Bachelor Informatica

University of Amsterdam

Informatica — Universiteit van Amsterdam

A Performance Analysis of High-Level Synthesis for FPGAs

Pim Kunis

July 3, 2021

Supervisor: Ana-Lucia Varbanescu

Signed:

Abstract

It is becoming increasingly popular to use Field-Programmable Gate Arrays (FPGAs) as hardware accelerators in order to speed up certain parts of an algorithm. FPGAs promise more energy efficiency and increased performance compared to CPUs. They are, however, traditionally programmed with Hardware Description Languages (HDLs), which are notoriously hard to use.

To accelerate their adoption in different contexts and domains, FPGAs can nowadays *also* be programmed with high-level languages, such as C/C++ or OpenCL, in a process called High-Level Synthesis (HLS). However, it can be a challenge to efficiently accelerate algorithms using HLS.

In this thesis, we investigate the performance of HLS for a non-trivial case-study. To this end, we devise a performance comparison between a sequential CPU algorithm and its FPGA version, programmed with OpenCL for the Xilinx Vitis platform. Our case study is a string-searching algorithm using the FM-index method, which is able to efficiently locate substrings in arbitrarily long texts.

Our naive reference implementation is a simple port from CPU to OpenCL. In search for more performance, on top of this naive OpenCL implementation, we also propose and evaluate two FPGA-specific optimizations, suggested by literature.

Our empirical analysis shows that optimizations can greatly improve the performance of algorithms programmed with HLS for FPGAs. However, the performance of our HLS-based FPGA version for our case study could not match the performance of the CPU, despite being more work efficient.

Contents

1 Introduction & Research Question **7**
- 1.1 Introduction . . . 7
- 1.2 Research Question . . . 8
- 1.3 Outline . . . 8

2 Background & Related Work **9**
- 2.1 Field-programmable gate arrays . . . 9
  - 2.1.1 Choosing hardware platform . . . 9
  - 2.1.2 FPGA architecture . . . 10
- 2.2 OpenCL . . . 10
- 2.3 String matching algorithm . . . 11
  - 2.3.1 Burrows-Wheeler transform . . . 11
  - 2.3.2 FM-index . . . 13
- 2.4 Related Work . . . 15
  - 2.4.1 OpenCL for high-level synthesis . . . 15
  - 2.4.2 Optimizing OpenCL . . . 15
  - 2.4.3 Synthesizing FM-index . . . 16

3 Reference Application for CPU **17**
- 3.1 Reference implementation . . . 17
  - 3.1.1 Interface . . . 17
  - 3.1.2 String searching . . . 18
  - 3.1.3 Burrows-Wheeler transform construction . . . 19
- 3.2 Experimental setup . . . 19
- 3.3 Results . . . 20

4 Reference Application for FPGA **25**
- 4.1 FPGA implementation . . . 25
  - 4.1.1 Host program . . . 25
  - 4.1.2 FPGA kernel . . . 25
- 4.2 Empirical evaluation . . . 26
- 4.3 Results . . . 27

5 Optimizations **33**
- 5.1 Optimizations . . . 33
  - 5.1.1 Memory optimizations . . . 33
  - 5.1.2 OpenCL's NDRange . . . 34
  - 5.1.3 Combined optimizations . . . 35
- 5.2 Empirical evaluation . . . 35
- 5.3 Results . . . 35

6 Challenges **41**
6.1 Toolchain . . . 41
6.2 Development process . . . 41
6.2.1 Software emulation . . . 41
6.2.2 Hardware emulation . . . 41
6.2.3 Hardware compilation . . . 42
6.3 Results gathering . . . 42

7 Conclusion and future work **43**
7.1 Main findings . . . 43
7.2 Contributions . . . 44
7.3 Limitations and threats to validity . . . 44
7.4 Ethical considerations . . . 45
7.5 Future work . . . 45

A Optimized kernels **49**
A.1 NDRange optimized kernel . . . 50
A.2 Memory-access optimized kernel . . . 51
A.3 Final optimized kernel . . . 52

CHAPTER 1

Introduction & Research Question

1.1 Introduction

Scientists and companies are always looking to speed up the execution of algorithms, especially those that are slow on CPUs. One way to achieve better performance is by using accelerators which can execute part of an algorithm faster. For example, graphical processing units (GPUs) are a common accelerator to speed up graphics and vector processing.

A recent trend is to utilize field-programmable gate arrays (FPGAs) for hardware acceleration and speed up parts of algorithms. An FPGA is a device with circuits that can be reprogrammed after manufacturing. They are often compared with application-specific integrated circuits (ASICs), which are very fast chips designed for one purpose, but cannot be reprogrammed after manufacturing. In terms of performance and reconfigurability, FPGAs offer a good trade-off between CPUs and ASICs: a CPU is often slower, but can be instructed to do anything, while the specialized ASICs are much faster [27]. Another great benefit of FPGAs is the low power consumption in comparison to CPUs and GPUs.

Traditionally, FPGAs are programmed using low-level Hardware Description Languages (HDL) that describe the circuits needed to run an algorithm. Unlike sequential programming languages, like C or python, HDLs use a concurrent model where data flow is done in parallel [27]. Because of the difference in programming paradigms, many computer programmers experience difficulty programming with HDLs.

To alleviate this problem, past research has focused on High-Level Synthesis (HLS) which aims to synthesize more high-level programming languages and models like C, Haskell or OpenCL into programs that run on FPGAs [20]. While programming FPGAs using high-level models is convenient and accessible, it can be challenging to efficiently synthesize algorithms.

In this project, we analyze the performance of FPGA algorithms developed using HLS. To this end, we will analyze the Vitis HLS toolkit, developed by Xilinx.

Vitis supports several high-level languages for HLS, among which we have chosen OpenCL. OpenCL is an open standard for heterogeneous computing, meaning it is possible to execute programs across multiple platforms, including CPUs, GPUs, and FPGAs [17].

Our analysis is based on a detailed case-study: a full-text substring index called FM-index, which has applications in fields including bioinformatics [21]. Using the FM-index, it is possible to quickly find the position of any substring in an arbitrarily long text.

To evaluate the performance of the FPGA-accelerated FM-index, we compare its throughput and energy consumption when finding patterns inside a text, against those measured for a CPU. For this comparison, we provide a sequential CPU application as well as a reference FPGA application. We further optimize the reference FPGA implementation, and further measure its performance improvement against the reference versions.

1.2 Research Question

The goal of this work is to analyze the performance of HLS for FPGAs. Therefore, our main research question is formulated as follows: *How well does high-level synthesis perform for programming FPGA accelerators?*

To help answer this question, we formulate three subquestions:

1. How does a naive HLS-based FPGA application compare against its CPU counterpart?
2. What optimizations can improve the performance of an HLS-based FPGA application?
3. How difficult is it to develop FPGA applications using HLS?

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 discusses theoretical background regarding FPGAs, introduces the selected case-study for performance analysis, and provides a short analysis of related work. In chapter 3, we present our CPU reference implementation and analyze its performance using a comprehensive experimental setup. Chapter 4 presents our reference implementation for the FPGA, and perform an empirical evaluation of its performance. In chapter 5 we present and evaluate two optimizations for the reference FPGA application; we further present the empirical evaluation of these optimizations, and a performance comparison with the reference applications. Chapter 6 states several challenges we have encountered when developing applications with HLS for FPGAs. Lastly, in chapter 7 we formulate our answer for the research question, and present possible future work.

CHAPTER 2

Background & Related Work

In this section we provide a brief introduction to the main concepts required to understand the remainder of this work: FPGAs and HLS, OpenCL, and the chosen case-study. Finally, we also summarize relevant related work.

2.1 Field-programmable gate arrays

Field-programmable gate arrays (FPGAs) are devices that have circuits which can be programmed after manufacturing [10]. In this section we explain the motivation for using FPGAs and how they work.

2.1.1 Choosing hardware platform

Figure 2.1 shows a simplified relation between CPUs, FPGAs and Application-Specific Integrated Circuits (ASICs) in terms of flexibility and efficiency. On the one end of the spectrum, we have CPUs which are easy to program and can be used to perform any task. However, the CPU has comparatively inefficient power consumption and performance. However, CPUs can be inefficient in terms of power consumption and performance for certain applications. On the other side of the spectrum we have ASICs, which are specialized chips, designed for one specific application. This means the design of an ASIC must be near-perfect when fabricating, which leads to a long time-to-market. However, the big advantage of ASICs is their efficiency: they are very energy efficient and fast. Between these extremes of flexibility and efficiency, fits the FPGA. Compared to CPUs, FPGA often perform more work per clock cycle as we program their circuits, instead of software. On the other hand, programming FPGAs is often harder and compilation is slow. Finally, FPGAs are more flexible than ASICs as they can be reprogrammed, but are slower.

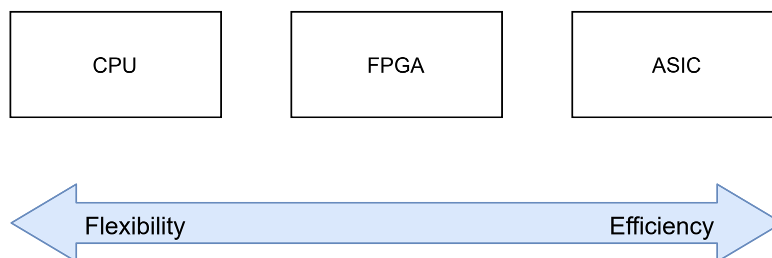

Figure 2.1: The relation between CPU, FPGA and ASIC in terms of flexibility and efficiency.

FPGAs are traditionally programmed using Hardware Description Languages (HDLs), such as VHDL and Verilog [18]. These languages describe how the hardware should behave in order to execute a specific algorithm. Because programming with HDLs requires a thorough understanding

of the targeted hardware, it is often a difficult task. This problem is addressed by the use of High-Level Synthesis (HLS). Using HLS, it is possible to automatically produce a circuit specification from a high-level language, such as C, C++ or OpenCL. This allows programmers to exploit the efficiency of the FPGA using a more familiar language.

2.1.2 FPGA architecture

As previously explained, the internal circuits of FPGAs can be reprogrammed. We will explain how this works, using terminology sometimes specific to Xilinx products (note: this thesis uses Xilinx hardware).

The basic, repeating element of an FPGA is the Configurable Logic Block (CLB) [24]. The CLB consists of several Lookup Tables (LUTs) and flip-flops. A LUT is basically a truth table used to implement a Boolean function with N inputs. The flip-flops are small storage units that store results between clock cycles.

The CLBs wired together are capable of implementing complex algorithms. An illustration of the wiring of CLBs is shown in figure 2.2.

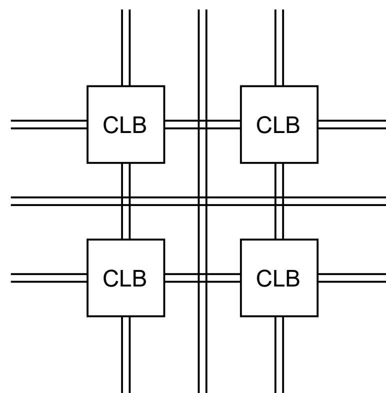

Figure 2.2: Several Configurable Logic Blocks (CLBs) wired together.

Modern FPGAs also have different options for memory. Physical memory chips are attached to FPGAs to provide large amount of storage, accessible to both the FPGA and the host processor. Faster memory is implemented as Block RAM (BRAM), located in the FPGA fabric itself.

Finally, FPGAs also include DSP blocks, which are Arithmetic Logic Units (ALUs) embedded in the fabric of the FPGA. These are used to perform more complex arithmetic computations.

2.2 OpenCL

OpenCL is a standard for heterogeneous computing, which allows programmers to write parallelized code to improve performance [17]. The term “heterogeneous” refers to executing programs on different kinds of processors or cores, such as CPUs, GPUs and FPGAs.

The OpenCL standard and API are managed by the Khronos Group [22]. Hardware vendors that support OpenCL implement the OpenCL standard for their own products (usually combining compilers, toolchains, and run-time systems). The standard is effectively defined by four core components: the platform model, the memory model, the execution model and the programming model.

Platform model

OpenCL’s platform model consists of one host computer connected to one or more OpenCL devices. The host is most often a CPU. The OpenCL devices can be divided into multiple compute units, which are further divided into processing elements. The host submits commands to the OpenCL devices, which run computations on the processing elements.

Execution model

The execution model is also divided into two parts: a host program runs on the host, while OpenCL kernels run on the OpenCL devices.

When the host program submits a kernel to the OpenCL devices, it must specify an index space. The index space determines how the execution can be subdivided into smaller stand-alone parts. These parts are called work-items, and each is executed by an instance of the kernel. For example, the execution of vector addition could be divided such that one work-items calculates one item in the final vector. The work-items can be grouped together in work-groups, to divide the index space in a more coarse-grained manner.

The index range supported by OpenCL is called "NDRange". An NDRange divides the index space into one, two or three dimensions.

Memory model

The OpenCL memory model specifies different memory regions that kernels have access to:

1. *Global* and *constant memory*: memory accessible to both the host and all work-items. In case of constant memory, only the host program is allowed to initialize it, while work-items can only read it.

