Exploring the Epiphany manycore architecture for the Lattice Boltzmann algorithm

MASTER THESIS

Volvo Penta, Göteborg
and
IDE Department,
Högskolan i Halmstad

Name:	Sebastian Raase
Program:	Embedded and Intelligent Systems
Year:	2014
Supervisor Högskolan:	Prof. Tomas Nordström
Supervisor Volvo:	Lars Johansson

Declaration

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. Material and ideas taken, directly or indirectly, from foreign sources are noted as such.

This work has not been submitted for any other degree or diploma.

Halmstad, November 21, 2014

Acknowledgement

To my grandfather.

I would like to thank Prof. Tomas Nordström from Högskolan i Halmstad, who supported me continuously throughout the thesis, took the time needed and always provided very helpful feedback.

Also, I would like to thank Lars Johansson from Volvo Penta AB, without whom this thesis would not have been possible, and Jonas Latt from the University of Geneva for the immensely helpful discussions about the Lattice Boltzmann algorithm.

A special thanks goes to my family, who always stand by my side.

Finally, I thank all those people who suffered my frustrations and encouraged me.
You know who you are.

Thank you.

Contents

Abstract **1**

1 Introduction **2**
1.1 Research Question and Methodology 3

2 Background **4**
2.1 Computational Fluid Dynamics 4
2.1.1 Fluids 4
2.1.2 Test Case: lid cavity 5
2.2 Lattice Boltzmann Method 5
2.2.1 Lattice Models 7
2.2.2 Boundary Conditions 9
2.3 Adapteva Epiphany 10
2.3.1 Parallella Board 10
2.3.2 Mesh Architecture 11
2.3.3 Processor Node 13
2.3.4 Programming 16

3 Implementation **17**
3.1 Data Layout 19
3.2 Algorithm 20
3.3 Host Code 20

4 Results **22**
4.1 Computation 23
4.2 Shared Memory Access 28

5 Conclusion **32**

6 Future Work **33**

A Appendix **34**
A.1 Build System 34

A.2 Shared Header File 35
A.3 Epiphany 36
A.4 Host 38

B References 39

Abstract

Computational fluid dynamics (*CFD*) plays an important role in many scientific applications, ranging from designing more effective boat engines or aircraft wings to predicting tomorrow's weather, but at the cost of requiring huge amounts of computing time. Also, traditional algorithms suffer from scalability limitations, making them hard to parallelize massively.

As a relatively new and promising method for computational fluid dynamics, the Lattice Boltzmann algorithm tries to solve the scalability problems of conventional, but well-tested algorithms in computational fluid dynamics. Through its inherently local structure, it is well suited for parallel processing, and has been implemented on many different kinds of parallel platforms.

Adapteva's Epiphany platform is a modern, low-power manycore architecture, which is designed to scale up to thousands of cores, and has even more ambitious plans for the future. Hardware support for floating-point calculations makes it a possible choice in scientific settings.

The goal of this thesis is to analyze the performance of the Lattice Boltzmann algorithm on the Epiphany platform. This is done by implementing and testing the lid cavity test case in two and three dimensions. In real applications, high performance on large lattices with millions of nodes is very important. Although the tested Epiphany implementation scales very good, the hardware does not provide adequate amounts of local memory and external memory bandwidth, currently preventing widespread use in computational fluid dynamics.

1 Introduction

Fluids are all around us, from the air we breathe to the water we drink. Additionally, fluid mechanics play a key role in many different scientific and technical domains, being used to decide the shapes of energy-efficient cars and airplanes, or propeller and turbine blades. Understanding the behaviour of air in the athmosphere is important for both today's weather forecast and for building storm-resistant buildings, as is understanding the behaviour of water in the ocean for predicting dangerous waves. Although the list of applications could be extended much further, fluid mechanics are far from being a solved problem. In fact, one of the seven most fundamental yet unsolved Millennium Prize problems named by the Clay Mathematics Institute in 2000 revolves around the Navier-Stokes equations[6], which describe the behaviour of Newtonian fluids[1].

