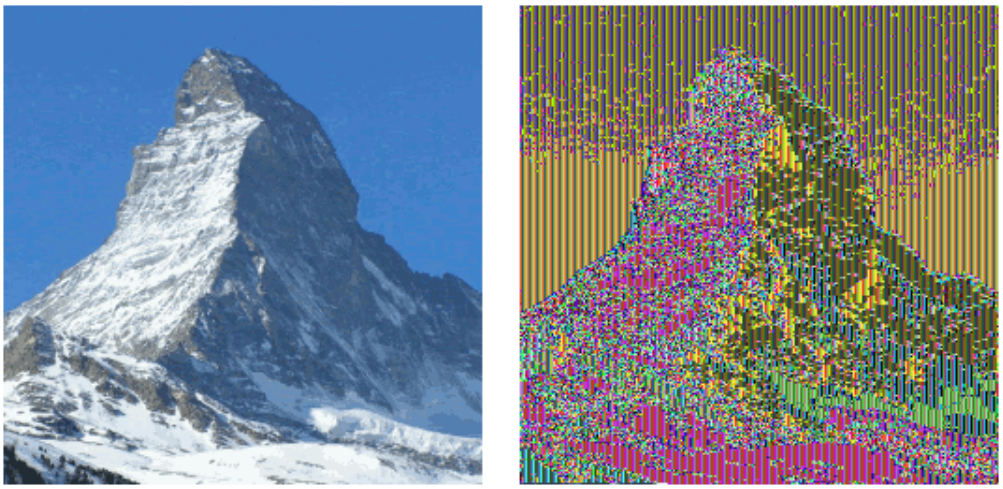


Figure 3.2: On the left, the original picture, on the right, the encrypted picture with ECB mode of operation

Source: Wikipedia's example of picture encryption with the ECB mode, ref. URL03 & URL04

A very good example of a statistical attack is the encryption of a bitmap picture. Looking at the encrypted file block by block, the picture seems encrypted. But when we looking at figure 3.2, we can effortlessly guess the content of the picture. Moreover, the last issue is that in ECB mode, the encrypted file will always be the same for a given key. That issue means that if one can get the plaintext corresponding to a block or the entire file, every time this encrypted block or file passes through the channel, the attackers will know the content.

Otherwise, we have the CBC mode of operation, which applies an exclusive or between each block and the previously encrypted block. This dependency between the blocks par-

tially forbids concurrency. Thus, this mode of operation is sequential for encryption, but the decryption is parallelizable. There exist some modes of operation that are compatible with parallelism, such as the CounTeR (CTR) mode. It is a mode of operation that, for each block, increment the counter, encrypts the value plus the nonce, and applies an exclusive or with the block. There is also the Galois/counter mode, based on CTR, but adds the integrity and authenticity guaranty. There are plenty of modes of operation, and the NIST describes five modes of operation in its 2001 recommendation (18).

Nonetheless, considering cryptographically secure software implementation, we cannot solely think about the algorithm; we need to consider attacks on the implementation itself, called side-channel attacks. For instance, one can measure, but not only, the power consumption, time of execution, harvest error messages, or returned values(19). The risk of side-channel attacks increases with a high-level language like Futhark. To take advantage of abstraction, we have to relinquish some control over memory management, the order of execution, and optimizations. Hence, to avoid Futhark related problems, one should scrutinize the trans-compiled code, and start all over again at each version. Even though these problems are well-known, this project will not focus on those kinds of issues. The goal pursued here is to determine if Futhark can obtain satisfying performances and not make a deemed secure implementation of block ciphers.

3.2. Review of the situation

Block ciphers are some of the most used ciphers today. Some of them like AES profit from hardware implementations. The AES primary hardware implementation is the AES-NI instructions set. It allows fast and secure implementations. One of the most popular tools, OpenSSL, provides an implementation with AES-NI and state of the art software implementations with Camellia. If one wants to evaluate his implementation performances, OpenSSL seems to offer a fair and square comparison.

Performance comparison of block ciphers GPU based implementation is not an easy task. The main issue is the numerous hardware found in the literature. It is not easy to compare an implementation on a brand new Nvidia Tesla V100 and one on an Nvidia GTX 580. First, we are separated by generations of architecture. Each generation adds something new like, not exhaustively, cache, faster memory, or faster instructions. Furthermore, We have the type of GPU series, grand public series like Nivida's GeForce, or professional series like the Nvidia's Tesla. Professional series usually propose error-detection, a higher bandwidth, or more graphical memory. Finally, the last issue is the measurement method. For instance, one can measure the time of sending data to the GPU, encrypting the block, and getting it back from the GPU or measure only the encryption itself with the data already present on the GPU.

In 2017, an Egyptian team achieved a throughput of 277 Gbits/s(20), on an Nvidia GTX 1080, for AES encryption without transfer. For their research, they used a T-Box based implementation. This technic pre-computed some tables for a total of 4KB; this pre-computation fused some parts of the algorithm. Hence, it reduces the number of instructions needed. Moreover, they pre-compute the scheduled key on the CPU and then store it in the shared memory. It is a memory shared by a group of threads on the GPU called a block. Their main work specifity relies on that they tried to vary the kernel granularity encrypting multiple blocks at the time. They also compared different strategies to store the lookup tables. They got the peek performance when storing their T-Boxes on shared memory with a two blocks per kernel granularity.

In October 2019, a searcher Iranian team announced a throughput of 1,4 Tbits/s(21), on an Nvidia Tesla V100, for AES encryption without transfer. With their implementation, they got a throughput of 805 Gbit/s on an Nvidia GTX 1080, which is 2.9 times better than the results obtained by the Egyptian team(20). They did not use the T-Box implementation; instead, they used a bit-slicing approach, thus avoiding costly I/O operations. The bit slicing approach split bytes in multiple variables containing the different components bits(22). For instance, we can replace a substitution box with multiple small boolean functions.

As for Camellia, the literature is a bit scarce; nonetheless, in 2014, a Malaysian team got a throughput of 61.1 Gbit/s (23) with the CTR mode of operation, on an Nvidia GTX680. Like the AES Egyptian team(20), they pre-computed the scheduled key on the CPU and then stored it in the GPU registers. They stored the Camellia lookup table in the GPU shared memory. Finally, they overlapped the execution of the kernels with the data transfers.

Chapter 4:

Implementation

To evaluate the performance of Futhark in practical application, we chose two block ciphers AES and Camellia. The original name of AES is Rijndael. Initially, the NIST organized a contest in 1997 to find a successor for their standard block cipher at the time, DES. In October 2000, the NIST announced that Rijndael had won the contest and ratified the standard in February 2001 with the Federal Information Processing Standards (FIPS) 197(24). Today, AES is the most spread block cipher around the world. AES is present in various applications, from HyperText Transfer Protocol Secure (HTTPS) with Secure Sockets Layer (SSL), to disk encryption with tools like Microsoft's BitLocker. Since 2008, Intel proposed a new instruction set for AES called AES-NI(25). This hardware implementation offers outstanding performance and avoids software side-channel attacks. The second cipher Camellia, NTT and Mitsubishi designed it and published the first specification in July 2000. NTT's based Camellia on another cipher, E2(26). NTT submitted E2 for the AES contest. In 2014, in their "*Algorithms, key size and parameters report*", the ENISA qualified Camellia as a suitable block cipher for legacy and future use along with AES.

In this project, we will implement AES and Camellia in ECB mode with a key size of 128 bits and Public Key Cryptography Standards #7 (PKCS7)'s padding for compatibility with OpenSSL.

4.1. Analysis of the chosen ciphers

a) AES

AES has blocks of 16 bytes, and we implemented the 16-bytes key version. It is composed of two main steps the key scheduling and the encryption rounds. The key scheduling step is unique for a given key; it does not depend on the plaintext.