2. *Local memory*: memory accessible to a single work-group. All work-items within the work-group can access this memory. The host cannot access this memory.

3. *Private memory*: memory accessible only to a single work-item. The host cannot access this memory.

Programming model

Finally, OpenCL supports two programming models: data parallel and task parallel.

In the data parallel model, OpenCL makes use the index space to parallelize execution over the work-items. In the task parallel model, a single kernel is executed for each task, irregardless of index space. Parallelism can instead be expressed by enqueuing multiple tasks.

2.3 String matching algorithm

A string matching algorithm is used to find a query string, called the *pattern*, within a much bigger string, called the *text* [15]. There are two kinds of string matching algorithms: sequential and indexed algorithms. Sequential string matching relies on the original unprocessed text, which it traverses sequentially to find the requested pattern. Indexed string matching algorithms operate on data structures constructed from the original text called an *index*. The use of specialized data structures allows for finding a pattern without searching through the whole text. This method is preferred in the following scenarios: Firstly, when sequentially searching the original text simply takes to much time. Next, the text must not change often, as the whole index would have to be rebuilt. Lastly, sufficient storage space is available to store the index.

The string matching algorithm we have chosen for our case study on the FPGA is the FM-index invented by Ferragina and Manzini in 2000[4]. The FM-index is a so-called self-index, which means that it is capable of reproducing the original text and its suffixes [15]. An integral part of the efficient string matching of the FM-index is the Burrows-Wheeler transform. Both the Burrows-Wheeler transform and the FM-index will be described in the following section.

2.3.1 Burrows-Wheeler transform

The Burrows-Wheeler transformation (BWT) is an operation on a block of text T, which produces a permutation of the text called the Burrows-Wheeler transform T^{bw}, first described by Burrows and Wheeler in 1994 [1]. T^{bw} can often be more easily compressed than the original text and,

interestingly, can be reversed to reproduce the original text. This section will explain how the BWT works.

Creating T^{bw} from a block of text can be achieved using a few simple steps, as described in the original paper:

1. Append a special character, here `$`, to the original text to denote the end of the text.
2. List all rotations of the text, by putting a character from the end of the text to the front.
3. Sort the rotations in lexicographical ordering, where the special character `$` has priority over any other character.
4. T^{bw} is acquired by taking the last column of the matrix with the sorted rotations as rows (also known as the Burrows-Wheeler matrix).

A graphical representation of the steps of this algorithm can be seen in figure 2.3 for the text `ALALA`, which will be our running example.

It may not be immediately clear how T^{bw} can be more easily compressed than the original text. To better see this, take for example the string

“`zeven␣schotse␣scheve␣schaatsers␣schaatsen␣scheef`”

which is transformed into

“`fsenenhhaassssssvshesvshzeccccceeher␣␣␣␣␣tttoaaee$`”

according to the ASCII character ordering. In T^{bw} there are multiple runs of the same character, which can be more efficiently encoded by, for example, a run-length encoder.

To understand why these runs occur in T^{bw}, we look at the sorted rotations in figure 2.3, and especially to the last two rows. We see that a character in the last column essentially comes before the character in the first column. What's more, we see that there are two places in the text where an `A` precedes an `L`. Because we sorted the rotations starting from their first character, there will be a run of the character `A` in the last column and therefore in T^{bw}. In most texts, there are usually only a few distinct characters that can precede a character, which results in runs in the last column [1].

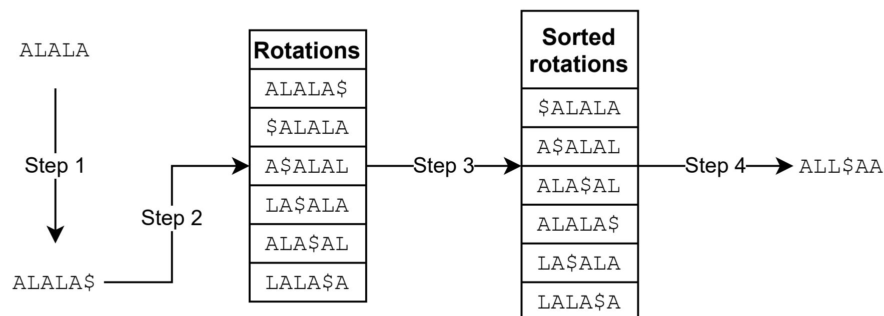

Figure 2.3: The steps to acquire the Burrows-Wheeler transform of a block of text.

T^{bw} can also be reversed to reproduce the original text. To this end, we recognize that we can obtain the first column of the Burrows-Wheeler matrix by lexicographically sorting T^{bw}. Next we can use the principle of Last-First-Mapping (LF-mapping) to find a character in the first column, given the same character in the last column [4]. The LF-mapping recognizes that the ordering, or rank, of a character in the first column is the same as in the last column. We can empirically check this by looking at the sorted rotations in figure 2.3. The first `L` encountered in the first column is the same as to the first one encountered in the last column, and vice versa for the second `L`.

The whole process of finding the original text is shown in figure 2.4. We start with the unique character `$` which is always the last character. Then we can find the position of `$` in the first

column using the LF-mapping. The character preceding `$` can then be obtained by looking in the last column on the same row. These steps are performed until the last character is encountered again and we are done.

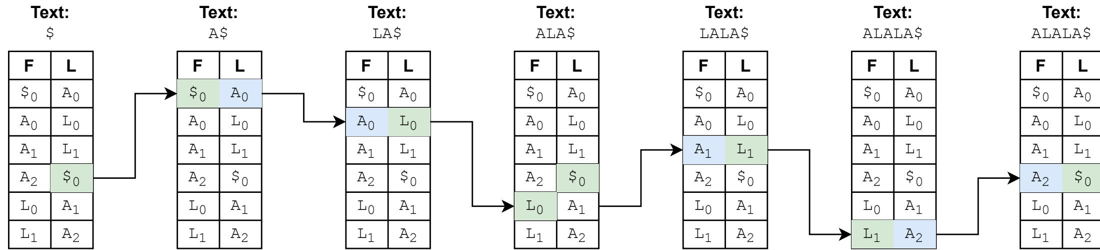

Figure 2.4: Reverting the BWT to obtain the original text. F and L indicate the first and last column of the Burrows-Wheeler matrix respectively. The highlighted row in the table correspond to the LF-mapping currently being considered, and the text above the table shows the intermediate resulting text.

2.3.2 FM-index

The FM-index is an index, which consists of the BWT and some small auxiliary array-based data structures, first described by Ferragina and Manzini [4]. ”FM” supposedly means ”Full-text Minute-space” but could also refer to the authors [12]. The FM-index allows for two operations: counting the occurrences of a query pattern, and finding the original position indices of the pattern in the original text. Both operations are accompanied with an array-based data structure to speed up searching.

Finding the number of occurrences of a pattern in a text also makes use the LF-mapping, just like in figure 2.4 where we recovered the original text from T^{bw}. The procedure involves keeping track of a range of rows in the Burrows-Wheeler matrix which currently match the pattern. To demonstrate the procedure we use our running example of the text `ALALA` and the pattern `ALA`, which is shown in figure 2.5. We iterate over the characters in the pattern backwards, and therefore start with matching the last character, namely `A`. The character `A` occurs three times in the Burrows-Wheeler matrix, namely the rows starting with A_0, A_1 and A_2. Using the LF-mapping property, we can then find which characters precede these `A`s. We find that two preceding characters matching the `L` in the pattern, namely the characters L_0 and L_1. Continuing, we further find that each of the `L`s is preceded by an `A` character, which match the first character of the pattern. We conclude that the pattern `ALA` occurs twice in the text `ALALA`.

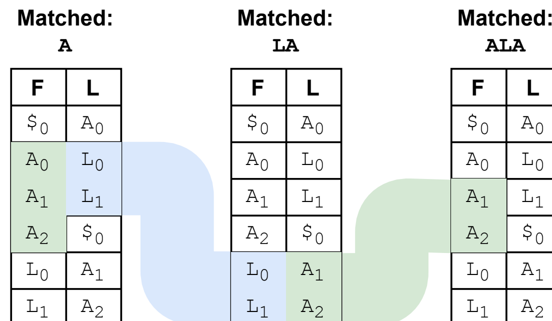

Figure 2.5: Iteratively finding the range of occurrences of a pattern. Each table represents a Burrows-Wheeler matrix and its first and lats column. The text above the table shows the pattern that has been matched. F and L indicate the first and last column of the Burrows-Wheeler matrix respectively.

In practice, we do not keep track of each row that is matched. Rather, we can save the

beginning and ending row of the *range* of matches. This is possible because the Burrows-Wheeler matrix is sorted lexicographically. Any matches are guaranteed to appear next to each other.

Calculating the LF-mapping each time is actually very costly and therefore Ferragina and Manzini propose the use of a precalculated *rank matrix* [4, 13]. The rank matrix stores for each position in the Burrows-Wheeler matrix, how many characters have been encountered thus far. The rank matrix for our running example of text `ALALA` is shown in figure 2.6. If we start again with the rows beginning with A_0, A_1 and A_2 in the Burrows-Wheeler matrix, we use the rank matrix to find which `L` characters are preceding. We see that before the range, no `L` has been encountered. Furthermore, at the end of the range, two `Ls` have been encountered. Therefore we conclude that L_0 up to L_1 (in this case all `Ls`) precede an `A`.

Burrows-Wheeler matrix

F	L
$\$_0$	A_0
A_0	L_0
A_1	L_1
A_2	$\$_0$
L_0	A_1
L_1	A_2

Rank matrix

$	A	L
0	1	0
0	1	1
0	1	2
1	1	2
1	2	2
1	3	2

Figure 2.6: The calculated rank matrix for a Burrows-Wheeler matrix.

The resulting procedure `bw_count_occurences` is shown as pseudocode in listing 1. The variables `start` and `end` denote the current range, and is initialized with whole range of the last character in the pattern. Then, we loop backwards through the pattern and narrow the range to matching rows each iteration. The auxiliary procedure `start_range` returns the position of the first row that start with the given character, which can be calculated in constant time due to the ordering.

```
def bw_count_occurences(pattern):
  c = pattern[-1]
  i = len(pattern) - 1
  start, end = character_range(c)
  while start <= end and i > 0:
    c = pattern[i-1]
    start = start_range(c) + count_occurences(c, start-1)
    end = start_range(c) + count_occurences(c, end-1)
    i -= 1

  if end < start:
    return 0
  else
    return end - start
```

Listing 1: Pseudocode for finding the number of matches of a pattern in a text.

Having found the range of starting characters for our pattern, we want to find their positions in the original text as well. To this end, we can use an additional data structure, called the suffix array [11]. A suffix array stores for each character in T^{bw} the position in the original text that the character corresponds to.

In figure 2.7, the suffix array is shown for our running example of the text `ALALA`. We are finally ready to identify where the pattern `ALA` occurs in the original text. We have previously found that the `A`s with rank one and two are the starting positions of the pattern within the text. These in turn correspond to the last two `A`s in T^{bw}. Consulting the suffix array in figure 2.7 tells us the pattern occurs at index zero and two in the original string. It can be easily checked that this is correct.

T	T^{bw}	Suffix array
A	A	4
L	L	3
A	L	1
L	$	5
A	A	2
$	A	0

Figure 2.7: The original text T (`ALALA`), its Burrows-Wheeler transform T^{bw} and the suffix array are shown alongside each other.

2.4 Related Work

In this section, we summarize related work about high-level synthesis using OpenCL and string-search algorithms.

2.4.1 OpenCL for high-level synthesis

The use of OpenCL in conjunction with FPGAs goes as far back as 2010. OpenRCL is a system that enables executing code on an FPGA, and supports parallelization and multithreading through the OpenCL API. However, the OpenCL code is not synthesized into hardware, but rather runs on computing cores on the FPGA [14]. The SOpenCL system is the first synthesis tool capable of converting OpenCL C code into synthesizable HDL [19]. In the industry, Intel has developed its FPGA SDK for OpenCL for its FPGA hardware, while Xilinx provides its Vitis platform (formerly SDAccel) to program Xilinx FPGAs [8, 26].

In this thesis, we use the OpenCL implementation from Xilinx's Vitis platform. Although our code and optimizations are designed for this specific OpenCL implementation, they should be portable to other vendors' platforms.

2.4.2 Optimizing OpenCL

Case studies have shown that OpenCL HLS can outperform CPUs for certain algorithms. Additionally, OpenCL HLS kernels are often more power-efficient than their CPU counterparts.

For example, Zohouri et al. present three optimization techniques to optimize the performance of OpenCL HLS [27]. The first technique they show is explicit vectorization of loops in the algorithm. This replaces a loop with operations that can be parallelized. Secondly, the authors show an NDRange approach where a task is split into independent parts that are executed by separate kernel. Finally, they show that data transfer between the host system and the kernels can sometimes be done in bursts depending on the amount of memory on-board, to limit the number of transfers while transferring as much data as possible at once. However, the data-transfer technique could result in non-portable code, as the technique is optimized for a certain FPGA architecture. The paper shows how these optimizations lead to 1.5× speedup for four of twelve experiments conducted. Moreover, no matter what optimization applied, using the FPGA to offload the algorithm results in 60% to 80% less energy consumption.