In Computational Fluid Dynamics (CFD), instead of trying to solve problems from fluid dynamics, numerical methods are used to simulate them. These methods require huge amounts of computation, making the choices of hardware platform and algorithm even more important. The Lattice Boltzmann algorithm is a relatively new algorithm used in CFD, and is especially well suited for parallelization. This algorithm has been studied extensively on different architectures, e.g. by evaluating the single-core performance on common off-the-shelf and specialized vector processors [27] or looking at cluster performance [18]. However, many studies have been focused on GPUs [16][25][11], or compared GPU and cluster performance [21]. So far, there has been little work on manycore architectures, although early results on Intel's Xeon Phi have been reported [7].

[1]Newtonian fluids are characterized by a stress-independent viscosity.
Many common fluids, e.g. water and air, are commonly approximated as being Newtonian.

1.1 Research Question and Methodology

The central question of this thesis is the suitability of Adapteva's Epiphany architecture[1] for the Lattice Boltzmann algorithm. To the author's knowledge, this has not been studied before, although research involving this architecture is ongoing in other areas (e.g. [23]). Being a manycore architecture, parallelization is the key to use the Epiphany efficiently and as such, this work will mainly focus on scalability and performance.

To do the evaluation, the Lattice Boltzmann algorithm will be implemented on the Parallella board, which currently serves as the reference implementation of the Epiphany architecture and contains sixteen processor cores. This implementation will be used to simulate the lid cavity test case in two and three dimensions. Code instrumentation and benchmarking of different problem sizes will then be used to measure scalability and performance of the algorithm. Through analysis of the results, performance problems and bottlenecks will be identified and a conclusion will be drawn on whether the Epiphany architecture is suitable for the Lattice Boltzmann algorithm or not.

However, a universal and highly optimized implementation of the Lattice Boltzmann algorithm is outside the scope of this thesis and left for future study (see also chapter 6).

2 Background

In Computational Fluid Dynamics (CFD), problems from the domain of fluid mechanics are solved numerically through simulation. In order to achieve sufficient accuracy of the (approximate) solutions, good fluid models are necessary. This chapter provides a short introduction to fluids in general and describes the test case used. Since this thesis focuses on the Lattice Boltzmann and the Epiphany hardware architecture, they are described in more detail afterwards.

2.1 Computational Fluid Dynamics

2.1.1 Fluids

"A fluid is defined as a substance that deforms continuously whilst acted upon by any force tangential to the area on which it acts."[26] These forces are called *shear forces*, and while solids do resist them (possibly after a slight, non-permanent deformation), fluids will change their shape and not be able to reach a stable equilibrium. Fluids are further divided into gases and liquids, with liquids usually having a much higher density and being much less compressible than gases. Also, gases tend to spread, filling any available volume, while liquids will form boundary layers to the surroundings instead.

Although a complete analysis of a fluid would have to account for each individual molecule, in most engineering applications only the local and/or global averages for different properties like density or velocity are interesting. Two different approaches have been developed to deal with this. In classical fluid mechanics, fluids are thus treated as a continuum and

the basic principles from mechanics (i.e. conservation of energy and matter, and Newton's laws of motion) are applied to them. On the other hand, approaches like the Lattice Boltzmann method treat fluids as statistical particle distributions, averaging the fluid behaviour locally.

2.1.2 Test Case: lid cavity

As a result of the no-slip boundary condition, fluid nodes attached to a wall will always move at the same speed as the wall itself. The lid cavity test case consists of a box with zero-speed no-slip walls on all sides, except for a single wall, which moves at a constant velocity. Initially, the fluid is of constant density and does not move at all. Figure 2.1 shows the normalized absolute velocity of a simulation after 150 (left) and 1000 (right) time steps. Higher speeds are shown with higher intensity. It can be seen that in the beginning, the moving top wall (moving to the right) drags the fluid nearby along, increasing velocity. Since the fluid cannot pass the right wall, it starts to move downwards and, over time, forms a circular stream. After some time has passed, a steady state is reached.

