Zynq SoC based Acceleration of the Lattice Boltzmann Method

Xiaojun Zhai[1,2*], Abbes Amira[2], Faycal Bensaali[3], AlMaha Al-Shibani[2], Asma Al-Nassr[2], Asmaa El-Sayed[2], Mohammad Eslami[2], Sarada Prasad Dakua[4], Julien Abinahed[4]

[1]School of Computer Science and Electronic Engineering, University of Essex, UK
[2]Department of Computer Science and Engineering, Qatar University, Qatar
[3]Department of Electrical Engineering, Qatar University, Qatar
[4]Department of Surgery, Hamad General Hospital, Qatar

[*]xzhai@essex.ac.uk

Abstract: Cerebral aneurysm is a life-threatening condition. It is a weakness in a blood vessel that may enlarge and bleed into the surrounding area. In order to understand the surrounding environmental conditions during the interventions or surgical procedures, a simulation of blood flow in cerebral arteries is needed. One of the effective simulation approaches is to use the Lattice Boltzmann (LB) method. Due to the computational complexity of the algorithm, the simulation is usually performed on high performance computers. In this paper, efficient hardware architectures of the LB method on a Zynq system-on-chip (SoC) are designed and implemented. The proposed architectures have first been simulated in Vivado HLS environment and later implemented on a ZedBoard using the software-defined SoC (SDSoC) development environment. In addition, a set of evaluations of different hardware architectures of the LB implementation are discussed in this paper. The experimental results show that the proposed implementation is able to accelerate the processing speed by a factor of 52 compared to a dual-core ARM processor-based software implementation.

Keywords: Computational Fluid Dynamic; Zynq; Lattice Boltzmann; Cerebral Aneurysm

1. Introduction

Six million people in the United States have an unruptured brain aneurysm and about 30,000 people suffer a brain aneurysm rupture [1]. Among each fifty people, there is one patient with unruptured brain aneurysm and there is a brain aneurysm rupturing every 18 minutes. Statistics demonstrate as well, that about 15% of patients with aneurysmal subarachnoid hemorrhage (SAH) die before arriving to the hospital and most of the deaths are due to rapid and massive brain injury from an initial bleeding [2]. A major solution to cerebral aneurysm is the clipping surgery where a clip is placed across the neck of the aneurysm preventing blood from leaking (as shown in Figure 1). However, in order to reduce the risk of having inaccurate blood flow measurements and efficiency of the clipping surgery, one of the solutions is

to apply fluid dynamics to provide hemodynamic estimates for the simulation of blood flow in cerebral arteries. The impacts are severe since it targets one of the most critical organs in the human body, which is the brain and any minor mistake could lead to fatal problems. Therefore, the required measurements (i.e. velocity and blood pressure) for clipping surgery can be generated to provide the surgeons with the desired support to apply the clipping surgery.

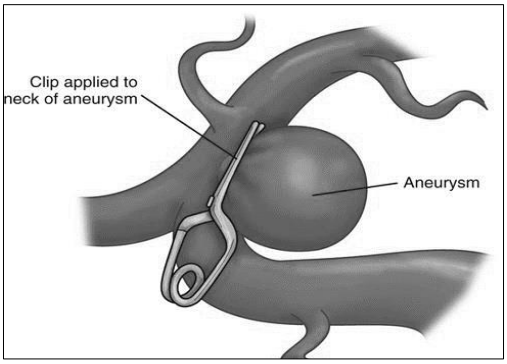

Figure. 1. Surgical clipping of an aneurysm[1].

Sun et al. present an analysis of the interactions of red and white blood cells into post-capillary venues by using a Lattice Boltzmann (LB) approach is presented in their work[3]. A computational simulation for the separation of Red Blood Cells (RBCs) is suggested by Zai et al. [4] where the LB method is used to solve the Navier-Stokes equations. He et al. [5] used LB for flow simulation in cerebral vasculature geometry along with the level-set method for medical imaging processing. Similarly, LB has been quite useful to simulate the blood flow in aneurysms. For example, the LB method is used to simulate the non-Newtonian blood flows in deformable vessels [6] and model the blood flow and clotting in intracranial aneurysms with different sizes[7]. The simulation of blood flow in cerebral aneurysms by Bernsdorf et al. [8] used optimized LB solver to capture non-Newtonian rheology. The work described in [9] focuses on applying the inclusion of critical components (e.g. RBCs, the corrugated wall and the glycocalyx) within a single unified computational framework to allow them to reproduce blood rheology in complex flows and geometrical conditions. The results achieved in this work shows that LB algorithm can provide a reliable and robust estimation for studying biological fluids at different scales. Similarly, a computational hemodynamics application called HemoSolve that uses LB for the simulation of blood flow in human vascular system is proposed in [10].

Since the LB algorithm involves massive computationally intensive operations, parallelization has been widely used to improve the simulation speed of LB method. Although this means it needs more memory storage to parallelize its iterations as well as other computing resources, the straightforward formulation of LB algorithm should naturally be adapted to various software and hardware architectures so that it could be accelerated in high performance computing (HPC) systems with various structures to exploit efficient parallelization of the simulations, even when slow interconnection network is available

[11,12]. For this reason, several frameworks and packages are available such as MUPHY [13], LUDWIG [14], HARVEY [15,16], HemeLB [7], Musubi [17], LB3D [18] and OpenLB [19]. MUPHY is a multi-physics/scale code based upon the combination of microscopic molecular dynamics (MD) with a hydro-kinetic LB method which evaluated on IBM BlueGene supercomputer [13]. Randles et al.[16] designed a framework based on HARVEY to enable scalable simulations of large, high-resolution arterial geometries. In fact, the main aim behind developing HARVEY is a parallel LB system in order to study cardio vascular diseases based on the vessel geometry derived from the segmentation of MRA data. The HARVEY code successfully addresses key challenges of image-based hemodynamics on supercomputers, such as limited memory capacity and bandwidth, flexible load balancing, and scalability. In summary, most of previous HPC implementations mainly provided solutions to exploit cache-based approach to reuse spatial data for accelerating the LBM computing [20-22], however, those solutions are not optimized for low-cost and particularly memory limited embedded platforms.

