Maxeler Open Dataflow Design Competition

Emanuele Del Sozzo, Marcello Pogliani
Politecnico di Milano
{emanuele.delsozzo, marcello.pogliani}@polimi.it

1 Introduction

The analysis of retinal blood vessels has its purpose in preventive screening tests both for ocular diseases, such as diabetic retinopathy, degenerative maculopathy, and for other pathologies, such as the investigation of thrombi. In order to detect blood vessels within a retinal image, it is necessary to process the image at different levels (Figure 1). For instance, an initial preprocessing step is fundamental to prepare the image and remove noise. This procedure may require to separate the image channels, as well as resizing and cropping it. However, the main and most computational intensive phase of blood vessel detection is the filtering one. As a consequence, this phase may be the bottleneck of the overall analysis. For this reason, the main goal of this project is the hardware-acceleration a medical application for blood vessel detection by means of Maxeler's dataflow engines. In particular, we accelerated a filter called `NoBorderFilter`, while the rest of the computation is performed on the host CPU.

To develop and benchmark the design presented in this project, we used a Maxeler Maxworkstation with the following characteristics: an Intel i7 quad-core CPU with simultaneous multithreading (Intel HyperThreading) and 16 GB of RAM; a Vectis dataflow engine with 24 GB of RAM for the LMem, connected to the host system by means of a PCIExpress 2.0 x8 link. The dataflow engine is based upon a Xilinx Virtex 6 FPGA. MaxCompiler version is 2015.2.

We compared the hardware implementation against a single-thread serial CPU implementation, as well as a multi-threaded CPU implementation. In order to parallelize and process the input image, we relied on state-of-the-art libraries like OpenMP and OpenCV.

2 Implementation

This section presents the implementations we developed. It starts by describing the first, serial implementation, then it moves to the parallel ones.

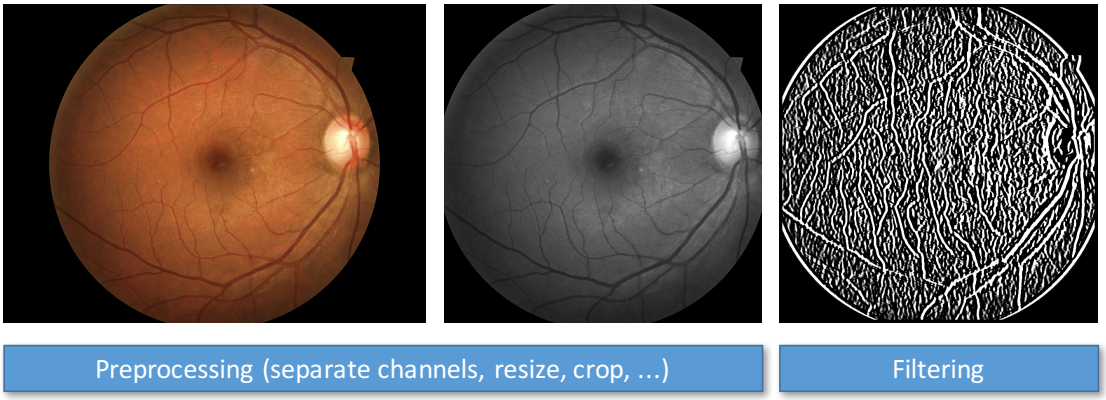

Figure 1: Retinal image processing.

2.1 Serial

`NoBorderFilter` is a fairly simple and standard image processing filter; it works by iterating over all the pixels of the (single channel) input image, taking a window of 16×16 pixels centered around this point, and computing a function of the values of all the pixels in this window. In particular, the kernel computes the sum of each pixel multiplied by a coefficient; the coefficients for each pixel are different (matrix kernel).

Despite the apparent simplicity of the kernel, it was not possible to implement it naively using the Maxeler abstraction of window in a stream: as the image is bidimensional, we would have needed to consider a window of 16 rows of an image, which may not fit the FMem (BRAM), especially when the image width increases. Therefore, this approach would not be scalable.