Figure 2.1: Velocity field of a 104x104 2D lid cavity simulation, at iterations 150 (left) and 1000 (right)

2.2 Lattice Boltzmann Method

Austrian physicist Ludwig Boltzmann is seen as one of the founders of statistical mechanics and developed a kinetic theory of gases in the late 19th century. Based on the

idea that ideal gases consist of many small particles, a statistical approach can be used to predict the behaviour of a gas on a macroscopical scale by using the particle properties on a microscopical scale. By taking particle probabilities into account instead of doing a statistical analysis of all particles, he formulated the *Boltzmann equation*:

$$\left(\frac{\partial f}{\partial t}\right) = \left.\frac{\partial f}{\partial t}\right|_{\text{collision}} + \left.\frac{\partial f}{\partial t}\right|_{\text{force}} + \left.\frac{\partial f}{\partial t}\right|_{\text{diffusion}} \tag{2.1}$$

in which $f = f(\vec{x}, \vec{v}, t)$ describes the particle density function in the phase space[1], and depends on position $\vec{x}$, velocity $\vec{v}$ and time t. While the *collision* and *force* terms account for internal (particle-particle) and external forces, respectively, the *diffusion* term accounts for the diffusion of particles, and can be formulated more explicit:

$$\left(\frac{\partial f}{\partial t}\right) = \left.\frac{\partial f}{\partial t}\right|_{\text{collision}} + \left.\frac{\partial f}{\partial t}\right|_{\text{force}} + \xi \cdot \left(\frac{\partial f}{\partial \vec{x}}\right) \tag{2.2}$$

with ξ being the average (macroscopic) particle velocity. Defining ξ suitably and neglecting external forces, this equation may be arranged differently:

$$\left(\frac{\partial f}{\partial t}\right) + \xi \cdot \left(\frac{\partial f}{\partial \vec{x}}\right) = \left.\frac{\partial f}{\partial t}\right|_{collision} \tag{2.3}$$

The *collision* term itself is a nonlinear integral differential equation and particularly hard to solve [24]. The BGK[2] collision operator, published in 1954, provides an approximation for small Mach numbers [4] by introducing a single relaxation time τ and the equilibrium distribution function $f^{(0)}$:

$$\left(\frac{\partial f}{\partial t}\right) + \xi \cdot \left(\frac{\partial f}{\partial \vec{x}}\right) = -\frac{1}{\tau}\left[f - f^{(0)}\right] \tag{2.4}$$

[1] The phase space describes all possible states of a physical system. For every degree of freedom in a given system, the corresponding phase space contains one dimension.

[2] Bhatnagar-Gross-Krook

The Lattice Boltzmann method obviously needs a lattice, which is defined by discretizing the velocity space into a finite number of velocities $\vec{e}_i$, leading to a discretization in both space and time:

$$\left(\frac{\partial f_i}{\partial t}\right) + \vec{e}_i \cdot \left(\frac{\partial f_i}{\partial \vec{x}}\right) = -\frac{1}{\tau}\left[f_i - f_i^{(0)}\right] \tag{2.5}$$

Approximating the equilibrium distribution function $f_i^{(0)}$ can be done for non-thermal fluids by doing a Taylor series expansion and choosing an appropriate lattice model[17], providing the last missing element needed for simulation:

$$f_i^{(0)} = \rho \cdot w_i \cdot \left[1 + 3 \cdot (\vec{e}_i \cdot u) + \frac{9}{2} \cdot (\vec{e}_i \cdot u)^2 - \frac{3}{2} \cdot u^2\right] \tag{2.6}$$

In each node, the macroscopic mass density ρ is calculated by summing all particle distributions f_i, and the macroscopic velocity u is calculated by adding all particle densities going in a specific direction, subtracting all particle densities going in the opposite direction, and normalizing the result by the mass density.

It should be noted that the right-hand side of equation 2.5 still denotes the BGK collision operator, which is a relaxation towards a local equilibrium, while the left-hand side describes the streaming operator. Both operations need to be implemented independently at least for solid boundaries [24]. The weighting factors w_i depend on the chosen lattice model (see chapter 2.2.1) and the velocity direction i.

It has been shown that the Navier-Stokes equations can be recovered from equations 2.5 and 2.6, given an appropriate lattice model and some other constraints (see e.g. [5]).

2.2.1 Lattice Models

The standard naming scheme for Lattice Boltzmann models is D*m*Q*n*, in which m denotes the number of dimensions and n denotes the number of discrete velocities in the model. As

representatives for a two- and a three-dimensional model, the D2Q9 and D3Q19 models are shown in figures 2.2 and 2.3. Please keep in mind that the circles denote the directional probability densities f_i, not positions. Each of those structures represent a single lattice cell, with the center point at index zero.