Paulino, Ferreira, and Cardoso also present a number of techniques to optimize OpenCL HLS performance, and compare its performance to both CPUs and GPUs [20]. Firstly, using OpenCL pragmas, one can indicate that a kernel can be replicated as separate pipelines to exploit data parallelism. Other optimizations are loop unrolling and using shift registers when performing calculations on floating-point numbers. Finally, for algorithms that exhibit a structured grid computation pattern, the advanced sliding window technique can be used. The authors investigate the performance benefits of these techniques using benchmarks from the Rodinia suite. Experimental results from these benchmarks show that FPGAs are slightly faster than CPUs, but not better than GPUs. The FPGAs are however up to 3.4× more energy efficient than GPUs at executing the benchmarks using the presented optimizations.

These studies show that optimization techniques for OpenCL FPGA kernels can improve speed and energy consumption.

2.4.3 Synthesizing FM-index

The implementation string-searching using the FM-index in hardware has been studied by Fernandez, Najjar, and Lonardi in 2011 [3]. However, in their study, the FM-index was designed using a hardware description language. The authors indicate up to 196× speedup when compared to a brute-force CPU implementation. Moreover, Ullah, Ben Ahmed, and Amano researched the synthesis of FM-index using HLS on a multi-FPGA system [23]. On a single board implementation, the implementation was 10× faster than on a CPU. They effectively showed that using multiple FPGAs connected in a ring network, throughput can be scaled and increased. These studies show that accelerating string-searching using FM-index on FPGAs can result in increased throughput.

CHAPTER 3

Reference Application for CPU

In this chapter, we present our reference CPU application. We also present an experimental setup, which we use the performance of the FM-index implementations.

3.1 Reference implementation

To analyze the performance of an FPGA implementation, we have programmed a CPU-based reference implementation in the C language. We have chosen this language for its performance and for easy portability to an FPGA kernel written in OpenCL. As will be explained in section 3.2, we will focus on analyzing the performance of string-matching, and not the construction of the FM-index. Therefore we can precalculate all auxiliary data structures that were previously explained in section 2.3.

3.1.1 Interface

The data structures comprising the FM-index are implemented as a C struct shown in listing 2. The suffix array is henceforth abbreviated as `sa` in code listings.

```c
typedef unsigned ranges_t;
typedef unsigned ranks_t;
typedef unsigned sa_t;

typedef struct fm_index {
  char *bwt;
  size_t bwt_sz;
  char *alphabet;
  size_t alphabet_sz;
  ranks_t *ranks;
  sa_t *sa;
  ranges_t *ranges;
} fm_index;
```

Listing 2: The C struct for the FM-index. The `bwt`, `ranks` and `sa` data structures correspond to the Burrows-Wheeler transform, rank matrix and suffix array respectively. The data structure `ranges` is used to lookup a character in the Burrows-Wheeler matrix.

The exposed functions are presented in listing 3. The `FMIndexConstruct` function constructs the FM-index from the given text, and `FMIndexFree` frees the FM-index and all underlying data structures. The functions `FMIndexDumpToFile` and `FMIndexReadFromFile` can be used to save and read an FM-index to and from a file. The flag `aligned` specifies whether to allocate

memory page-aligned, which is useful for communication with the FPGA in chapter 4. Finally, the `FMIndexFindMatchRange` and `FMIndexFindRangeIndices` functions are used to find the indices of a given pattern within a text, as explained in section 2.3.2. `FMIndexFindMatchRange` finds the start and end of the range of matches for a pattern, and saves these in `start` and `end` respectively. `FMIndexFindRangeIndices` finds all indices using the earlier acquired range, and saves them in `match_indices`.

```
fm_index *FMIndexConstruct(char *s);
void FMIndexFree(fm_index *fm);

int FMIndexDumpToFile(fm_index *fm, char *filename);
fm_index *FMIndexReadFromFile(char *filename, int aligned);

void FMIndexFindMatchRange(fm_index *fm, char *pattern, size_t pattern_sz,
                           ranges_t *start, ranges_t *end);
void FMIndexFindRangeIndices(fm_index *fm, ranges_t start, ranges_t end,
                             unsigned long **match_indices);
```

Listing 3: The FM-index C functions exposed by the interface.

3.1.2 String searching

The two functions from section 3.1.1 to perform string searching using the FM-index are implemented as shown in listing 4. The `FMIndexFindMatchRange` function closely follows the pseudocode in listing 1. The `FMIndexFindRangeIndices` uses the found range of matches, and performs a simple lookup in the suffix array to find the original indices.

```
void FMIndexFindMatchRange(fm_index *fm, char *pattern, size_t pattern_sz,
                           ranges_t *start, ranges_t *end) {
  int p_idx = pattern_sz - 1;
  char c = pattern[p_idx];

  *start = fm->ranges[2 * string_index(fm->alphabet, c)];
  *end = fm->ranges[2 * string_index(fm->alphabet, c) + 1];

  p_idx -= 1;
  while (p_idx >= 0 && *end > 1) {
    c = pattern[p_idx];
    ranges_t range_start = fm->ranges[2 * string_index(fm->alphabet, c)];
    int alphabet_idx = string_index(fm->alphabet, c);
    *start =
        range_start + fm->ranks[fm->alphabet_sz * (*start - 1) + alphabet_idx];
    *end = range_start + fm->ranks[fm->alphabet_sz * (*end - 1) + alphabet_idx];
    p_idx -= 1;
  }
}

void FMIndexFindRangeIndices(fm_index *fm, ranges_t start, ranges_t end,
                             unsigned long **match_indices) {
  for (unsigned long i = 0; i < end - start; ++i)
    (*match_indices)[i] = fm->sa[start + i];
}
```

Listing 4: The full implementation of string searching using the FM-index.

3.1.3 Burrows-Wheeler transform construction

We previously explained in section 2.3.1 that, in order to obtain T^{bw}, we list all rotations of the text and then lexicographically sort them. This would unfortunately lead to a space complexity of $O(n^2)$ which is infeasible for large data sets. To avoid this issue, we can derive T^{bw} from the suffix array of the text [23]. The suffix array is essentially a Burrows-Wheeler transform, but holds indices instead of characters. Deriving T^{bw} is therefore as easy as indexing the text for each index in the suffix array.

The C code for obtaining the suffix array from a text is shown in listing 5. First we simply list all indices of the text. Then we use quicksort to sort the indices: for each pair of indices, the strings starting from these positions are lexicographically compared.

```c
sa_t *ConstructSuffixArray(char *text, size_t text_sz) {
  sa_t *suffix_array = calloc(text_sz + 1, sizeof(sa_t));

  for (size_t i = 0; i < text_sz + 1; ++i)
    suffix_array[i] = i;

  qsort_r(suffix_array, text_sz + 1, sizeof(sa_t), &CompareSuffixArray, text);

  return suffix_array;
}
```

Listing 5: Generating the suffix array for a text. The auxiliary function `CompareSuffixArray` lexicographically compares two suffixes, taking the sentinel character `$` into account which is always sorted first.

3.2 Experimental setup

We investigate the performance on the CPU and FPGA using two metrics found in literature. To evaluate the overall performance, throughput is measured by counting the average amount of patterns matched (finding the corresponding original positions) per second. As one of the main benefits of FPGAs is energy efficiency, we also measure the energy consumption when running the experiments.

The string searches are performed on FM-indices which we preprocess on the CPU. We use several corpora provided by the Pizza & Chili corpus collection [5] to compute the FM-indices. This collection is often used for text compression and indexing, and features corpora from a diverse set of sources [15]. We use the *dna*, *proteins* and *dblp.xml* corpora for our experiments. An explanation for each corpus and the alphabet size is shown in table 3.1.

Corpus	**Description**	$\lvert\Sigma\rvert$
dna	DNA sequences provided by Project Gutenberg. The DNA bases are encoded as A, C, G and T, but also contains special characters.	16
proteins	Protein sequences provided by the Swiss-Prot database. The 20 amino acids are encoded as upper-case letters. Also contains special characters.	27
dblp.xml	Bibliographic information on computer science journals in XML format, provided by DBLP.	97

Table 3.1: Description and alphabet size of the three corpora used from the Pizza & Chili corpus collection.

To measure throughput, we generate as input for the experiment 7500 random patterns that occur in the text. Generating random patterns from the alphabet alone gives too many patterns that have no occurrences in the original text, especially for the *dblp.xml* corpus which has a high alphabet size. We therefore think taking patterns that actually occur in the original text better

reflects a normal string search. We experiment with different pattern lengths, namely lengths 4, 6 and 8. We chose these lengths because smaller lengths would give way too many matches for corpora with a small alphabet size, while bigger lengths would give almost no matches at all for a large alphabet size.

We also investigate the effect of the size of the corpus. Therefore, for each corpus, we perform experiments on 10, 15 and 20 MB versions of the corpus. We decided to use these sizes, as larger texts would result in buffers bigger than 4 GB which are unsupported on the chosen FPGA platform.

Energy consumption is measured for each run using Intel's Running Average Power Limit (RAPL). This interface enables to enforce power limits, but also to read energy consumption at run-time [7]. It is only possible to measure the energy consumption of the entire CPU socket, so we cannot measure the energy usage of the process our experiments are running on. This makes energy measurements less accurate but still serves as an illustration of how much energy a computer would consume when running the algorithm. To read the RAPL values, we use the Power Capping Framework of the Linux kernel [2].

Lastly, each experiment is performed five times to limit variability.

The CPU experiments are performed on an Intel Xeon Gold 6128 processor [9]. This CPU has a base frequency of 3.40 GHz.

We expect that a higher pattern length will result in a lower throughput, as string searching using the FM-index loops over the characters of the pattern. Subsequently, we expect that the size of the corpus has a negligible effect on the throughput. Lastly, we hypothesize that corpora with a larger alphabet, i.e. the *dblp.xml* corpus, have a lower throughput, as the data structures are larger which could negatively impact memory performance.

3.3 Results

Figure 3.1 shows the average throughput of the three corpora with different pattern length and corpus sizes. The top throughputs reach more than 400 thousand patterns matches per second, while we also see some in the lower ten thousands. We see that, contrary to the hypothesis, a longer pattern generally leads to higher throughput. To explain this fact, table 3.2 shows the average number of matches per pattern. Not surprisingly we see that longer patterns have less occurrences than shorter ones. For shorter patterns, we therefore have to look up way more positions in the suffix array. The memory lookups explain why longer patterns in figure 3.1 are faster to search for.

When we look at the effect of the corpus size, we see that for the *dna* and *dblp.xml* corpora, a bigger text leads to lower throughputs. However, the lower throughput does not seem to be linear to the corpus size. We hypothesize that the lower throughput could be the result of worse memory caching, as random accesses in the data FM-index data structures are more likely to result in page faults.

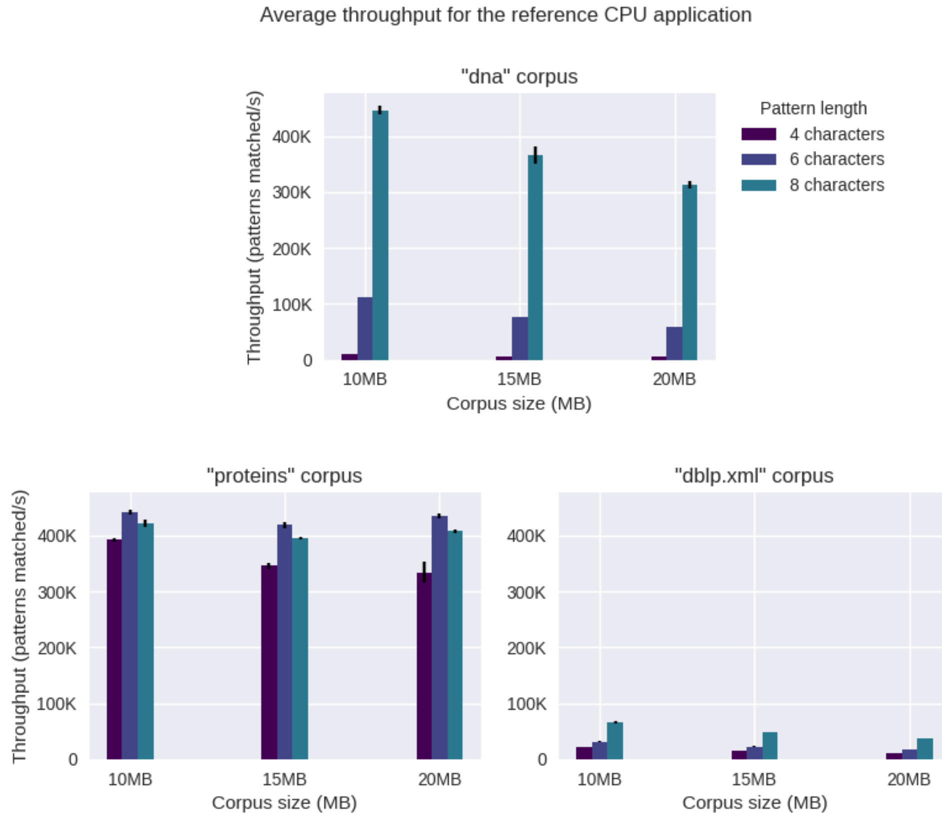

Figure 3.1: Average throughput (in patterns matched per second) for the reference CPU application. Different corpora, text sizes and pattern lengths are investigated and the average is taken over 5 runs for each configuration.