For this reason, we decided to exploit the large (24 GB) amount of on-board (even though off-chip) DDR memory (LMem, large memory) to hold the intermediate steps of the computation. Thus, we implemented the algorithm having the DFE iterate over the input matrix multiple times, computing partial results, and accumulating them to produce the final matrix. Given a window of 16×16 pixels, we iterate over the input matrix 16 times, and compute on each iteration the sum of the (products of the) elements belonging to a different row of a window. To do this, the algorithm works in the following way:

- At the beginning, the input matrix (image) is streamed from the CPU and copied to the large memory, while the coefficient matrix is stored in a ROM memory;
- The DFE kernel is invoked 16 times (one for each row of the window); for each iteration, the kernel receives two input streams and one output stream: the input streams are the original matrix and a feedback matrix; the output stream will contain the partial results. The output stream of the kernel invocation i will be the feedback matrix of the kernel invocation $i + 1$. Furthermore, at each kernel invocation, the input matrix will start from a memory location incremented of the

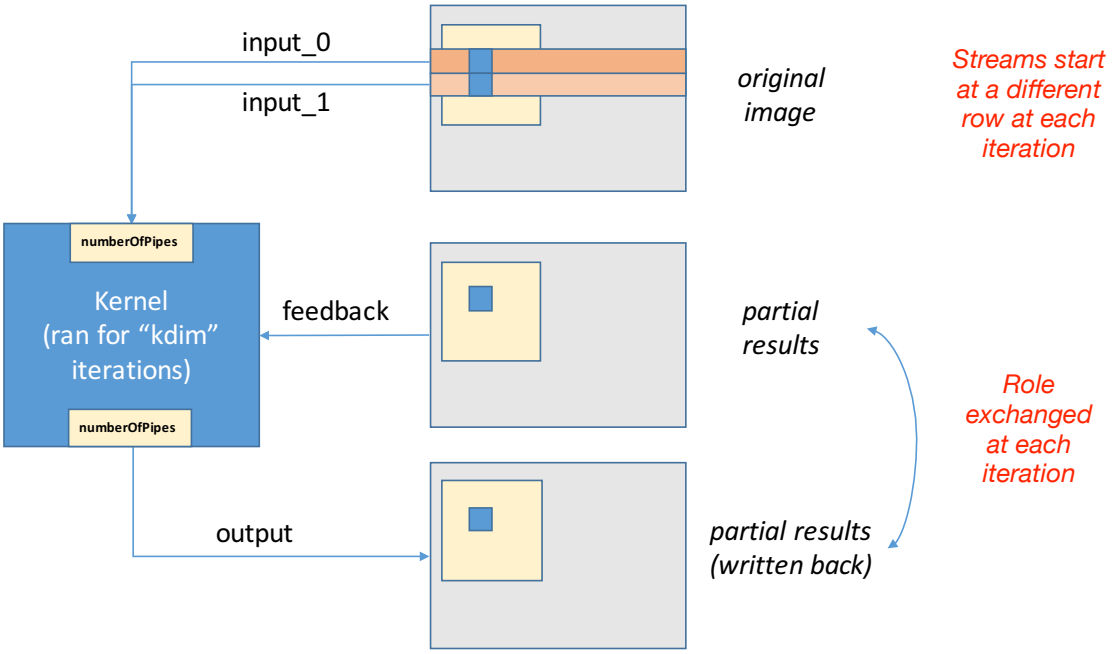

Figure 2: Proposed design.

image width, so that the same computation performed by the kernel will output the correct row of the input window. While it was possible to perform this computation in the DFE kernel, we found it simpler to apply an offset to the memory address to be read for the input stream. The drawback of this approach is that the width of the image needs to be burst-aligned, i.e., must be a multiple of 384 bytes.

- At the end, the output matrix is streamed from the LMem to the CPU. This matrix contains the final result.

Figure 2 shows a diagram of the proposed design.