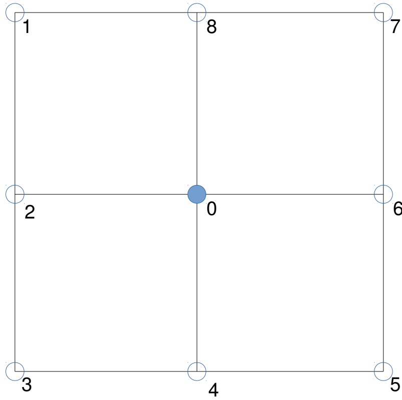


Figure 2.2: D2Q9 model

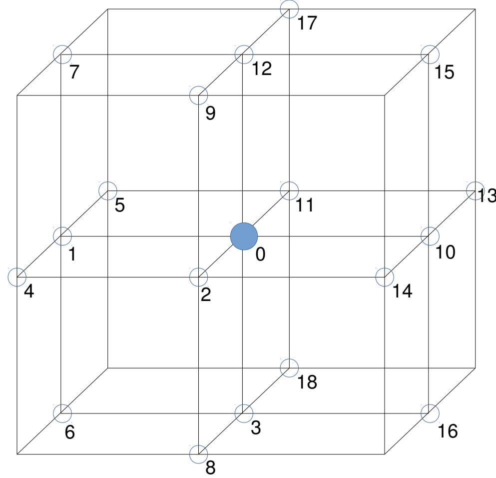


Figure 2.3: D3Q19 model

To be able to implement a Lattice Boltzmann algorithm, some lattice-specific values have to be calculated, namely the discrete velocities e_i and the weights w_i. The former is given in lattice units and is defined by a vector of size m for each of the n velocities, while the latter are calculated from the momentums [4]. In equations 2.7, the solutions for the D2Q9 lattice are shown, and equations 2.8 show them for the D3Q19 lattice. Indices are chosen so that $i + \frac{n-1}{2}$ (for $i = 1 \dots \frac{n-1}{2}$) always refer to the opposite node, which simplifies the implementation.

$$\vec{e}_{i,D2Q9} = \begin{cases} (0,0), & i = 0, \\ (\pm 1,0), & i = 2,6, \\ (0,\pm 1), & i = 4,8, \\ (\pm 1,\pm 1), & i = 1,3,5,7 \end{cases}$$

$$\vec{w}_{i,D2Q9} = \begin{cases} {}^4/_9, & i = 0, \\ {}^1/_9, & i = 2,4,6,8 \\ {}^1/_{36}, & i = 1,3,5,7 \end{cases} \tag{2.7}$$

$$\vec{e}_{i,D3Q19} = \begin{cases} (0,0,0) & i = 0, \\ (\pm 1,0,0) & i = 1,10, \\ (0,\pm 1,0) & i = 2,11, \\ (0,0,\pm 1) & i = 3,12, \\ (\pm 1,\pm 1,0) & i = 4,5,13,14, \\ (\pm 1,0,\pm 1) & i = 6,7,15,16, \\ (0,\pm 1,\pm 1) & i = 8,9,17,18 \end{cases}$$

$$\vec{w}_{i,D3Q19} = \begin{cases} {}^1/_3, & i = 0, \\ {}^1/_{18}, & i = 1..3, 10..12, \\ {}^1/_{36}, & i = 4..9, 13..18 \end{cases} \tag{2.8}$$

2.2.2 Boundary Conditions

Boundary conditions describe the behaviour of a fluid at the boundary of a domain. Since the results of the Chapman-Enskog expansion are only valid for the fluid nodes themselves [10], the macroscopic values might not be correct for wall nodes.

The simplest boundary condition is the *Full Bounce-Back* boundary condition, in which the microscopic particle densities leaving the domain get reflected back unchanged. In this case, the boundary nodes are not considered part of the fluid, since the boundary is located halfway between the boundary nodes and the adjacent fluid nodes. In macroscopic terms,

the boundary behaves as a no-slip, or zero-velocity, wall and is orientation-independent.

