Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Mateo Vázquez Maceiras

# Accelerating LBM on a Tightly-Coupled Field Programmable Gate Array

Master's thesis in Embedded Computing Systems
Supervisor: Magnus Jahre
June 2021

NTNU
Kunnskap for en bedre verden

Mateo Vázquez Maceiras

# Accelerating LBM on a Tightly-Coupled Field Programmable Gate Array

Master's thesis in Embedded Computing Systems
Supervisor: Magnus Jahre
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

# Accelerating LBM on a Tightly-Coupled Field Programmable Gate Array

Mateo Vázquez Maceiras

June 9, 2021

# Abstract

With the end of Dennard Scaling and the imminent end of Moore's Law, the search for new ways to improve performance in computing systems is increasing. Nowadays, the main approach is to use hardware accelerations to offload the application. However, while this is a power-efficient approach, their development process is costly and time-consuming. In this Thesis, we have implemented a hardware accelerator for LBM, what we initially expected to be one of the most accelerator-friendly benchmarks of SPEC CPU 2017. We have implemented it in a Field Programmable Gate Array (FPGA) using High-Level Synthesis (HLS), which simplifies the developing process. With our acceleration strategy we have achieved speedups between 1.3× and 1.5× relative the software implementation for realistic data sets. Moreover, we have analyzed HLS and found out that, while it actually simplifies the developing process, this is still not trivial. Developers still need to know and understand the target architecture and guide tool if we want to achieve near-optimal results.

# Sammendrag

Det er ikke lenger mulig å anvende Dennard's prinsipper til å skalere integrerte kretser, og det forventes at Moore's lov snart vil opphøre. Dette har ført til en voldsom interesse for nye metoder for å oppnå ytelsesforbedring i datamaskiner. Dagens hovedtilnærming er å benytte akseleratorer — spesialiserte kretser som er spesielt effektive for en spesifikk applikasjon eller et applikasjonsdomene. Selv om akseleratorer er effektive, er de også tidkrevende og kostbare å utvikle. I denne oppgaven har vi implementert en akselerator for den antatt akselleratorvennlige SPEC CPU 2017 applikasjonen LBM. Akseleratoren er realisert i en Field Programmable Gate Array (FPGA), og for å forenkle utviklingsprosessen har vi implementert akseleratoren ved hjelp av High-Level Synthesis (HLS). Akseleratoren forbedrer ytelsen med mellom 1.3× og 1.5× for et realistisk datasett sammenliknet med kjøre applikasjonen på en vanlig prosessor. Videre har vi vurdert HLS som implementasjonsmetode mer generelt og kommet til at selv om den forenkler utviklingsprosessen betraktelig må utvikleren fortsatt ha en god forståelse av applikasjon, målarkitektur og utviklingsverktøy for å oppnå et godt resultat.

# Contents

# Figures

# Tables

# Code Listings

# Acronyms

**ASIC** Application-Specific Integrated Circuit. 6, 7, 35

**BGK** Bhatnagar Gross and Krook. 10

**CLB** Configurable Logic Block. 6, 7, 40, 41, 45

**DSA** Domain Specific Accelerator. 2, 3, 5, 6

**DSL** Domain Specific Language. 6

**DSP** Digital Signal Processing. 7, 20, 32, 40

**FPGA** Field-Programmable Gate Array. ix, 6–8, 17, 20, 22, 27, 30, 31, 34, 35, 39, 43, 44, 50

**GUI** Graphical User Interface. 19

**HLS** High-Level Synthesis. ix, 3, 4, 6, 17–20, 22, 23, 39, 42, 49

**HPC** High Performance Computing. 2, 10

**ISL** Iterative Stencil Loop. 2, 10, 22, 26

**LBM** Lattice Boltzmann Method. ix, 3, 9, 10, 12, 14, 25–27, 49

**LUT** Look-Up Table. 7, 20, 40, 45

**RTL** Register-Transfer Level. 6, 17, 19, 22, 49

**TRT** Two-Relaxation-Time. 10

# Chapter 1

# Introduction

## 1.1 Current Landscape of Computer Architecture

The field of computer architecture has seen a huge development since the creation of the first general purpose electronic computer in 1945. Parameters such as performance, power consumption and size have been greatly improved since then. An important factor of this progress is the improvement in manufacturing technology. Manufacturers were able to constantly increase the transistor density of our chips, while keeping the power density constant and increasing the frequency at which they work.

In 1965, Gordon Moore made the observation that the number of transistors in a chip would double every year [1] and, in 1975, he reviewed his observation, saying that said number would double every two years [2]. This prediction held true for several decades, and it is now known as Moore's Law. In 1974, Dennard *et al.* analyzed the scaling of MOSFETs, and how they can be reduced in size [3]. Here they described how threshold voltage can be reduced together with the transistor size, thus making possible to also reduce voltage supply. Therefore, while transistors get smaller, and thus transistor density increases, the power density can remain constant. This is known as Dennard scaling.

We have leveraged Moore's Law and Dennard scaling for decades, but we are reaching the end of this ride. Dennard scaling ended around 2004 [4], not being possible to further decrease the voltage supply due to the proximity of the threshold voltage. Moreover, we are also facing the imminent end of Moore's Law. With this, the performance growth seen in recent decades, with over 50% increase per year, is stagnating, as seen in Fig 1.1.

Now we are facing the power wall. The end of Dennard scaling meant that the power density started to increase as the transistor size decreased. This lead to the point were power dissipation is becoming an issue. Nowadays it is common to under-utilize the transistors available, being those transistors unused during a certain application execution known as Dark Silicon [6]. Moreover, with growing importance of embedded and IoT devices, energy and power have become the main design constraints.

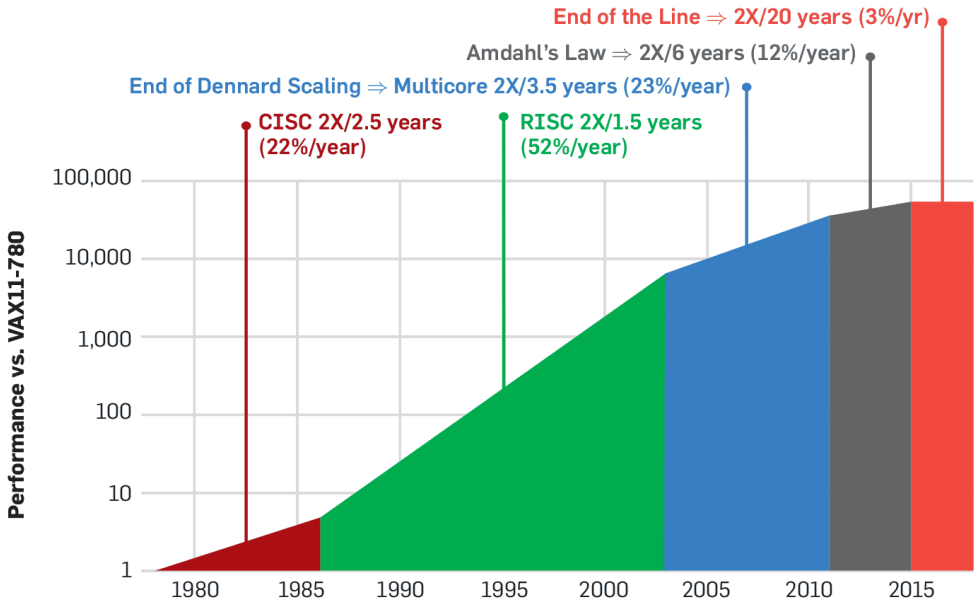



**Figure 1.1:** Growth in processor performance over 40 years (Reproduced from [5])

The approach initially taken to solve this was the use of multicore architectures, no longer focusing only on instruction-level parallelism, but giving more importance to data- and thread-level parallelism. However, this approach is also facing limitations, mainly finding applications with a level of parallelism high enough that can take advantage of hundreds or thousands of cores, as it is explained by Amdahl's Law [7]. This was later reviewed by Gustafson [8], in what is known as Gustafson's law. This new analysis defends that the parallelism is not a problem as long as we can scale the data proportionately to the cores, but that the bottleneck will be in the memory bandwidth, as we may not be able to feed all the cores.

Against this new obstacle, the approach taken is the use of hardware accelerators: circuits highly optimized for executing specific algorithms as fast and efficiently as possible. This is a trade-off of area against performance and energy efficiency [9]. These accelerators are also known as Domain Specific Accelerators (DSAs). They allow to take advantage of the specific characteristics of the target application, and provide an architecture more suited to it. An important consideration about DSAs is that are not designed to accelerate full applications, but to target small compute-intensive kernels. This way is possible to obtain the maximum performance improvement with minimal area increase. Areas where DSAs have great importance are machine learning, with special focus on deep neural networks [10], and Iterative Stencil Loops (ISLs), key kernel in many High Performance Computing (HPC) applications [11].

However, the design process of these accelerators is costly and time-consuming.

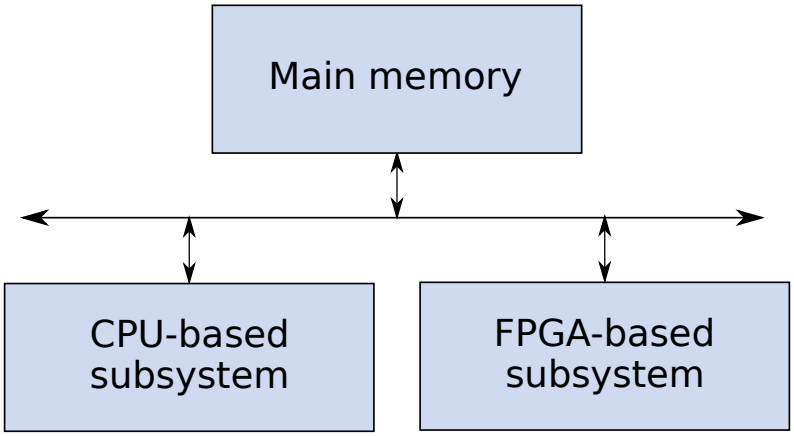



**Figure 1.2:** Simplified platform block diagram

While designing DSAs, there are two main obstacles: their complexity and the economical feasibility. Overcoming these obstacles is one of today's main challenges for computer architects.

## 1.2 Assignment Interpretation

In this thesis, we will continue with the work done in [12]. The goal is to design and implement a hardware accelerator for LBM, one of the benchmarks in SPEC CPU 2017. We chose this specific benchmark because it appeared to be one of the most accelerator-friendly SPEC benchmark, as it spends 99% of the time in a single function. However, while we can easily deduce what we need to accelerate, how to do so is not an easy task. Here, we aim to make efficient use of the bottleneck resources in the target platform. Said platform offers a heterogeneous architecture, with a processor-based subsystem and a reconfigurable fabric, both sharing main memory, as seen in Fig. 1.2.

In our previous work we have profiled the application, running it single core of the target platform. This analysis of both the algorithm and the original implementation is presented once again in Ch. 3.

For this Thesis, we have interpreted the assignment text to ask us to address the following three tasks:

- **T1** Propose and implement a hardware accelerator for LBM.
- **T2** Evaluate how the accelerator utilizes the resources of the target architecture.
- **T3** Do so using HLS, and evaluate the utilization of this tool as a way to simplify the development process.

## 1.3 Contributions

To begin with, for addressing task T1, we have proposed several acceleration strategies. We have initially started with a simple one, implemented it, and then iterated over it, finding its weak points and trying remove them. For the implemented proposals, we compared them to the original software implementation, and analyzed the results.

Besides implementing these hardware accelerators and analyzed their results, we have also analyzed how efficiently they use the resources available, where are the bottlenecks, and what is the margin for improvement, thus addressing task T2.

While doing this, we were taking note of the advantages and disadvantages of using HLS. This way, we have been able to contrast our initial expectations with our actual experience, as well as answering task T3.

# Chapter 2

# Background

As we said in the Sec. 1.1, the design process of hardware accelerators is costly and time-consuming. While designing DSAs, there are two main challenges hardware developers have to face: overcoming their complexity and doing so while also being economically feasible.

## 2.1 Transparent Acceleration

Regarding the complexity, designing no longer requires the traditional skill set of hardware designers. Until recently, they just needed to focus on the hardware side, as they had an instruction set architecture to separate them from the software. However, nowadays this is not enough, and they require also knowledge of algorithms and application domains. This is needed to understand which parts of the application can be offloaded and how this can be done. It is no longer purely hardware design, but hardware/software co-design.

To solve this problem appears the concept of transparent acceleration, which aims to offload the application with minimal effort. The ideal would be to write the code in a high-level language, and then the tool transparently offloads the application into an accelerator. While there are domain specific frameworks and languages that can do this, it is not clear if it can be extended to general purpose code. As it is hardware/software co-design, it is divided in two main areas: the architecture and the framework.

### 2.1.1 Architecture

Traditional DSAs sacrifice programmability to improve performance and/or power efficiency. This implies that it is harder to map different kernels to the same hardware. The more specific an accelerator is, the better it will perform for its target application, but the more difficult it will be to offload similar applications to it, and the improvements achieved for these applications will be smaller. If we want to achieve the highest speedup possible for different applications, we would have

to design different accelerators, which would increase both design and validation costs and time, as well as area consumption [13].

To solve this issue there are approaches such as LSSD [14, 15]. In this work, Nowatzki *et al.* propose a programmable architecture which achieves good performance and efficiency while still providing a certain degree of programmability. Although it is not as area efficient as a traditional DSA, it can be used to replace several of them, thus achieving a better area efficiency in total. This way, the effort needed to develop several highly specific accelerators can be reduced to develop a more programmable one, but still obtaining good performance and power results.

### 2.1.2 Framework and other software tools

On the software side, the idea is to use frameworks and other software tools to simplify the offloading process. Frameworks which support this are able to compile the code and map the algorithm to the target application, or make use of HLS to generate the Register-Transfer Level (RTL). They can also handle the hardware/software interfaces, as well as the memory accesses. In short, they bring the problem to a higher abstraction level. This way, many of the hardware details that would have been taken care of manually are simplified out of the development process. In this area there are some works [16–18], but in most of the times these tool are not yet enough. The designer still needs to adapt the algorithm so that the tool can properly map it onto the target architecture, or generate the RTL design for the kernel and the hardware/software interface. Commercial approaches, such as Xilinx's Vitis, also suffer from these problems.

Other software tools that can be used are Domain Specific Languages (DSLs). General purpose languages may not be able to express the particularities of the target architecture, so that the algorithm cannot be properly mapped, or the generated hardware is not the one the developer had in mind. The use of DSLs can help overcoming this barrier, as they can properly express the inherit parallelism of the application.