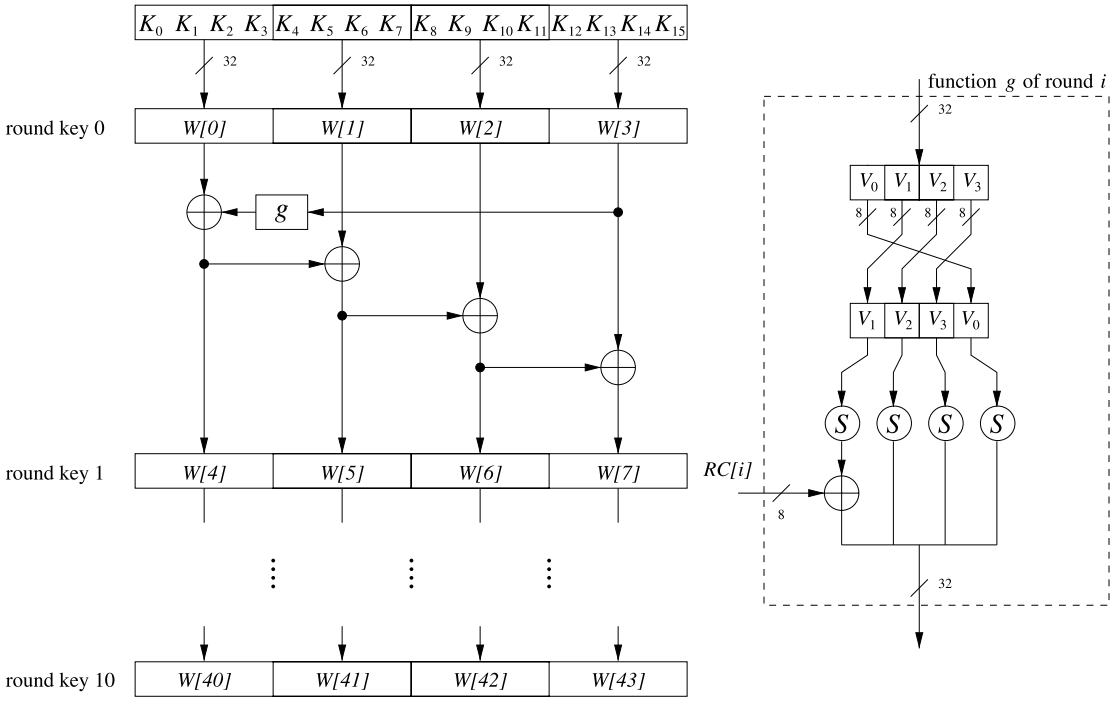


Figure 4.1: AES key scheduling with a 128-bit key

Source: Cropped by El Kharroubi Michaël, taken from Pr. Paars's book: «Understanding Cryptography: A Textbook for Students and Practitioners»

In figure 4.1, we have K_n, which are the 16 bytes of the key. Then we have W_m for the 44-words scheduled key. A word is a sequence of four bytes. The firsts four words of the scheduled key, W_m, are the four words of the key, K_n. The primary operations used are exclusive or, left logical rotation and the function S. The function S swaps a byte with a substitution table called S-BOX. We also have an exclusive or with *RC[i]*, an array containing round constants. In the details of function g in figure 4.1, we observe at the end that the round constant is one byte long, and with it, we applied an exclusive or on $S(V_1)$. If we store our round constants on a word with 3 bytes of trailing zeros, we can directly xor the concatenated outputs of $S(V)$ with our redefined $RC[i]$.

We could translate this key scheduling algorithm with the following pseudo-code :

```
function g (V: word, i: integer32):
    V_prime:= rotate_left(V, 8) -- logical left rotation of 8 bits
    return swap_bytes_with_SBOX_table(V_prime) xor RC[i]

function aes_key_schedule (K: [4]word):
    W:= init [44]word
    W[0:4]:= K --The firsts 4 W's words take the values of the 4 K's words
    for i in 1 to 10 do
        last_words:= W[4*(i-1):4*i]
        new_words:= init [4]word
        new_words[0]:= last_words[0] xor g(last_words[3], i)
        new_words[1]:= last_words[1] xor last_words[0]
        new_words[2]:= last_words[2] xor last_words[1]
        new_words[3]:= last_words[3] xor last_words[2]
        W[4*i:4*(i+1)]:= new_words
    done
    return W
```

As we said, this code is not critical because we do it only once. Hence, we could execute it on the host. On the contrary, the encryption step is critical because we have to do it for each block.

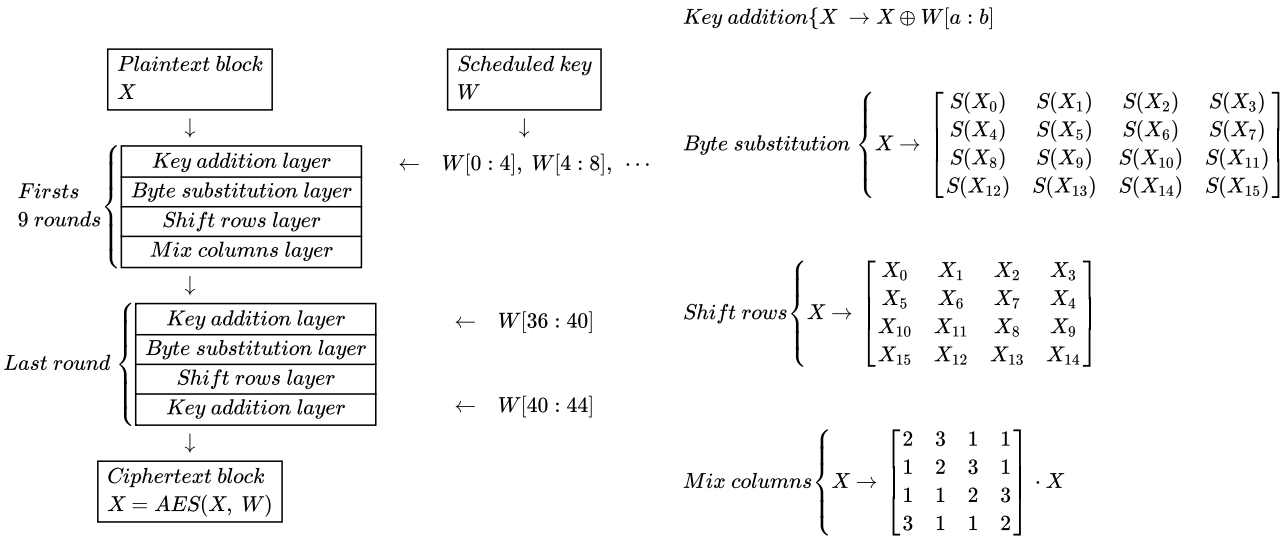


Figure 4.2: AES block encryption with a 128-bit key

Source: Made by El Kharroubi Michaël with mathcha.io

The block encryption function takes two arguments, a block, and the scheduled key. Like the key scheduling function, we found substitutions, logical shifts, and exclusive or oper-

ations. The new element is the mix columns operation, which is a matrix multiplication between a constant matrix and the block. The operations composing the matrix multiplication, are non-conventional because they take place in a Galois field. Without going further in the mathematical details of the Galois fields, we can say that the additions are replaced by exclusive or. Moreover, the multiplication operation between two bytes should be invertible. Which we could formulate like this, with $\otimes$ the multiplication in $GF(2^8)$:

$$\begin{cases} \forall B_n \ \in \ GF\left(2^8\right), & B_i \otimes B_j \ = \ B_k \ \in \ GF\left(2^8\right) \\ \forall B_n \ \in \ GF\left(2^8\right), \ \exists!\ B_n^{-1} \in GF\left(2^8\right), & \left(B_i \otimes B_j\right) \otimes B_j^{-1} \ = \ B_i \end{cases}$$

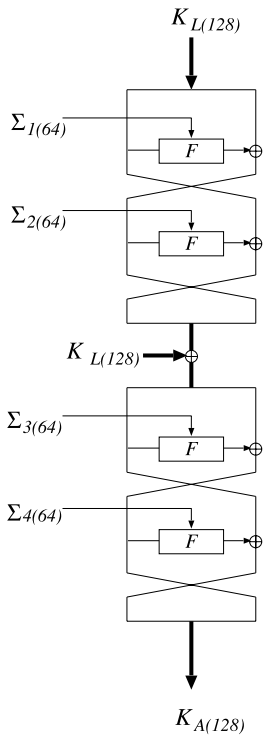
We could give the following pseudo-code for the encryption function:

```
function aes_encrypt (X: [4]word, W: [44]word):
    Y:=X
    for i in 1 to 10 do
        round_key:= W[4*(i-1):4*i]
        Y:= Y xor round_key
    Y:= swap_block_bytes_with_SBOX_table(Y) -- See aes encryption figure
    Y:= shift_rows(Y) -- See aes encryption figure
    if i < 10 then
        Y:= mix_columns(Y) -- See aes encryption figure
    else
        Y:= Y xor W[4*i:4*(i+1)]
    done
    return Y
```

The decryption process is almost the same. The operations do not change; they are reversed, as well as their order. The central difference is that we have an inverse SBOX for the inverse bytes substitution operation and an inverse matrix for the inverse mix columns function.

b) Camellia

Like AES, Camellia use 16-byte blocks, and, similarly, as with AES, we also implemented the 16-byte key version. Camellia has a particularity; it posses a Feistel network. The essential notion with Feistel networks is that the encryption process and the decryption process are the same; we only have to reverse the order of the scheduled keys(27). For the sake of simplicity, we will not further explain Feistel networks. The interested readers will found more pieces of information in Pr Paars's book(17) and Pr Khono's book (16).



$$F(X_{(64)},\ K_{(64)}) = P(S(X \oplus K))$$

$$P(Z_{(64)}) = Z'_{(64)} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} z_8 \\ z_7 \\ \vdots \\ z_1 \end{pmatrix}$$

$$S(L_{(64)}) = Z_{(64)} = \begin{pmatrix} l_8 \\ l_7 \\ l_6 \\ l_5 \\ l_4 \\ l_3 \\ l_2 \\ l_1 \end{pmatrix} \longrightarrow \begin{pmatrix} s_1(l_8) \\ s_2(l_7) \\ s_3(l_6) \\ s_4(l_5) \\ s_2(l_4) \\ s_3(l_3) \\ s_4(l_2) \\ s_1(l_1) \end{pmatrix}$$