In our implementation, the orchestration of the data transfer (LMem address computation, invocation of the DFE kernel the correct number of times, ...) is performed by the CPU code. While this approach works reasonably well, it poses some constraints on the input format:

- The data (i.e., the image size) should be aligned with the LMem burst (384 bytes)
- Every row should be aligned with the LMem burst (to start reading the LMem from a different offset at each iteration)

The CPU code we implemented will automatically add the required padding to the image; unfortunately, this can add overhead to the process as it requires that the CPU iterates over the image, and performs copies, to prepare the memory locations to be transferred to the FPGA via PCIe. The second requirement could be lifted by applying

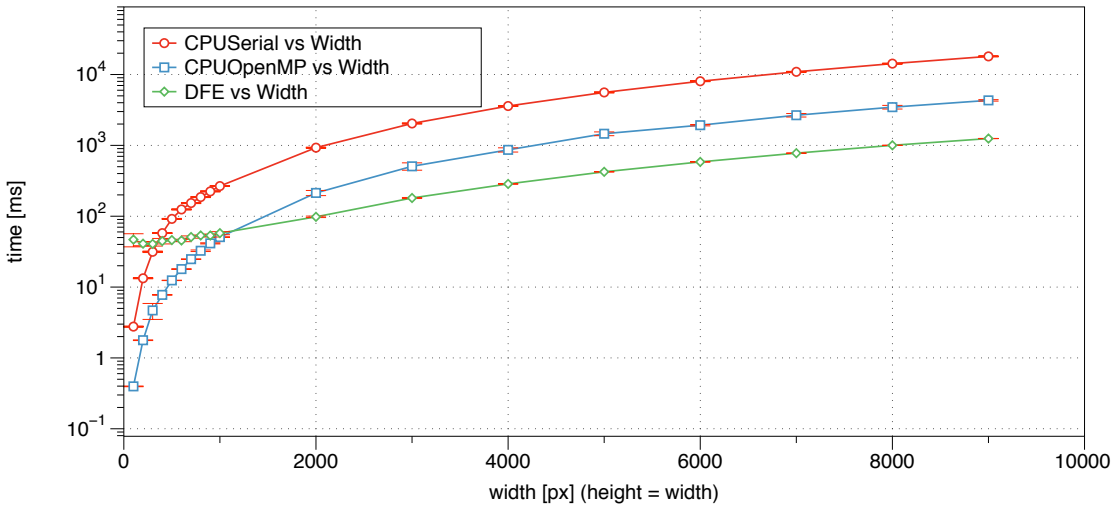

Figure 3: Performance of a parallel implementation with 1 stream and 24 pipes compared to serial and parallel CPU implementations.

the offset computation in the DFE; the first requirement cannot be lifted, as it is not possible to perform data transfer less than a multiple of the memory burst.

A simple implementation of the aforementioned approach leads to the following resource usage report:

```
Logic utilization:      55548 / 297600 (18.67%)
 LUTs:                  35042 / 297600 (11.77%)
 Primary FFs:           44799 / 297600 (15.05%)
 Secondary FFs:          8864 / 297600 (2.98%)
Multipliers (25x18):       48 / 2016   (2.38%)
 DSP blocks:               48 / 2016   (2.38%)
Block memory (BRAM18):    298 / 2128   (14.00%)
```

As it can be seen, the DFE results quite underutilized; therefore, we decided to exploit parallelism to better utilize the available resources and increase performance.