## 2.2 Field-Programmable Gate Arrays

In addition to the increased complexity, the other main barrier for hardware accelerators is to design and produce them while being economically viable. The non-recurring engineering costs for designing and manufacturing the Application-Specific Integrated Circuits (ASICs) are really high. Therefore, they need a high volume domain such as graphics or machine learning to be amortized [11, 19, 20].

For low and medium volumes, a feasible alternative is to use reconfigurable chips, such as FPGAs. These devices are integrated circuits composed mainly of multiple Configurable Logic Blocks (CLBs), an interconnect network and I/O ports. Programming these devices means configuring the logic blocks so that they will behave according to the RTL. How this blocks are configured is vendor dependent.

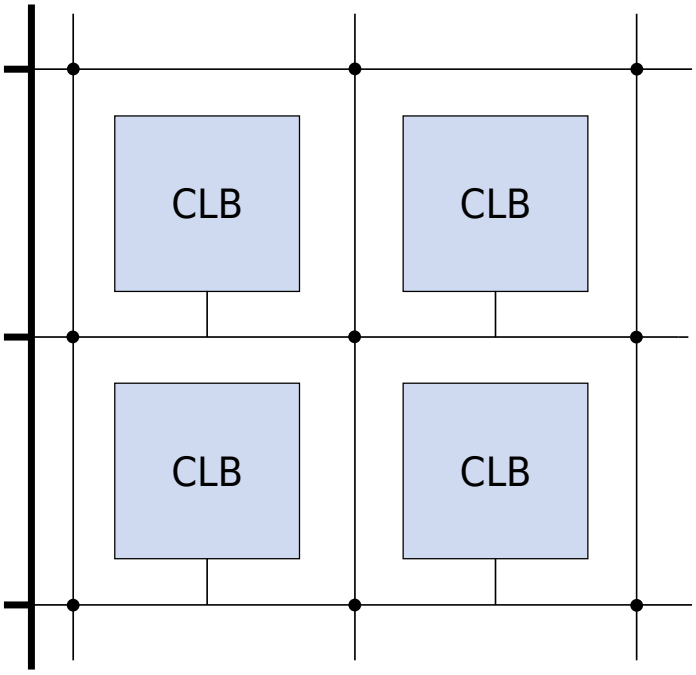



**Figure 2.1:** Simple FPGA layout

While their performance is not as good as their ASIC equivalent (with the same manufacturing technology), they have lower non-recurring engineering costs and are more flexible, being possible to reuse them in the future without needing to buy new devices. Moreover, nowadays manufactures produce hybrid architectures, providing both CPU and FPGA on the same chip, such as the Zynq-UltraScale+ from Xilinx.

As we just mentioned, FPGAs consist in several CLBs connected with an interconnect network, not only with other CLBs, but also with I/O ports, as exemplified in Fig. 2.1. As for what it is inside the CLBs, it is vendor dependant. In the case of Xilinx UltraScale+, it implements function generators as six-input Look-Up Tables (LUTs), with two independent outputs each. Each CLB contains 8 LUTs and 16 flip-flops, as well as arithmetic carry logic and multiplexers. Moreover, some CLB slices contain distributed RAM, bringing memory next to computation. Another important component of Xilinx FPGAs' fabric are Digital Signal Processing (DSP) slices, which implement operations in hardwired hardware. In this architecture, DSPs implement a 27x18-bit signed multiplication, followed by a 48-bit adder/accumulator. Moreover, they have a 27 bit pre-adder. Besides the distributed RAM in the CLBs, UltraScale+ devices come with block RAM: dedicated 36kb blocks, with two read and write ports. By cascading them, we can generate much more larger memories on the fabric. Regarding the frequency, this is customizable in the FPGA. It has specific clock busses, but we can configure which frequency goes through them.

## 2.3 Benchmark suites

Up until now we have discussed the development of computer architecture until today, as well as the challenges we are currently facing. A key parameter to analyze this development has been the performance. Therefore, it is needed to establish a way to measure it. Performance can be defined as the reciprocal of execution time, and the best way to measure it is to benchmark it against real applications [4]. While benchmarking individual applications can lead to unfair situations, specially for general purpose architectures, they can be collected together in benchmark suites. These suites have the advantage that the weakness of any individual benchmark can be compensated by the others. Examples of this are PARSEC, Rodinia and MachSuite.

Between the most popular benchmarks suites are the ones provided by SPEC, a non-profit organization formed in 1988 to establish, maintain and endorse standardized benchmarks [21]. Their benchmarks have evolved together with the computer architecture, and nowadays they offer a wide range of benchmark suites to cover multiple areas and requirements.

## 2.4 Accelerating LBM

Among the different benchmarks suits provided by SPEC, one of the most known ones is SPEC CPU, dating its last version from 2017. One specific benchmark included in this suite is LBM. This algorithm is presented in detail in Ch. 3.

Being part of this popular suite, starting from the 2006 version, there is already previous work in accelerating algorithm. Sano *et al.* proposed an acceleration strategy, although for the 2D version of the problem, using an FPGA[22]. In their work, they propose to stream the data in multiple cycles (3 values in parallel per cycle), to optimize the bandwidth usage without saturating it. In later work, they extended this to a cluster of FPGAs [23]. While they have done so using different vendors for both the actual hardware and IPs, they still had to write low level HDL code in Verilog.

Leaving FPGAs, Ren *et al.* proposed an optimized implementation in GPUs [24]. In this case, they used the 3D version of the algorithm, as in the SPEC CPU benchmark. This work also places great importance in establishing a good memory strategy.

# Chapter 3

# Lattice Boltzmann Method

In this chapter we will analyze the LBM algorithm, as well as its software implementation. The contents of this chapter were the focus of our previous work [12], which this Thesis continues. We have included this here for completeness. Secs. 3.1 and 3.2 have been barely changed, while Sec. 3.3 is the summarized version of the corresponding section in the previous work.

## 3.1 The Mathematical Model

The Lattice Boltzmann Method (LBM) is an algorithm widely used in the field of computational fluid dynamics (CFD) for fluid simulation. Compared to traditional algorithms in this field, which do the calculations based on macroscopic properties, LBM makes use of micro-particles to establish a velocity model. Then, based on the statistical physics method, it obtains the macro-flow characteristics according to probability distribution functions [25].

In the computational model, the simulated fluid is placed into a grid, representing the simulation space, and divided into cells placed at its nodes, representing the particles. The distribution function used can be expressed as $f(\vec{x}, \vec{u}, t)$, where $\vec{x}$ represent the position in the space, $\vec{u}$ represents the particle's velocity and $t$ represent the time [27]. With this, the idea is to calculate the probability $f$ of a particle with speed $\vec{u}$ appearing in position $\vec{x}$ at time $t$. Here, the micro-particle motion model is obtained from the Boltzmann equation:

$$\frac{\delta f}{\delta t} + \left\langle \vec{u} \, , \, \nabla f \right\rangle = \frac{1}{\lambda}\left(f - f^{(0)}\right) \tag{3.1}$$

Particles move in the discrete space with a certain probability and in several directions, colliding with others. This is done with two operations per time step: stream, where the data from neighbouring cells are retrieved, and collide, where the new values are computed. This update process is represented in Fig. 3.2a for a 2D space with 9 speed directions (D2Q9). However, this can also be done otherwise, reading the values from the central cell and calculating the values for the neighbouring cells, as seen in Fig. 3.2b.

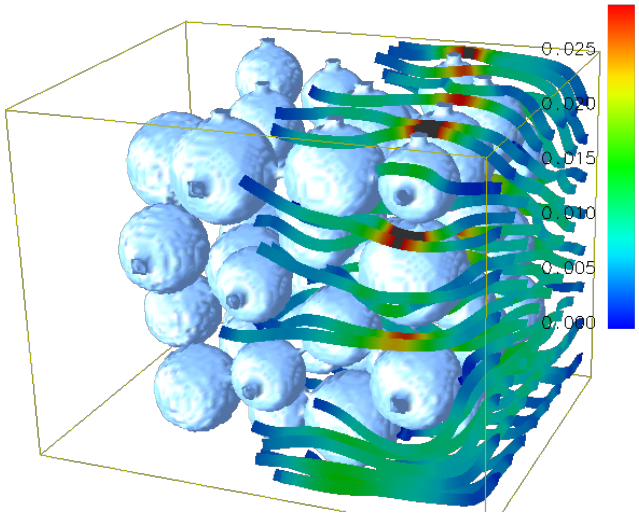



**Figure 3.1:** Flow of a fluid through a space with spherical obstacles (Reproduced from [26])

There are different space models for LBM, both in 2D and in 3D. In this case, a D3Q19 model is used: it represents a 3D space and every cell has 19 speed directions. A cell in this model is represented as seen in Fig. 3.3. Here it is possible to observe that every speed vector points to the opposite one in the neighbouring cell (except the one in the center), thus modeling collisions.

## 3.2 SPEC CPU Implementation

The LBM algorithm to optimize is the one provided by SPEC CPU, and developed by Thomas Pohl in C [28]. This is a problem based on an ISL, key kernel in many HPC applications [11]. This type of kernel iteratively modifies the elements of an array, taking the values from the current iteration step to compute the values of the next one.

In this specific implementation of LBM, it works as shown in Alg. 1. In this case, the stencil is the collide-stream operation and the iteration step is the time step. Then, there are two possible variations of the kernel: the Bhatnagar Gross and Krook (BGK) and the Two-Relaxation-Time (TRT). The latter is the one used in the SPEC benchmark and, therefore, the one that will be analyzed here. The pseudo-code for this stencil is in Algorithm 2. Fig. 3.4 is the data dependency graph for this kernel.

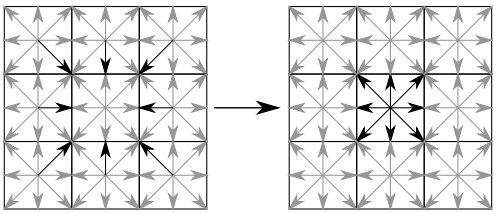
In this implementation, cells are sorted starting by their $z$ coordinate, then by their $y$ coordinate and finally by their $x$ coordinate, as exemplified in Fig. 3.5. Each cell consists of 20 values: 19 for the speed vectors, as seen in Fig. 3.3, and one extra to use as a flag. All of this is stored into a 1D array instead of on a 4D one, thus being needed just one pointer for the whole grid. The data type used is

**Algorithm 1** Main LBM ISL

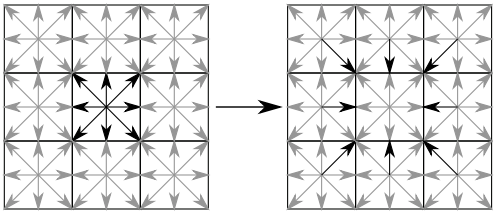
1: **for** step in time_steps **do**
2: **for** cell in array **do**
3: $collide_stream(src[], dst[])$
4: **end for**
5: Swap arrays
6: **end for**

**Algorithm 2** Collide-stream kernel

1: **if** cell is obstacle **then**
2: $dst[] = src[]$
3: **end if**
4: $\rho = \sum(src[])$
5: $u_x = \left(\sum(src[i] \cdot C_{xi})\right)/\rho$
6: $u_y = \left(\sum\left(src[i] \cdot C_{yi}\right)\right)/\rho$
7: $u_z = \left(\sum(src[i] \cdot C_{zi})\right)/\rho$
8: **if** cell is accel **then**
9: $u_x = 0.005$
10: $u_y = 0.002$
11: $u_z = 0.000$
12: **end if**
13: $u_2 = 1.5 \cdot \left(u_x \cdot u_x + u_y \cdot u_y + u_z \cdot u_z\right)$
14: **for** vector in cell **do**
15: $feqs[vector] = f(\rho, u_x, u_y, u_z, u_2)$
16: $feqa[vector] = f(\rho, u_x, u_y, u_z)$
17: $fs[vector] = f(src)$
18: $fa[vector] = f(src)$
19: **end for**
20: **for** vector in cell **do**
21: $dst[vector] = f(src, feqs, feqa, fs, fa)$
22: **end for**



**(a)** Stream - Collide



**(b)** Collide - Stream

**Figure 3.2:** Operations performed by LBM

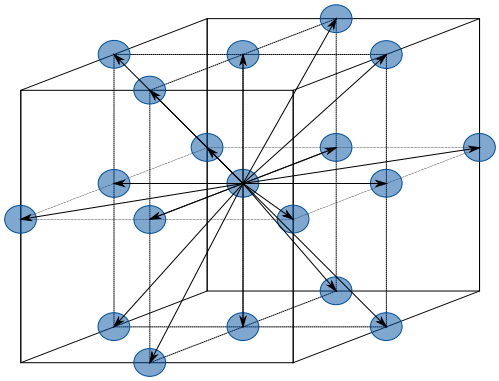


**Figure 3.3:** D3Q19 cell

double floating precision, for both the speed vectors and for the flags. With all of this, a grid is represented as an array such that $Array\ Size = d_x \times d_y \times d_x \times 20 \times sizeof(double)$. The memory for this array is allocated dynamically with *malloc*. Moreover, extra space is allocated for supporting padding, thus solving the issue of dealing with the borders. As the grid is stored in a 1D array, calculating the index for the required speed vector is not trivial. Thus, several macros are provided for handling this complexity.