$Subkey_{(64)}$	$Value_{(64)}$
kw_1	$(K_L \lll_0)_L$
kw_2	$(K_L \lll_0)_R$
k_1	$(K_A \lll_0)_L$
k_2	$(K_A \lll_0)_R$
k_3	$(K_L \lll_{15})_L$
k_4	$(K_L \lll_{15})_R$
k_5	$(K_A \lll_{15})_L$
k_6	$(K_A \lll_{15})_R$
kl_1	$(K_A \lll_{30})_L$
kl_2	$(K_A \lll_{30})_R$
k_7	$(K_L \lll_{45})_L$
k_8	$(K_L \lll_{45})_R$
k_9	$(K_A \lll_{45})_L$
k_{10}	$(K_L \lll_{60})_R$
k_{11}	$(K_A \lll_{60})_L$
k_{12}	$(K_A \lll_{60})_R$
kl_3	$(K_L \lll_{77})_L$
kl_4	$(K_L \lll_{77})_R$
k_{13}	$(K_L \lll_{94})_L$
k_{14}	$(K_L \lll_{94})_R$
k_{15}	$(K_A \lll_{94})_L$
k_{16}	$(K_A \lll_{94})_R$
k_{17}	$(K_L \lll_{111})_L$
k_{18}	$(K_L \lll_{111})_R$
kw_3	$(K_A \lll_{111})_L$
kw_4	$(K_A \lll_{111})_R$

$\lll_n$ *Circlular shit left of n bits*
$(X)_L$ *The most significant half of X*
$(X)_R$ *The less significant half of X*

Figure 4.3: Camellia key scheduling with a 128-bit key

Source: Formatted by El Kharroubi Michaël, partially taken from Camellia's specification paper, with mathcha.io

The key scheduling has two steps. In the first step, we have to compute K_A. We take our 128-bits key; we will call it K_L. We use four rounds of a Feistel network with Camellia's F function, with an exclusive or with K_L in the middle, and we have K_A. Camellia's F function is a composition of two other functions, P and S, as we can see in figure 4.3. The S function is a byte-wise substitution like in AES, except that Camellia has four substitution boxes. Like the mix columns operation, the P function is a matrix multiplication with a constant matrix(27). The F function in the key schedule's Feistel network takes constants, like the RC_i constants in AES, called Σ_i. We have to make a set of rotations and splitting of K_L and K_A to get each word of the scheduled key. Eventually, we have a 52-words scheduled key.

The following pseudo-code can describe the key scheduling function:

```
function F(X: [2]word, K: [2]word):
    return P(S(X xor K)) -- See Camellia's key scheduling figure

function camellia_feistel(K: [4]word, S: [2]word):
    K_L:= F(K[0:2], S) xor K[2:4]
    return K_L || K[0:2] -- Array concatenation

function camellia_key_schedule(K_L: [4]word):
    K_A:=K_L

    for i in 1 to 4 do
        if i == 2 then
            K_A:= feistel(K_A, Sigma[i]) xor K_L
        else
            K_A:= feistel(K_A, Sigma[i])
    done

    W:= init [56]word
    W[0:4]:= K_L -- kw1 & kw2
    W[4:8]:= K_A -- k1 & k2
    -- For rotate_left see AES's key-scheduling pseudo-code
    W[8:12]:= rotate_left(K_L, 15) -- k3 & k4
    ... -- See Camellia's key scheduling figure
    W[52:56]:= rotate_left(K_A, 111) -- kw3 & kw4

    return W
```

In the pseudo-code above, we voluntarily elapsed some code to stay as concise and straightforward as possible.

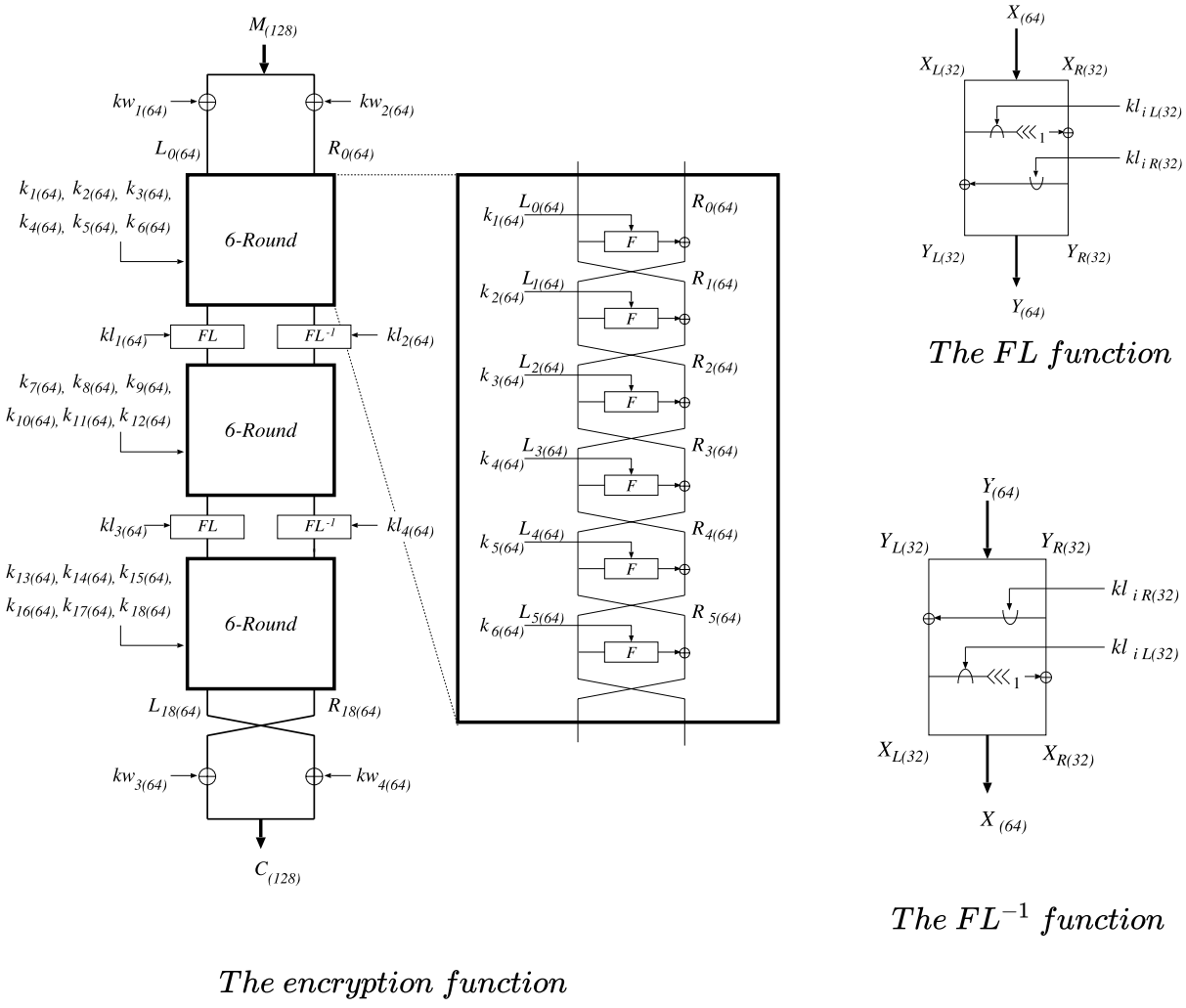


Figure 4.4: Camellia block encryption with a 128-bit key

Source: Grouped by El Kharroubi Michaël, taken from Camellia's specification paper

As we can see in figure 4.4, encryption takes two parameters like AES, the scheduled key, and the plaintext block. Two new operations appear in the encryption algorithm, the FL and FL^{-1} functions. They are used at the end of every six rounds of the Feistel network function, except for the last ones. We can translate the encryption function into the following pseudo-code:

```
function FL(X: [2]word, K: [2]word):
    -- For rotate_left see AES's key-scheduling pseudo-code
    Y_R:= rotate_left(X[0] and K[0], 1) xor X[1]
    return ((Y_R or K[1]) xor X[0]) ||  Y_R -- Concat arrays

function FL_1(Y: [2]word, K: [2]word):
    X_L:= (Y[1] or K[1]) xor Y[0]
    return X_L || (rotate_left(X_L and K[0], 1) xor Y[1])
```

```
function camellia_encrypt (M: [4]word, W: [56]word):
    C:= M xor W[0:4] -- M xor (kw1 || kw2)

    idx:= 4 -- Current position in W
    for i in 1 to 3 do
        for j in 1 to 6 do
            -- For camellia_feistel see Camellia's
            -- key-scheduling pseudo-code
            C:= camellia_feistel(C, W[idx: idx+2])
            idx:= idx+2
        done

        if i < 3 then
            C[0:2]:= FL(C[0:2], W[idx:idx+2])
            C[2:4]:= FL_1(C[2:4], W[idx+2:idx+4])
            idx:= idx + 4
        else
            C:= rotate_left(C, 64) -- Swap halfs
            C:= C xor W[idx: idx+4]
    done

    return C
```

4.2. Experiments

This part of the project aims to implement both ciphers efficiently and respect functional programming principles as much as possible. One of our primary references to find state of the art, efficient codes are OpenSSL's implementations(28). We employed a step by step explanation of the AES algorithm(29) and the specifications of AES(24) and Camellia(27). We also used NTT's proposed software optimization technics(26). Finally, we worked with Pr. Paar's book(17).