We also observe that the *proteins* corpus has a comparable throughput independent of corpus size or pattern length. If we again look at table 3.2, we see that the average number of matches per pattern for the *proteins* corpus is comparatively low for each text size. This is seemingly a property of this corpus: any substring of the text does not have many occurrences. This results in fewer memory lookups for the original positions, resulting in a better throughput.

Lastly, when looking at figure 3.1 we do indeed see that the *dblp.xml* corpus with the high alphabet size has a throughput almost 7× lower than the other two corpora.

Average match count

Pattern length	dna			proteins			dblp.xml		
	10MB	15MB	20MB	10MB	15MB	20MB	10MB	15MB	20MB
4	52714	79095	99805	640	802	786	22930	33816	43540
6	4090	6166	8249	364	402	335	14903	21964	28237
8	400	591	811	270	315	250	6876	9954	12882

Table 3.2: Average amount of matches per pattern. Different corpora, text sizes and pattern lengths are investigated and the average is taken over 5 runs for each configuration.

The average energy consumption of running the experiments is shown in figure 3.2. Note that this figure shows the total energy consumption over the 7500 patterns. We see that the *dna* corpus causes the most energy usage, with at most 160 joules when searching four-character long

patterns. Longer pattern lengths significantly reduce the energy consumption. We believe this is again due to the amount of occurrences as shown in table 3.2. Searching for four-character long patterns inside a 20MB *dna* corpus results in more than a hundred thousand occurrences. The original position of each occurrence needs to be looked up, and this could cause the high energy consumption. Furthermore, we see a very low energy consumption, around 2.5 joules, for all experiments with the *proteins* corpus. Just like we concluded for the throughput results, we believe the low energy usage could be due to the low amount of matches.

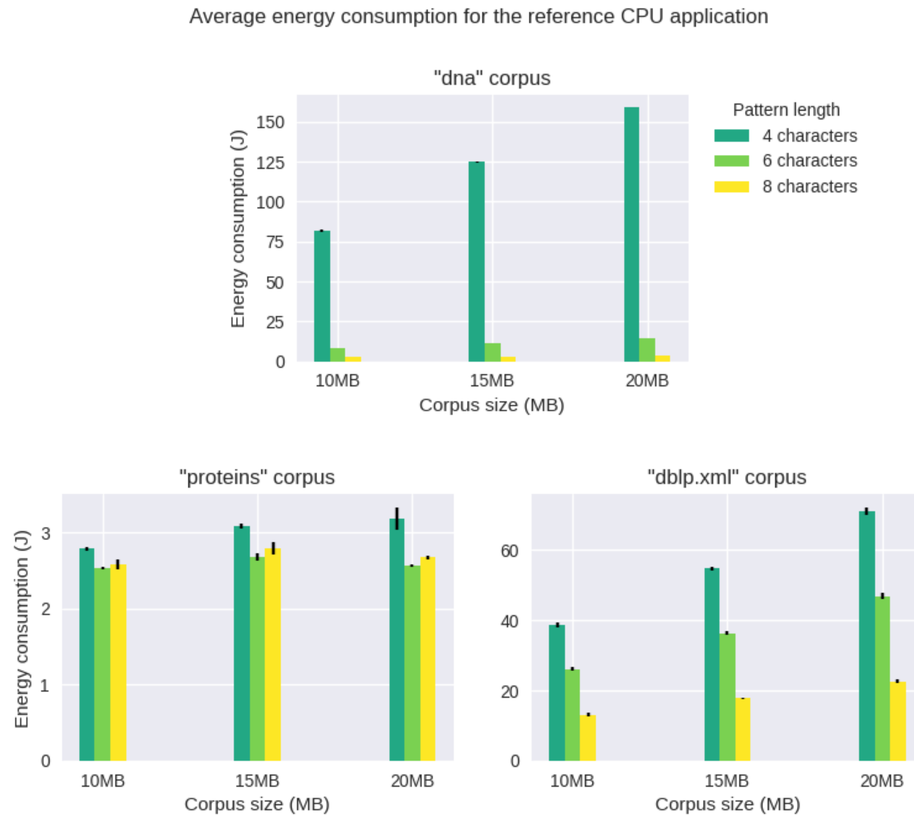

Figure 3.2: Total energy consumption (in joules) when matching 7500 patterns using the reference CPU application. Different corpora, text sizes and pattern lengths are investigated and an average is taken over 5 runs for each configuration.

Because the FM-index is essentially a collection of data structures, we feel it is worth mentioning the memory footprints. Table 3.3 shows the memory footprint for the 20MB versions of the three chosen corpora. It can be seen that there is a big difference in memory footprint between the three corpora. This difference is the result of the alphabet size, which is highest for the *dblp.xml* corpus.

Memory footprint of FM-indexes			
Text size	Corpus		
	dna	*proteins*	*dblp.xml*
10MB	0.3GB	0.5GB	1.3GB
15MB	1.1GB	1.6GB	2.2GB
20MB	3.8GB	5.7GB	7.6GB

Table 3.3: The memory footprints of the FM-index with different corpora and text sizes.

CHAPTER 4

Reference Application for FPGA

In this chapter, we present our reference FPGA application. We also evaluate the performance of the reference FPGA implementation using the same experimental setup used in chapter 3.

4.1 FPGA implementation

We have chosen Xilinx's Vitis platform as it seems to be the most popular platform currently [26]. The Vitis platform allows us to write a host program that runs on the CPU that communicates, using OpenCL bindings, with a kernel that runs on the FPGA. Both the host program and the FPGA kernel are discussed in the following section.

4.1.1 Host program

The host program is implemented using the C++ language. OpenCL is used in the host program to instruct and communicate with the FPGA kernel. It further uses our FM-index C library from chapter 3 to load an FM-index file from disk into memory.

The steps performed by the host program can be seen in the following timeline:

1. Load kernel from disk and program the FPGA
2. Load FM-index and experiment data from disk
3. Enqueue OpenCL commands:
   (a) Migrate experiment data and FM-index data structures to FPGA
   (b) Execute FPGA kernel
   (c) Migrate result data from FPGA to host program
4. Wait until all commands are handled and terminate

Note that enqueuing the OpenCL commands does not block the CPU. Therefore we must wait until OpenCL is done handling the commands, and the data is migrated to the host.

4.1.2 FPGA kernel

The FPGA kernel receives a list of patterns, for which it must find the positions in the original text using the FM-index. These positions are saved in a buffer and sent to the host program when the kernel is done executing.

The FPGA kernel is programmed as an OpenCL kernel and presented in listing 6. As can be seen, the kernel is nearly identical to the string search algorithm presented in chapter 3. Only minimal changes are made that are required for OpenCL to function.

First of all, the `reqd_work_group_size` attribute is used to specify how much of the input the kernel can process. However, we don't use this feature here and we simply set the attribute to all ones.

Next, all input buffers to the kernel have the `__global` attribute. This tells the compiler these buffers are located in the global memory on the FPGA.

Lastly, the resulting positions are saved in the global `out` buffer. The amount of matches is also saved in this buffer, so we can more easily verify the result on the host.

```
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void fmindex(__global char *bwt,
             __global char *alphabet,
             __global unsigned *ranks,
             __global unsigned *sa,
             __global unsigned *ranges,
             __global char *patterns,
             __global unsigned long *out,
             size_t bwt_sz, size_t alphabet_sz, unsigned pattern_count,
             unsigned pattern_sz, unsigned out_sz) {
  for (unsigned i = 0; i < pattern_count; ++i) {
    int p_idx = pattern_sz - 1;
    char c = patterns[i * pattern_sz + p_idx];
    unsigned start = ranges[2 * string_index(alphabet, c)];
    unsigned end = ranges[2 * string_index(alphabet, c) + 1];

    p_idx -= 1;
    while (p_idx >= 0 && end > 1) {
      c = patterns[i * pattern_sz + p_idx];
      unsigned range_start = ranges[2 * string_index(alphabet, c)];
      int alphabet_idx = string_index(alphabet, c);
      start = range_start + ranks[alphabet_sz * (start - 1) + alphabet_idx];
      end = range_start + ranks[alphabet_sz * (end - 1) + alphabet_idx];
      p_idx -= 1;
    }

    unsigned long match_count = end - start;
    out[i * out_sz] = match_count; // Save match count
    for (unsigned j = 0; j < match_count; ++j)
      out[i * out_sz + j + 1] = sa[start + j]; // Save match position
  }
}
```

Listing 6: The OpenCL Kernel for the reference FPGA application.

4.2 Empirical evaluation

The same experimental setup is used to evaluate our reference FPGA application as for the reference CPU application in section 3.2.

We use timing and power measurements generated by Vitis [1]. Vitis reports the power consumption in watts at discrete intervals. Therefore we calculate the energy consumption by integration using Simpson's rule.

We expect the throughput of the reference FPGA application to be much lower than the CPU version. However, we expect that they are similar if we look at the throughput independent of clock cycles, because the CPU runs much faster than the FPGA. In terms of energy consumption, we expect the FPGA to use less energy than the CPU.

[1] Thanks Tristan Laan for the data extraction script

4.3 Results

In figure 4.1 the throughput (in patterns matched per second) is shown for the reference FPGA application. Memory transfer times are not incorporated in this figure. In general, we see that each figure resembles the CPU throughput as shown in section 3.3. The throughput of the *dna* and *proteins* corpora is best, with 30 and 25 thousand top throughput respectively. The *dblp.xml* corpus only reaches a top throughput of barely 3000 patterns matched per second.

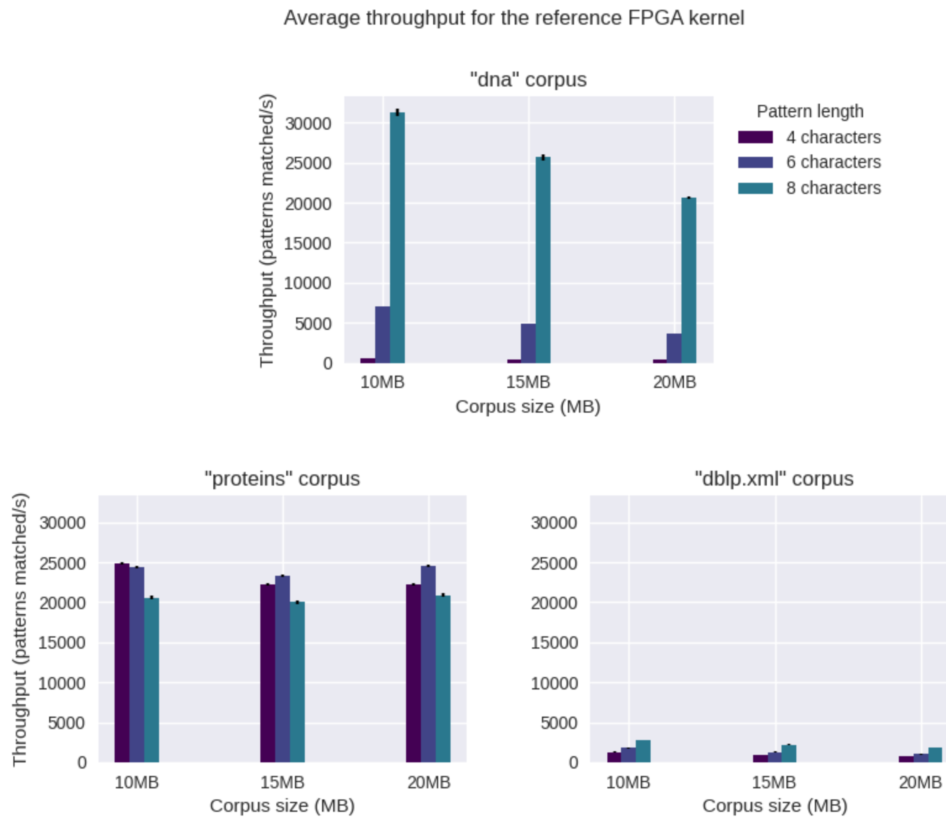

Figure 4.1: Average throughput (in patterns matched per second) for the reference FPGA application. Different corpora, text sizes and pattern lengths are investigated and an average is taken over 5 runs for each configuration.

We believe it is fair not to count the memory transfer times for the FPGA application like we did in figure 4.1. In a real-world scenario, the data would be loaded onto the FPGA only once, and after that the real processing starts. But we still want to show how long memory transfers take, which is shown in table 4.1. As can be seen, these times are significant and range between 0.2 to 2.4 seconds.