HemeLB (Hemodynamic Lattice Boltzmann) [9,23,24] is a superior parallel LB library for large-scale fluid stream in complex geometries. The core HemeLB code comprises of a parallelized LB application upgraded for sparse geometries, for example, vascular systems by utilization of indirect addresses. Segmented angiographic information from patients HemeLB setup tool permits the client to show the geometric space to simulate by utilizing a graphical user interface. Itani et al. presented a tool to automatically create an ensemble of multiscale blood flow simulations and run these simulations using supercomputing resources, scales near-linearly up to 32,768 cores [25].

The intention of using LB method for practical purposes requires large computational power such as supercomputers as mentioned above. However, alternatively, several attempts have been made to implement LB on different platforms such as graphic processing unit (GPU) and field programmable gate array (FPGA). The proposed work [26] describes the porting of the LB component of MUPHY to GPUs using ad-hoc techniques for optimized addressing.

The simulations of D3Q19 LB model were executed successfully on multi-node GPU clusters using CUDA platform and MPI library [27,28,29]. Januszewski et al. [30] present Sailfish, an open source fluid simulation package implementing the LB on modern GPUs using both CUDA and OpenCL programming packages. Obrecht et al. [29] extend the software design to achieve more efficient and highly scalable multi-GPU parallelization within waLBerla framework which is capable to heterogeneous simulations using CPUs and GPUs in parallel. They evaluates the results on the Tsubame 2.0 cluster for more than 1000 GPUs.

The ultimate limitation for GPU implementation is the global memory access that bounds the performance of the LB code, however, FPGAs provide flexibility that allows compilers to create memory topologies for the acceleration of the LB algorithm with higher precision [31,32]. The method proposed in [31] used OpenCL to accelerate and implement D2Q9 lattice model of LB on FPGA. Using the OpenCL tool

makes it possible to address well-known HPC problems on FPGA more easily [33]. Two scalable approaches for simulating LB on coupled multiple FPGAs are proposed in [34] and [35] which use accelerator-domain network (ADN) for low-latency and high-speed data transfer between FPGAs. Investigation study in [36] shows that the network bandwidth is much more important than the memory bandwidth in multiple FPGAs framework.

In summary, the existing LB implementations did not fully explore the memory utilization and flexibility of heterogeneous architectures for accelerating the LB operations. In this paper, an efficient hardware architecture based LB algorithm for a Zynq System-on-Chip (SoC) hardware implementation are proposed. The proposed implementation utilizes the advantages of heterogeneous SoC, and divides the calculations of the LB algorithm into two parts: Programmable Logic (PL) and the Processing System (PS) [32,37], where the PL contains a Xilinx 7 series Artix-based programmable logic and the PS contains a dual ARM Cortex-A9 based hard core processors[38]. The major contributions of this work can be summarized in the following two points: 1) Efficient real-time implementation of LB method on a heterogeneous SoC; 2) An efficient model partition scheme to increase the efficiency of memory utilization. The achieved experimental results show that the proposed implementation accelerate the processing speed by a factor of 52 compared to a pure software implementation and only consume 2 W, which is only 15% overhead than the software implementation.

The rest of the paper is organized as follows. Section 2 briefly introduces the LB method. The corresponding software and hardware implementations are presented in Section 3 and 4 respectively. The experimental results are discussed in Section 5. Finally, Section 6 concludes the paper and highlights some perspectives of future work.

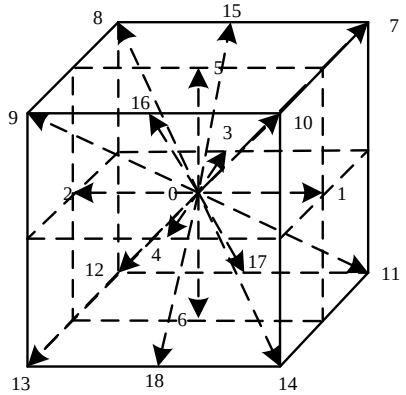

Figure 2. Lattice nodes of D3Q19 model.

2. Lattice Boltzmann Method

The LB method is known as a technique being used for the simulation of fluid flow, which models fluids with particles performing propagation and collision processes over a discrete lattice mesh. In this paper, a three dimensions, 19 speeds model called D3Q19 model is used, as shown in Figure 1. Each

point in the model is joined to its neighbours by a set of lattice vectors e_i. In the current implementation of the LB algorithm, we use a cubic lattice with 19 lattice vectors joining each node with its neighbours. The LB model adopted has the distribution function that is assumed to evolve towards its local equilibrium value, at a rate controlled by a single relaxation parameter τ [21]:

$$\Omega \approx \frac{f^{(eq)} - f}{\tau} \tag{1}$$

where f is the distribution function of the particles, and Ω is the collision. D3Q19 models are three-dimensional with 19 directions of velocity as shown in the Figure 1. The resulting lattice equation is