We will not present intermediate results in this sub-section. For the sake of brevity, we will only present the results of the final implementation. Nonetheless, all the experiments are available on Githepia[1].

As we have seen in the situation analysis chapter, AES and Camellia, have many elements in common, like the block size or the substitution boxes. Hence, some results of the experiments we have done for AES were also relevant for Camellia.

[1] Accessible at the following URL: https://gitedu.hesge.ch/michael.elkharro/bachelor

The first step was to implement AES. The only objective with the first implementation was for it to encrypt a given block and for the result to be equivalent to our reference example(29). First, we made a small python script to understand the arithmetic in $GF(2^8)$. Then we passed on Futhark and implemented each essential bricks of AES separately. At this point, we already used two optimization technics. The first one was to regroup the bytes per words; we used unsigned 32-bits integer to store them. As for the mix columns operation, we used a technic proposed by Ilmari Karonen on the cryptography section of stackexchange.com(30). Which only used multiplications by two in $GF(2^8)$ and additions in $GF(2^8)$ with exclusive or. This technic uses 36 operations, instead of at least[2] 176 operations for classic matrix multiplication in $GF(2^8)$. We called this first version *AES_0.1*.

The next step was to compare data structures. We compared three data structures, arrays, vectors, and tuples. Vectors are hidden tuples, but they use a nested structure of tuples, and they use conditions to access the elements. Therefore we dimmed useful to add simple tuples to the comparison. On Futhark 15.8, we measured each version's speed, and as we suspected after our exchanges with Futhark's creator, arrays were about 30% slower than vectors. We found out that as announced, the tuples and the vectors have similar performances. That was done months ago. We reran the measures in early august with Futhark 16.2. They seemed to have perfected the arrays; right now, we get non-statistically relevant differences between the three types.

The next step was to try to implement the algorithm implemented by OpenSSL. We called this version *AES_0.2*. OpenSSL used the implementation technic proposed in the original submission paper of Rijndael(31). They fuse the operations of an AES round, and they compute in advance four tables of 256 words for a total of 4KB. We firstly made a pure CUDA version to have measurement standards, but it quickly revealed itself to be too slow. This version meant to be faster was, in fact, slower than our *AES_0.1* version. The objective of this project was not to make an efficient pure CUDA version, so we quickly passed on Futhark. In Futhark, the result was about 10% slower than the plain tuple version. This difference was due to the access pattern of the algorithm; it is almost exclusively array lookups. Accessing global memory (DRAM) on a GPU has some latency. Usually, the GPU tries to group memory access and partly spread it with computing operations on the other threads to hide this latency partially(32). At the time, we did not spend time on it, because we had with the comparison we made earlier, the information that array lookups were slow. Now things seem to have changed with Futhark 16.2. It would be pertinent to re-evaluate this solution, but we will not be able to do it in this work due to a lack of time.

[2]It may vary, depending on the modulo operation used on each multiplication.

Since lookups seem to be a bottleneck for our algorithm, we decided to try several lookups technics. The first one was a simple array; then, we tried a function with pattern matching, and finally, we tried a version with a vector of 256 elements. The best technic was the array. We can explain this with our access pattern due and our data. We substitute bytes in the SBOX, and those bytes will only be known at runtime. GPUs have specialized cores; they do not aim to be as polyvalent as CPU cores. There are designed for a type of instructions called Single Instructions Multiple Data (SIMD). Those instructions perform very well with logical or arithmetic operations on multiple data simultaneously, for instance, vector addition or matrix multiplication. On the other and, they do not perform well with branching instructions like jumps that compose conditions. To get the best performances out of our GPU, we need each thread to do the same work, and for that, we need to determine the path of execution statically. That is not an issue with arrays because they do not depend on branching instructions. However, with Futhark's pattern matching technic, the substitution cannot be done statically in our situation. It is the same for the vectors because, as we have said earlier, they use conditions to access the values if those conditions cannot be resolved statically, we will suffer performance losses.

We tried some variation of the number of blocks by kernel, and it does not seem to have a real impact on Futhark's performances. We suspect Futhark's compilator to optimize the SOAC either way.

Our final version experiment on AES, *AES_0.9*, added the support for differents endianness and some abstraction to our implementation to avoid duplicated code with Camellia.

Ultimately, we took Camellia's reference code proposed by NTT and Mitshubishi, and we tried to implement a Futhark version. We quickly stopped this process because it was a loss of time. As we said earlier, due to similarities between AES and Camellia, each test we did on our AES implementation had significant results for Camellia. We directly reoriented ourselves toward the OpenSSL implementation. This implementation uses a software implementation technic targeted for 32-bits processors, described in the sub-section 4.2.7 of the report on Camellia's implementation technics(26). Instead of having four substitution boxes of 256 byte-entries, we pre-compute four tables of 256 word-entries. With those tables, we can evaluate the F functions with fewer instructions, therefore computing the Feistel function of our Camellia implementation faster. To make our code more explicit, we switched from vectors to tuples. The member access of a tuple is more straightforward than the vector's one.

4.3. FHA

In this sub-section, we will present FHA. A small library that we made to fulfill a need we discovered during our experimentations. It is available on Github[3]. We reached a milestone in development, where we tried to maximize our performance. We used a profiling tool provided by Nvidia, called *nvprof*. This tool gives us pieces of information about our program, like the time spent on each device-related API calls. The profiling told us something we suspected from the beginning, without realizing how huge it was; more than 75% percent of the execution time was due to transfers between the host and the device. At this point, the only relevant optimization we could make was on transfer time.

a) Pinned memory

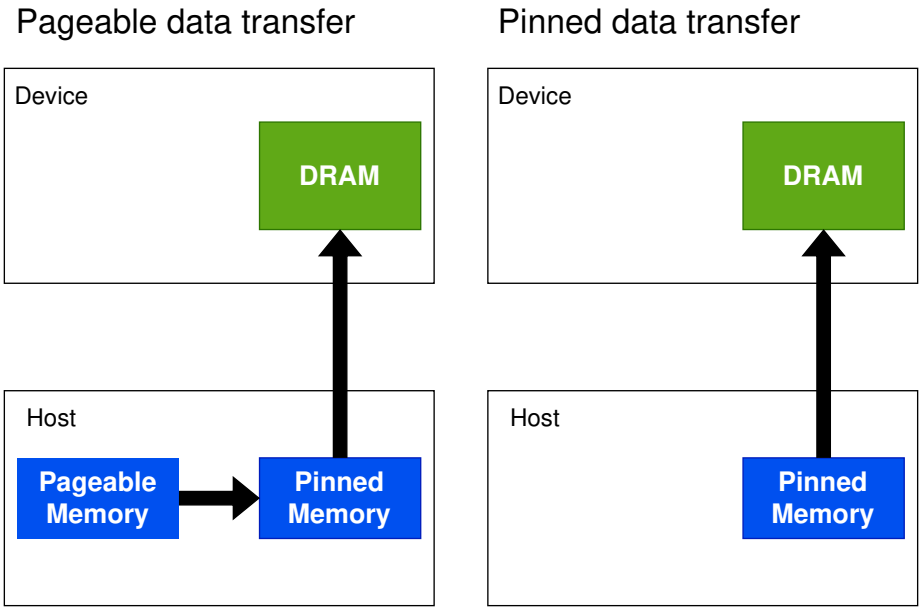


Figure 4.5: Pageable data transfer versus Pinned data transfer

Source: Remade by El Kharroubi Michaël after the schematics on URL05

After some research(33), we found out about pinned memory. As we can see in figure 4.5, pinned memory saves us a transfer in comparison with pageable memory. Pageable memory is a sort of memory that separates the data in pages of fixed size. The operating system can save those pages on secondary storage like an SSD, if the primary storage, usually RAM, is full. This technic allows us to allocate more memory than available on the primary storage. When we use `malloc`, the OS allocates pageable memory by default. Unfortunately, the GPU cannot directly access pageable data; it has to go through the

[3]Accessible at the following URL: https://github.com/michael-elkh/cellular_automaton-futhark-cuda-opencl

CPU; it would have to retrieve the data from pageable memory and store it on pinned memory. Pinned memory is fixed in the primary storage. Thus, the GPU can directly access it and avoid requesting the CPU to transfer the data between pageable and pinned memory.

b) Library

Now that we understand the interest of pinned memory, the issue is; how to use it and keep Futhark's portability simultaneously. For instance, with CUDA, we have to use the function `CudaMallocHost` to allocate pinned memory, but with OpenCL, we have to use `clCreateBuffer` and `clEnqueueMapBuffer`. Those backend-specific, snippets of code are an issue for Futhark's portability. We brought this issue to Troels Henriksen, the creator of Futhark. He suggested that we make a single file library on our side, and he added a preprocessor constant in the generated header file, which indicates the backend used at compile time to Futhark's 0.16.1 release. Hence, we made a library that we called FHA.

```
FHA_RESULT futhark_host_alloc(struct futhark_context *ctx, void **mem,
    size_t size);
FHA_RESULT futhark_host_free(struct futhark_context *ctx, void *mem)
```

We created two functions with behaviors as simple as possible. We wanted to avoid using a new context or any specificity proper to a backend. Those two functions return a result, which is success or failure. They take Futhark's context as an argument to recover the backend-specific context. For CUDA and C, it was pretty basic. We only had to call respectively, `cudaMallocHost` and `malloc`. For OpenCL, there was a bit of overhead. Firstly, we had to create an OpenCL buffer variable of type `cl_mem`. This buffer is problematic. We can not use it as a standard pointer; we had to map a pointer on it with `clEnqueueMapBuffer`. Once it was done, we had a standard pointer, but we could not free the buffer yet, we would have to free it with our pointer later.