2.2 Parallel

Replicating the main computational pipeline of the aforementioned kernel N times, it is possible to greatly increase performance and better exploit data locality. We implemented the kernel using N parallel pipes, exploiting the `DFEVector<DFEVar>` composite datatype to simplify and make more readable the implementation. In this version, for every tick of every iteration, N pixels are processed at once instead of just 1. Note that for efficiency reasons, N must be a multiple of 4. This approach increases DFE space utilization, to achieve better performance than the CPU implementation, and is scalable, allowing the designer to reduce the performance to save resource utilization or

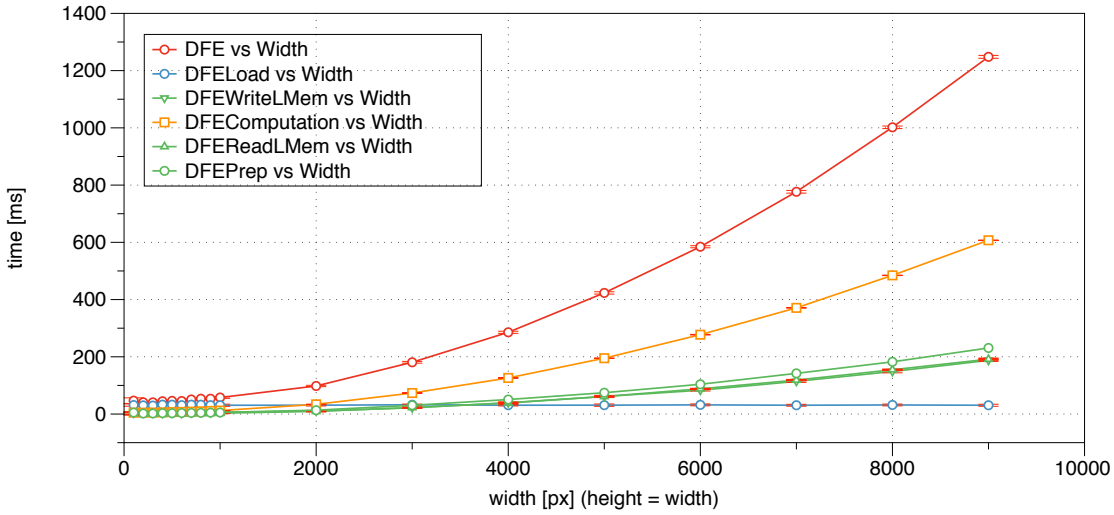

Figure 4: DFE runtime with 1 stream and 24 pipes.

vice-versa. A first implementation of such approach leverages 1 stream of data and 24 pipes (*1s24p*). This leads to the following resource usage:

```
Logic utilization:      98163 / 297600 (32.98%)
 LUTs:                  58664 / 297600 (19.71%)
 Primary FFs:           84963 / 297600 (28.55%)
 Secondary FFs:         15684 / 297600 (5.27%)
Multipliers (25x18):     1152 / 2016   (57.14%)
 DSP blocks:             1152 / 2016   (57.14%)
Block memory (BRAM18):    361 / 2128   (16.96%)
```

We then decided to further improve such implementation by loading multiple rows at once from the LMem. In this way, we were able to use 2 streams starting from different address. At the same time, we needed to use just 12 computational pipes instead of 24 (*2s12p*). This was done to reduce the usage of multipliers, since they are a scarce resource. As a result, the resource usage was the following:

```
Logic utilization:      91551 / 297600 (30.76%)
 LUTs:                  62179 / 297600 (20.89%)
 Primary FFs:           79245 / 297600 (26.63%)
 Secondary FFs:         18259 / 297600 (6.14%)
Multipliers (25x18):     1152 / 2016   (57.14%)
 DSP blocks:             1152 / 2016   (57.14%)
Block memory (BRAM18):    384 / 2128   (18.05%)
```