In each iteration step, the grid is traversed through all its cells. After doing the computations for the corresponding cell, the speed vectors of the neighbouring cells have to be updated. While the classic and intuitive stream-collide approach would be to update the central cell with values from the neighbouring cells, this would have some negative implications: for each time step, a single cell would take part in the update process of 19 different cells, including itself. Thus, all its values would have to be loaded 19 times. However, these values are used to calculate

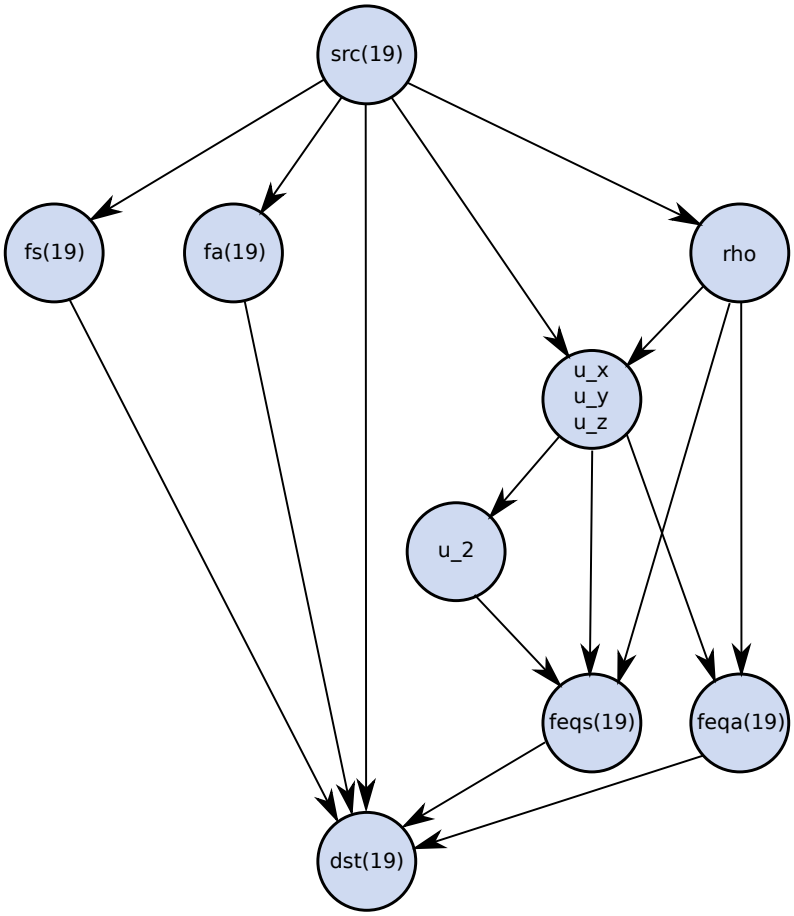



**Figure 3.4:** LBM Kernel data dependency graph. Between brackets, number of elements in the array.

some intermediate parameters, which are then used to update the speed vector of the central cell, all within the same kernel. These intermediate parameters are the same for all the update cases, and thus loading the needed values and calculating them every time should be avoided. By calculating the intermediate parameters of the central cell and then sending the value to the neighbouring cells, this is avoided. Therefore, this algorithm implements the collide-stream method shown in Fig. 3.2b, and not the classical stream-collide method as in Fig. 3.2a.

Also, the update cannot be done overwriting the values of the currently traversed grid, as some of those values still need to be read within the same iteration. The approach taken here is to have the array duplicated, using a source grid and a destination grid. Data is read from the former and written into the latter. At the end of each iteration, the pointers to the grids are swapped, and thus in the next iteration the source grid will have the values written in the previous one.

To describe the space, an obstacle file is used. It is a text file with one character for every position in the grid. The dot character represents a fluid cell, while other

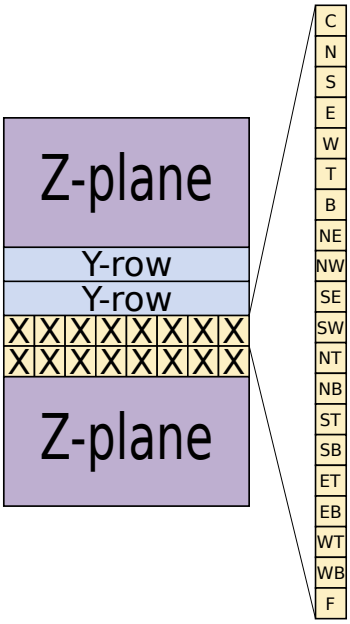



**Figure 3.5:** Grid distribution in memory

characters refer to obstacles. During initialization, this file is read, and the flag is set to its obstacle value when needed. Besides this obstacles, all the borders of the grid are also set up as obstacles, thus creating a closed environment. Also, the inner side of one of the border faces is flagged as acceleration particles. They are used for simulating an input flow.

Regarding the speed vectors, they are initialized with constant values. This is done independent of their position and their flag. These constants are given by the application for every type of speed vector.

In each iteration the first task is to check whether the current cell is an obstacle or not. If it is, the speed vector's directions from the cell are directly transmitted to the neighbouring cells, and the calculations are omitted.

## 3.3 Profiling

In our previous work we have analyzed and profiled this implementation of LBM in the target platform (see Sec. 6.1) [12]. We showed that execution time is directly proportional to the number of time steps, as well as directly proportional to the number of nodes in the grid. Moreover, we found that the execution time was shorter for grids with greater percentages of obstacle particles. For a fixed size, the best case would be when the grid is full of obstacles (although this is not a realistic data set), and the worst case would be when the grid is empty of obstacles. We can see this in Fig. 3.7.

The key main point about this kernel is that it is heavily memory bound. In Fig. 3.7a it is possible to see how the cache size architecture affects the instructions

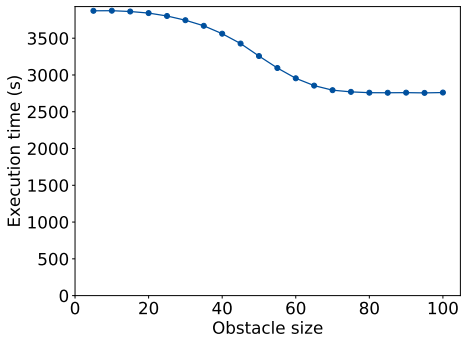
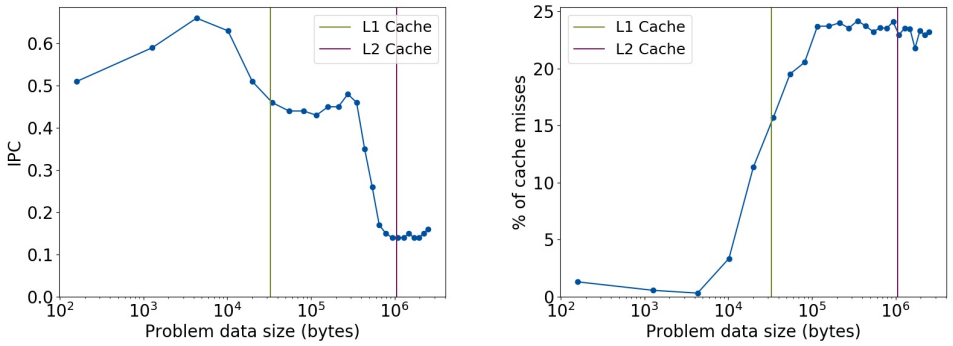



**Figure 3.6:** Runtime vs radius (Cubic grid with side size 100, centered spherical obstacle) [12]




**(a)** IPC

**(b)** L1 cache misses

**Figure 3.7:** Effects of problem data size [12]

per cycle. Moreover, in Fig. 3.7b we can also see how the L1 cache misses greatly increase at the same size at which the previous parameter has its first step. To give a better description on how heavily memory bound this problem is, we can look back to Fig. 3.7. Points in the left represent the worst case scenario, while points in the right represent the best case scenario. In the latter, the grid is completely full of obstacle particles. Due to this, there is no actual computation: the flag is checked and, as it is an obstacle, the read speed vectors are immediately written to their corresponding position in the destination grid. However, even if there is no computation besides one comparison, the execution time is only 70% of the most computational demanding case.

# Chapter 4

# Accelerating Applications with High-Level Synthesis

As discussed in Ch. 2, designing an accelerator is not an easy task. One of the approaches we mentioned to reduce complexity is to use High-Level Synthesis (HLS). This technology allows to convert algorithmic descriptions written in high level languages, such as C/C++, into hardware description (RTL), thus elevating the abstraction level from RTL to algorithms. We can see how this works in Fig. 4.1. Ideally, we no longer need to worry about implementation details such as clocks or technology, following the concept of transparent acceleration (Sec. 2.1).

The tool is capable of extracting the control- and data-flow from the source code, and implement designs based on them. Moreover, it is possible to use directives to inform the tool about specific ways how we want it to behave. This allows to get multiple different implementations form the same source description, thus optimizing design exploration.

However, as we can see in Fig. 4.1, while HLS in an important part of the process, it is not the end. Once it has generated the hardware specification, the tool still needs to execute other steps, such as logic synthesis and implementation. This means that the hardware details, such as resource consumption and delays given by the HLS tool are just indicative. The actual results are given in the implementation report.

This is important because there can be huge differences between what resources HLS estimates and what resources are actually used after implementation. This means that is not trivial to know if the resources available on the actual FPGA will be enough based on the HLS report. However, HLS takes much less time compared to logic synthesis and implementation. Therefore, it is good to know how to estimate the actual resource consumption based on the HLS results. This can be done by having a reference result, and doing an estimation based on them.

Besides the hardware side, we also need to handle the software side. For this, parallel to HLS, a compiler is in charge of generating the binary file. It will be the one to offload the data to the reprogrammable fabric. For doing so, the accelerator has to be exposed to the software. The way this is done depends on the specific

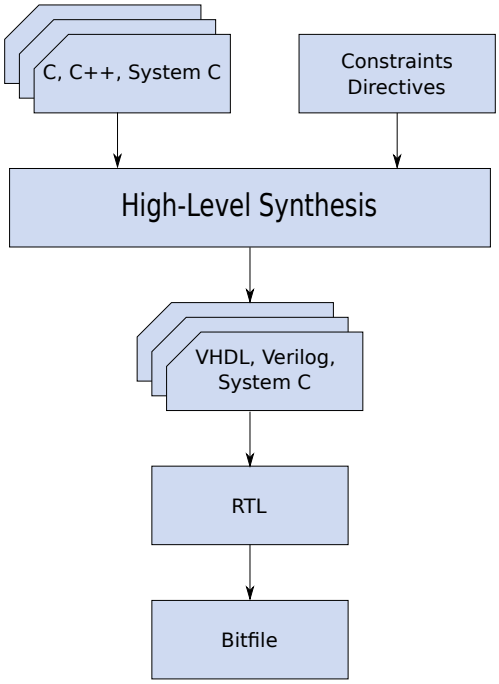



**Figure 4.1:** HLS process flow

tool. It can be done by simply calling C/C++ functions, or by using other standards such as OpenCL.

## 4.1 Stages of HLS

The process of HLS can be divided in two main steps or stages: the scheduling and the allocation and binding.

### Scheduling

Scheduling consists in arranging the operations on a time scale, assigning them to specific clock cycles, while fulfilling the inter-dependencies between them, that is, respecting the precedence constraints. The goal is to execute all the operations in the shortest time possible. Moreover, the tool also has to consider how long operations take, as they can take several cycles, and if there is also any resource constraints.

### Allocation and Binding

The next step is to build a datapath based on the given schedule. Here, operations are mapped to actual hardware cores. These cores are taken from the hardware library of the corresponding target device.

**Code listing 4.1:** Source code for Vitis HLS example

```
int foo(char x, char a, char b, char c) {
    char y;
    y = x*a+b+c;
    return y;
}
```

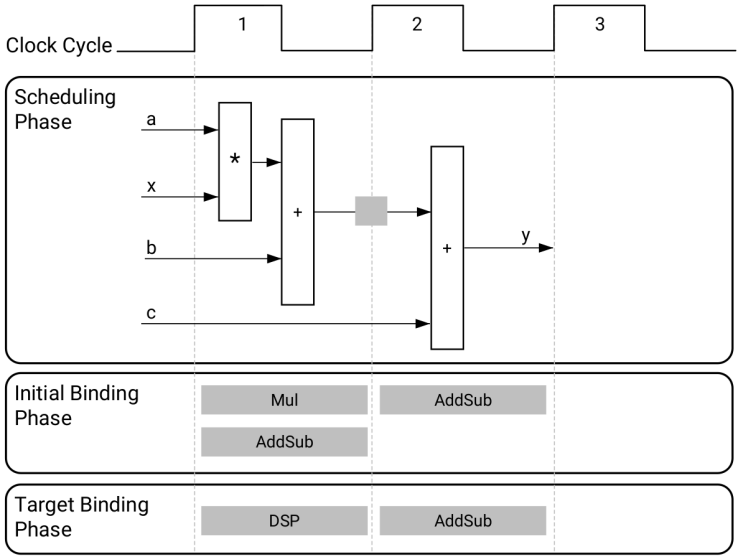



**Figure 4.2:** Scheduling and Binding example for Vitis HLS [29]

## 4.2 Vitis HLS

Once knowing how HLS works, we can familiarize ourselves with how the specific tool we are working with uses it. As our target platform will be based on a Xilinx's chip (see Sec. 6.1), we will use the manufacturer's tool, Vitis HLS. In this tool, the HLS process consists on three different stages:

1. Scheduling.
2. Binding.
3. Control logic extraction

While the first to stages correlate to the ones mentioned before, the last one creates a finite state machine in order to sequence the operations in the RTL design, so that the schedule can be fulfilled. One example of how the tool would operate is shown in Fig. 4.2, were we can see how the tool would do the scheduling and binding stages for the C function showed in Lst. 4.1.

In Vitis HLS, user directives can be given as pragmas embedded inside the code or throught the user interface. This way, depending on their placement, they will affect specific parts of the code. Besides doing the HLS process, the tool also offers a Graphical User Interface (GUI) that gives information about the resulting

schedule. We can see the schedule itself, the inter-dependencies between operations, the allocated hardware resource for each operation, the bit width and the delay, among other parameters.

### 4.2.1 Resource Limitation in Vitis HLS

When developing for FPGAs, one of the tasks is to make our implementation fit into the target platform. While Vitis HLS has its information, this does not imply that it will make any code fit in it by default. We need to know how to reduce the resource consumption.

One of the simplest ways is to make use of the *allocation* pragma. With it, we can limit the resource consumption of the units used during the binding stage. While this may lead to the original schedule not being fulfilled, the tool will just generate a new one. Following an iterative process, it will end up finding an schedule that can be satisfied in the binding stage while also respecting the resources limitation. The main penalty for doing so is not being able to achieve the desired number of pipeline stages (which is 1 by default). To transition between pipeline stages, registers will have to be instantiated for temporary storage.

Other option is to change the data type. For example, if the base algorithm works with floating point data types, the tool will use the DSPs slices. As they have hardware multipliers, the tool tries to map multiplications to them. For dealing with floating point numbers, several multiplications are needed, so their usage increases greatly. To solve this, we can use fixed point data types. For that, Xilinx provides its custom implementation of fixed point data types. Besides the potential precision loss, this will greatly decrease the DSP usage in exchange of LUT usage. This trade-off shall be analyzed for the specific case.

## 4.3 OpenCL

Regarding HLS, last, but not least, we need to know how to integrate its kernels into our code. This is important, because by this we will also be describing how the offloading is done. This depends on which options are supported by the developer of the tool.

In the case of Xilinx, this is done in their Vitis IDE tool (which internally calls Vitis HLS and other tools needed). This software platform supports for acceleration normal C/C++ functions. We just need to select the function we want to be accelerated, and the tool will do so for us. However, this approach is supported because it is inherited from the previous tool, which Vitis has replaced. In Vitis, it is encouraged to use OpenCL for handling the offloading.

OpenCL is an open standard for parallel programming in heterogeneous architectures. It uses C/C++, and it is widely supported by the industry. This way, Vitis adopts the industry standard, thus allowing code compatibility with applications originally targeted at other hybrid architectures.