For the boundary nodes, only some of the particle densities are unknown, namely those pointing into the domain from the outside. These densities can be calculated by solving a lattice-dependent linear equation system, as shown in 1995 by Zou and He [12][28]. Although their results have been improved since then (cf. Ho et al [13]), they have been chosen for this implementation because of their relative simplicity. The Zou/He boundary nodes are considered a part of the fluid, and provide a suitable boundary for the lid cavity problem.

2.3 Adapteva Epiphany

The hardware used in this thesis is the Parallella board, which is an open platform built around the Adapteva Epiphany E16G3. Started in 2012 as a Kickstarter project, the devices are now available from Adapteva and others [19].

2.3.1 Parallella Board

The Parallella board [20] is a fully open-source credit-card sized computer containing a Xilinx Zynq 7010/7020, an Epiphany E16G3 and 1 GiB of RAM. The Xilinx Zynq is a System-on-Chip (SoC) with two ARM Cortex-A9 processor cores and some reconfigurable FPGA logic and is fully supported by Linux. The board also contains GBit-Ethernet, USB and HDMI interfaces, can boot from a MicroSD card and is able to run the Epiphany SDK. The E16G3 chip is a 16-core implementation of the Epiphany-III architecture.

Adapteva's e-Link interface is implemented using FPGA logic and is used to exchange data between the ARM cores and the Epiphany. By default, a 32 MiB block of memory is shared between both systems and starts at address `0x8e000000`, which translates to mesh coordinates between (35,32) and (36,0). On the Parallella board, the 4x4 grid of processor nodes uses the coordinates between (32,8) and (35,11) inclusive.

To allow for daughter boards, four expansion connectors are provided, allowing access to the power supplies, general I2C, UART, GPIO and JTAG interfaces as well as the northern and southern eLink interfaces to the Epiphany chip.

2.3.2 Mesh Architecture

The architecture reference [1] defines the Epiphany architecture as "a multicore, scalable, shared-memory, parallel computing fabric", consisting of "a 2D array of compute nodes connected by a low-latency mesh network-on-chip". Although designed for good performance in multiple application domains, low power usage is a key factor. At 800 MHz, the E16G3 chip uses less than one watt [2].

This mesh operates in a single shared, flat 32-bit address space. All nodes use a continuous 1 MiB large block of it, so that the first 12 bits of an address specify the node by row and column, with the remaining 20 bits being local to that node. In theory, there can be at most 4096 mesh nodes operating in a single 64x64 mesh.

bits	31..26	25..20	19..0
address	row	column	local

Table 2.1: Mesh address structure

A system of that size would not be practical, however, since without available address space, data exchange to other devices is hard. Also, the node address 0x000 is defined as being the local node, which makes addressing that node from any other node impossible. It is possible to have different types of nodes, although the current implementations only use *processor* nodes.

The mesh itself consists of three meshes with different purposes (see figure 2.4). Read requests travel through the *rMesh*, while *cMesh* and *xMesh* carry write transactions destined for on-chip and off-chip nodes, respectively. To the application, off-chip traffic is indistinguishable from on-chip traffic, apart from lower bandwidth and higher latency. Also, writes are heavily favoured over reads, since reading a foreign address involves sending a read request and waiting for the answer to arrive. Writes, on the other hand, are of a fire-

and-forget type, allowing the node to continue processing, while the data will eventually reach its destination.

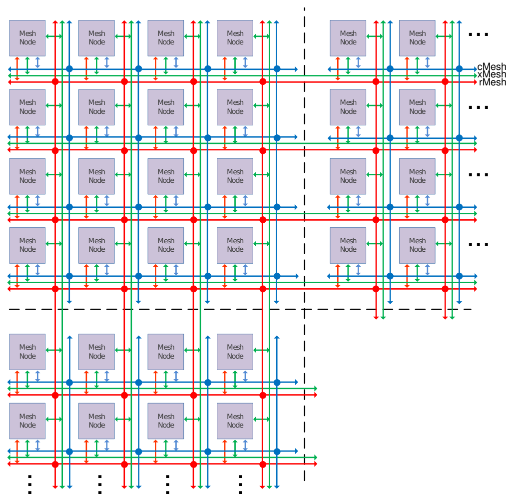


Figure 2.4: Mesh network [1]