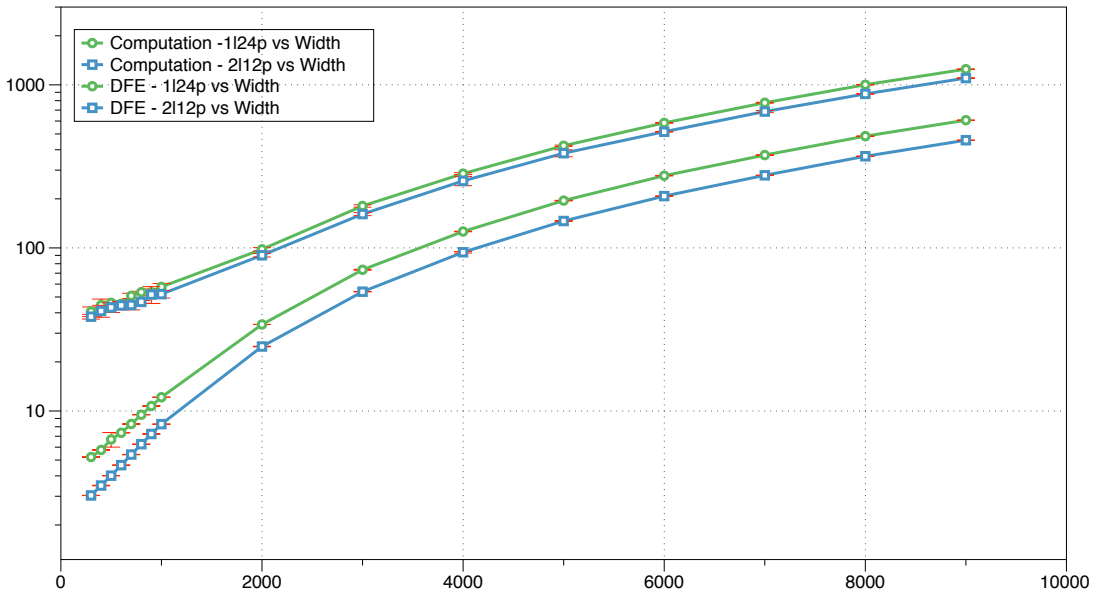

Figure 5: 1 LMem stream w/ 24 pipes vs. 2 LMem streams w/ 12 pipes.

3 Experimental Results

This section reports the performance of the proposed implementations. In all implementations we used 32-bit signed integer as both input and output data types. Nonetheless, the proposed design is parametric with respect to the input/output data types, therefore other data types may be employed.

The first implementation (the serial one) of the `NoBorderFilter` kernel had very poor performance. Indeed, a single-thread, non optimized CPU implementation of the same filter performed 4 times faster. For this reason, we decided to exploit parallelism by means of multiple computational pipes. As shown in Figure 3, *1s24p* implementation outperforms both serial and parallel CPU implementations; indeed, the DFE implementation is about 3.5 times faster than the OpenMP one. It is important to notice than the DFE performance here reported includes also CPU-side preprocessing and data transfer to/from LMem. Following this statement, it is interesting to analyze how the DFE runtime is split. According to Figure 4, we can see that computation is just about 50% of the DFE time. This implies that, if we ignore data transfer and CPU preprocessing, DFE is about 7 times faster than CPU.

Finally, let us consider the parallel implementation with 2 streams from LMem and 12 pipes. Such an implementation overcomes *1s24p* thanks to the parallel reads from the LMem. In particular, if we compare these two parallel implementations in terms of both overall DFE time and computational time, the runtime of *2s12p* is 1.3 times faster than the runtime of *1s24p* (Figure **??**). As a consequence, now the only computation is about 9 times faster than CPU.

4 Conclusions

The analysis of retinal blood vessels is crucial for diagnosis and prevention of ocular diseases, like diabetic retinopathy, degenerative maculopathy, and so on. In order to analyze and detect blood vessels within a retinal image, it is necessary to process and filter the input image. Such procedure may result computational intensive, hence it would definitely benefit from an hardware acceleration. FPGAs have potential for acceleration of this kind of computational kernels. To this end, Maxelers toolchain provides a model to program FPGAs in a dataflow fashion. However, in some cases, it is not possible to implement a computational kernel as a pure dataflow way. Therefore, it is fundamental to evaluate different implementations and trade-offs to obtain the desired performance.

This project proposes a DFE implementation of the `NoBorderFilter` able to outperform both serial and parallel CPU implementations. The proposed design leverages multiple computational pipes, as well as optimized parallel reads from off-chip memory.