$$f_i(\mathbf{x} + \mathbf{e_i}\Delta x, t + \Delta t) - f_i(\mathbf{x}, t) = -\frac{f(\mathbf{x}, t) - f^{(eq)}(\mathbf{x}, t)}{\tau} \tag{2}$$

where the local equilibrium distribution functions are:

$$f_i^{eq} = w_i\left(p + \frac{\mathbf{e_i} \cdot \mathbf{u}}{c_s^2} + \frac{(\mathbf{e_i} \cdot \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u} \cdot \mathbf{u}}{2c_s^2}\right) \tag{3}$$

where f_i^{eq} is the equilibrium distribution and τ is the relaxation time towards the equilibrium for collision which is calculated separately from streaming. w_i is a weight coefficient, c_s is the speed of sound, e_i is the particle's velocity in the direction i and the hydrodynamic density p and macroscopic velocity u are determined by the distribution functions based on:

$$p = \sum_{i=0}^{18} f_i = \sum_{i=0}^{18} f_i^{(eq)} \tag{4}$$

$$\mathbf{u} = \sum_{i=0}^{18} \mathbf{e_i} f_i = \sum_{i=0}^{18} \mathbf{e_i} f_i^{(eq)} \tag{5}$$

For each of the 19 directions, the distribution functions are propagated along the lattice velocity e_i to the adjacent sites. More specifically, the equilibrium distribution f_i, with velocity e_i moves from the site at position (x, y, z) to the site at position $(x, y, z) + e_i$.

The lattice nodes of macroscopic velocity is defined as:

$$\vec{e_l} = \begin{cases} (0,0,0) & i = 0 \\ (\pm 1,0,0),(0,\pm 1,0),(0,0,\pm 1) & i = 1,2,3,4,5,6 \\ (\pm 1,\pm 1,0),(\pm 1,0,\pm 1),(0,\pm 1,\pm 1) & i = 7,8,9,...,18 \end{cases} \tag{6}$$

The streaming calculation moves to the direction of 19 directions velocity f_i. Consequently, the macroscopic density and velocity are calculated using their equations from f_i. In addition, the equilibrium distribution and the distribution function in collision step are also calculated. Finally, the steps of streaming to the collision are repeated.

3. Implementation of Lattice-Boltzmann Algorithm

The LB method was implemented using C/C++ and then simulated using Vivado HLS [39]. Various pragmas have been applied in order to generate different architectures in Vivado HLS environment. During the implementation, the LB algorithm, has been divided into two main calculation phases: 1)

collision phase; and 2) streaming phase. The collision phase has been implemented on PL and the streaming phase on PS.

Figure 3 shows the overall flowchart of the proposed implementation. The memory is firstly initialised to accommodate the geometry model, and each partition of geometry model is then loaded to the collision phase for the calculations of hydrodynamic density p and macroscopic velocity u until the end of geometry model. After this, the movements of each particle in 19 directions are calculated in the streaming phase for the entire geometry model. Finally, this process is repeated until the end of simulation.

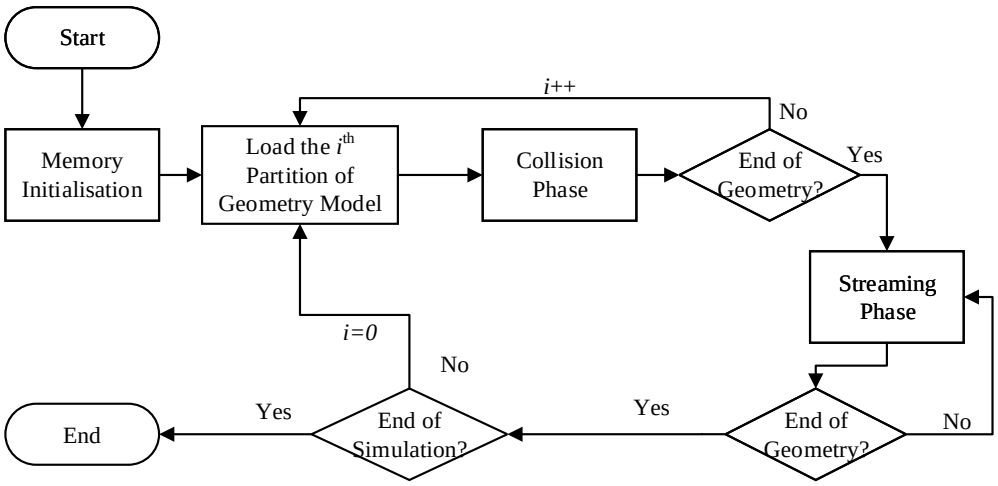

Figure 3. Flowchart of the proposed LB implementation.

3.1 Collision phase

Since the collision phase is one of the main calculation stages in LB method and in order to achieve good data locality, a simple data structure is applied, where the 3D data are reshaped and stored in 1D arrays. The code to handle the collision phase is illustrated in the Pseudocode 1.