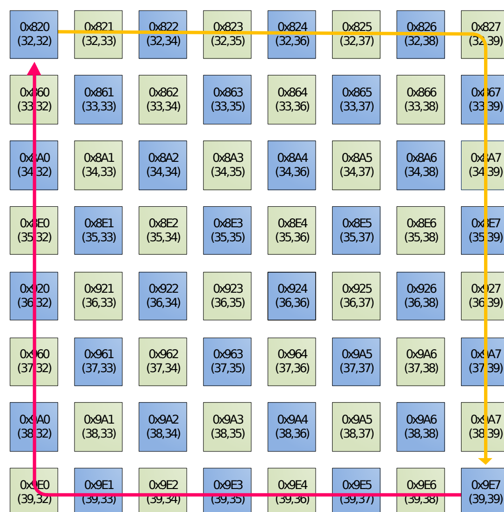


Figure 2.5: Routing example [1]

As a result of the employed *weak memory-order model*, the order of memory transactions is not always deterministic, see table 2.2. If deterministic behaviour is necessary, applications need to synchronize manually.

First Transaction	Second Transaction	Deterministic Order
Read from X	Read from X	Yes
Write to X	Write to X	Yes
Write to X	Read from X	No
Read from X	Write to X	Yes
Read from X	Read from Y	Yes
Read from X	Write to Y	Yes
Write to X	Write to Y	No
Write to X	Read from Y	No

Table 2.2: Weak memory-order model

Routing of traffic follows a few simple, static rules. At every hop, the router compares its own address with the destination address. If the column addresses are not equal, the packet gets immediately routed to the east or west; otherwise, if the row addresses are not equal, the packet gets routed to the north or south; otherwise the packed gets routed

into the hub node, which then is the final destination. A read transaction consists of a read request on the *rMesh*, and a write request on either the *cMesh* or *xMesh*. Figure 2.5 shows how a read transaction is routed through the mesh. The node at (32,32) sends a read request to (39,39), which is routed along the yellow path until it reaches the target node. The target node reads the data and sends the result back to the source of the original request. Because both the row and the column addresses are different, the data will be routed through a different path, shown in red.

When using the *multicast* feature of the mesh, a different routing algorithm is used instead. In this case, the data is sent radially outwards from the transmitting node. All nodes compare the destination address with a local multicast register and if both values are equal (the node is *listening* to that traffic), it enters the node. This feature allows writing to multiple nodes using a single transaction.

2.3.3 Processor Node

Currently, all implementations of the Epiphany mesh architecture only contain *processor* nodes. A processor node contains an eCore RISC CPU, 32 KiB of high-speed local memory, a DMA controller, two event timers, and the obligatory mesh controller.

eCore

Every eCore is a superscalar 32-bit RISC CPU and contains two computational units called Integer ALU (IALU) and Floating-Point Unit (FPU), a register file and a control unit called Program Sequencer. Additionally, an interrupt controller and a debug unit are part of each eCore.

The Program Sequencer handles control flow. It reads instructions from memory to keep the 8-stage pipeline filled, handles jumps and calls and may issue up to two instructions in a single clock cycle. If some restrictions on the code are followed, special zero-overhead loops are also supported by using special loop hardware registers. The interrupt controller may redirect the control flow if interrupts are enabled.

Priority	Name in SDK	Description
0 (high)	SYNC	*Sync* hardware signal asserted
1	SW_EXCEPTION	Invalid instruction, alignment error or FP-exception
2	MEM_FAULT	Memory protection fault
3	TIMER0_INT	Timer 0 expired
4	TIMER1_INT	Timer 1 expired
5	MESSAGE_INT	Message interrupt
6	DMA0_INT	DMA channel 0 finished
7	DMA1_INT	DMA channel 1 finished
8	WAND	*Wired-AND* signal asserted (not defined in SDK 5.13.07.10)
9 (low)	USER_INT	User interrupt

Table 2.3: Interrupts on Epiphany

Fast working storage is provided by the register file. As a RISC architecture, load/store instructions are the only way to move data from or to the memory. All of the 64 registers are 32-bit wide and can be used by both the IALU and the FPU. In every clock cycle, up to nine operations are possible: 4 by the FPU (3 reads, 1 write), 3 by the IALU (2 reads, 1 write) and 2 by the memory interface (one 64-bit read or write).