Pattern length	dna			proteins			dblp.xml		
	10MB	15MB	20MB	10MB	15MB	20MB	10MB	15MB	20MB
4	1.05 s	1.54 s	1.46 s	0.26 s	0.36 s	0.44 s	1.29 s	1.86 s	2.41 s
6	0.32 s	0.49 s	0.70 s	0.24 s	0.34 s	0.42 s	1.24 s	1.84 s	2.39 s
8	0.19 s	0.30 s	0.47 s	0.23 s	0.33 s	0.42 s	0.94 s	1.35 s	1.76 s

Average memory transfer times for the reference FPGA application

Table 4.1: Average memory transfer times for the reference FPGA application (in seconds). Different corpora, text sizes and pattern lengths are investigated and the average is taken over 5 runs for each configuration.

Energy measurements (in joules) of the reference FPGA application are shown in figure 4.2. Note that this graph does not include the energy usage needed for memory transfers. The figure closely resembles the energy measurements of the reference CPU application as shown in figure 3.2. We see again that the highest energy consumption, almost 800 J, belongs to the *dna* corpus for four-character long patterns. As we have seen earlier, longer patterns cost less time to process, and also result in lower energy consumption here. The *proteins* again has a comparable energy consumption for all corpus sizes and pattern lengths of around 9 to 12 J.

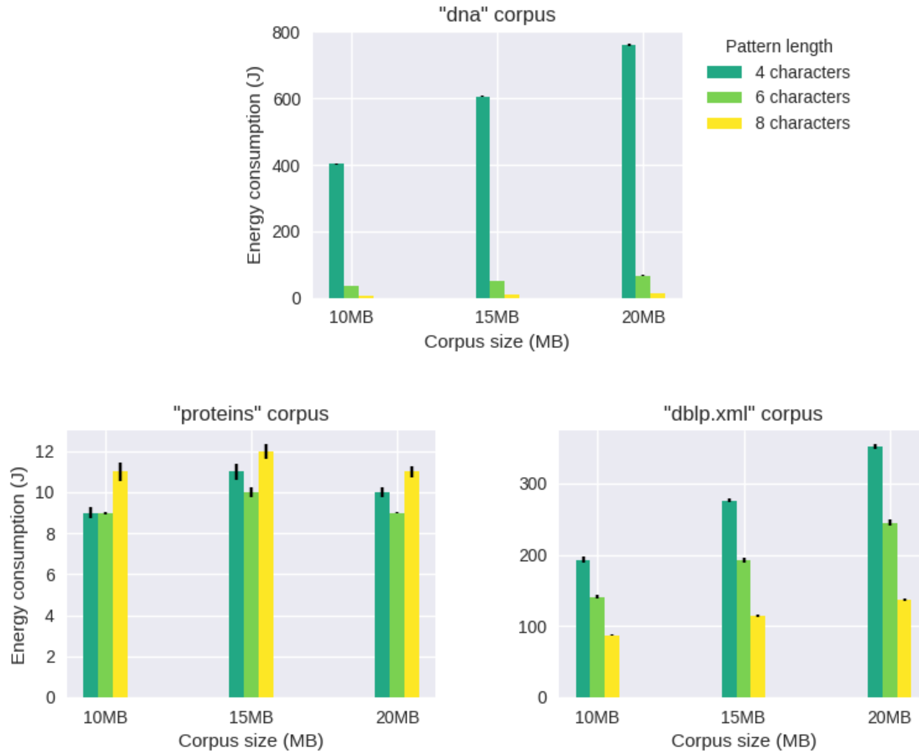

Figure 4.2: Total energy consumption (in joules) when matching 7500 patterns using the reference FPGA application. Different corpora, text sizes and pattern lengths are investigated and an average is taken over 5 runs for each configuration.

The following list shows the resource utilization of the FPGA of the reference kernel:

1. **Lookup tables (LUTs)**: 4313 (0.25%)

2. **Registers**: 5477 (0.19%)
3. **BRAM**: 2 (0.07%)
4. **DSP**: 14 (0.11%)

Table 4.2 shows a comparison of energy consumption between the reference FPGA and CPU applications. It can be seen that reference FPGA application uses more energy than the CPU version in all cases. This varies between 3.0 and 6.7 times the CPU energy consumption. The *dblp.xml* corpus induces the most increases in energy usage, while the *proteins* corpus less. Overall, we conclude that the reference FPGA application performs a lot worse than CPU in terms of energy consumption.

Comparison of energy consumption between the reference FPGA and CPU applications

Pattern length	dna			proteins			dblp.xml		
	10MB	15MB	20MB	10MB	15MB	20MB	10MB	15MB	20MB
4	4.9×	4.9×	4.8×	3.6×	3.6×	3.4×	5.0×	5.1×	5.0×
6	4.4×	4.4×	4.6×	3.9×	3.9×	3.9×	5.4×	5.3×	5.2×
8	3.1×	3.1×	3.7×	4.6×	4.5×	4.4×	6.6×	6.4×	6.1×

Table 4.2: An energy consumption comparison of the reference FPGA application and the CPU version, expressed as $\text{energy}_{\text{FPGA}}$ / $\text{energy}_{\text{CPU}}$. Different corpora, text sizes and pattern lengths are investigated and an average is taken over 5 runs for each configuration.

Figure 4.3 shows the speedup of the FPGA reference application compared to the reference CPU application. We see that in all cases that the FPGA performs 13 to 23 times worse than the CPU version.

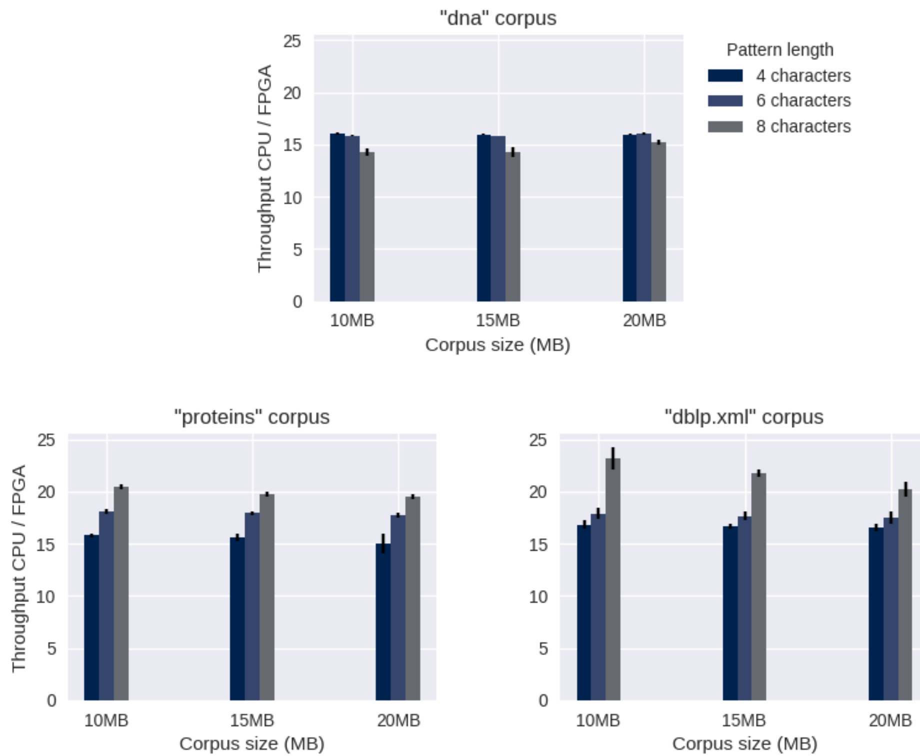

Figure 4.3: Throughput comparison between the reference FPGA application and the CPU version, expressed as $\text{throughput}_{\text{CPU}}$ / $\text{throughput}_{\text{FPGA}}$. Different corpora, text sizes and pattern lengths are investigated and an average is taken over 5 runs for each configuration.

To explain why the reference FPGA application performs much worse than the CPU version, we turn to the difference in clock speeds. The CPU has a base clock speed of 3.40 GHz, while our FPGA kernel runs at 300 MHz. Therefore we also compare the amount of clock cycles needed to complete each experiment on the CPU and FPGA. Figure 4.4 shows a graph of this, expressed in the amount of CPU clock cycles divided by that of the FPGA. We see that the CPU is able to perform the experiments in around 0.5 to 0.8 times the amount of clock cycles of the FPGA. This means that the FPGA still performs less work per clock cycle, but it looks less pessimistic than looking solely at the throughput.

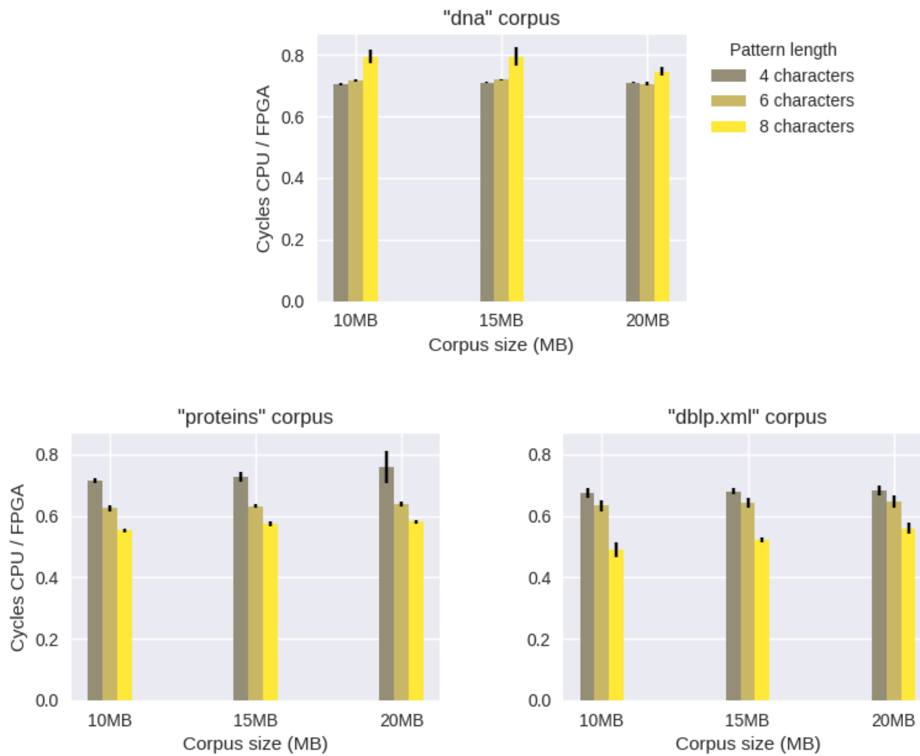

Figure 4.4: Comparison of the amount of clock cycles between the reference FPGA application and the CPU version, expressed as clock cycles$_{\text{CPU}}$ / clock cycles$_{\text{FPGA}}$. Different corpora, text sizes and pattern lengths are investigated and an average is taken over 5 runs for each configuration.

Another reason why the reference FPGA application could perform worse than the reference CPU version, is the nature of the case study. String-searching using the FM-index relies heavily on random memory accesses in big data structures, making it a memory-bound algorithm. It makes sense that FPGAs perform better for more compute-intensive algorithms, because operations like these can be synthesized on the FPGA.

CHAPTER 5

Optimizations

In this chapter, we present our optimizations to our reference FPGA application as presented in chapter 4. We also evaluate the performance of each optimization, as well as the combination of each optimization.

5.1 Optimizations

In this section we present two optimizations for the reference FPGA application that are found in literature. Each optimization is applied to the reference FPGA kernel from chapter 4. Finally, a final kernel with both optimizations applied is presented.

5.1.1 Memory optimizations

An important optimization for the FPGA is the efficient use of memory accesses. We have used two methods to improve memory accesses.

The first method is making use of faster memory types available on the FPGA. As explained in section 2.2, OpenCL has different memory types, that vary in scope. These different memory types also have different characteristics, as explained in the following physical mapping of the OpenCL memory types [25]:

- *Global* and *constant memory*: implemented as physical memory chips attached to the FPGA.
- *Local memory*: implemented as Block RAM (BRAM) elements inside the FPGA fabric.
- *Private memory*: implemented as registers inside the FPGA fabric.

The physical RAM chips are the slowest type of memory available on the FPGA, but can hold the most data. BRAM is faster, but it has less storage capacity. Finally, registers are the fastest type of memory available, but is in short supply.

In order to minimize memory latency, we should access as much memory from the fastest memory types as possible. One way of doing this is caching memory into, for example, private memory. This is only possible for low amounts of data however.

We have identified two data structures in global memory in the reference FPGA application which can be cached this way. These are the `alphabet` and `ranges` buffers. The `alphabet` buffer simply holds the alphabet for a text, which in our case has a maximum size of 97 bytes for the *dblp.xml* corpus. The `ranges` buffer stores the first column of the Burrows-Wheeler transform, in the form ranges of distinct characters. This buffer is a maximum of 194 bytes long for the *dblp.xml* corpus. The other buffers in global memory are hundreds of megabytes large and are no candidates.