Pseudocode 1: Collision Phase

1. **Input**: f_{old} is source buffers of particle distribution function.
2. **Output**: f_{old} is updated source buffers of particle distribution function after collision phase.
   // N_x, N_y, and N_z are the dimensions of grid // d is the number of vectors
   float * f_{old} = (*float* *) *malloc*($N_x \times N_y \times N_z \times d$ *sizeof*(*float*));
3. ***for*** *all the sites in* ***z*** *direction*
4. ***for*** *all the sites in* ***y*** *direction*
5. ***for*** *all the sites in* ***x*** *direction*
   // Calculate macroscopic parameters
6. p_{new} = calculate_p(f_{old}, x, y, z); // fluid density
7. u_{new} = calculate_u(f_{old}, p_{new}, e, x, y, z); // fluid velocity
8. ***for*** *all the 19 directions*
   // BGK approximation collision operator
9. $f_{old}[x + ((y + z \times n_y) \times n_x) + (n_x \times n_y \times n_z \times i)]$ =
   bgk ($f_{old}[x + ((y + z \times n_y) \times n_x) + (n_x \times n_y \times n_z \times i)]$, τ, p_{new}, w_i, e_i, u_{new});
10. ***end***
11. ***end***
12. ***end***
13. ***end***

During the collision phase, the fluid density and velocity are calculated for each cell within the predefined grid network. Once the new fluid density and velocity are calculated, the Bhatnagar-Gross-Kroop (BGK) approximation collision operator is applied in each direction vector of the lattice nodes of D3Q19 model. Finally, the buffers of particle distribution function are updated with the new calculations. As it can be seen from Pseudocode 1, the most computational tasks are the calculation of three nested loops (i.e. line 4, 5 and 6), therefore our main optimisation efforts would be applying pipelining and parallelism to accelerate the process. In order to optimize the throughput of the operations, a set of pragmas have been used in the code to guide the complier to optimize the code, for instance, "*HLS PIPELINE*" is one of the pragmas to increase the pipelining and parallelism of the implementation. In addition, arrays are implemented as block-random access memory (BRAM) which has only a maximum of two data ports. This can limit the throughput of a read/write (or load/store) intensive algorithm. The bandwidth can be improved by splitting the array (a single BRAM resource) into multiple smaller arrays (multiple BRAMS), which would effectively increase the number of ports of memory.

In order to effectively adopt the pipeline mechanism, it will need extra memory and other logic during the implementation. However, it is not possible to duplicate all the cells in the source buffer f_{old} due to the limitation of the available on-chip memory as well as the size of simulation model. Therefore,

in order to reduce the usage of the on-chip memory, the entire simulation buffer f_{old} has been divided by n. This means we only calculate the collision within the small partition, and once this is completed, we will calculate the following partitions until the end of the simulation grid. Since the partitions are independent from each other, this makes it very suitable to implement them in parallel. Figure 4 demonstrates this partition process, where N_x, N_y and N_z are the dimensions of simulation grid. An example of the simulation model was divided by a factor of n in x axis direction.

In Figure 4, the total number of cells in the simulation buffer f_{old} is $N_y \times N_z \times N_x \times d$ when we partition the buffer by a factor of n. The total number of cells in each partition is then equal to $N_y \times N_z \times N_x \times d/n$. When $N_x = n$, the total number of cells within the simulation becomes $N_y \times N_z \times d$. Let f_y and f_z denote the total number of cells in y and z axes directions respectively, and the partition buffer is f_{pold}. If the f_{pold} is partitioned into two smaller arrays (i.e. applying *#pragma HLS ARRAY_PARTITION cyclic factor* = 2), and then each array will contain $f_y \times f_z$ cells. By applying these pragmas, the buffer f_{pold} will therefore be divided into two arrays, one is with even memory addresses and the other one is with odd memory addresses, and the three nested loops in pseudocode 1 (i.e. line 4, 5 and 6) could be flatten and unroll in a pipeline manner, which would potentially significantly increase the throughput of the design. Based on this optimization, Pseudocode 1 can then be rewritten as Pseudocode 2. In addition to this optimisation, a number of implementations with different other optimisation parameters have been introduced and compared in the result section.

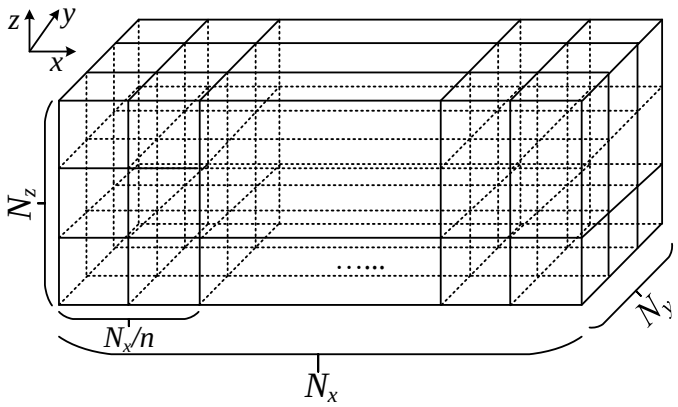

Figure. 4. Partitions of simulation grid.

Pseudocode 2: Collision Phase with optimisation

1. **Input**: f_{old} is source buffers of particle distribution function.
2. **Output**: f_{new} is updated source buffers of particle distribution function after collision phase.
3. *#pragma HLS ARRAY_PARTITION variable=f_{pold} cyclic factor=2 dim=3*
4. ***for*** *all the sites in* ***z*** *direction*
5. #pragma HLS PIPELINE
6. ***for*** *all the sites in* ***y*** *direction*

   // Calculate macroscopic parameters
7. p_{new} = calculate_p(f_{old}, y, z); // fluid density
8. u_{new} = calculate_u(f_{old}, p_{new}, e, y, z); // fluid velocity
9. ***for*** *all the 19 directions*

   // BGK approximation collision operator
10. $f_{old}[y + ((z + i \times n_z) \times d)]$ = $bgk(f_{old}[y + ((z + i \times n_z) \times d)], \tau, p_{new}, w_i, e_i, u_{new})$;
11. ***end***
12. ***end***
13. ***end***