**Code listing 4.2:** OpenCL example

```
  std::vector<DATA_TYPE, aligned_allocator<DATA_TYPE>> grid_0_HW(DATA_SIZE);
  std::vector<DATA_TYPE, aligned_allocator<DATA_TYPE>> grid_1_HW(DATA_SIZE);
  size_t vector_size_bytes = sizeof(DATA_TYPE) * DATA_SIZE;

  // Allocate Buffer in Global Memory
  OCL_CHECK(err, cl::Buffer buffer_0(
                    context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE,
                    vector_size_bytes, grid_0_HW.data(), &err));
  OCL_CHECK(err, cl::Buffer buffer_1(
                    context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE,
                    vector_size_bytes, grid_1_HW.data(), &err));

  DATA_TYPE inc = INCR_VALUE;
  int size = DATA_SIZE;

  // Set the Kernel Arguments
  int narg = 0;
  OCL_CHECK(err, err = krnl_adder.setArg(narg++, buffer_0));
  OCL_CHECK(err, err = krnl_adder.setArg(narg++, buffer_1));
  OCL_CHECK(err, err = krnl_adder.setArg(narg++, inc));
  OCL_CHECK(err, err = krnl_adder.setArg(narg++, size));

  for(int t=0;t<TIME_STEPS;t++){
    // Copy input data to device global memory
    OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_0},
              0 /* 0 means from host*/));

    // Launch the Kernel
    OCL_CHECK(err, err = q.enqueueTask(krnl_adder));

    // Copy Result from Device Global Memory to Host Local Memory
    OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_1},
              CL_MIGRATE_MEM_OBJECT_HOST));

    // Swap the arrays
    std::swap(buffer_0,buffer_1);

    // Update the arguments
    narg = 0;
    OCL_CHECK(err, err = krnl_adder.setArg(narg++, buffer_0));
    OCL_CHECK(err, err = krnl_adder.setArg(narg++, buffer_1));

  }

  // Wait until all queued tasks are finished
  q.finish();
}
```

In Lst. 4.2 we can see an example of how Xilinx's implementation of OpenCL can be used to offload a simple application into the reprogrammable fabric, and swap the input and output arrays after each iteration (thus behaving like an ISL). The swapping is done in the processor.

From this code, it is important to understand what memory migrations means: copying data to and from hardware memory. In Xilinx's hybrid architectures, the processor-based subsystem and the FPGA-based subsystem share main memory. However, the tool divides this memory, assigning a specific memory region for the hardware fabric. As the swapping is done in processor, for each iteration the application needs send the data to the processor memory, swap the arrays, and sent it back to the fabric again. Moreover, while the swapping is done in constant time, moving the data in memory is not, requiring linear time. Therefore, the bigger the array, the more time is spent doing this.

## 4.4 Our Experience with HLS

One of the goals of this Thesis was not only use HLS, but also to evaluate it. While in theory it was promising, this does not necessarily correlates with reality, and one of the tasks of this Thesis is to verify this.

From our experience during this work, HLS has advantages over traditional hardware design procedures, but also drawbacks. Between its main advantages, it raises the level of abstraction, taking care of many low level details. Programming this tool is more similar to traditional high-level programming than to programming RTL. Moreover, it handles many important elements, such as memory accesses and interrupts, that otherwise would have to be manually configured.

However, while doing many things for us, it does not always do what we wanted or expected from it. This should be added to the fact that we still need be knowledgeable about the target architecture. While by default it offloads application in a way that they work (even though correctness it is not guaranteed, cases were this does not happen are very rare exceptions), to achieve optimal implementations we need to know how to express this in a way that the tool will make use of the hardware resources that we want it to use. This implies not only that, but also that we need to know which hardware resources are available, and how to use them efficiently, as the tool will not do this for us. Also, debugging tools can still not handle all the different layers present and, although it is possible to see the RTL code generated, it is far from readable. Moreover, as we are programming at high levels of abstraction, it is not possible to implement all we want and how we want. We are limited, for example, by the usage of high level languages, which do not allow us to express everything that Verilog or VHDL would. Other problem we faced was hardware optimization. When dealing with issues such as resource consumption or timing constraints, the approaches found were quite limited. As we cannot go to the lower level, we cannot do small improvements to handle such issues.

As a summary, HLS left us a lukewarm impression. While raising the abstraction level reduces the complexity of designing hardware accelerators, it is not trivial to make the tool generate the hardware we actually want, and sometimes being able to do lower level modifications would be helpful. However, it remains to see if the problem lies in HLS being flawed as a concept, or if this applies only for Xilinx's implementation.

# Chapter 5

# Acceleration Strategies

Once having analyzed LBM, and knowing the target platform, we can move on to proposing how to accelerate it. First we will go over the general approach and then we will go into the implementation details.

## 5.1 Memory Model

As discussed in Sec. 3.3, LBM is a memory bound problem (for interesting data sets, cache bound if the data set is small enough). If we analyze the memory model shown in Fig. 3.5 we can see that, while it works well for reading the values from source grid, the same does not apply for writing into the destination grid. As explained in Ch. 3, in each iteration step the kernel works with the speed vector of one cell to then transfer the results to the neighbouring cells. Therefore, it needs to read the 20 values that compose a cell, which in this memory model are in consecutive memory positions. However, for writing, it needs to access 19 different cells. Taking as reference the central cell, two more destination cells will be in the same row, placed right before and right after in memory. 6 more cells will be in different rows within the same plane, and the 10 remaining cells will be in different planes. Therefore, these cells are placed within a range equivalent to 2 planes and 2 rows. While some cells are pretty close one to each other, in most cases they will be too far away to the system to fetch a cache block including more than one of them. Moreover, out of the 19 possible values of those target cells, we only need to write one. This means that, when working with non-unitary cache blocks, we will be fetching elements which are not needed.

Due to this we can see that we need to optimize data locality in order to achieve a better access pattern an reduce the amount of unused fetched data. Our proposal for this is to organize the memory placement in an element-major fashion. We can do this by using individual arrays for each speed vector, as well as one for the flags. Instead of creating an array of structures (although the actual implementation is a 1D array, conceptually it is an an array of structures), we will create a structure of arrays. This will allow sequential memory access not only

when reading, but also when writing. Moreover, this way we can minimize the amount of unused data fetched from memory.

Not only that, but this approach will also allow us to remove the flag duplicity. In the original memory model, they were always read from the source grid, and thus they needed to be stored in both grids. In this proposed model, flags can be stored in an array independent of both grids, thus saving not only unnecessary fetches, but also memory space.

## 5.2 Kernel Computation

Another important part to analyze is how to increase the number of operations without a bandwidth increase, i.e. to improve the operational intensity. This would increase the arithmetic intensity, moving closer to the compute-bounded area of the Roofline model [30]. With ISLs, one common way to do this is to calculate more than one iteration at the same time, and only saving to memory the results of the last iteration. We can see this in proposals such as SST [31], CA [32] and DCMI [11]. However, this approach relies on the intra- and inter-cell data dependencies and, in this case, they are not trivial. There are two main obstacles for this approach: multiple origins for the source and a division of sums.

Regarding the former, the speed vectors from the source cell come from 19 different cells. For the computation of the collide-stream operation of cell $(x, y, z)$ in time $t+n$, we would need to use the values from as many cells as $\frac{20}{3}n^3 + 8n^2 + \frac{10}{3}n + 1$ cells. Besides this, the intermediate calculations would also be required for other cells, moving in every direction. This values would have to be calculated again for those cells, unless we implement some sort of intermediate storage in the fabric. Handling all of this is far from trivial.

Regarding the latter issue, in the data dependency for the LBM kernel there is a division of sums. $rho$ is the sum of all the coefficients of the source array, while $u_x$, $u_y$, and $u_z$ are different sums of the source values multiplied by a coefficient ($\pm 1$), and then divided by $rho$. For this to be correct, we need to add all the summands before the division is done, at least for $rho$. Therefore, it is challenging (if not even mathematically impossible) to compute the partial results of the individual speed vectors, each of them being received from a unique neighbouring cell in the previous time step. Moreover, as we can see in Fig. 3.4, the output values directly depend on the input ones, so it is also not possible to do some precomputation before hand and reduce the data describing each cell into some intermediate values present inside the kernel.

Therefore, we propose to accelerate the algorithm by streaming the data sequentially, while also reducing the amount of unused data fetched.

## 5.3 Data Types

Data representation is an important part to consider in computing systems. Not only they define the precision of the data, but may also affect the execution time and memory utilization. In hardware accelerators, they also affect the resource consumption.

In the original implementation form SPEC CPU, all the data uses the double floating data type. While this format provides high precision, it also consumes quite a lot of memory compared to other possible alternatives. For example, by using single floating point, we could save not only 50% of memory space, but also of memory bandwidth, which is key in a memory bound problem like this. Moreover, this would also allow us to reduce resource consumption, as arithmetic units would be smaller.

Despite that, we are not going to change the grid's data type. Changing it to one with a narrower bit width would reduce its precision, thus achieving worse results. As we are no expects in the field of fluid dynamics, is not up to us to decide acceptable error margins. Being faithful to the original implementation, we have decided to continue using this data type, and aim for the lowest possible error, if any.

However, this does not apply to the flags. They are used for comparison and have integer values given upon initialization. Thus, precision will not be a problem. As they have only three possible values, we have decided to use 8-bit unsigned integers. This is the smallest data type provided by standard C/C++. With this, we will not have only removed the flag duplicity, as explained in Sec. 5.1, but also have greatly reduced the contribution to memory space and bandwidth of the remaining flag array.

## 5.4 Accelerators

Now that we know how we want to accelerate LBM, we can go to the specific implementation details. We started with an simple initial implementation that could offload the algorithm. Once this first implementation was working correctly, we analyzed it (Sec. 7.2). Then, with the newly gathered information we were able to improve our original proposal.

### 5.4.1 LBM-S

With the use of an FPGA, new possibilities for dealing with memory appear. While the FPGA does not have a dedicated cache, as we will see in Ch. 6, we could emulate this with the resources available in the chip. However, as our proposed memory model allows data to be accessed sequentially, we can stream the data in and out of the fabric (thus the "S" in LBM-S). We will do this following the idea of decoupled access/execute architectures presented in [33].

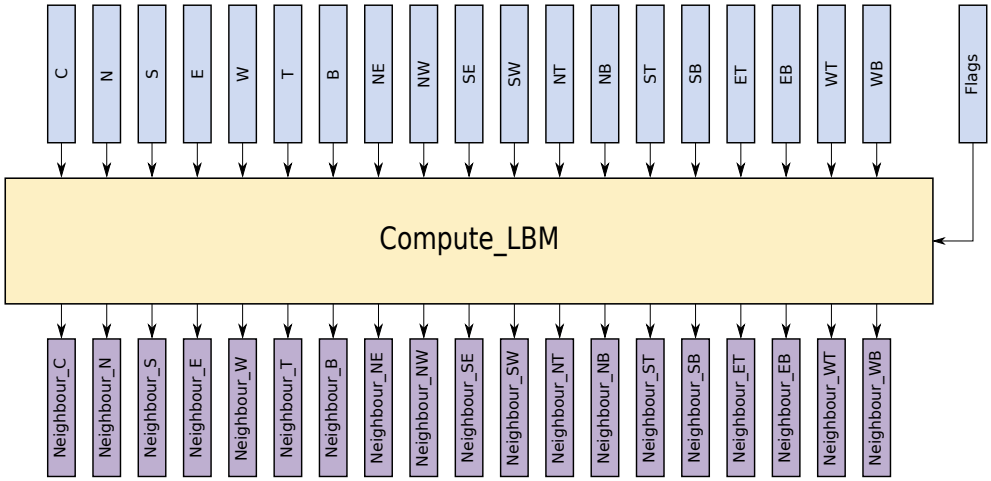



**Figure 5.1:** Scheme for streaming implementation

In order to do so, we will implement two units for communicating with the main memory: an input unit, composed by 19 + 1 input buffers for the values of the current cell and the flags, and an output unit, composed by 19 output buffers for writing the new values back to memory. We can see this in Fig. 5.1. The buffers in both units will act as FIFOs. The input data will be read from memory and stored in the corresponding buffer. Then it will be processed by the computing unit when all the needed data is in the inputs. When there is space available, the system will fetch new data from memory before it gets fully emptied. This will be done in parallel to the computing. This way, data will be fetched from main memory before the compute module runs out of data available and needs to wait for new data. The equivalent process applies for the output buffers, which allow the compute module to write data without having to wait for it to be properly stored, as the buffers will be the ones handling that. However, instead of having one processor focused on computing addresses and performing read/write requests and a second processor focused on doing the main computations, this time we have an input unit in charge of reading from memory filling the 19 + 1 input buffers, a computation unit, and an output unit in charge of writing to memory the data stored in the 19 output buffers. In this case, the computational unit is completely isolated from memory, and communicates only with the input and output units. Despite this, the main condition remains: the units communicating with the main memory need to be ahead the computing unit. If one of the input buffers is empty or one of the output buffers is full, the computing unit will have to wait. Ideally, the units should work at an equivalent rate, so that none of them have any dead time.

Moreover, with how the data is placed in memory, consecutive values are in consecutive positions, so handling this is quite straight-forward. Besides, there are no control conditions for reading data: since the initial position until the end, all data is needed, we will not need to handle flushing elements out of the input FIFOs.

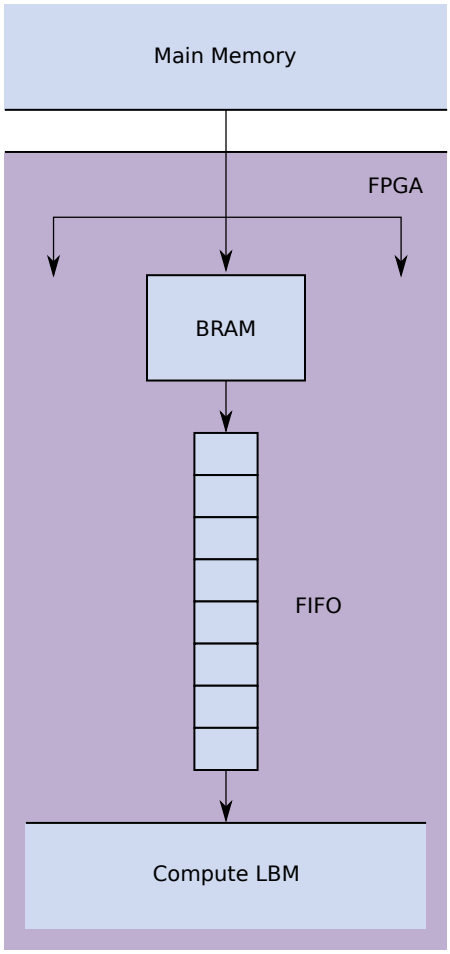



**Figure 5.2:** Memory subsystem

### 5.4.2 LBM-SM

As we will see in Sec. 7.2, our first implementation approach was not accelerating the application. This was due to the data being read and pushed to the FIFOs value by value, accessing to non-consecutive memory positions. In this second accelerator, we will focus on improving the memory system (thus the "M" in LBM-SM).