Listing 7 shows how this optimization is done for the `alphabet` buffer. The `alphabet` buffer resides in global memory and will have poor memory access performance. To improve this, we cache this data into the `_alphabet` buffer, which resides in private memory.

```
__private char _alphabet[MAX_ALPHABET_SZ];
for (unsigned j = 0; j < alphabet_sz; ++j)
  _alphabet[j] = alphabet[j];
```

Listing 7: Caching global memory in private memory for more efficient memory access.

Another way to optimize memory accesses, is by using burst memory transfers. Burst memory transfers means transferring larger amounts of memory in one go, instead of small transfers sporadically throughout a kernel. Notice how listing 7 already does this. Global memory is transferred in one burst to private memory, instead of sporadically accessing global memory.

We can however improve this further by pipelining the loop; listing 8 shows this concept. The `xcl_pipeline_loop` attribute is a Xilinx extension of the OpenCL specification, and tells the compiler to pipeline the loop. This means that phases of the loop, for example accessing the `alphabet` buffer, can execute simultaneously with other parts of the loop. We write `xcl_pipeline_loop(1)` to indicate the loop as an initiation interval (II) of 1. The II means how many clock cycles to wait until the next loop iteration can execute. In case of loop dependencies, where a loop iteration depends on the previous one, the II can increase. In the case of listing 8, there is no loop dependency and the next loop iteration can start on the very next clock cycle.

```
__private char _alphabet[MAX_ALPHABET_SZ];
__attribute__((xcl_pipeline_loop(1)))
for (unsigned j = 0; j < alphabet_sz; ++j)
  _alphabet[j] = alphabet[j];
```

Listing 8: Pipelining a loop to improve performance.

The entire kernel with the above optimizations can be seen in appendix A.2.

5.1.2 OpenCL's NDRange

As explained in section 2.2, NDRange is an important concept in OpenCL to achieve parallelism. Xilinx's Vitis platform supports OpenCL's NDRange by the use of multiple compute units (CUs).

A CU is a kernel that is synthesized on the FPGA. It is possible to have multiple CUs of the same kernel, or multiple CUs of different kernels and a mix of these. OpenCL can use multiple CUs of the same kernel to schedule work-groups. Each CU then executes a single work-group at a time.

Listing 9 shows the reference FPGA kernel altered for NDRange use.

First of all, the `reqd_work_group_size` OpenCL attribute is changed to create an appropriately large work-group. The index space of 7500 patterns is partitioned into work-groups consisting of 300 work-items. The two "1"s in the work-group size means we do not use the last two NDRange dimensions, which makes it a one-dimensional range.

An additional attribute, `xcl_zero_global_work_offset`, is added to the kernel. Normally, an offset can be specified which is used to calculate the index of work-items in the index space. This attribute promises that this offset is always zero, and can improve performance.

Inside the kernel, the `xcl_pipeline_workitems` attribute encompasses the kernel code. This enables the pipelining of work-items inside the work-group.

Finally, the OpenCL function `get_global_id(0)` is used to get the index of the pattern we are processing inside a work-item. This function call essentially replaces the loop variable of `i` from the reference FPGA kernel in listing 6. The argument of "0" specifies the dimension of the index space to query. Because our kernel is one-dimensional, this is zero (the first dimension).

```
kernel
__attribute__((reqd_work_group_size(300, 1, 1)))
__attribute__((xcl_zero_global_work_offset))
void fmindex(/* Parameters */) {
  __attribute__((xcl_pipeline_workitems)) {
    int i = get_global_id(0);
    /* Work-item code */
  }
}
```

Listing 9: The NDRange version of the FPGA kernel.

The host program must also be changed to execute an NDRange kernel. This change can be seen in listing 10. The `enqueueNDRangeKernel` method is used to enqueue an NDRange kernel. The arguments also specify that the offset is zero, the total amount of work is 7500 patterns, and we partition the total work in work-groups of 300 work-items.

```
command_queue.enqueueTask(kernel);
// Becomes
command_queue.enqueueNDRangeKernel(kernel, 0, 7500, 300);
```

Listing 10: Enqueuing an NDRange kernel in the host program instead of a single kernel.

The entire kernel with the NDRange optimization can be seen in appendix A.1.

5.1.3 Combined optimizations

In addition to two kernels implementing the above optimizations, we have also developed a kernel with both optimizations applied. This kernel can be seen in appendix A.3.

5.2 Empirical evaluation

The same experimental setup is used to evaluate our three optimized kernels as the setup in section 3.2.

For the kernels optimized with NDRange we need to make a choice of how many compute units to use. Therefore we have compiled the NDRange optimized kernel with different numbers of compute units. Unfortunately, the only configuration that successfully compiled was 5 compute units; more compute units results in errors in the compilation process. Chapter 6 goes into more detail on this.

We expect that the NDRange optimization will improve by a significant amount, as it really exploits paralellization. We also think the memory optimizations will improve performance a bit. Finally, we expect the kernel with all optimizations to exceed performance of the reference CPU application both in terms of throughput and energy usage.

5.3 Results

Figure 5.1 shows the throughput (in patterns matched per second) for the optimized kernels alongside the references applications. We also show the same results in logarithmic scale in 5.2 because the difference in throughput is very small in some cases.

In general we see that the memory optimizations improve throughput over the reference FPGA kernel in all cases. However, for the *dblp.xml* corpus and *dna* corpus with pattern lengths 4 and 6, this improvement is only very minor. In the other cases, the memory optimizations improve throughput by 0.5× to almost 4×.

Interestingly, the usage of NDRange alone actually lowers throughput compared to the reference FPGA kernel in most cases. Only for the *proteins* corpus with pattern lengths 6 and 8

does our usage of NDRange improve throughput. However, the lowered throughput for the other cases is again minor. The low throughput seems to be the result of worse memory bandwidth utilization, which we will address in the fully optimized kernel.

Finally, using both optimizations improves throughput most. This is best seen for the *dna* corpus with pattern length 8 and the *proteins* corpus. Across these cases, the kernel with both optimizations induces an improved throughput compared to the reference FPGA kernel of between 3.5× and 9.3×. Again, for the other cases, the improvement is minor.

We believe that combining the memory and NDRange optimizations causes the best throughput, because the memory optimization is replicated across multiple compute units. Each compute unit can therefore take advantage of better memory usage resulting in better performance.

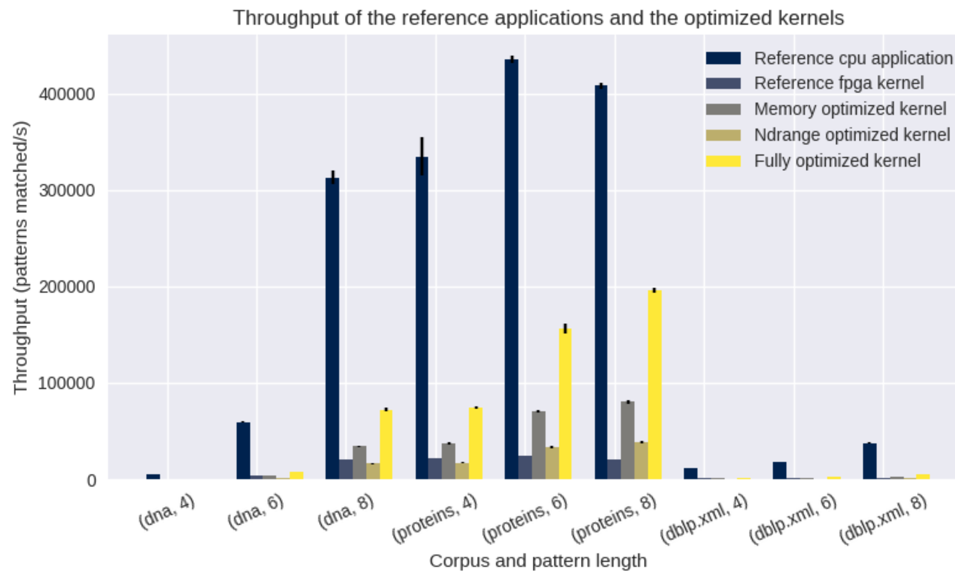

Figure 5.1: Average throughput (in patterns matched per second) for the reference applications and optimized kernels. Different corpora and pattern lengths are investigated, and an average is taken over 5 runs for each configuration.

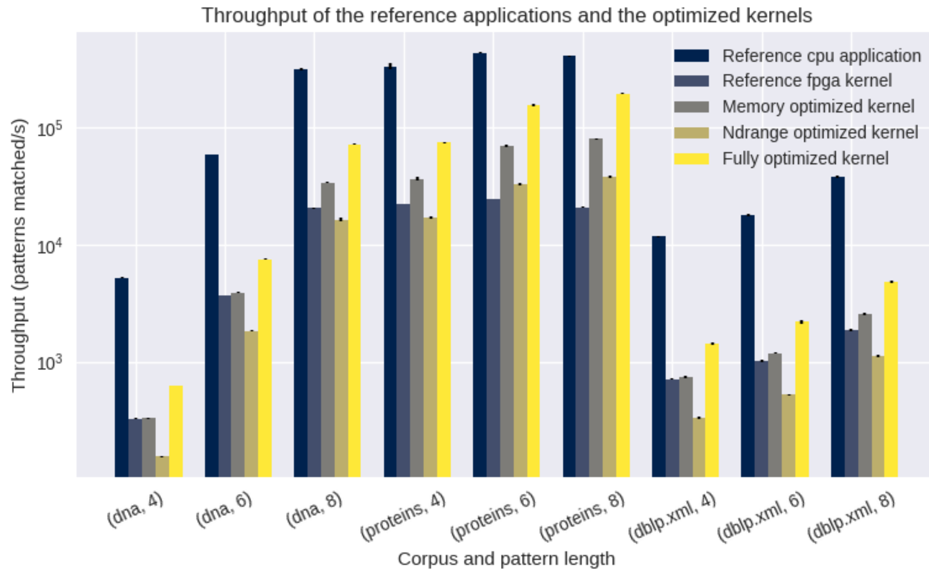

Figure 5.2: Average throughput (in patterns matched per second) for the reference applications and optimized kernels. Different corpora and pattern lengths are investigated, and an average is taken over 5 runs for each configuration. Note the y-axis has logarithmic scale.

While we have seen that the optimizations gives us better performance over the reference FPGA application, we still cannot top the CPU version. Just like in chapter 4, we will therefore look at the amount of clock cycles needed to perform each experiment. The results of this can be seen in figure 5.3. We see that the FPGA kernel with both optimizations is able to perform the experiments in less clock cycles in each case. This is most apparent for the *dna* corpus with pattern length 8 and the *proteins* corpus, where CPU uses between 2.6 and 5.5 times the amount of clock cycles as the FPGA.

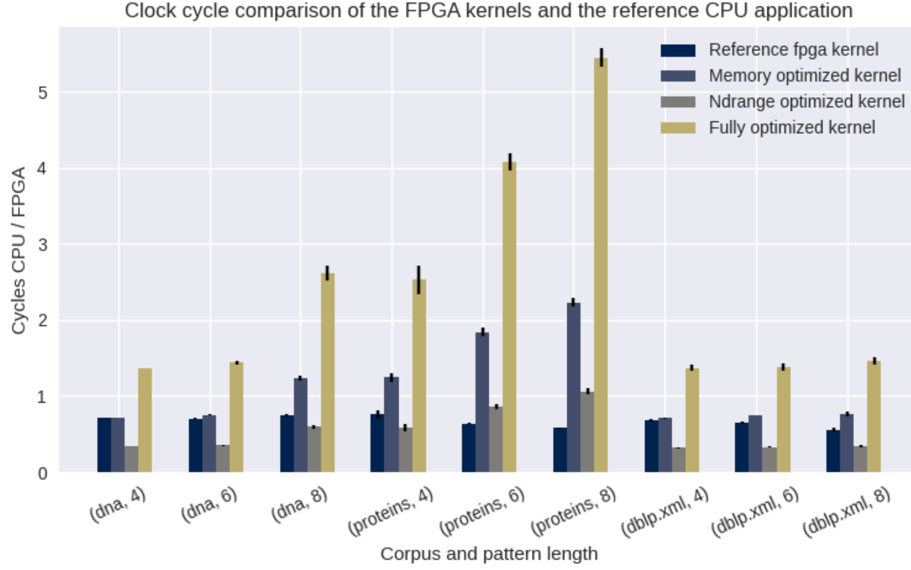

Figure 5.3: Comparison of the amount of clock cycles FPGA kernels and the reference CPU application, expressed as clock $\text{cycles}_{\text{CPU}}$ / clock $\text{cycles}_{\text{FPGA}}$. Different corpora, text sizes and pattern lengths are investigated and an average is taken over 5 runs for each configuration.

Table 5.1 shows an energy consumption comparison between the reference CPU application and the three optimized kernels. It can be seen that the memory-optimized and NDRange-optimized kernels both still consume more energy than the CPU version. Only the fully optimized kernel consumes less energy for the *proteins* corpus with pattern lengths 6 and 8. This is not surprising as we have previously seen that these configurations also lead to a good throughput.