3.2 Streaming phase

In the streaming phase, the particle distribution function is caculated among lattice points in the predefined directions (i.e. D3Q19 model). Generally, the fluid particles are streamed from one cell to a neighbouring cell according to the velocity of the fluid particles in this cell. Depending on the velocity of each cell, the streaming can either be performed as a pushing operation from one cell to the surrounding cells or pulling operation in a reverse direction. In addition, the boundary conditions are also considered within this phase. The bounce back rule is applied to the no-slip wall surrounding the system, and the procedure of handling the streaming phase is illustrated in the Pseudocode 3. Since the streaming phase involves mainly I/O operations, e.g. memory accesses, and there is little arithmetic calculation within this phase, therefore, the implementation of this phase has been performed on the PS side.

Pseudocode 3: Streaming Phase

1. **Input**: f_{old} is source buffers of particle distribution function.
2. **Output**: f_{new} is updated source buffers of particle distribution function after streaming phase.
3. ***for*** *all the sites in* ***z*** *direction*
4. ***for*** *all the sites in* ***y*** *direction*
5. ***for*** *all the sites in* ***x*** *direction*
   // Calculate forces in different directions
   //*i and i'* are from 0 to *d* directions,
   // calculate the directions of the streams one by one,
   // *d* is the total number of the vectors.
6. $f_{new}[x + ((y + z \times n_y) \times n_x) + (n_x \times n_y \times n_z \times i)] =$
   $f_{max}[0.0, f_{old}(x + ((y + z \times n_y) \times n_x) + (n_x \times n_y \times n_z \times i'))]$
7. ***end***
8. ***end***
9. ***end***

4. Experimental Results

The proposed LB implementation has first been validated in Vivado HLS, and then synthesised and translated to a hardware description language (HDL) code. A set of pragma directives have been used to optimise the hardware implementation with the goal to achieve the optimal throughput with reasonable usage of hardware recourses. The IP core is connected via AXI4 interfaces to the accelerator coherency port (ACP) of the ARM CPU in the Zynq-7000 SoC device. The solution is then exported as an IP core connected with AXI4 interface to the ACP on AP SoC PS. The connection is made through a direct memory access (DMA) core in the PL subsystem. SDSoC (v2016.4) [40] has been used to interface the AP SoC PL hardware, the peripheral, the DMA engine, an AXI timer as well as other data mover logics. The SDSoC is also used to design the AP SoC PS software to manage the peripherals and loading the testing data.

The collision phase block have been implemented in Vivado HLS, and has been integrated with the other blocks of the design to be a heterogeneous embedded system as shown in Figure 5. The proposed hardware implementation uses 32-bit floating-point arithmetic, a C/register transfer level simulation is performed before exporting the RTL as a Vivado's IP core. The generated IP cores have later been used in SDSoC (v2016.4) in order to move the IP cores to hardware and generate the corresponding firmware for the hardware/software codesign. As shown in Figure 5, the IP core (i.e. collision phase) is interconnected with Zynq PS 7 via AXI interconnection blocks. The AXI2FIFO adapter block is used to

convert the interface from AXI to FIFO in order to connect with the IP cores. The AXI DMA is used to move the processed data back to Zynq PS7.

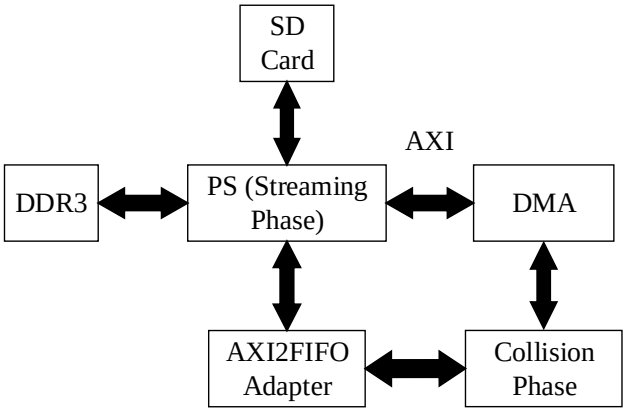

Figure 5. Implementation overview.

4.1 Vivado HLS Simulation

Prior to the hardware implementation, the proposed system was validated using Vivado HLS C simulator. Once the implementation has passed the Vivado C simulation, the C++ codes were translated to HDL, and then register transfer level simulation is performed in order to validate the generated HDL architecture. The same C++ test bench used in the C/C++ simulation was used for the C/ register transfer level co-simulation; however, instead of using the C++ function, the synthesized register transfer level architectures are used to perform the calculation. The simulator used in the C/ register transfer level co-simulation was XSIM where VHDL was selected as the generated HDL. The clock period for the simulation was set to 10 *ns*. Table 1 summarises the hardware utilization results for implementing the collision phase.