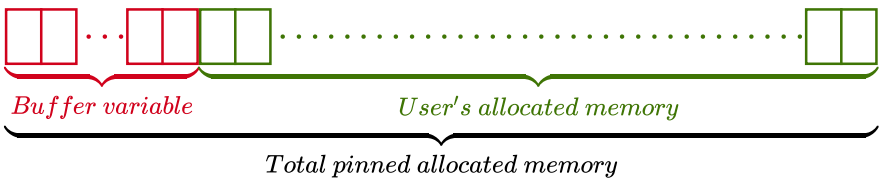


Figure 4.6: Pinned memory allocation for OpenCL with FHA

Source: Made by El Kharroubi Michaël

We used a trick to pass the buffer object to the user without using a structure to encapsulate the buffer. Let say that one wants to allocate n bytes. Rather than allocate only them, we allocate $n + m$ bytes, where m is the size of our buffer variable. Then, as we can see in figure 4.6, we store our buffer variable at the beginning of the total allocated

space, and we return the address following our stored buffer. Hence, to free this space, we have to subtract the size of the buffer type to the pointer to recover it and free it together with the allocated space.

Table 4.1: Sample comparison in CUDA of 1GiB of allocated pageable memory vs. pinned memory, allocation's, and round trip transfer's duration.

Type of memory	Allocation's duration in [ms]	Transfer's duration in [s]
Pageable	0.017	242.266
Pinned	337.559	165.394

As we can see in table 4.1, pinned memory allocation is slower by several orders of magnitude. If we sum the duration of the allocation and the round trip transfer, we found out that pinned memory is slower than pageable memory. Those results hint us that we have to reuse the buffer to be performant. For example, if we make a simple calculation, one wants to make ten round-trip transfers with the same buffer, the pageable memory would be 20% slower than pinned memory. The total duration is a linear function of the total number of transfers. The allocation's duration is constant, and the transfer time is the slope of the function. With any two linear functions, if the first function's slope is steeper than the second, the first function will inevitably exceed the other.

4.4. File encryptor

The final step for this project was to make a file encryptor compatible with OpenSSL. We did this in three steps. First, we had to make a library to use the results of our experiments. Then we had to validate it with official test vectors, and lastly, we had to make a console program to use it.

a) Library

Our library had to follow some guidelines. Mostly, we had to be modular, and we needed to abstract anything related to Futhark. First of all, we had to make a modular Futhark library. We took our experiment *AES_0.9* as a base. To harmonize the AES and Camellia implementations, we removed vectors, and we used a tuple type called `word_block`, which is a tuple of four unsigned 32-bit integers. Furthermore, we had to swap the endianness if necessary and process the block with a given function.

With those rules, we proposed the following signatures for the general higher-order functions:

```
let schedule_key
  (sk: [4]u32 -> [][4]u32)
  (swap_endianness: bool)
  (key: [4]u32) : [][4]u32 = ...
```

The first function takes three arguments. A function called `sk` will schedule the key; it takes an array of four 32-bit unsigned integers and returns a bi-dimensional array with an unknown number of rows and four columns. Then we have a boolean value to precise if we have to swap the endianness. The last one is the key, which is an array of four 32-bit unsigned integers. As expected with the `sk` function, the result is the above-mentioned bi-dimensional array.

```
let process_buffer [m][n]
  (pb: [m](word_block) -> (word_block) -> (word_block))
  (swap_endianness: bool)
  (scheduled_key: [m][4]u32)
  (buffer: [n][4]u32) : [n][4]u32  = ...
```

The next function serves for encryption and decryption. It has two size parameters, `m` and `n`. The parameter `m` is the number of rows in the scheduled key, and `n` is the number of 128-bits blocks in the buffer. The `pb` function takes the scheduled key a block of data and returns a processed block. Like before, we have a boolean to deal with the endianness. Next, we have the scheduled key, and ultimately, the buffer to be processed. To illustrate how we use those functions, we can take an example with the exposed AES functions:

```
entry aes_schedule_encryption_key (swap_endianness: bool) (key: [4]u32)
  : [][4]u32 =
  schedule_key aes_encrypt.schedule_key swap_endianness key
entry aes_encrypt_buffer [m][n]
  (swap_endianness: bool)
  (scheduled_key: [m][4]u32)
  (buffer: [n][4]u32)
  : [n][4]u32 =
    process_buffer
      aes_encrypt.encrypt_block
      swap_endianness
      (scheduled_key:>[11][4]u32)
      buffer
```

We just pass the received arguments to the universal functions we defined, and we add the concrete functions with it, respectively, `aes_encrypt.schedule_key` and `aes_encrypt.encrypt_block`.

On the C wrapper side, we do something similar with the wrapping functions. Let us take, for instance, the key scheduling :

```
// The key scheduling function type
typedef int (*schedule_key_f_t)(struct futhark_context*,
    struct futhark_u32_2d**,
    const bool,
    const struct futhark_u32_1d*);
// The universal key scheduling function
void schedule_key(schedule_key_f_t f, ciphers_params_t *params,
    uint8_t key_size, void *key) {...}
// The concrete usage with aes key schedule entry.
void aes_schedule_key(ciphers_params_t* params, uint8_t key_size,
    void* key) {
    schedule_key(&futhark_entry_aes_schedule_encryption_key, params,
        key_size, key);
}
```

To allow the user to extend this principle, the library proposes two types to represent a generalization of the functions exposed by the library :

```
//ciphers_params_t* params, uint8_t key_size, void* key
typedef void (*ciphers_schedule_key_f_t)(ciphers_params_t*, uint8_t,
    void*);
//ciphers_params_t* params, uint32_t nb_blocks, void *buffer
typedef void (*ciphers_encrypt_decrypt_f_t)(ciphers_params_t*, uint32_t,
    void*);
```

b) Validation with official known answer test vectors

After implementing a satisfying version of the block ciphers, we had to validate our implementations. The NIST offers official test vectors for AES(34). They propose three types of tests: the known answer, the multi-block message, and the Monte Carlo. The first type gives a key, a plaintext, and a ciphertext, either we have to encrypt the plaintext to get the given ciphertext or the other way around. The second aims to evaluate the capacity to encrypt multiple blocks at the time. Those kinds of tests are particularly useful for a chained mode of operation, or a parallel one like CTR, to control the counter values. The

last one uses a probabilist approach for validation. The plaintext is encrypted a thousand times with the same key, and then we make an exclusive or between the result and the key, generating the next key, and we repeat this process a hundred times. NTT's proposes only the known answer test type for Camellia(35).

In this project, we only verified the known answer tests for both ciphers. We justify this choice by the very nature of this project; the objective here is to evaluate the performance and not make a cryptographically secure implementation. We estimate that an implementation passing all the official known answer tests is good enough for that purpose.

Given that we have two different ciphers with two different formats for their tests, we decided to harmonize their formats with a python script.

Table 4.2: Format for harmonized known answer tests.

Field	Possible values	Size in bytes
Type	$\{0,1\}_1$	1
Key	$\{0,1\}_{128}$	16
Plaintext	$\{0,1\}_{128}$	16
Ciphertext	$\{0,1\}_{128}$	16
Total		49

The first step to harmonize the different tests was to define a format. As you can see on table 4.2, we choose to set one byte for the type, which can be zero for encryption and one for decryption. Then, we have 48 bytes that we have split equally between the key, the plaintext, and the ciphertext.

We decided to make the python script as general as possible to be extended should the need arose. We made a necessary data encoder, and we only need to implement a line parser for each new test suite.