Comparison of energy consumption between optimized FPGA kernels
and reference CPU application

	dna			proteins			dblp.xml		
Pattern length	4	6	8	4	6	8	4	6	8
memory	4.7×	4.3×	2.3×	2.1×	1.4×	1.1×	4.8×	4.5×	4.5×
NDRange	11.1×	10.2×	5.3×	5.0×	3.2×	2.7×	11.6×	11.2×	10.8×
full	2.8×	2.5×	1.1×	1.1×	0.6×	0.5×	2.8×	2.8×	2.7×

Table 5.1: An energy consumption comparison of the reference FPGA application and the CPU version, expressed as $\text{energy}_{\text{FPGA}}$ / $\text{energy}_{\text{CPU}}$. Different corpora, text sizes and pattern lengths are investigated and an average is taken over 5 runs for each configuration.

To understand the relation between energy consumption and time better, figure 5.4 shows two ratios alongside each other. The first is execution time, by dividing the FPGA's execution time by that of the reference CPU version. The second is energy, by dividing the FPGA's energy consumption by that of the reference CPU version. Only the fully optimized FPGA kernel is considered for brevity. We see that in each case, while the execution time for the FPGA is higher than for the CPU, the energy consumption does not increase accordingly. For example for the experiment with *dna* corpus and pattern length 4, execution time on the FPGA is about 8× that of the CPU, while energy consumption is only about 3× that of the CPU.

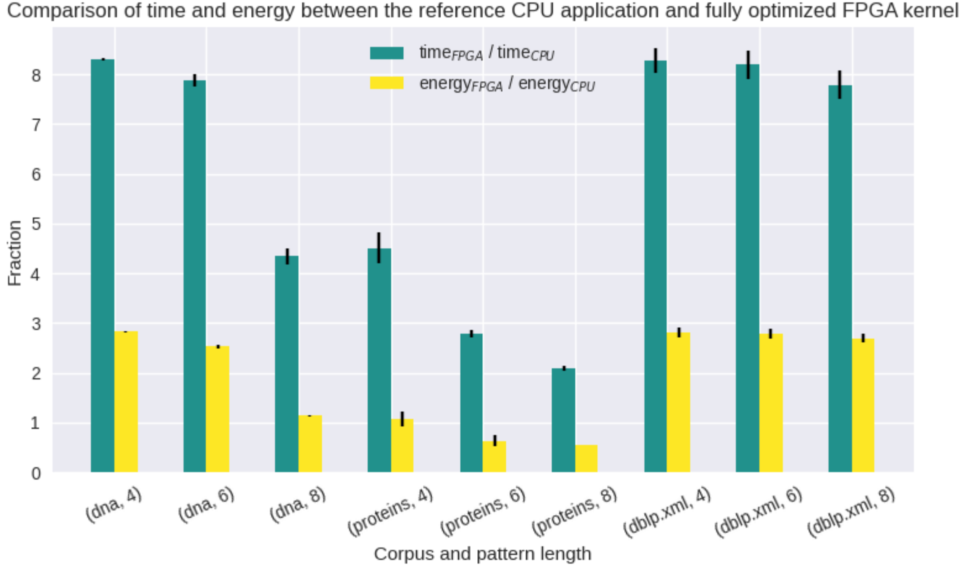

Figure 5.4: Comparing execution time and energy consumption between the reference CPU application and the fully optimized FPGA kernel, expressed as $\text{time}_{FPGA}/\text{time}_C PU$ and $\text{energy}_{FPGA}/\text{energy}_{CPU}$. Different corpora, text sizes and pattern lengths are investigated and an average is taken over 5 runs for each configuration.

Table 5.2 shows the resource utilization on the FPGA for each optimized kernel. As the NDRange and fully optimized kernels both use multiple compute units, these kernels use much more resources, but not nearly all resources available.

FPGA resource utilization

	LUTs		Registers		BRAM		DSP	
	Count	%	Count	%	Count	%	Count	%
memory	4374	0.25	5607	0.19	2	0.07	8	0.07
NDRange	21368	1.25	29061	1.00	10	0.35	100	0.80
full	28982	1.70	36266	1.25	15	0.55	75	0.60

Table 5.2: Resource utilization for the three optimized FPGA kernels.

CHAPTER 6

Challenges

In this chapter, we discuss several challenges we encountered when developing FPGA applications using Xilinx's Vitis toolchain.

6.1 Toolchain

First of all, we had quite a lot of problems installing the Vitis toolchain on our development machine. The installation process itself takes up to 5 hours, a huge portion of which is caused by the installation server's low upload speed of around 200 kB/s. What's more, the installation often hangs at the finishing phase of the installation process. After careful examination of the installation log files, we found that the installation depends on a library which was not documented. It would have been more useful to let the installer crash instead of hanging indefinitely.

Because of these problems, installation actually took us several days to complete. Luckily, after installation, we did not encounter any more problems with the toolchain itself.

6.2 Development process

The development of HLS kernels with the Vitis toolchain can be split into three parts: software emulation, hardware emulation, and hardware execution.

6.2.1 Software emulation

Compilation for software emulation is generally quite fast, with our application taking between one and two minutes to compile. Software emulation can be used to quickly verify whether the algorithm is functional without the constraints of the FPGA. Because of the low compilation time, it can be used for iterative development. However, software emulation does not guarantee that the HLS kernel can actually successfully be compiled for the FPGA.

6.2.2 Hardware emulation

Hardware emulation provides a better indication on whether an HLS kernel can successfully run on the target FPGA. To do this, the target hardware is emulated on the CPU, and the kernel is executed on that emulator.

For our application, compilation for hardware emulation takes between 10 and 15 minutes, so it is best used after software emulation shows the kernel works correctly. Hardware emulation can provide more guarantees that a kernel can run under the constraints of the target FPGA, because the actual hardware is emulated.

A downside of hardware emulation is that only a small amount of data can be used as input for the kernel (i.e., larger inputs result in very long emulation times). On the other hand, a small input set might not reflect the actual input used in hardware.

In our specific case, hardware emulation did not work on the personal development machine at all. Fortunately, it did work on the server where the FPGA was also hosted.

6.2.3 Hardware compilation

Finally, hardware compilation generates the bitstream of HLS kernel for the actual hardware. This process takes the longest: for our application, it took between 2.5 and 5 hours. The upper limit (5 hours) is only reached when increasing the number of compute units we want, because the toolchain has to place and route these on the FPGA, which takes a lot of time.

Also, several times we encountered compilation errors when increasing the number of compute units. These errors did not occur during hardware emulation, making hardware emulation somewhat unreliable as a clear indicator of success in hardware. The compile errors also often occured after several hours of compilation, making iterative development difficult.

Overall, development of OpenCL kernels for the FPGA using the Vitis toolchain was easy, as it mostly adheres to the OpenCL standard. In some cases, vendor-specific functions were needed, but these were well documented. As discussed above, what made development rather difficult, is that it could take hours until feedback is given on a program's correctness.

6.3 Results gathering

Generating timing and energy consumption reports was easy, because the Vitis toolchain handles that directly. One point of criticism here is the format of these reports: they are generated as CSV files, but the format of the content is not documented anywhere. This is because the files are supposed to be interpreted by the *vitis_analyzer* tool. After examining this tool, we were able to infer what the data inside the reports mean.

CHAPTER 7

Conclusion and future work

The use of field-programmable gate arrays (FPGAs) for hardware acceleration is a recent trend, which captured many practitioners' attention because it promises both increased performance and lower energy consumption compared to native applications.

FPGAs are traditionally programmed using low-level hardware-description languages (HDLs), which are very different from modern programming languages, and thus hard to program with by non-experts. To alleviate this challenge, and thus accelerate the adoption of FPGAs for regular applications, many advocate the use of high-level synthesis (HLS) for FPGAs. HLS allows users to program an FPGA using a high-level language, such as C, C++ or OpenCL. However, it can be a challenge to *efficiently* program algorithms with HLS to run on an FPGA.

To investigate how well HLS-based FPGA applications perform, we have proposed a detailed, non-trivial case-study: a substring index called FM-index, which enables quickly finding any substring in a text. For this case-study, we ported the algorithm from CPU to FPGA using HLS, and further investigated possible improvements for increasing performance. Our HLS language of choice is OpenCL (a standard designed for heterogeneous computing), and we evaluated the results in hardware, using a Xilinx U250 FPGA as accelerator.

7.1 Main findings

We structured our research in three phases, each driven by one research question. We formulate the answers to these questions in the following paragraphs.

Subquestion 1: How does a naive HLS-based FPGA application compare against its CPU counterpart? We addressed this question by comparing a CPU and FPGA reference application. The design and implementation of the reference CPU application is presented in chapter 3, while chapter 4 presented the reference FPGA version. We further compared the FPGA and CPU reference versions through a comprehensive empirical analysis. In this setup, we measure string-search throughput and energy consumption, and investigate different corpora, text sizes and search pattern lengths.

Our results show that, compared to the sequential CPU version, our naive FPGA application has a throughput 13× to 23× lower. However, we attribute much of this difference to the difference in clock speed: the CPU runs at 3.4 GHz while the FPGA runs at 300 MHz. If we instead compare clock cycles, we find that the FPGA's performance reaches as high as 80% of the performance of the CPU version. Moreover, the energy consumption of the reference FPGA application is considerably worse when compared against the CPU version: the FPGA uses 3.0× to 6.7× more energy than the CPU.

Subquestion 2: What optimizations can improve the performance of an HLS-based FPGA application? In chapter 5, we have explored two different kinds of optimizations for our reference FPGA application: improving memory usage and using multiple compute units with OpenCL's NDRange.

The memory-optimized kernel makes more efficient use of the memory model of OpenCL, and transfers memory in burst where possible. This optimization resulted in at most 4× higher throughput over the reference FPGA kernel.

The NDRange-optimized kernel exploits parallelism by using multiple compute units on the FPGA. This optimization has generally led to decreased throughput. The low performance seems to be the result of worse memory bandwidth utilization, but this is addressed in the fully optimized kernel.

The fully optimized kernel, employing both optimizations, achieves the best throughput with 3.5× to 9.3× that of the reference FPGA kernel. However, the FPGA was not able to beat the CPU version in any experiment in terms of throughput.

We have also looked at the clock cycles needed to perform each experiment with the optimized kernels. We have found that the fully optimized kernel needed less clock cycles than the CPU version for each experiment: in fact, the CPU version uses 2.6× to 5.5× more clock cycles than the fully-optimized FPGA kernel.

Lastly, the energy consumption of the fully optimized FPGA kernel is, in most cases, worse than that of the CPU version. However, there are two instances where the fully optimized FPGA kernel use half the energy that the CPU version does.

Subquestion 3: How difficult is it to develop FPGA applications using HLS? Overall, programming FPGA kernels in OpenCL itself has been easy, as OpenCL is widely-used and many resources exist. However, using the Vitis toolchain has been sifnificantly more difficult. Installing the toolchain took several days because of errors during installation. While software and hardware emulation is useful for quickly checking functionality of a kernel, we have experienced several instances where it ran successfully, while the compilation for hardware failed. The long compile times for hardware also made iterative development difficult.

We conclude that using HLS does result in quick development time, but the overall performance is limited. With the added optimizations, however, the HLS version of our case-study did outperform the CPU version in terms of cycles, but not in terms of execution time (due to the clock speed difference).

7.2 Contributions

This research makes the following contributions:

- We propose the first Xilinx Vitis implementation for the FM-index algorithm, together with its performance evaluation.
- We demonstrate the implementation of two optimizations recommended by literature in our HLS version of he application, and evaluate their impact on performance.
- We provide considerations on the difficulties of going from beginner to programmer for FPGAs using Xilinx's Vitis platform.

7.3 Limitations and threats to validity

A major limitation of our research is that we have only investigated the performance of a single algorithm. The case study of a string-searching algorithm is mostly memory-bound, so investigating a more compute-bound algorithm would help generalize our conclusions.

Another limitation is the fixed number (5) of compute units for the NDRange-optimized kernel. We encountered compile errors when increasing the number of compute units, so only 5 compute units are used.

Furthermore, we have used relatively small text sizes during our experiments. Larger sizes can result in the data structures of the FM-index becoming larger than 4GB, which is a buffer size not supported by the Vitis platform. Real-world applications use the FM-index for searching in DNA,

which can be multiple gigabytes large (if each chromosome is a character). Our implementation does not scale to these ranges.

Lastly, experiments were performed in only one environment. To better generalize our conclusions, the experiments should be executed on multiple CPUs, FPGAs and FPGA platforms.

7.4 Ethical considerations

We believe there are no ethical problems with our methods nor the research field. The selected case study is mainly used in bioinformatics, and used to understand genomes and proteins and ultimately could help medicine research. Therefore, we think this case study is morally responsible. FPGAs simply provide another way to perform calculations, so in essence it does not introduce any new ethical considerations.

7.5 Future work

Although we have explored two different kinds of optimizations for the FPGA kernel, there are many more optimizations possible. For example, the Vitis documentation has extensive guidance on optimizing the use of different memory banks available on the FPGA.

Another angle for future research is to propose improvements for the FM-index. Since the release of the original paper on FM-index, Manzini and Venturini have developed improvements on the search algorithm [16]. Another improvement by Ferragina et al. reduces the space occupancy of the index [6]. It would be interesting to see whether these are portable to our FPGA implementation.