Table 1. Hardware utilization estimate of collision phase (C/RTL co-simulation)

Collision Phase	**LUT (%)**	**FF (%)**	**DSP48E (%)**	**BRAM_18K (%)**
BGK	41	13	62	0
Fluid Velocity Function	16	6	7	0
Others	8	2	3	2
Total	65	21	72	2

The hardware utilization results are based on the implementation using only the pipeline pragma on a Zynq-7000 xc7z020 SoC. Based on the utilisation report, 41% of the LUTs, 13% of the FFs and 62% of the DSP48E are used to implement the BGK collision operator within the collision phase (i.e. line 10 in Pseudocode 2), as this function contains most of the arithmetic calculations of the entire collision phase, it uses most of the resources. In addition to the BGK function, the calculation of fluid velocity function

consumes 16%, 6% and 7% of the used LUTs, FFs and DSP48E resources. It is worth noting that the target Zynq SoC has a relatively small chip capacity in Xilinx 7 series family, which means that the proposed architecture is area-efficient, and can be easily deployed on a low-cost FPGA or integrated on a large chip. In the proposed work, we firstly performed system level profiling of the software implementation, and then optimised the software in terms of throughput, latency and area trade-offs for various hardware acceleration configurations in the Zynq based multi-processor system-on-chip. Figure 6 presents a diagram that shows a comparison of hardware resource usage for different implementations with various optimisation conditions. Table 2 summarises the tested optimisation conditions in Figure 6.

Table 2. Optimization conditions for the different implementations in Figure 6

Configurations	Array Partition factor (dimension = 3)	Array Partition factor (dimension = 2)	Array Partition factor (dimension = 1)	Pipeline
Configuration 1	0	0	0	No
Configuration 2	2	0	0	Yes
Configuration 3	2	4	4	Yes
Configuration 4	4	4	4	Yes
Configuration 5	19	4	4	Yes

As it can be seen in Figure 6, the implementation requiring the least hardware resources in the configuration 1, where no pipeline or array partition is required. However, the processing speed of this implementation is limited, it needs 2105 *us* to process the collision phase. In configuration 2, the array partition and pipeline pragmas have been applied, it improves the processing speed to 335 *us* and as expected using more hardware resources. For configurations 3, 4, and 5, although the factors of array partitions have been increased accordingly and more hardware resources have been used, however, the processing speed has not improved significantly. Therefore, configuration 2 has been chosen for the final hardware implementation.

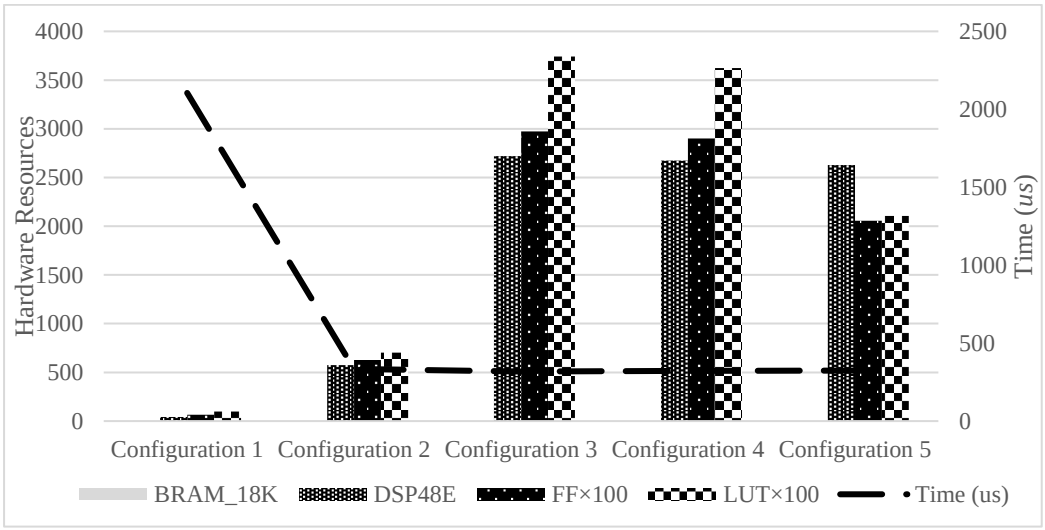

Figure. 6. A comparison of hardware resources usages with different optimization configurations.

4.2 Hardware Implementation

The proposed system has been implemented on the Xilinx Zedboard, which is equipped with a Zynq-7000 All Programmable SoC XC7Z020-CLG484, 512 MB DDR3 memory and 16 GB SD card. In addition, the corresponding software (i.e. drivers, control codes and streaming phase) and hardware (i.e. collision phase) are partitioned and implemented using the Xilinx SDSoC development environment. In the proposed implementation, the model size is $N_x = 8$, $N_y = 4$ and $N_z = 4$, and a pure software processor based and a heterogonous SoC implementations are performed and implementation results are compared in the following sections.

4.2.1 Power Consumption

The on-chip power consumption consists mainly of two parts, which are static and dynamic power consumption. The static power is consumed due to transistor leakage. The dynamic power is consumed by fluctuating power as the design runs, i.e. Zynq7 Processing System (PS7), clock, power, logic power, signal power, BRAMs power, etc., which are directly affected by the chip clock frequency and the usage of chip area. The details of estimated power consumption of the implementation are summarised in Table 3. The PS7 consumes much more power than the PL; this is because that the ARM dual core Cortex-A9 based processing system has much higher running frequency than the PL and it runs at a fixed clock frequency. Compared to the PS7, the programmable logic blocks consume only a small portion of the total on-chip power consumption, and it handles manly the collision phase and consumes only additional 15% of the overall power consumption. The total on-chip power consumption estimations for the heterogeneous SoC implementation is about 2 W.

Table 3. Power consumption

	Utilization Details	Power (W)	Utilization (%)
Dynamic Power Consumption	Clock	0.084	4%
	Signals	0.081	4%
	Logic	0.055	3%
	BRAM	0.019	1%
	DSP	0.065	3%
	PS7	1.526	76%
Static Power Consumption	Device Static	0.169	9%

4.2.2 Timing Analysis

The ARM processor runs at 650 MHz and the PL clocked at 100 MHz. The processing time of the proposed system is measured by counting the number of clock cycles of the ARM processor for obtaining the calculated results from the collision phase. Table 4 shows the comparison between the software and hardware implementations of collision phase in terms of the processing time in CPU clock cycles and millisecond.