In C, we can directly load the formatted test in memory with the following types :

```
typedef enum __attribute__ ((__packed__)) operation {
    ENCRYPT=0,
    DECRYPT=1
} operation_t;

typedef struct __attribute__((__packed__)) test_vector {
    operation_t type;
```

```
    uint8_t key[16];
    uint8_t plaintext[16];
    uint8_t ciphertext[16];
} test_vector_t;
```

We made a nonexclusive test reader and validator in C, which allows us to keep the test file for a given cipher as simple as possible. To that end, we used the library defined types for the different functions needed. Namely, `ciphers_schedule_key_f_t` and `ciphers_encrypt_decrypt_f_t`.

We have to load the tests from a given file into an array of test vectors. Then we can iterate on it to try to pass each test. We only have to pass as arguments the appropriate functions, the context, and the test. The `try_test` function will terminate the program with a message containing appropriate pieces of information about the failed test. Suppose we take the AES validator; the code should like this :

```
...
test_vector_array_t tv_array;
load_tests_vectors(filename, &tv_array);

for (int i = 0; i < tv_array.nb_test_vector; i++)
{
    try_test(aes_schedule_key,
        aes_encrypt_buffer,
        aes_decrypt_buffer,
        &params, tv_array.array + i);
}
free_test_vectors(&tv_array);
...
```

c) Sequential buffer encryption

We made the first file encryptor sequential and used the buffered I/O proposed by the standard library with `fread` and `fwrite`.

We can summarize the algorithm with the following pseudo-code :

```
parse_aguments()
open_files()
schedule_key()
for i in 0 to nb_buffer_in_file-1 do:
    load_buffer_from_file()
    -- encrypt or decrypt depending on the input action
    process_buffer()
    write_buffer_in_file()
done

if action == ENCRYPT then
  padd_final_buffer()
else
  unpadd_final_buffer()
write_final_buffer()
close_files()
```

For the padding, we used PKCS7's format as described in section 6.3 of the *RFC 5652*(36).

d) Parallel I/O

For the second version, we decided to make task parallelism with concurrent queues. To make a concurrent queue, we firstly made a sequential queue. Then we made our concurrent queue with Portable Operating System Interface uniX (POSIX)'s threads, using two mutexes and two semaphores. We can recapitulate the push and pop operations with the following pseudo-code:

```
function push(element):
    wait(can_push)
    lock_push()
    push_in_write_queue(element)
    unlock_push()
    post(can_pop)

function pop():
    wait(can_pop)
    lock_pop()
    res:=try_pop_from_read()
```

```
if res == empty:
    post(can_pop)
    lock_push()
    swap(pop_queue, push_queue)
    unlock_push()
    unlock_pop()
    return pop()
post(can_push)
unlock_pop()
return res
```

At the concurrent queue's initialization, we set the `can_pop` semaphore at zero and the `can_write` semaphore at the wanted capacity of our concurrent queue. With those two sequential queues, we can push and pop simultaneously. However, it is tough to know the size of the queue at an instant t without locking both queues. Furthermore, with the semaphores, we cannot guarantee that the serving order will follow the arrival order.

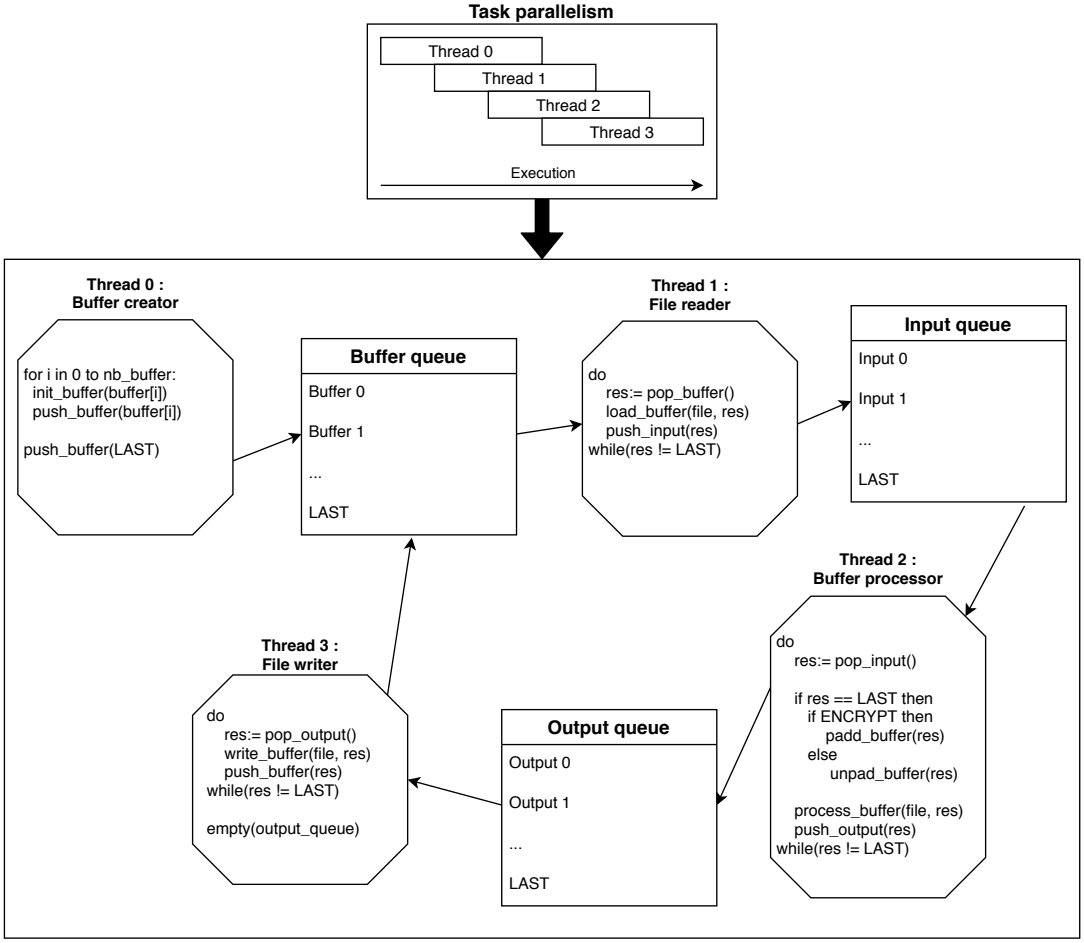


Figure 4.7: Task parallelism with our file encryptor

Source: Made by El Kharroubi Michaël with draw.io

The objective was to parallelize the file I/O with the encryption of the blocks. We can perceive in figure 4.7 that we used as planned task parallelism. To avoid losing time on buffer initialization, we used a separate thread to allocated the pinned-memory, which allows the file reader to start as soon as the first buffer is ready. Then we are in a close loop, and we recycle the buffers to avoid time-consuming expendable operations. As for the sequential version, we used the standard library's buffered I/O functions, `fread`, and `fwrite`.

e) Memory mapping

The first two versions gave us good results, but we were not yet at OpenSSL's level. After some research, we found out that the bottleneck was the I/O operations. So we decided to try a non-standard alternative, which is *mmap*. Although there is some equivalent on windows and mac os, it is not a part of the standard library. The principle of memory mapping is to access the file through an address as if it was present in memory. One of the advantages is that you do not have to load the file buffer after buffer; one can map the file entirely. The limit is the addressing space; on a 32-bits system, the highest address is $2^{32} - 1$, which is approximately $4GB$, and on a 64-bits system, it is approximately $18EB$.

The following pseudo-code can summarize this version:

```
parse_aguments()
file_in_ptr= mmap(file_in)
file_out_ptr= mmap(file_out)
position=0

schedule_key()
for i in 0 to (file_in_size-block_size)/chunk_size do:
    -- encrypt or decrypt depending on the input action
    process_buffer(file_in_ptr+position, file_out_ptr+position)
    position:= position + size_of_chunk
done

final_block=read(file_in+position)
if action == ENCRYPT then
  padd_final_block(final_block)
else
  unpadd_final_block(final_block)

write(file_out, final_block)
```

As you can see in the pseudo-code above, we have to use a write call at the end. It is because mapping a file implies knowing its size in the beginning. However, we do not know the output file's size at the beginning because of the padding. Computing the output file's size at the beginning would uselessly complicate the code, to save less than a second at runtime. That is why we simply deal with the last block separately.