Finally, to assess the performance of HLS against more traditional FPGA programming languages, a VHDL or Verilog version of the application should be developed. In turn, this will demonstrate whether HLS can be a competitive alternative when compared with a low-level, dedicated FPGA solution.

Bibliography

[1] Michael Burrows and David Wheeler. *A Block-Sorting Lossless Data Compression Algorithm.* Tech. rep. DIGITAL SRC RESEARCH REPORT, 1994.

[2] The kernel development community. *Power Capping Framework — The Linux Kernel documentation.* URL: https://www.kernel.org/doc/html/latest/power/powercap/powercap.html (visited on 05/21/2021).

[3] E. Fernandez, W. Najjar, and S. Lonardi. "String Matching in Hardware Using the FM-Index". In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines.* May 2011, pp. 218–225. DOI: 10.1109/FCCM.2011.55.

[4] P. Ferragina and G. Manzini. "Opportunistic data structures with applications". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science.* ISSN: 0272-5428. Nov. 2000, pp. 390–398. DOI: 10.1109/SFCS.2000.892127.

[5] Paolo Ferragina and Gonzalo Navarro. *Pizza&Chili Corpus – Compressed Indexes and their Testbeds.* URL: http://pizzachili.dcc.uchile.cl/index.html (visited on 04/21/2021).

[6] Paolo Ferragina et al. "An Alphabet-Friendly FM-Index". en. In: *String Processing and Information Retrieval.* Ed. by Alberto Apostolico and Massimo Melucci. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 150–160. ISBN: 978-3-540-30213-1. DOI: 10.1007/978-3-540-30213-1_23.

[7] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer Manual: Vol 3.* en. 2016. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html (visited on 05/21/2021).

[8] Intel Corporation. *Intel® FPGA SDK for OpenCL™ - Intel FPGA SDK for OpenCL.* URL: https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html (visited on 05/17/2021).

[9] Intel Corporation. *Intel® Xeon® Gold 6128 Processor (19.25M Cache, 3.40 GHz) Product Specifications.* en. URL: https://ark.intel.com/content/www/us/en/ark/products/120482/intel-xeon-gold-6128-processor-19-25m-cache-3-40-ghz.html (visited on 05/13/2021).

[10] Ian Kuon, Russell Tessier, and Jonathan Rose. *FPGA Architecture: Survey and Challenges.* ISBN: 978-1-60198-127-1.

[11] Ben Langmead. "Burrows-Wheeler Transform and FM Index". en. In: *Technical Report* (Jan. 2014), p. 41. URL: https://www.cs.jhu.edu/~langmea/resources/lecture_notes/bwt_and_fm_index.pdf (visited on 05/26/2021).

[12] Ben Langmead. "Introduction to the Burrows-Wheeler Transform and FM Index". en. In: (Nov. 2013), p. 12. URL: https://www.cs.jhu.edu/~langmea/resources/bwt_fm.pdf (visited on 05/26/2021).

[13] Ben Langmead et al. “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome”. en. In: *Genome Biology* 10.3 (Mar. 2009). Number: 3 Publisher: BioMed Central, pp. 1–10. ISSN: 1474-760X. DOI: 10.1186/gb-2009-10-3-r25. URL: https://genomebiology.biomedcentral.com/articles/10.1186/gb-2009-10-3-r25 (visited on 04/18/2021).

[14] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. “OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices”. In: *2010 International Conference on Field Programmable Logic and Applications*. ISSN: 1946-1488. Aug. 2010, pp. 458–463. DOI: 10.1109/FPL.2010.93.

[15] Veli Mäkinen and Gonzalo Navarro. “Compressed full-text indexes”. In: *ACM Computing Surveys* 39.1 (Apr. 2007), 2–es. ISSN: 0360-0300. DOI: 10.1145/1216370.1216372. URL: https://doi.org/10.1145/1216370.1216372 (visited on 04/19/2021).

[16] Paolo Manzini and Rossano Venturini. *FM-Index Version 2*. Tech. rep. Sept. 2005. URL: https://pages.di.unipi.it/ferragina/Libraries/fmindexV2/index.html (visited on 05/30/2021).

[17] Aaftab Munshi. “The OpenCL specification”. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. Aug. 2009, pp. 1–314. DOI: 10.1109/HOTCHIPS.2009.7478342.

[18] R. Nane et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (Oct. 2016). Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 1591–1604. ISSN: 1937-4151. DOI: 10.1109/TCAD.2015.2513673.

[19] Muhsen Owaida et al. “Synthesis of Platform Architectures from OpenCL Programs”. In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. May 2011, pp. 186–193. DOI: 10.1109/FCCM.2011.19.

[20] N. Paulino, J. C. Ferreira, and J. M. P. Cardoso. “Optimizing OpenCL Code for Performance on FPGA: k-Means Case Study With Integer Data Sets”. In: *IEEE Access* 8 (2020). Conference Name: IEEE Access, pp. 152286–152304. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3017552.

[21] Jared T. Simpson and Richard Durbin. “Efficient construction of an assembly string graph using the FM-index”. In: *Bioinformatics* 26.12 (June 2010), pp. i367–i373. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btq217. URL: https://doi.org/10.1093/bioinformatics/btq217 (visited on 06/04/2021).

[22] The Khronos Group Inc. *The Khronos Group*. en. Section: General. May 2021. URL: https://www.khronos.org (visited on 05/25/2021).

[23] M. M. Imdad Ullah, Akram Ben Ahmed, and Hideharu Amano. “Implementation of FM-Index Based Pattern Search on a Multi-FPGA System”. en. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Fernando Rincón et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 376–391. ISBN: 978-3-030-44534-8. DOI: 10.1007/978-3-030-44534-8_28.

[24] Xilinx Inc. “Introduction to FPGA Design with Vivado High-Level Synthesis (UG998)”. en. In: (Jan. 2019), p. 92. URL: https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf (visited on 05/24/2021).

[25] Xilinx Inc. *Private Memory*. en-us. concept. URL: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/dli1504034322329.html (visited on 05/26/2021).

[26] Xilinx Inc. *Vitis Platform*. en. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html (visited on 05/14/2021).

[27] Hamid Reza Zohouri et al. “Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’16. Salt Lake City, Utah: IEEE Press, Nov. 2016, pp. 1–12. ISBN: 978-1-4673-8815-3. (Visited on 04/07/2021).

APPENDIX A

Optimized kernels

A.1 NDRange optimized kernel

```
#define LOCAL_SIZE 300

kernel
__attribute__((reqd_work_group_size(LOCAL_SIZE, 1, 1)))
__attribute__((xcl_zero_global_work_offset))
void fmindex(__global char *bwt,
             __global char *alphabet,
             __global unsigned *ranks,
             __global unsigned *sa,
             __global unsigned *ranges,
             __global char *patterns,
             __global unsigned long *out,
             size_t bwt_sz, size_t alphabet_sz, unsigned pattern_count,
             unsigned pattern_sz, unsigned out_sz) {
  __attribute__((xcl_pipeline_workitems)) {
    int i = get_global_id(0);
    int p_idx = pattern_sz - 1;
    char c = patterns[i * pattern_sz + p_idx];
    int alphabet_idx = string_index(alphabet, c);
    unsigned start = ranges[2 * alphabet_idx];
    unsigned end = ranges[2 * alphabet_idx + 1];

    p_idx -= 1;
    while (p_idx >= 0 && end > 1) {
      c = patterns[i * pattern_sz + p_idx];
      alphabet_idx = string_index(alphabet, c);
      unsigned range_start = ranges[2 * alphabet_idx];
      start = range_start + ranks[alphabet_sz * (start - 1) + alphabet_idx];
      end = range_start + ranks[alphabet_sz * (end - 1) + alphabet_idx];
      p_idx -= 1;
    }

    unsigned long match_count = end - start;
    out[i * out_sz] = match_count;
    for (unsigned j = 0; j < match_count; ++j)
      out[i * out_sz + j + 1] = sa[start + j];
  }
}
```

A.2 Memory-access optimized kernel

```
#define MAX_PATTERN_SZ 8
#define MAX_ALPHABET_SZ 97
#define MAX_RANGES_SZ (2 * MAX_ALPHABET_SZ)

kernel
__attribute__((reqd_work_group_size(1, 1, 1)))
void fmindex(__global char *bwt,
             __global char *alphabet,
             __global unsigned *ranks,
             __global unsigned *sa,
             __global unsigned *ranges,
             __global char *patterns,
             __global unsigned long *out,
             size_t bwt_sz, size_t alphabet_sz, unsigned pattern_count,
             unsigned pattern_sz, unsigned out_sz) {
  __private char _alphabet[MAX_ALPHABET_SZ];
  __attribute__((xcl_pipeline_loop(1)))
  for (unsigned j = 0; j < alphabet_sz; ++j)
    _alphabet[j] = alphabet[j];

  __private unsigned _ranges[MAX_RANGES_SZ];
  __attribute__((xcl_pipeline_loop(1)))
  for (unsigned j = 0; j < 2 * alphabet_sz; ++j)
    _ranges[j] = ranges[j];

  for (unsigned i = 0; i < pattern_count; ++i) {
    __private char pattern[MAX_PATTERN_SZ];
    __attribute__((xcl_pipeline_loop(1)))
    for (unsigned j = 0; j < pattern_sz; ++j)
      pattern[j] = patterns[i * pattern_sz + j];

    int p_idx = pattern_sz - 1;
    char c = pattern[p_idx];
    int alphabet_idx = string_index(_alphabet, c);
    unsigned start = _ranges[2 * alphabet_idx];
    unsigned end = _ranges[2 * alphabet_idx + 1];

    p_idx -= 1;
    while (p_idx >= 0 && end > 1) {
      c = pattern[p_idx];
      alphabet_idx = string_index(_alphabet, c);
      unsigned range_start = _ranges[2 * alphabet_idx];
      start = range_start + ranks[alphabet_sz * (start - 1) + alphabet_idx];
      end = range_start + ranks[alphabet_sz * (end - 1) + alphabet_idx];
      p_idx -= 1;
    }

    unsigned long match_count = end - start;
    out[i * out_sz] = match_count;

    __attribute__((xcl_pipeline_loop(1)))
    for (unsigned j = 0; j < match_count; ++j)
      out[i * out_sz + j + 1] = sa[start + j];
  }
}
```

A.3 Final optimized kernel

```
#define LOCAL_SIZE 300
#define MAX_PATTERN_SZ 8
#define MAX_ALPHABET_SZ 97
#define MAX_RANGES_SZ (2 * MAX_ALPHABET_SZ)
#define PATTERNS_SZ (MAX_PATTERN_SZ * LOCAL_SIZE)

kernel
__attribute__((reqd_work_group_size(LOCAL_SIZE, 1, 1)))
__attribute__((xcl_zero_global_work_offset))
void fmindex(__global char *bwt,
             __global char *alphabet,
             __global unsigned *ranks,
             __global unsigned *sa,
             __global unsigned *ranges,
             __global char *patterns,
             __global unsigned long *out,
             size_t bwt_sz, size_t alphabet_sz, unsigned pattern_count,
             unsigned pattern_sz, unsigned out_sz) {
  int group_id = get_group_id(0);

  __local char _patterns[PATTERNS_SZ];
  __attribute__((xcl_pipeline_loop(1)))
  for (unsigned i = 0; i < PATTERNS_SZ; ++i)
    _patterns[i] = patterns[group_id * PATTERNS_SZ + i];

  __attribute__((xcl_pipeline_workitems)) {
    int work_id = get_global_id(0);
    int local_id = get_local_id(0);

    __private char _alphabet[MAX_ALPHABET_SZ];
    __attribute__((xcl_pipeline_loop(1)))
    for (unsigned i = 0; i < alphabet_sz; ++i)
      _alphabet[i] = alphabet[i];

    __private unsigned _ranges[MAX_RANGES_SZ];
    __attribute__((xcl_pipeline_loop(1)))
    for (unsigned i = 0; i < 2 * alphabet_sz; ++i)
      _ranges[i] = ranges[i];

    int p_idx = pattern_sz - 1;
    char c = _patterns[local_id * pattern_sz + p_idx];
    int alphabet_idx = string_index(_alphabet, c);
    unsigned start = _ranges[2 * alphabet_idx];
    unsigned end = _ranges[2 * alphabet_idx + 1];

    p_idx -= 1;
    while (p_idx >= 0 && end > 1) {
      c = _patterns[local_id * pattern_sz + p_idx];
      alphabet_idx = string_index(_alphabet, c);
      unsigned range_start = _ranges[2 * alphabet_idx];
      start = range_start + ranks[alphabet_sz * (start - 1) + alphabet_idx];
      end = range_start + ranks[alphabet_sz * (end - 1) + alphabet_idx];
      p_idx -= 1;
    }

    unsigned long match_count = end - start;
    out[work_id * out_sz] = match_count;

    __attribute__((xcl_pipeline_loop(1)))
    for (unsigned i = 0; i < match_count; ++i)
      out[work_id * out_sz + i + 1] = sa[start + i];
  }
}
```