Going back to the decoupled access/execute architecture mentioned previously, the problem in our first implementation is that units in charge of accessing memory cannot keep with the compute unit due to the delays derived from the bad access pattern.

In order to improve this, we have to use burst accesses from and to memory for reading and writing. We have seen that the implemented FIFOs are working with single values. Therefore, one possible approach would be to read bigger chunks of data from memory, store them in an intermediate storage, closer to the buffers,

and push it from there (equivalent would be done for writing). As we will see in Tab. 7.1, we have barely used the BRAMs. Therefore, we could copy bigger parts of the input arrays into the BRAMs, using bursts to read/write the data. These storage units would be much closer to the buffers, so the delay for the unitary push/pop operations would be much smaller.

Fig. 5.2 shows how this new memory subsystem would be implemented (in this specific case, for reading the value of one array). The first thing to notice is that there is only one bus which comes from memory, and it is in the FPGA where the data is redirected to the specific BRAM. This means that the BRAMs cannot communicate with main memory in parallel. However, once the data is on the fabric, it can move from BRAMs to FIFOs (and vice versa) in parallel. When it is the turn of the corresponding BRAM, it will be filled/emptied, and then it will continue interacting with its corresponding FIFO while waiting for its next turn.

These way, there will be burst accesses from and to memory, providing a better utilization of the memory model. Then the push/pop operations will interact not with main memory, but with the BRAMs. This will not only be done in parallel, but the delays involved will be much shorter.

### 5.4.3 LBM-SMS

With the kernel computation being more efficient, the main problem of the second version of our accelerator is that it uses the processor for swapping the grids, which incurs in some overhead (see in Figs. 7.8 and 7.13 for a quantitative analysis). Although physically there is only one main memory, Xilinx's OpenCL implementation divides it in different regions for the processor and the hardware fabric. As in the previous accelerators, the swapping is done by software, the data is sent back and forth processor memory in every time step. However, we don't need to swap the whole arrays, just their pointers. And right now we are moving the whole arrays just for swapping their pointers. Therefore, in order to save the time spent by doing this, we shall do this on the reprogrammable fabric, without having to go back to the processor (from here comes the second "S" in LBM-SMS). Moreover, by doing this, we will improve the pipeline on the memory subsystem. In the previous accelerators, when the data arrived from memory first we needed to fill the BRAMs. As this cannot be done in parallel, the computing unit will be starving data until the last BRAM can read new data and push it into the FIFO. The equivalent happens when writing. However, if we are doing the swapping in hardware, we could create a better pipeline, where the elements of one time step can be loaded into the BRAMs while the values from the previous time step are still being calculated.

# Chapter 6

# Experimental Setup

## 6.1 Target Platform

The target platform in which the hardware accelerator was implemented is the Ultra96v1 development board from Avnet. It is based on a Zynq UltraScale+ MPSoC ZU3EG SoC from Xilinx [34]. It provides an heterogeneous architecture with an ARM-based side, denominated as Processing System, and a FPGA side, denominated as Programmable Logic. In order to connect all the components, ARM's AMBA interconnect is used, together with ARM's AXI protocol, a burst-oriented protocol intended for high bandwidth while providing low latency.

The processing system contains on four ARM Cortex-A53 cores for general purpose, with their own L1 cache and a shared L2 cache. As for the FPGA, this platform provides the UltraScale+ FPGA technology we explained in Sec. 2.2. Regarding the frequency, we have chosen the lowest one configured in our platform: 100MHz. While choosing higher frequencies would help achieving better performance, it would be more prone to timing issues, and we wanted to prioritize a working implementation. Table 6.1 contains the main technical specifications of the board, while Fig. 6.1 shows a simple block diagram of the platform.

As for the software side, the board was used running a Linux kernel, generated with Petalinux. Besides, the original implementation was not build on the board, but cross-compiled with the Vitis tool together with the hardware build. The Linux image is stored in a SD card from which the system boots.

## 6.2 Parameters to Analyze

When evaluating the hardware accelerator, we are going to measure the impact of the following parameters on the execution time: time steps, grid size and obstacle size.

**Table 6.1:** Ultra96v1 specifications

| Processing System | |
|---|---|
| CPU | Quad-Core Cortex-A53 |
| Frequency | 1.2GHz |
| L1d Cache | 32 kB |
| L1i Cache | 32 kB |
| L2 Cache | 1 MB |
| On-Chip Memory | 256 kB |
| **Programmable Logic** | |
| Frequency | 100MHz |
| Logic cells | 154.350 |
| Flip-flops | 141.120 |
| LUTs | 70.560 |
| Distributed RAM | 1.8 Mb |
| Block RAM (block) | 36 kb |
| Block RAM (total) | 7.6 Mb |
| DSP Slices | 360 |
| **Memory** | |
| RAM | 2GB LPDDR4<br>533 MHz |
| SD Card | 16 GB |

### 6.2.1 Time Steps

The first parameter to analyze is the number of time steps. Although it does not affect the kernel itself, it determines how many times it is executed. Thus, it will have significant effects on the execution time.

### 6.2.2 Grid Size

Other important parameter is the data size, which in this case is defined by the grid size. It will define how many iterations are done in each time step. For simplicity, we will use cubic grids (except when using the original reference grid, which is 100×100×130). Testing all possible relations between the three dimensions of the grid is not feasible and, as the implementation actually works over 1D arrays, it does not matter now the elements are actually distributed. The only difference would be upon initialization, since some shapes with the same size would have more borders, i.e. bigger perimeter with the same volume. However, this would be an issue of obstacle particles compared to the total (as explained in Sec. 3.2, border particles are initialized as obstacles), which we can better analyze by means of the obstacle file. A cubic shape is the best one, since it minimizes the effect of the border obstacles, and we can refer to it based on its size length, which makes it easier to compare to spherical obstacles measured by its radius. This is

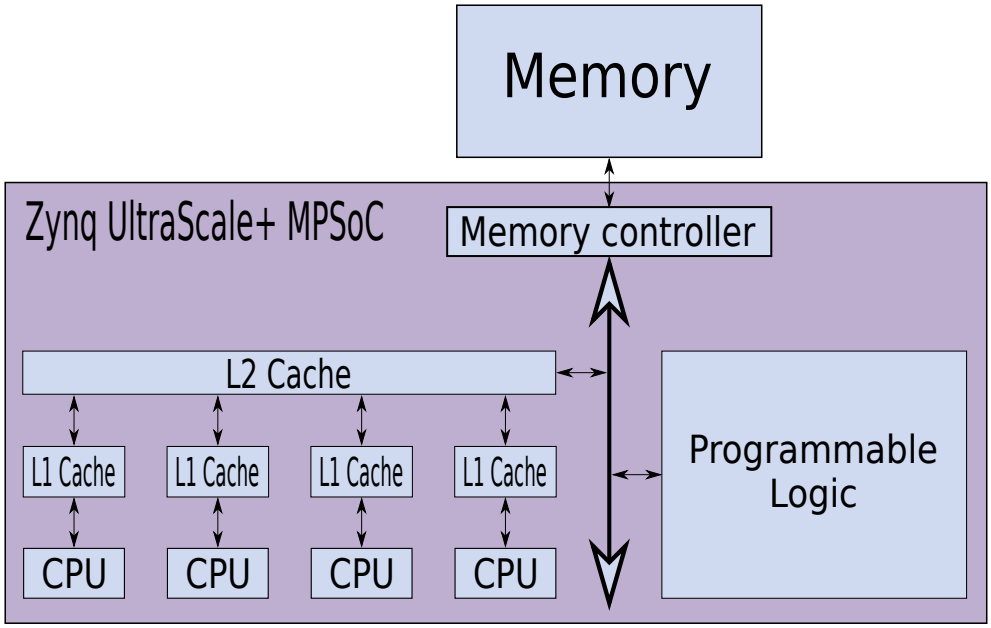



**Figure 6.1:** Target platform's block diagram

important because we can test obstacles with radius bigger than half the grid size, and thus comparing volumes would not be correct, since part of the obstacle would be out of the grid.

### 6.2.3 Obstacle Size

The last important parameter to analyze is the obstacle size. However, more than the absolute size, the key part is its relative size compared to the grid in which it is place. That is important because it defines the ratio between free and obstacle particles. Besides the original reference data set, the custom ones will be spherical obstacles placed in the middle of the grid, and we will change their radii. Moreover, we will allow these radii to be bigger than half the grid size, thus having the sphere going out of the grid. While this is not a realistic data set, it is interesting for a computational analysis, and can give some useful insight.

## 6.3 Measuring Execution Time

Until now we have mentioned that we are going to measure the execution time, but we have not defined what it is. Also known as wall-clock time, response time or elapsed time, it is the latency to complete to fully complete the task, including memory accesses, input/output activities, operating system overhead, etc. That is, including everything that causes a delay in the response [4].

### 6.3.1 Tools

We have used three different tools for trying to measure the execution time.

#### Unix *time* Command

One of the simplest ways to measure execution time in an UNIX-based operating systems is to use the *time* command. It does not only give the real (execution) time, but also user time and system time. However, this applies to the whole command line, and cannot be used for finer grained time analysis [35].

#### C++ *ctime* Library

Inherited from the C *time.h* library, this library provides its timing functionalities [36]. From it, we have used the *clock* function, which returns the processor time used by the process since the beginning of the program's execution. This time is given in clock cycles, but can be easily converted to seconds by diving it by the CLOCKS_PER_SEC macro. Moreover, as this is a C/C++ function, we can place it in the middle of the code and get partial timing results. However, this only measures processor time. Therefore, when computation is done outside it, this time will not be counted. This applies to kernels executed in the FPGA.

#### C++ *chrono* Library

Due to the previous issue with *ctime* library, the alternative is to use the *chrono* library [37]. This library allows us to access to different system clocks, and get their absolute times. Knowing this time points, we will be able to get the different duration measurements we are interested in. As it works with the absolute time points of system clocks, this method will provide the actual execution time, independent of some computation being offloaded to the hardware fabric.

### 6.3.2 Measuring the Execution Time of LBM

Regarding how execution time was measured, we have used the three options we just presented. While the *time* command measures the whole program, we used the *ctime* and *chrono* libraries to measure only the main loop in which the kernel is executed. With this setup, we were able to find out several observations.

In first place, we found out that the *ctime* library did not work as expected. It did not report the same as the *chrono* library or the real time from the *time* command. However, what it reported was equivalent to the addition of the system and user time reported by said command. This means that for some time, which is the difference between the value measured by the *chrono* library and the one measured by *ctime*, the program is not being executed in the processor. This shows that our program has been successfully offloaded.

Also, while the values measured by *chrono* and *time*, they are not exactly the same. That difference is due to the initialization. In this case, initialization means

not only initializing the arrays with data, but also preparing the system to the offloading. Detecting and setting up the accelerator is constant for any grid size, while initializing the data depends on its size. However, this time is quite small, in most cases smaller than 1% (exceptions are for small grids, where it can go slightly higher). Compared to the total time, this time in negligible, and thus it will not be considered.

With this in mind, we would have to choose between *chrono* and *time*. The former has the advantage that we can measure specific parts of the code, providing more flexibility. It is even possible to measure both the original implementation and the offloaded one using the same binary file. Therefore, we will use as execution time the one measured by the *chrono* library.

Finally, it is important to notice that the measured times are not fully consistent, with differences of up to 5% with regards to the average value for the same test. This does not happen for grids with size equal or smaller than 20. For grids with size between 30 and 50 this starts happening, but it is specially seen in the first execution after start up. For greater grids, this no longer matters. As the absolute difference increases with the grid size, interrupts are not enough to explain this. Also, we know this happens in the processor and not in the FPGA, as this difference is also observed in the time measured using *ctime*. With this information, we think it is a memory-related issue. For each time step, data is copied (we will explain this more in detail in Ch. 7), and the difference is likely to be here. Moreover, as we were dealing with some problems deallocating memory, the cases were the first test is the fastest can be due to the data being allocated in favourable memory positions for that initial test, while that position is no longer available for the next tests.

While appropriate would be to do intensive testing to get maximums and minimums, as well an accurate standard deviation, this would have taken too much time, which we could not afford. With a few tests per case we can still get results relevant enough seeing the main trend, as the difference between test cases was big enough. Thus, we decided to measure 5 executions for grids up to size 60 and 3 for bigger ones.

## 6.4 Other considerations

In order to simplify the evaluation process, our implementation has a key different compared to the original one: while on the SPEC implementation the grid size was fixed in a config file, in ours it is given through the terminal. This is different because in the former case it was needed to recompile the program in order to change it. However, this approach would not be feasible for the hardware accelerator. While it is not an ASIC, but a FPGA, so it is reprogrammable, building the project consumes a considerable amount of time, and the board would have to be rebooted with a new boot file and kernel image. Changing the grid size is actually just changing the number of iterations inside the kernel, and this can be done by changing a value in a register, which will be compared to the counter. However,

when the tool detects any change in the kernel, it builds it again and, although does not need to be a clean build, it still takes a considerable amount of time. Therefore, we have decided to implement generic version which can work with different grid sizes, being given its $x$, $y$ and $z$ coordinates through the command line. Besides, during the initial tests we could compare both approaches, a fixed accelerator against a flexible one, and both resource consumption and execution time were equivalent, so this gain in flexibility came at no cost.

# Chapter 7

# Results

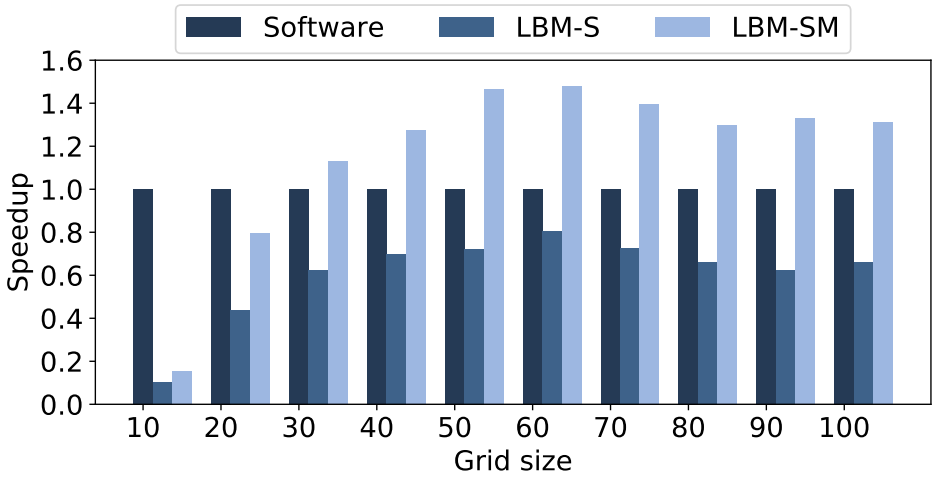
Finally, we have implemented the hardware accelerator in the target platform. As explained in Ch. 5, we have conceived three different accelerators: LBM-S, which implements streaming as a way to feed data to the compute kernel, LBM-SM, which adds a custom memory system, and LBM-SMS, which moves the swapping from the processor to the reprogrammable fabric. We have implement the two first proposals, while we did not have time to successfully implement the last one, facing issues with unfulfilled timing constraints.