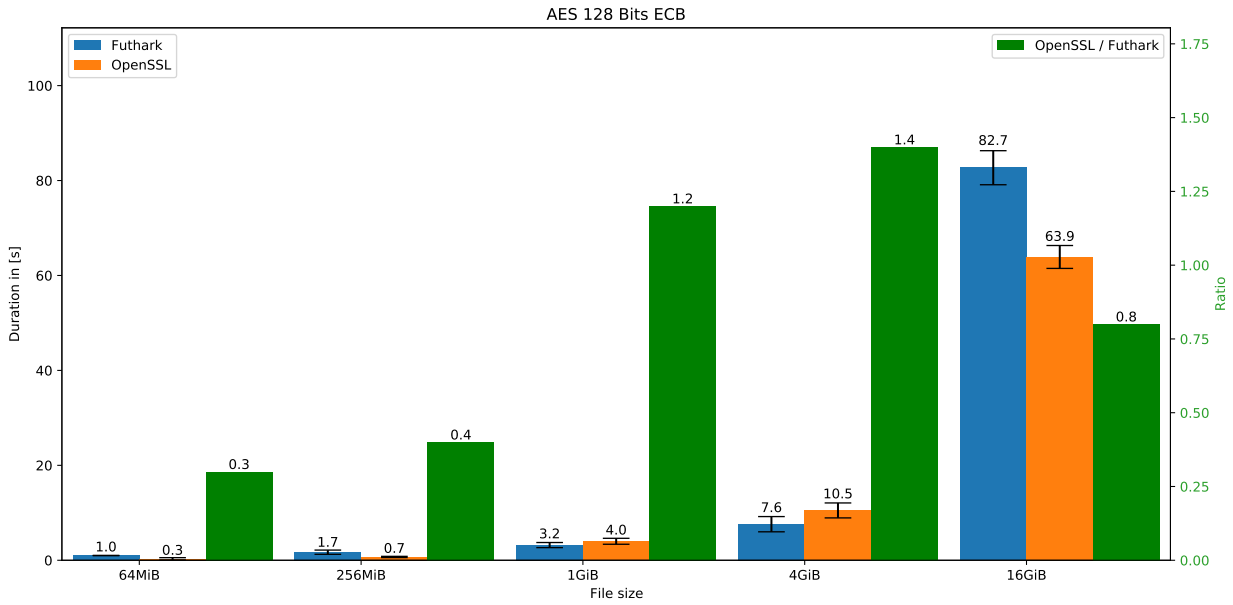


Figure 4.8: Benchmark of AES encryption performances, our file encryptor using Futhark with CUDA against OpenSSL's implementation with AES-NI

Source: Measures made by El Kharroubi Michaël

In figure 4.8, we have three pieces of information, the speed of both tools in blue and orange, and the ratio between OpenSSL and Futhark in green. The first thing we can note in this figure is that this implementation does not suit small files; there is simply too much overhead. We can find an optimal spot between a file size of $1GiB$ and $4GiB$. With the green bar, it seems that with a $4GiB$ file, OpenSSL is 1,4 times slower on average than our implementation. However, on the related blue bar and the orange one, we can see their error bars overlap, meaning that this difference is not statistically significant. Moreover, it seems that there is a slow down at $16GB$. We decline from approximately $542MiB/s$ with $4GiB$ to $198MiB/s$ with $16GiB$. We speculate that it is due to the mmap function and that it merits further investigations to try and fix it. However, we will not be able to do it in this work due to a lack of time.

Nonetheless, getting at least comparable results as an AES-NI implementation is an achievement for this project. We have to send the data from the host to the device, encrypt it, and send it back faster than the direct encryption done by the hardware

implementation. We took this benchmark on a notebook with a high-end CPU[4] and a mid-range GPU[5], on a PCI-E 3.0 bus.

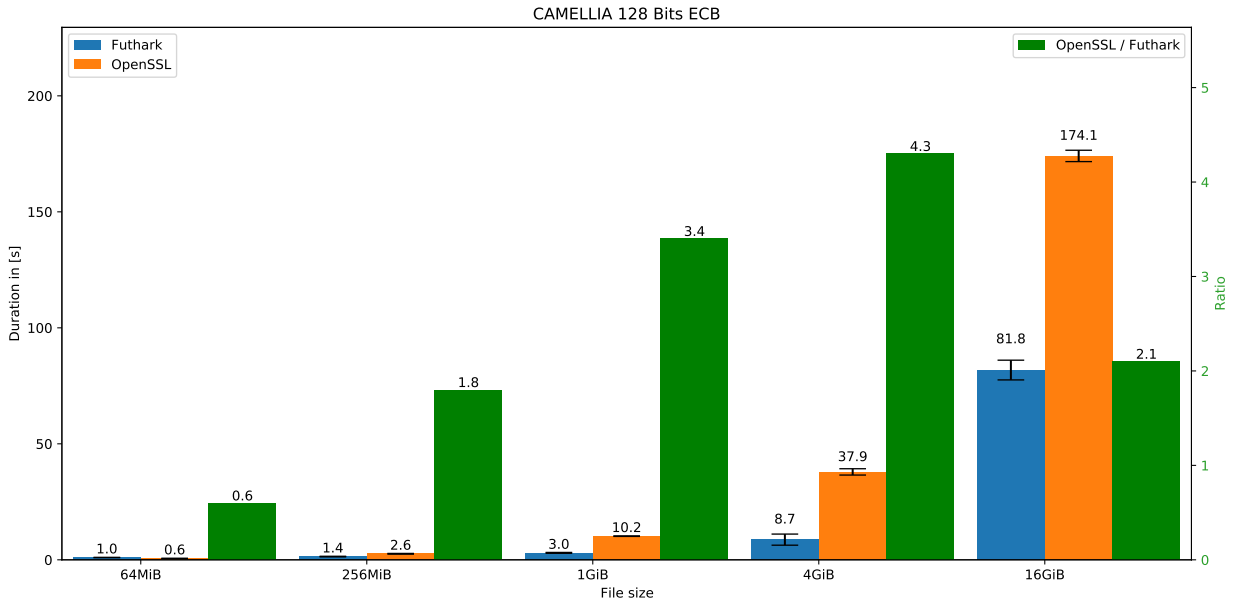


Figure 4.9: Benchmark of Camellia encryption performances, our file encryptor using Futhark with CUDA against OpenSSL's CPU implementation

Source: Measures made by El Kharroubi Michaël

In figure 4.9, we have the same type of information as in figure 4.8. However, we have better outcomes; indeed, Openssl's software implementation is sequential and running on CPU. We can see that we are faster from $256MiB$ and higher. Unlike the result in figure 4.8, we can see that for a $1GiB$ file, the error bars do not overlap, meaning that the difference is statistically significant. Nevertheless, we can observe the same speed loss as in figure 4.8. We decrease from approximately $467MiB/s$ with $4GiB$ to $200MiB/s$ with $16GiB$.

It is pertinent to compare the throughputs of our implementations between each other. Even though AES's algorithm is faster than Camellias, with their Futhark implementation, AES is only 16% faster than Camellia. It is a significant advantage of using a GPU software implementation; we can implement many algorithms and have similar performances. Whereas on CPU, we have AES, which profits from a fast hardware implementation, and all the other ciphers which only have a slow software implementation. In figure 4.9, one can note that with OpenSSL's implementations, for a $4GiB$ file, Camellia is 362% slower than AES.

[4]Intel I9-9980HK with $16MB$ of cache and a max turbo frequency (single-core) of $5GHz$.

[5]Nvidia GeForce GTX 1650 with $4GB$ of GDDR5, 1024 CUDA cores, and a max frequency of $1.56GHz$.

Conclusion

In this project, we implemented various algorithms with Futhark. Firstly, with cellular automata, we have compared Futhark and pure CUDA and OpenCL. Then we developed a cryptographic library with AES and Camellia. Furthermore, we made use of it in a file encryption tool. We made three versions of that tool to find the best solution to take full advantage of our library. All our results can be found at the address: https://gitedu.hesge.ch/michael.elkharro/bachelor, either in the repository itself or in the submodules.

During this process, we discovered Futhark by experimenting with it and interacting with his designer Troels Henriksen. Our work made us find a few bugs that we were able to report on Futhark's Github project. We also tried to improved Futhark by creating a library to allocate pinned memory. That library was made available for the community on Github. We were gladly surprised to found out that Futhark got better along with the project, from new functionalities like the attributes or significant performance improvement.

Although we deem the project to be a success, it was not necessarily a foregone conclusion at the beginning. Nonetheless, we achieved as meant to get near-equivalent performance, in some cases, against OpenSSL's implementation, with Intel's hardware instruction set. Furthermore, we got better performances against OpenSSL's Camellia implementation, starting from files with a size of $256MiB$.

Even though we have got promising results, this project could be improved. We should follow three axes for that purpose. Firstly we should improve disk to GPU transfers; it is the most time-consuming part of the process. We could pin a mapped file with the CUDA API and see if a performance gain is possible by saving another copy in memory. We could also fix the speed loss, that we have observed with $16GiB$ files in figure 4.8 & 4.9, by further investigate mmap's behavior. The second axis is Futhark. By proposing new attributes for the language, we could target the type of memory we would like to use. For instance, we could store the substitution boxes in shared memory or even better in registers. Furthermore, we could call Futhark's functions asynchronously to optimize task-

parallelism. The last axis regards the block ciphers algorithms. If we implement a mode of operation like the counter mode, we could spare sending the data to the device. We could generate the encrypted counter values directly on the GPU and apply an exclusive or with our data four blocks at the time with CPU hardware SIMD instructions sets like Advanced Vector Extensions (AVX)512. Besides, we could even use multiple CPU cores to improve our performances further. With the Galois/Counter Mode, we could also add authentication to our encryption. Finally, we should do a co-project, to identify potentials side-channel attacks and try patching them, to render our implementation safe to use.