Based on the implementation results from Table 2 and 3, the proposed heterogeneous SoC implementation achieves the throughput 1.7×10^5 grid-points/sec, which is 57 times higher than the PS7 only implementation. In addition, in terms of power consumption, the proposed heterogeneous implementation consumes only 15% overhead than the solo software implementation.

Table 4. Processing time of the proposed implementation

	Collision Phase (*clock cycles*)	**Collision Phase (*ms*)**
Heterogeneous Implementation	466556	0.718
Software Implementation	26691550	41.063

5. Conclusion

In this paper, efficient hardware architectures of the LB method on a Zynq System-on-Chip (SoC) platform are designed and implemented. The two main parts of the LB method, collision and streaming phases have been implemented on PL and PS respectively. In addition, of the proposed architectures have been tested and evaluated under different implementation configurations. The experimental results show that the proposed heterogeneous implementation is able to accelerate the processing speed by a factor of 52 with a power consumption of only 2 W. In the future, the performance of the proposed implementation

scheme will be further evaluated using different computing platforms (e.g. HPC and GPU, etc.), geometries and integrated with the aneurysm segmentation algorithm as well as the virtual reality facilities to create an interactive environment for treatment planning and training purpose.

6. Acknowledgments

This paper was made possible by National Priorities Research Program (NPRP) grant No. 5-792-2-328 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

7. References

1. Komotar RJ, Mocco J, Solomon RA. Guidelines for the surgical treatment of unruptured intracranial aneurysms: the first annual J. Lawrence pool memorial research symposium—controversies in the management of cerebral aneurysms. *Neurosurgery.* 2008;62(1):183-194.
2. Hunt WE, Hess RM. Surgical risk as related to time of intervention in the repair of intracranial aneurysms. *Journal of neurosurgery.* 1968;28(1):14-20.
3. Sun C, Migliorini C, Munn LL. Red blood cells initiate leukocyte rolling in postcapillary expansions: a lattice Boltzmann analysis. *Biophysical Journal.* 2003;85(1):208-222.
4. Zai-Yi S, Ying H. A lattice Boltzmann method for simulating the separation of red blood cells at microvascular bifurcations. *Chinese Physics Letters.* 2012;29(2):024703.
5. He X, Duckwiler G, Valentino DJ. Lattice Boltzmann simulation of cerebral artery hemodynamics. *Computers & Fluids.* 2009;38(4):789-796.
6. De Rosis A. Analysis of blood flow in deformable vessels via a lattice Boltzmann approach. *International Journal of Modern Physics C.* 2014;25(04):1350107.
7. Ouared R, Chopard B, Stahl B, Rüfenacht DA, Yilmaz H, Courbebaisse G. Thrombosis modeling in intracranial aneurysms: a lattice Boltzmann numerical algorithm. *Computer Physics Communications.* 2008;179(1):128-131.
8. Bernsdorf J, Wang D. Non-Newtonian blood flow simulation in cerebral aneurysms. *Computers & Mathematics with Applications.* 2009;58(5):1024-1029.
9. Groen D, Hetherington J, Carver HB, Nash RW, Bernabeu MO, Coveney PV. Analysing and modelling the performance of the HemeLB lattice-Boltzmann simulation environment. *Journal of Computational Science.* 2013;4(5):412-422.
10. Abrahamyan L, Schaap JA, Hoekstra AG, et al. A problem solving environment for image-based computational hemodynamics. Paper presented at: International Conference on Computational Science2005.
11. Schulz M, Krafczyk M, Tölke J, Rank E. Parallelization strategies and efficiency of CFD computations in complex geometries using Lattice Boltzmann methods on high-performance computers. *High performance scientific and engineering computing.* 2002;21:115-122.
12. Pohl T, Deserno F, Thurey N, et al. Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures. Paper presented at: Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference2004.
13. Bernaschi M, Melchionna S, Succi S, Fyta M, Kaxiras E, Sircar JK. MUPHY: A parallel MUlti PHYsics/scale code for high performance bio-fluidic simulations. *Computer Physics Communications.* 2009;180(9):1495-1502.
14. Desplat J-C, Pagonabarraga I, Bladon P. LUDWIG: A parallel Lattice-Boltzmann code for complex fluids. *Computer Physics Communications.* 2001;134(3):273-290.
15. Randles AP, Kale V, Hammond J, Gropp W, Kaxiras E. Performance analysis of the lattice Boltzmann model beyond Navier-Stokes. Paper presented at: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on2013.