In this chapter, first we will present the high-level results, comparing our implementations to the original one. Then, we will go into detail with the individual implementations, analyzing what is happening and how we could improve this.

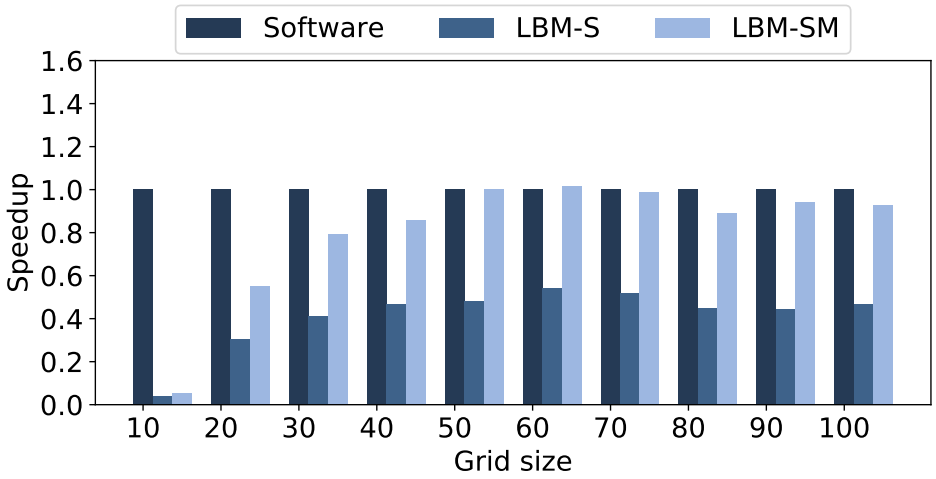
## 7.1 Accelerator Performance

Having implemented two accelerators, we can compare them to the original software implementation at the same time. In Fig. 7.1a we can see how, even in the best case scenario, LBM-S does never achieve its goal, while LBM-SM does so for grid sizes from 30 over, where the software implementation can no longer take advantage of the cache. Moreover, the latter accelerator provides almost double the performance of the former one. In Fig. 7.1b we can see how this relationship stays, and how LBM-SM barely decelerates in the worst case scenario of the grid being full. In Fig. 7.1c we can see how the transition from an empty grid to a full one affects the performance. As we can see in these three figures, the new custom memory subsystem has allowed LBM-SM to achieve a performance two times better than LBM-S.

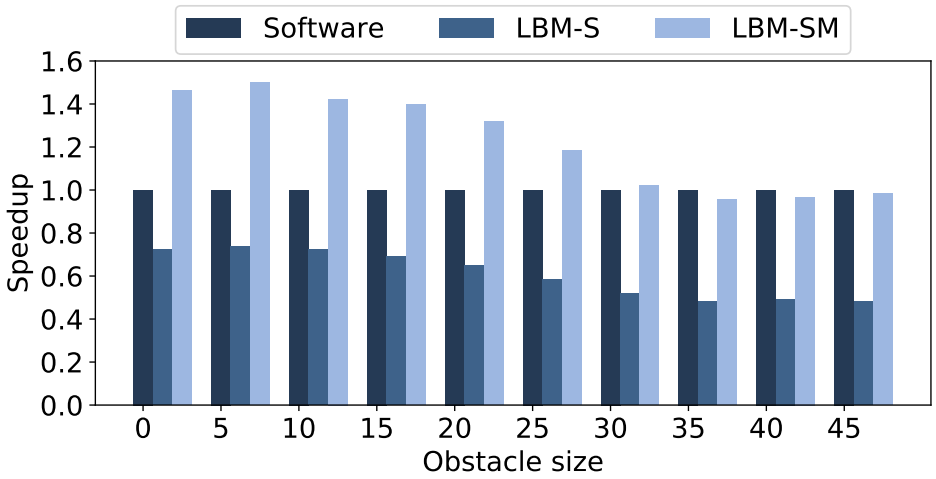
However, this does not mean there is still no room to improve in term of memory bandwidth. Fig. 7.2 shows the bandwidth utilization for the two accelerators. Here, only the kernel execution has been considered, not taking into account reading and writing data between memory regions for swapping in the processor. This way, we can analyze the bandwidth utilization comparing it to the maximum available for the reprogrammable fabric. As we can see, while LBM-SM has more




**(a)** Empty grid




**(b)** Full grid




**(c)** Obstacle size sweep

**Figure 7.1:** Speedup results (relative to software implementation)

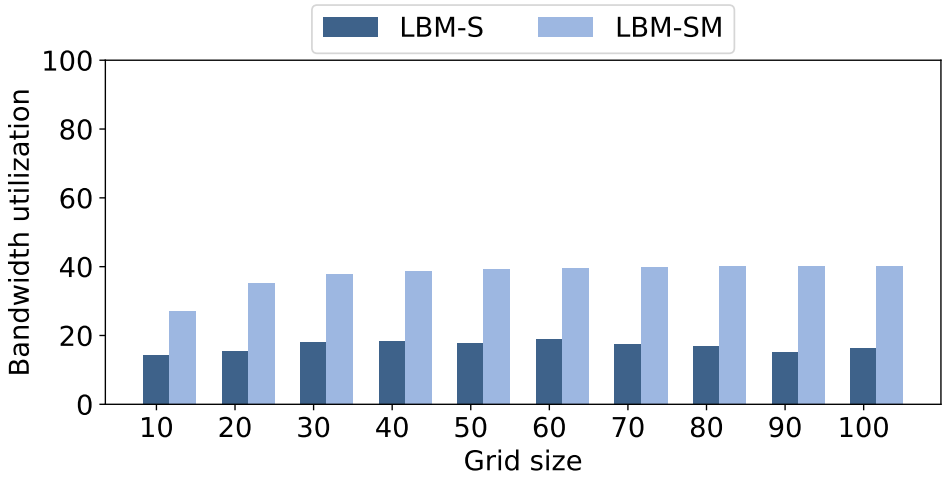



**Figure 7.2:** Bandwidth analysis with FPGA bandwidth as reference

than doubled the bandwidth utilization of the first one, it still has margin to improve. In order to improve this we do not need to apply conceptual modifications yet: there is still margin in the technical side. For example, we could increase the port width of the AXI memory module. We tried to implement it, but faced resource and timing constraints which did not allow us to do so. We considered changing to an equivalent platform with more resources, so that we could apply all that we have learned while avoiding resource limitations. However, due to the restrictions enforced by the COVID-19 pandemic, we could not have access to said platform.

Beside, this applies when comparing only to the maximum bandwidth utilization we have achieved with the FPGA. We have tested the maximum bandwidth on the processor side, and it is considerable higher. We can see the same results, but this time taking the processor bandwidth as a reference, in Fig. 7.3.

This difference can mean two things: either we are not being able to use the maximum bandwidth even in our reference tests, or the FPGA cannot make full use of the memory bandwidth (or both). What we found out is that achieving the maximum bandwidth possible in the FPGA is far from trivial, specially with HLS, as there are many elements and configurations which take part in this. Here, one of the parameters which may be causing this difference is the frequency, as the one of the reprogrammable fabric is $1/12$ th of the processor's.

## 7.2 LBM-S

For an initial implementation, we wanted to have one that would be able to run all the needed tests successfully, but without going far into optimization. We wanted to have an initial reference point, as well as understanding of how the tool worked, and were we needed to focus our efforts.

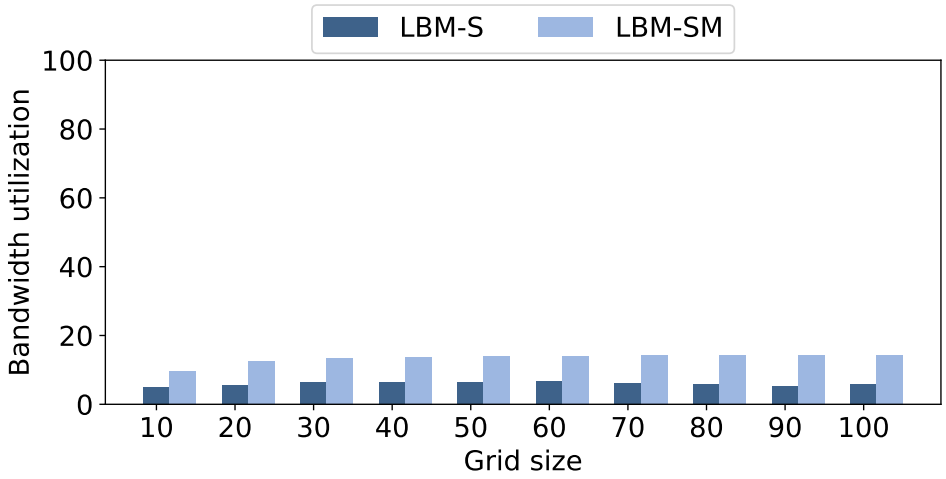



**Figure 7.3:** Bandwidth analysis with CPU bandwidth as reference

**Table 7.1:** Resource consumption

| Resource type | LBM-S (%) | LBM-SM (%) | Abs. difference | Rel. difference |
|---|---|---|---|---|
| CLB | 99.04 | 97.29 | -1.75 | -1.77 |
| LUT | 65.99 | 73.34 | 7.35 | 11.14 |
| As Logic | 59.32 | 66.18 | 6.86 | 11.56 |
| As Memory | 16.34 | 17.54 | 1.2 | 7.34 |
| Registers | 39.96 | 41.47 | 1.51 | 3.78 |
| BRAM | 0.93 | 18.98 | 18.05 | 1940.86 |
| DSP | 31.11 | 31.11 | 0 | 0 |

### 7.2.1 Resource Consumption

First point to analyze is the resource consumption. When we first tried to implement the kernel, it turned out to be too big for the board. Specially critical was the usage of DSPs slices. As discussed in Sec. 4.2.1, we first tried changing the data type, but the increase in LUT utilization was to high, so we moved to using *allocation* pragma. With it, we managed to get the resource consumption seen in Tab. 7.1.

As we can see, the CLB utilization is almost at 100%. However, this does not mean that they are being fully utilized. For example, LUT utilization is only 66%. A CLB has several LUTs. What we are seeing in this table means that, while almost all of the CLBs are being used, not all of their LUTs are. While we could improve even more the resource consumption, doing so was starting to lead to timing errors. Dealing with them was not a priority at the time, as we had a working implementation, so we decided to remain with the current values and start testing.

Moreover, this resource limitation will not allow us to make better use of data-

level parallelism. While the intra-cell data dependencies are quite strict, and the inter-cell data dependencies between different iterations are too, there are no data dependencies between cells in the same iteration. This means that it would be possible to compute the collide operation for the same time step of two different cells at the same time in parallel. However, as we do not have enough resources, we will not be able to benefit from this.

Another thing to observe is that the BRAM utilization is almost 0. This, mixed with other observations we have done while developing, indicates that the FIFO buffers are not being implemented in BRAM, but in the memory inside the CLBs.

### 7.2.2 Error

Once we implemented the LBM-S accelerator, we evaluated it. In first place, we checked its correctness. For this, we have compared the obtained results with the ones produced by the original implementation. For both the reference data set, as well as for empty arrays, errors were in the order of $10^{-13}$%. We have considered this a satisfactory result. In the other hand, for full grids, errors rose up to more than 1%, despite using the same data type and not having altered the computational part of the code. However, full grids are not a realistic data set and, while they are helpful for understanding how the implementation works, they will not be used in real cases. Therefore, as the error values for realistic cases are satisfactory, we consider no issues in this regards.

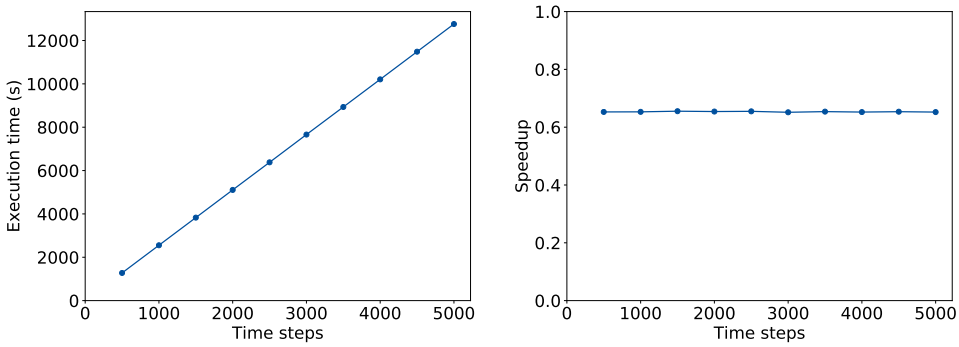
### 7.2.3 Sensitivity Analysis

Once we verified its correctness, we used the original data set to verify the effect of changing the number of time steps. As in the original implementation, execution time changes linearly with the number of time steps, as we can see in Fig. 7.4. Thus speedup is constant. However, we can see that this speedup is around $x0.65$, thus being not an accelerator, but a decelerator.

Next we tested how changing the grid size would affect the execution time. As we saw in Sec. 3.3, it depended not only on the grid size, but also on the percentage of obstacle particles. Thus, for each size, we did two types of tests: with the grid being fully empty and with the grid being completely full. We can see the results of these tests in Figs. 7.5 and 7.6 respectively. As in the original implementation, execution time increases cubically with the grid side size, which is the same as a lineal relationship with the volume.

Regarding the speedup, we can see that it is specially lower for small grid sizes. This is to be expected as for those sizes the cache utilization in the software version is much better. Then it increases until reaching a peak at size 60, to then slightly lower down and remain at its final value.