Bibliography

1. ORACLE. Chapter 1 Covering Multithreading Basics (Multithreaded Programming Guide). *Oracle's documentation* [en ligne]. 2010. [Consulté le 30 July 2020]. Disponible à l'adresse : https://docs.oracle.com/cd/E19455-01/806-5257/6je9h0329/index.html

2. HICKS, Michael. Concurrency Basics.. Lecture. Autumn 2007.

3. RUPP, Karl. 42 Years of Microprocessor Trend Data | Karl Rupp. *Karl Rupp* [en ligne]. 15 February 2018. [Consulté le 30 July 2020]. Disponible à l'adresse : https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

4. RUPP, Karl. CPU, GPU and MIC Hardware Characteristics over Time | Karl Rupp. *Karl Rupp* [en ligne]. 18 August 2016. [Consulté le 30 July 2020]. Disponible à l'adresse : https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

5. CUTRESS, Ian. Hands on with the 56-core Xeon Platinum 9200 CPU: Intel's Biggest CPU Package Ever. [en ligne]. [Consulté le 30 July 2020]. Disponible à l'adresse : https://www.anandtech.com/show/14182/hands-on-with-the-56core-xeon-platinum-9200-cpu-intels-biggest-cpu-package-ever

6. P. SINGH, Jaswinder. Computer Science 318 Operating Systems. [en ligne]. Lectures. Autumn 2019. [Consulté le 14 August 2020]. Disponible à l'adresse : https://www.cs.princeton.edu/courses/archive/fall19/cos318/schedule.html

7. EL KHARROUBI, Michaël. *AlephPlot - Visualiser des données avec Rust.* March 2020. HEPIA.

8. FALCONE, Jean-Luc. Programmation Fonctionnelle. *Cours de programmation fonctionnelle 3ème année.* January 2020.

9. FALCONE, Jean-Luc. Fonction anonymes. *Cours de programmation fonctionnelle 3ème année.* February 2020.

10. Reading 25: Map, Filter, Reduce. *6.005 - Software Construction* [en ligne]. Autumn 2015. [Consulté le 1 August 2020]. Disponible à l'adresse : https://web.mit.edu/6.005/www/fa15/classes/25-map-filter-reduce/

11. HUGHES, J. Why Functional Programming Matters. *The Computer Journal* [en ligne]. 1 January 1989. Vol. 32, n° 2, pp. 98–107. [Consulté le 18 March 2020]. DOI 10.1093/comjnl/32.2.98. Disponible à l'adresse : https://doi.org/10.1093/comjnl/32.2.98

12. HENRIKSEN, Troels. 2. The Futhark Language — Parallel Programming in Futhark. *Futhark's documentation* [en ligne]. 2020. [Consulté le 15 August 2020]. Disponible à l'adresse : https://futhark-book.readthedocs.io/en/latest/language.html

13. HENRIKSEN, Troels. Futhark 0.17.0 documentation. *Futhark documentation* [en ligne]. 2020. [Consulté le 1 August 2020]. Disponible à l'adresse : https://futhark.readthedocs.io/en/latest/usage.html#compiling-to-executable

14. GROUP, Khronos. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. *The Khronos Group* [en ligne]. 21 July 2013. [Consulté le 1 August 2020]. Disponible à l'adresse : https://www.khronos.org//

15. HENRIKSEN, Troels. Prelude/soacs. *Futhark documentation* [en ligne]. 2020. [Consulté le 2 August 2020]. Disponible à l'adresse : https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html#738

16. KOHNO, Tadayoshi, FERGUSON, Niels and SCHNEIER, Bruce. *Cryptography engineering: Design principles and practical applications*. Indianapolis, IN : Wiley Pub., inc, 2010. ISBN 978-0-470-47424-2.

17. PAAR, Christof and PELZL, Jan. *Understanding Cryptography: A Textbook for Students and Practitioners* [en ligne]. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. [Consulté le 2 August 2020]. ISBN 978-3-642-44649-8 978-3-642-04101-3. Disponible à l'adresse : http://link.springer.com/10.1007/978-3-642-04101-3

18. DWORKIN, Morris. NIST Special Publication (SP) 800-38A: *Recommendation for Block Cipher Modes of Operation: Methods and Techniques* [en ligne]. National Institute of Standards and Technology, 2001. [Consulté le 2 August 2020]. Disponible à l'adresse : https://csrc.nist.gov/publications/detail/sp/800-38a/final

19. AUMASSON, Jean-Philippe and GREEN, Matthew D. *Serious cryptography: A practical introduction to modern encryption*. San Francisco : No Starch Press, 2017. ISBN 978-1-59327-826-7. OCLC: ocn986236585

20. ABDELRAHMAN, Ahmed, FOUAD, Mohamed and DAHSHAN, Hisham. Analysis on the AES Implementation with Various Granularities on Different GPU Architectures. *Advances in Electrical and Electronic Engineering*. 1 October 2017. Vol. 15. DOI 10.15598/aeee.v15i3.2324.

21. HAJIHASSANI, Omid, MONFARED, Saleh Khalaj, KHASTEH, Seyed Hossein and GORGIN, Saeid. Fast AES Implementation: A High-Throughput Bitsliced Approach. *IEEE Transactions on Parallel and Distributed Systems*. October 2019. Vol. 30, n° 10, pp. 2211–2222. DOI 10.1109/TPDS.2019.2911278.

22. RAUBERT, Tim. Bitslicing, An Introduction - Data Orthogonalization for Cryptography. [en ligne]. 15 August 2018. [Consulté le 3 August 2020]. Disponible à l'adresse : https://timtaubert.de/blog/2018/08/bitslicing-an-introduction/

23. LEE, Wai Kong, GOI, Bok-Min, PHAN, Raphael and POH, Geong Sen. High speed implementation of symmetric block cipher on GPU.. 27 January 2015. pp. 102–107. DOI 10.1109/ISPACS.2014.7024434.

24. NIST. *FIPS 197, Advanced Encryption Standard (AES)* [en ligne]. 26 November 2001. NIST. Disponible à l'adresse : https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

25. GUERON, Shay. *Advanced Encryption Standard (AES) Instructions Set*. July 2008. Intel.

26. AOKI, Kazumaro, ICHIKAWA, Tetsuya, KANDA, Masayuki, MATSUI, Mitsuru, MORIAI, Shiho, NAKAJIMA, Junko and TOKITA, Toshio. *Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms*. 18 February 2002. NTT & Mitsubishi.

27. AOKI, Kazumaro, ICHIKAWA, Tetsuya, KANDA, Masayuki, MATSUI, Mitsuru, MORIAI, Shiho, NAKAJIMA, Junko and TOKITA, Toshio. *Specification of Camellia a 128-bit Block Cipher* [en ligne]. 26 September 2001. NTT & Mitsubishi. Disponible à l'adresse : https://info.isl.ntt.co.jp/crypt/eng/camellia/dl/01espec.pdf

28. PROJECT, The OpenSSL, YOUNG, Eric A. and HUDSON, Tim J. OpenSSL crypto library. *GitHub* [en ligne]. 8 December 2020. [Consulté le 12 August 2020]. Disponible à l'adresse : https://github.com/openssl/openssl

29. *AES Rijndael Cipher explained as a Flash animation* [en ligne]. [Consulté le 12 August 2020]. Disponible à l'adresse : https://www.youtube.com/watch?v=gP4PqVGudtg

30. KARONEN, Ilmari. Finite field - How are these AES MixColumn multiplication tables calculated? *Cryptography Stack Exchange* [en ligne]. [Consulté le 12 Au-

gust 2020]. Disponible à l'adresse : https://crypto.stackexchange.com/questions/71204/how-are-these-aes-mixcolumn-multiplication-tables-calculated

31. DAEMEN, Joan and RIJMEN, Vincent. *The design of Rijndael: AES–the Advanced Encryption Standard*. Berlin ; New York : Springer, 2002. ISBN 978-3-540-42580-9.

32. LUITJENS, Justin. Global Memory Usage and Strategy. *GPU Computing Webinar*. Webinar. 12 July 2011.

33. HARRIS, Mark. How to Optimize Data Transfers in CUDA C/C++. *NVIDIA Developer Blog* [en ligne]. 5 December 2012. [Consulté le 10 August 2020]. Disponible à l'adresse : https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

34. BASSHAM III, Lawrence E. *The Advanced Encryption Standard Algorithm Validation Suite*. USA : NIST, 2002.

35. NTT. Camellia Technical Information & Open Source. *NTT Cryptographic Primitive* [en ligne]. [Consulté le 12 August 2020]. Disponible à l'adresse : https://info.isl.ntt.co.jp/crypt/eng/camellia/technology/

36. HOUSLEY <HOUSLEY@VIGILSEC.COM>, Russ. RFC 5652 - Cryptographic Message Syntax (CMS). *IETF* [en ligne]. September 2009. [Consulté le 13 August 2020]. Disponible à l'adresse : https://tools.ietf.org/html/rfc5652#section-6.3