16. Randles A, Draeger EW, Bailey PE. Massively parallel simulations of hemodynamics in the primary large arteries of the human vasculature. *Journal of Computational Science.* 2015;9:70-75.
17. Hasert M, Masilamani K, Zimny S, et al. Complex fluid simulations with the parallel tree-based Lattice Boltzmann solver Musubi. *Journal of Computational Science.* 2014;5(5):784-794.
18. Schmieschek S, Shamardin L, Frijters S, et al. LB3D: A parallel implementation of the Lattice-Boltzmann method for simulation of interacting amphiphilic fluids. *Computer Physics Communications.* 2017;217:149-161.
19. Heuveline V, Krause MJ. OpenLB: towards an efficient parallel open source library for lattice Boltzmann fluid flow simulations. Paper presented at: International Workshop on State-of-the-Art in Scientific and Parallel Computing. PARA2010.
20. Li D, Xu C, Wang Y, et al. Parallelizing and optimizing large‐scale 3D multi‐phase flow simulations on the Tianhe‐2 supercomputer. *Concurrency and Computation: Practice and Experience.* 2016;28(5):1678-1692.
21. Tran N-P, Lee M, Hong S. Performance optimization of 3D lattice Boltzmann flow solver on a GPU. *Scientific Programming.* 2017;2017.
22. Liu S, Zou N, Cui Y, Wu W. Accelerating the Parallelization of Lattice Boltzmann Method by Exploiting the Temporal Locality. In Ubiquitous Computing and Communications (ISPA/IUCC), 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 1186-1193. IEEE, 2017.
23. Pontrelli G, Halliday I, Melchionna S, Spencer TJ, Succi S. The Lattice Boltzmann Method and Multiscale Hemodynamics: Recent Advances and Perspectives. *IFAC Proceedings Volumes.* 2012;45(2):30-39.
24. Mazzeo MD, Coveney PV. HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries. *Computer Physics Communications.* 2008;178(12):894-914.
25. Itani MA, Schiller UD, Schmieschek S, et al. An automated multiscale ensemble simulation approach for vascular blood flow. *Journal of Computational Science.* 2015;9:150-155.
26. Bernaschi M, Fatica M, Melchionna S, Succi S, Kaxiras E. A flexible high‐performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. *Concurrency and Computation: Practice and Experience.* 2010;22(1):1-14.
27. Xian W, Takayuki A. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing.* 2011;37(9):521-535.
28. Calore E, Gabbana A, Kraus J, Schifano SF, Tripiccione R. Performance and portability of accelerated lattice Boltzmann applications with OpenACC. *Concurrency and Computation: Practice and Experience.* 2016;28(12):3485-3502.
29. Obrecht C, Kuznik F, Tourancheau B, Roux J-J. Multi-GPU implementation of the lattice Boltzmann method. *Computers & Mathematics with Applications.* 2013;65(2):252-261.
30. Januszewski M, Kostur M. Sailfish: A flexible multi-GPU implementation of the lattice Boltzmann method. *Computer Physics Communications.* 2014;185(9):2350-2368.
31. Nallatech. *FPGA Acceleration Acceler Ation of Lattice Boltzmann Using OpenCL.* 2017.
32. Zhai X, Ait Si Ali A, Amira A, Bensaali F. ECG encryption and identification based security solution on the Zynq SoC for connected health systems. *Journal of Parallel and Distributed Computing.* 2017;106:143-152.
33. Abdelfattah MS, Hagiescu A, Singh D. Gzip on a chip: High performance lossless data compression on fpgas using opencl. Paper presented at: Proceedings of the International Workshop on OpenCL 2013 & 20142014.
34. Sano K, Kono Y, Suzuki H, et al. Efficient custom computing of fully-streamed lattice boltzmann method on tightly-coupled FPGA cluster. *ACM SIGARCH Computer Architecture News.* 2013;41(5):47-52.
35. Sano K. FPGA-Based Scalable Custom Computing Accelerator for Computational Fluid Dynamics Based on Lattice BoltzmannMethod. In: *Sustained Simulation Performance 2014.* Springer; 2015:187-201.

36. Kono Y, Sano K, Yamamoto S. Scalability analysis of tightly-coupled FPGA-cluster for lattice Boltzmann computation. Paper presented at: 22nd International Conference on Field Programmable Logic and Applications (FPL), 2012.
37. Djelouat H, Zhai X, Disi MA, Amira A, Bensaali F. System-on-Chip Solution for Patients Biometric: A Compressive Sensing-Based Approach. *IEEE Sensors Journal.* 2018;18(23):9629-9639.
38. Xilinx Inc. Zynq-7000 All Programmable SoC. 2012; http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm, 2013. [Accessed in Feb, 2018]
39. Xilinx Inc. Vivado HLS User Guide. 2017; www.xilinx.com. [Accessed in Feb, 2018]
40. Xilinx Inc. SDSoC User Guide. 2017; www.xilinx.com. [Accessed in Feb, 2018]

List of Tables:

Table 1. Hardware utilization estimate of collision phase (c/rtl co-simulation)

Collision Phase	LUT (%)	FF (%)	DSP48E (%)	BRAM_18K (%)
BGK	41	62	13	0
Fluid Velocity Function	16	6	7	0
Total	65	21	72	2

Table 2. Optimization conditions for the different implementations in Figure 6

Configurations	Array Partition factor (dimension = 3)	Array Partition factor (dimension = 2)	Array Partition factor (dimension = 1)	Pipeline
Configuration 1	0	0	0	No
Configuration 2	2	0	0	Yes
Configuration 3	2	4	4	Yes
Configuration 4	4	4	4	Yes
Configuration 5	19	4	4	Yes

Table 3. Power compsuption

	Utilization Details	Power (W)	Utilization (%)
Dynamic Power Consumption	Clock	0.084	4%
	Signals	0.081	4%
	Logic	0.055	3%
	BRAM	0.019	1%
	DSP	0.065	3%
	PS7	1.526	76%
Static Power Consumption	Device Static	0.169	9%

Table 4. Processing time of the proposed implementation

	Collision Phase (*clock cycles*)	Collision Phase (*ms*)
Heterogeneous Implementation	466556	0.718
Software Implementation	26691550	41.063