However, while the trends are equivalent for the empty and full grids, the speedup values are not the same. Contrary to the original implementation, in ours the execution time for both test cases is the same. That is, the hardware takes the




**(a)** Execution time **(b)** Speedup

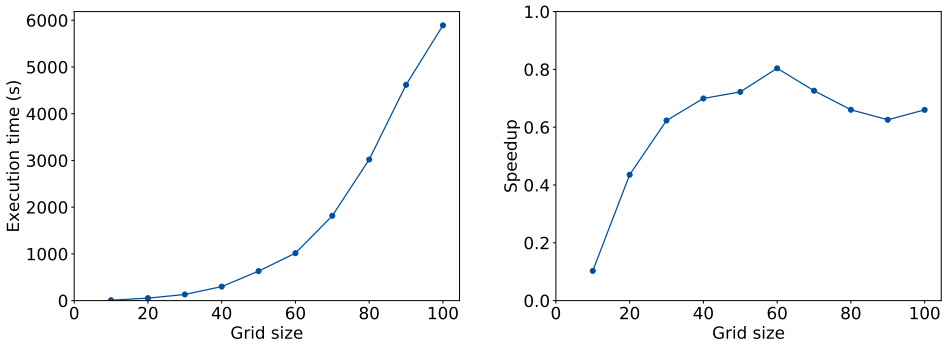
**Figure 7.4:** Execution time vs. number of time steps for LBM-S (speedup relative to processor execution time)

same time to compute the whole collide step than to just redirect the values. This can be explained by the hardware avoiding synchronization issues.

Let us consider the case where one particle is an empty one and the next is an obstacle. Being this a pipelined implementation, if the latter were sent directly to the output of the compute unit, it would reach the FIFO buffer sooner than the empty one. This would break the sequential order, and we would have to handle it. However, if we just enforce a sequential output at the compute unit, this would not happen. This approach is much easier than dealing with non-sequentiality in the FIFO. Thus, free and obstacle particles shall take the same time in the compute unit. We have checked this in the HLS scheduling, and this proved to be true. For the case where the particle is an obstacle one, the inputs are directly sent to the outputs, but this is not scheduled to happen until after all the computations for the collide step would have taken place.

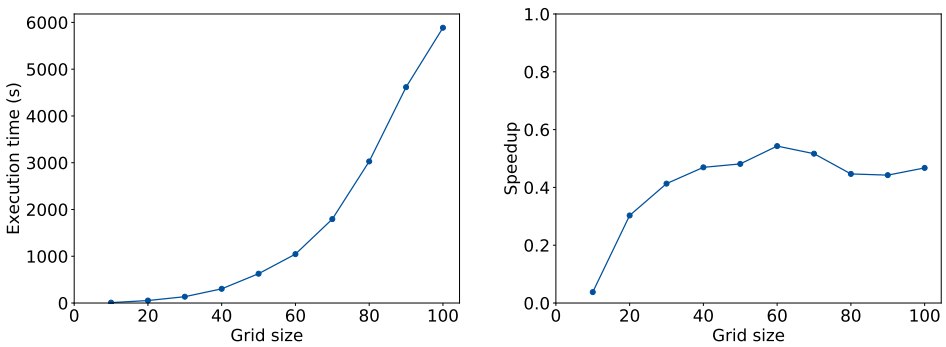
Therefore, while the execution time for both test cases is the same, the speedup is not, being lower for full grids. This means that, the emptier a grid is, the better the accelerator will work (or, for the time being, the less it will decelerate the application).

To have more insight about this, we have evaluated how the execution time is affected by changes in the obstacle size for a fixed grid size. For this, we have chosen 50 as grid size. This value is big enough that the software implementation does not have extra benefits due to the cache, but also small enough so that we can do the tests in a reasonable amount of time. We have started the tests without any obstacle, and then increasing the obstacle size, an sphere, up to a radius of 45 units, point at which the grid is full. As we can see in Fig. 7.7, while the execution time is constant, the speedup decreases as the obstacle size increases (because the processor time becomes smaller).




**(a)** Execution time **(b)** Speedup

**Figure 7.5:** Execution time vs array size (with empty grid) for LBM-S (speedup relative to processor execution time)




**(a)** Execution time **(b)** Speedup

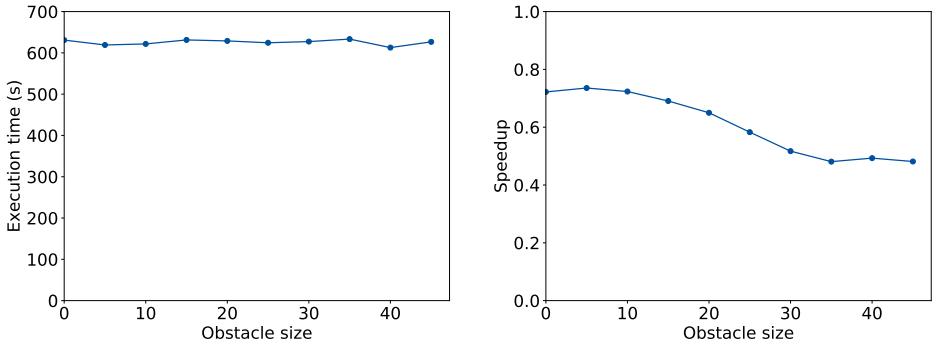
**Figure 7.6:** Execution time vs array size (with full grid) for LBM-S (speedup relative to processor execution time)

### 7.2.4 Breakdown

Knowing this data, we want to know exactly where time is going. For that, we have repeated the tests doing a more fine grained time analysis using the *chrono* library. For each time step, our implementation does four tasks:

1. Migrate memory objects to the FPGA.
2. Execute the kernel.
3. Migrate memory objects from the FPGA.
4. Swap the grids.

We have measured this four tasks, for each iteration and averaged them. We did the tests with 3000 time steps, and standard deviations were in an acceptable range. We show the obtained results in Fig. 7.8. As we can see, most of the time corresponds to kernel execution, while a smaller part corresponds to memory migrations. Swapping time itself is negligible.




**(a)** Execution time **(b)** Speedup

**Figure 7.7:** Execution time vs obstacle size (array size = 50) for LBM-S (speedup relative to processor execution time)

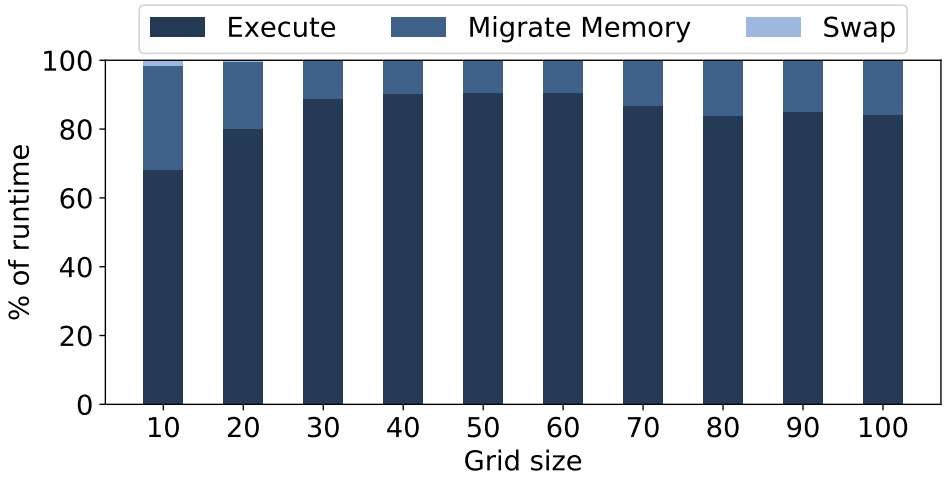



**Figure 7.8:** Fine-grained time analysis for LBM-S

Comparing this results with what we can see in Figs. 7.5b and 7.6b, we can see that the speedup peak at a size of 60 mentioned before corresponds to a lesser memory migration time. While this is copying data, and should be lineal, it did not behave exactly as that.

However, even if we solved this memory migration issue, it would not be accelerating. Therefore, we need to analyze why the kernel is not being executed faster. Regarding the compute, although the frequency at which the FPGA is working is just $1/12$ th of the processor frequency, parallel computation should be able to counter this. Moreover, as the original problem was heavily memory bound, memory is quite likely to be main issue in our implementation. Looking more in detail into what is going at the memory level, we saw that the memory accesses are being done individually: more than reading from memory, we are pushing individual values into the FIFOs directly from memory. This is far from being an

optimal memory utilization, specially because it is not taking advantage of the elements that correspond to the same FIFO being placed in order, not doing burst reads. Therefore, the current reading pattern is a bad one.

## 7.3 LBM-SM

After analyzing the results from the LBM-S accelerator, we have proposed improvements in Sec. 5.4.2, and in this section we will analyze their results.

### 7.3.1 Resource Consumption

First we need to analyze the new resource consumption. Looking at the data in Tab. 7.1, we can see that, besides the BRAMs, there has not been any major changes in the resource consumption. This means that the BRAMs have successfully been implemented in the design. The consumption of LUTs has also increased, specially for their usage as logic. This was expected, as extra logic is needed to handle the BRAMs.

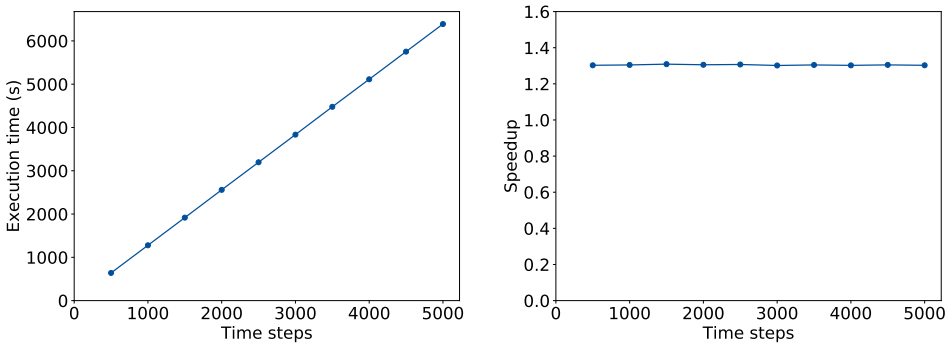
Also it is interesting to note that, while we have not removed anything, just added, the usage of CLBs have decreased. This means that the tool has been able to map more LUTs into less CLBs, while still respecting timing constraints.

### 7.3.2 Sensitivity Analysis

Now that we have successfully implemented this second iteration of the accelerator, we can move on to testing it. The same tests used for the initial implementation were used here. As we can see in Figs. 7.9, 7.10, 7.11 and 7.12, the trends observed in the original accelerator remained, while the execution times lowered (thus, achieving a higher speedup). Therefore, the analysis we have done of them for LBM-S still holds. Moreover, we can see that creating a custom memory subsystem for the accelerator has given us the results we expected: now, for realistic data sets (except the smallest grids, where the processor still takes advantage of the cache), the accelerator is actually accelerating.

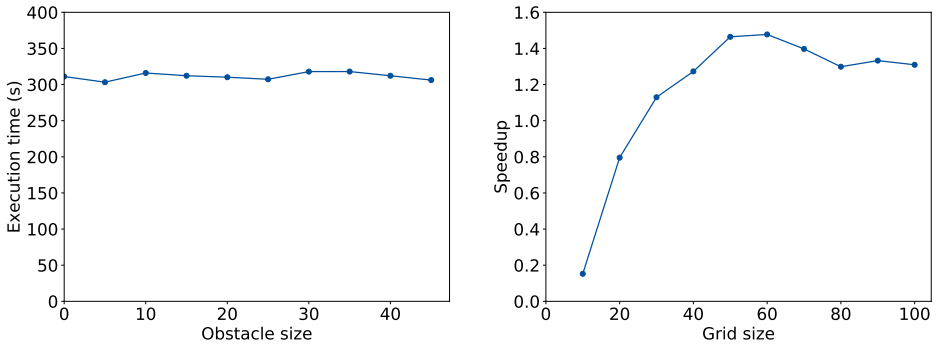
### 7.3.3 Breakdown

Moreover, in Fig. 7.13 we can see how the time distribution between the four different parts of each iteration has changed. While the swapping remains constant, the contribution made by the kernel execution has been significantly reduced. In this implementation, it represents between 70 and 80% of the total time, while moving the data from and to memory now gets to represent even more than 30%. This means that the more we optimize the kernel, the more the moving data between memory regions weight. Thus, the more optimized the kernel is, the more important is to avoid this, as we have proposed in Sec. 5.4.3.




**(a)** Execution time **(b)** Speedup

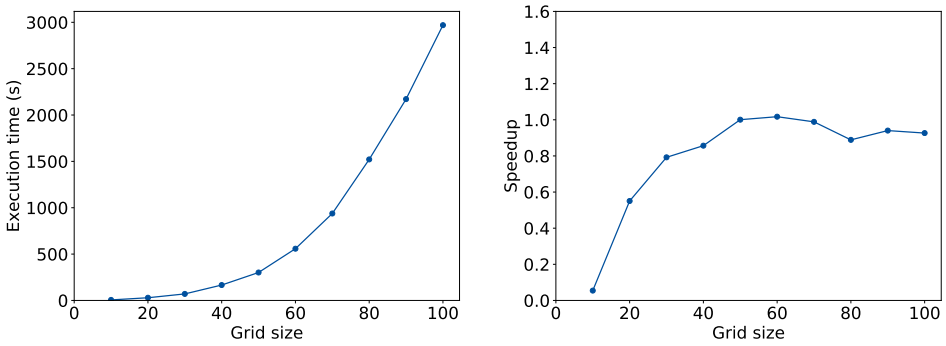
**Figure 7.9:** Execution time vs number of time steps for LBM-SM (speedup relative to processor execution time)




**(a)** Execution time **(b)** Speedup

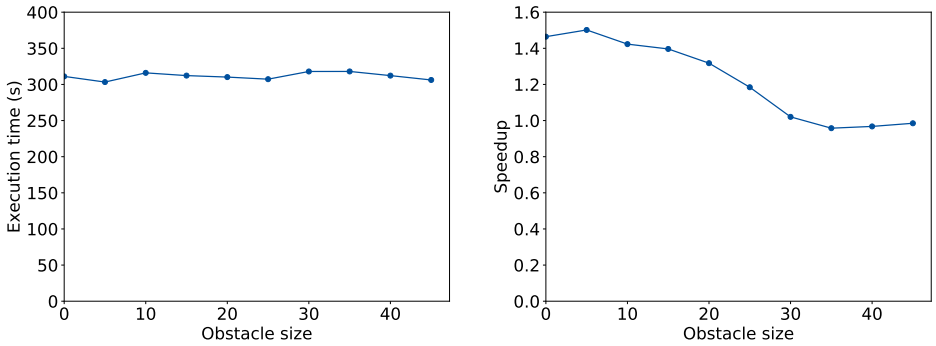
**Figure 7.10:** Execution time vs array size (with empty grid) for LBM-SM (speedup relative to processor execution time)

However, this does not imply that the execute stage cannot be improved. For example, we could increase port width of the AXI memory modules. By default they take the width of the data type they are working with (in this case, 64 bits for being double). However, it is possible to increase this width in order to read faster. The maximum port width available is 512. While we have done some testing in this direction, and it has actually improved the bandwidth utilization, we have not been able to add it to our accelerator due to problems with timing constraints and lack of time. These same reasons are also why we cannot present the implementation results for LBM-SMS. Going back two Fig. 7.13 we can see that we can expect a reduction of 20 to 30% of the total execution time over our second accelerator, that is, without even taking into account possible improvements in the memory system.




**(a)** Execution time **(b)** Speedup

**Figure 7.11:** Execution time vs array size (with full grid) for LBM-SM (speedup relative to processor execution time)




**(a)** Execution time **(b)** Speedup

**Figure 7.12:** Execution time vs obstacle size (array size = 50) for LBM-S (speedup relative to processor execution time)

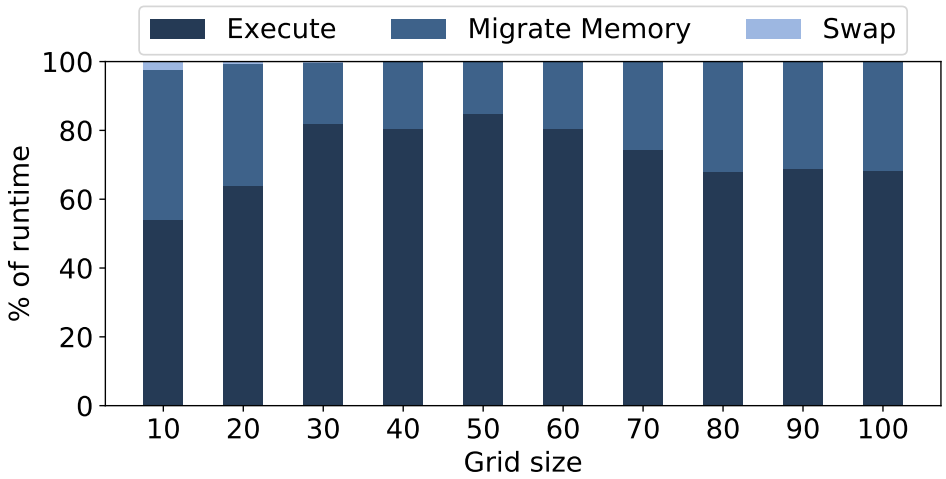



**Figure 7.13:** Fine-grained time analysis for LBM-SM

# Chapter 8

# Conclusion and Future Work

## 8.1 Conclusion

In this work we have proposed and implemented a hardware accelerator for LBM, one of the benchmarks in the SPEC CPU benchmark suit.

In Ch. 5 we have initially proposed a general approach to accelerating this application. Then we have gone into more specific implementation details, and proposed three accelerators: LBM-S, which implements streaming as a way to feed data to the compute kernel, LBM-SM, which adds a custom memory system, and LBM-SMS, which moves the swapping from the processor to the reprogrammable fabric. As the names suggest, LBM-S was the initial one, and the other two add improvements to it. Out of these three, we have implemented the two first, thus addressing task T1.

In Ch. 7 we have addressed task T2, discussing the results for the actual implementations, evaluating their usage of the resources of the target architecture, and finding their bottlenecks. For realistic data sets big enough to not fit in processor cache, we have achieved speedups between 1.3× and 1.5×. Moreover, there is still room to improvement while keeping the proposed acceleration strategy. Due to lack of time and the inability to have access to a bigger platform (yet with the same architecture) due to the COVID-19 pandemic, we have left on the table improvements for LBM-SM and the implementation of LBM-SMS. Despite being margin for improvement, we could have achieved some insightful conclusions. The main one was the importance of implementing memory systems that take advantage of the characteristics of the data. A good one will allow us to take advantage of the data locality, and achieve more efficient read and write patterns. Moreover, decoupled access/execute architectures have yet again shown to be good at hiding latencies of memory accesses.

Moreover, all of this has not been done using traditional hardware description languages such as Verilog or VHDL for designing the RTL, but with the use of HLS, which we have analyzed and evaluated in Ch. 4, answering task T3. This tool has reduced the complexity of designing the hardware, and also dealt with intricate details such as handling the communication with memory. It reaches the level of

abstraction bringing this process to a high-level domain. However, it is not free of disadvantages: it is not trivial to make the tool implement what we want the way we want it, there are challenges for debugging and, as we have raised the abstraction level, it is not possible to express everything we would have been able to express using lower-level languages.

## 8.2 Future Work

The first step to do after this work would be to polish the LBM-SM accelerator, as explained at the end of Sec. 7.3, and to implement LBM-SMS based on this. Other possible improvements would be statically storing the flags in the BRAMs, as they do not change over time steps. After this, we should analyze more in depth the memory subsystem. Among the first tests to do, we should test how changes in FIFO and BRAM sizes affect our accelerator. While we have done some brief checks of it while developing, we have not appreciated any relevant changes and, as it was not our priority, we have postponed it.

On the platform side, we should analyze why we are not able to achieve the same bandwidth in the FPGA as in the processor system (not even with tests designed to get the highest memory bandwidth possible). Also, we should analyze the possibility of using other platforms. Without leaving aside tightly-coupled FPGAs, other platforms can bring to the table architectures better suited for our application, or even just more raw power by providing more resources in the reprogrammable logic. For example, Xilinx provides systems-on-chip that come with high-bandwidth memory, from which our application could greatly benefit.

# Bibliography


[1] G. E. Moore *et al.*, *Cramming more components onto integrated circuits*, McGraw-Hill New York, NY, USA: 1965.

[2] G. E. Moore *et al.*, 'Progress in digital integrated electronics,' in *Electron devices meeting*, vol. 21, 1975, pp. 11–13.

[3] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous and A. R. LeBlanc, 'Design of ion-implanted mosfet's with very small physical dimensions,' *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[4] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, 6th. Elsevier, Dec. 2017.

[5] J. L. Hennessy and D. A. Patterson, 'A new golden age for computer architecture,' *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[6] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, 'Dark silicon and the end of multicore scaling,' in *2011 38th Annual international symposium on computer architecture (ISCA)*, IEEE, 2011, pp. 365–376.

[7] G. M. Amdahl, 'Validity of the single processor approach to achieving large scale computing capabilities,' in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[8] J. L. Gustafson, 'Reevaluating Amdahl's Law,' *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[9] M. B. Taylor, 'Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse,' in *DAC Design Automation Conference 2012*, IEEE, 2012, pp. 1131–1136.

[10] V. Sze, Y.-H. Chen, T.-J. Yang and J. S. Emer, 'Efficient processing of deep neural networks: A tutorial and survey,' *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[11] M. Koraei, O. Fatemi and M. Jahre, 'DCMI: A scalable strategy for accelerating iterative stencil loops on FPGAs,' *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–24, 2019.

[12] M. Vázquez Maceiras, *Accelerating LBM in a reconfigurable high-performance processor*, 2020.

[13] S. Borkar and A. A. Chien, 'The future of microprocessors,' *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[14] T. Nowatzki, V. Gangadhan, K. Sankaralingam and G. Wright, 'Pushing the limits of accelerator efficiency while retaining programmability,' in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2016, pp. 27–39.

[15] T. Nowatzki, V. Gangadhar, K. Sankaralingam and G. Wright, 'Domain specialization is generally unnecessary for accelerators,' *IEEE Micro*, vol. 37, no. 3, pp. 40–50, 2017.

[16] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm and A. Shriraman, 'Needle: Leveraging program analysis to analyze and extract accelerators from whole programs,' in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2017, pp. 565–576.

[17] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah and T. Nowatzki, 'DSAGEN: Synthesizing programmable spatial accelerators,' in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 268–281.

[18] Q. Huang, C. Yarp, S. Karandikar, N. Pemberton, B. Brock, L. Ma, G. Dai, R. Quitt, K. Asanovic and J. Wawrzynek, 'Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator socs using FPGA-acceleration,' in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2019, pp. 1–8.

[19] R. Jayaraman, 'Physical design for FPGAs,' in *Proceedings of the 2001 international symposium on Physical design*, 2001, pp. 214–221.

[20] M. Khazraee, L. Zhang, L. Vega and M. B. Taylor, 'Moonwalk: NRE optimization in ASIC clouds,' *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 511–526, 2017.

[21] *Standard performance evaluation corporation (spec)*. [Online]. Available: `http://www.spec.org/`.

[22] K. Sano, O. Pell, W. Luk and S. Yamamoto, 'FPGA-based streaming computation for lattice boltzmann method,' in *2007 International Conference on Field-Programmable Technology*, IEEE, 2007, pp. 233–236.

[23] K. Sano, 'FPGA-based scalable custom computing accelerator for computational fluid dynamics based on Lattice Boltzmann Method,' in *Sustained Simulation Performance 2014*, Springer, 2014, pp. 187–201.

[24] X. Ren, Y. Tang, G. Wang, T. Tang and X. Fang, 'Optimization and implementation of LBM benchmark on multithreaded GPU,' in *2010 International Conference on Data Storage and Data Engineering*, IEEE, 2010, pp. 116–122.

[25] M. Griebel, T. Dornseifer and T. Neunhoeffer, *Numerical simulation in fluid dynamics: a practical introduction*. SIAM, 1998.

[26] *519.lbm_r spec cpu 2017 benchmark description*. [Online]. Available: `https://www.spec.org/cpu2017/Docs/benchmarks/519.lbm_r.html`.

[27] D. A. Wolf-Gladrow, *Lattice-gas cellular automata and lattice Boltzmann models: an introduction*. Springer, 2004.

[28] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger and U. Rüde, 'Optimization and profiling of the cache performance of parallel lattice boltzmann codes,' *Parallel Processing Letters*, vol. 13, no. 04, pp. 549–560, 2003.

[29] *Vitis High-Level Synthesis User Guide*. [Online]. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1399-vitis-hls.pdf`.

[30] S. Williams, A. Waterman and D. Patterson, 'Roofline: An insightful visual performance model for multicore architectures,' *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[31] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto and M. D. Santambrogio, 'On how to accelerate iterative stencil loops: A scalable streaming-based approach,' *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–26, 2015.

[32] V. Rana, I. Beretta, F. Bruschi, A. A. Nacci, D. Atienza and D. Sciuto, 'Efficient hardware design of iterative stencil loops,' *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2018–2031, 2016.

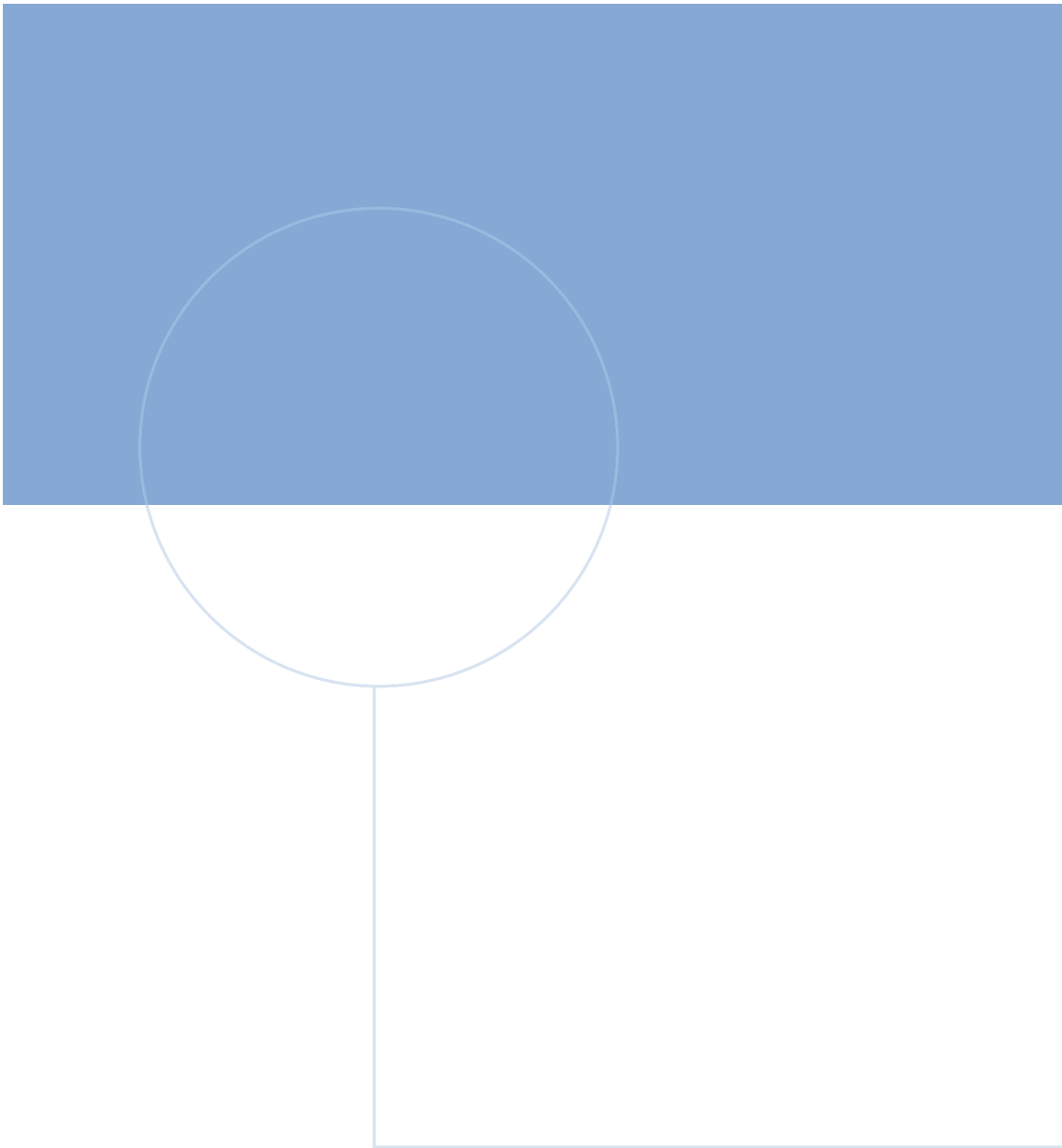
[33] J. E. Smith, 'Decoupled access/execute computer architectures,' *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 289–308, 1984.

[34] *Zynq UltraScale+ MPSoC data sheet*, Xilinx. [Online]. Available: `https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf`.

[35] *time documentation*, Issue 7, The Open Group, 2018. [Online]. Available: `https://pubs.opengroup.org/onlinepubs/9699919799/utilities/time.html`.

[36] *ctime reference*. [Online]. Available: `https://en.cppreference.com/w/cpp/header/ctime`.

[37] *chrono reference*. [Online]. Available: `https://en.cppreference.com/w/cpp/header/chrono`.



NTNU

Kunnskap for en bedre verden