While the IALU handles one 32-bit integer operation per clock cycle, the FPU executes one floating-point instruction per clock cycle. Integer operations are addition, subtraction, shifts and bitwise operations, as well as data load/store, and these can be scheduled anywhere without risking data-dependency stalls. Also, they are independent of floating-point operations as long as there are no register-use conflicts. Supported floating-point operations are addition, substraction, fused multiply-add, fused multiply-subtract, absolute value, fixed-to-float and float-to-fixed conversions. All of these operate on up to three single-precision operands (32-bit) and conform to IEEE754. The FPU does not support any double-precision operations natively.

The interrupt controller is responsible for handling interrupts and supports nested interrupts. All interrupts have a fixed priority and branch to addresses written in the Interrupt Vector Table (IVT). The IVT is local to every core and is located at the beginning of local memory. All interrupts can be masked independently and are listed in table 2.3. The Wired-AND interrupt can be used to implement efficient barriers for synchronization. However, the current SDK does not use this facility.

Memory

Current implementations of the Epiphany architecture contain 32 KiB of local memory per node, which is divided into four banks of 8 KiB each. Each bank can transfer 64 bits of data per clock cycle, independent of the other banks. There are also four users of local memory: The Program Sequencer reads instructions to keep the core's pipeline filled, the register file and the DMA controller exchange data with the memory, and the network interface may proxy request from the outside world as well.

By carefully distributing code and data among the banks, all four masters can access the local memory simultaneously and, if using doubleword accesses, use the maximum local bandwidth.

The Memory Protection Unit allows the application write-protect parts of local memory, at half-page granularity. Attempts to write to protected memory will then raise a memory protection fault interrupt.

DMA Controller

Every processor node contains a two-channel DMA controller running at the same clock frequency as the CPU. Each channel can be used to copy data between the the local memory and a remote node, or between two remote nodes. Additionally, a channel may be put in a *slave* mode, in which data arriving in a specific register will automatically be transferred to local memory. In this case, the receiving core decides the target address.

Two-dimensional transfers are possible by specifying inner and outer strides in a DMA descriptor, and descriptors can be chained automatically, allowing complex patterns to be transmitted without interrupting the CPU.

Both DMA channels are able to raise their interrupt after every handled descriptor. Also, a remote DMA controller may finish a transfer by raising a local interrupt after the last data item has arrived.

Event Timers

Using the event timers, it is possible to gather realtime data from every node. Each of the two timers may be programmed to count a specific type of event, that is, different kinds of clock cycles, instructions or stalls. The 32 bit counter register may be initialized with any value and will, while the timer is enabled, count down every time the event occurs. When the register reaches zero, the timer stops and an interrupt may be raised.

The Epiphany-IV architecture allows the timers to wrap around at zero and introduces chaining, so that both 32-bit counters may be used as one 64-bit counter instead.

2.3.4 Programming

Adapteva released a Software Development Kit called eSDK [3], which is fully open source. It provides a standard C programming environment based on the GNU toolchain (gcc, binutils, gdb) and the newlib [22] and allows the execution of regular ANSI-C programs on each core.

Additionally, the *Epiphany Hardware Utility Library* (eLib) provides access to architecture specific features, e.g. the DMA or interrupt controllers. Mutex and barrier functions are available as well, simplifying the handling of multiple processors.

To allow the host to easily control the Epiphany, which is used as a co-processor, the *Epiphany Host Library* (eHAL) is available. It provides outside access to the eCores, like loading programs, starting, resetting and more. Also, it provides read and write access to both the cores and the shared DRAM, if available.

On the host, all operating system functions are available, and any programming language able to interface to C libraries may be used to control the Epiphany. The eCores themselves don’t run any operating system, but provide a "bare metal" standard C environment for programming.

3 Implementation

To be able to evaluate the Epiphany architecture in the scope described earlier, the Lattice Boltzmann algorithm has been implemented on the Parallella platform. While this chapter describes the design and the data layouts, appendix A provides a more technical explanation of the code, which also covers the build system used.

Central to the Lattice Boltzmann algorithm is the lattice of nodes, which contains the simulation state of the fluid in the domain. The whole lattice is updated once per iteration (that is, for each time-step), and the data required for e.g. visualization is comparatively cheap to extract from the lattice, although some of the data show up when running the algorithm as well. To simplify the implementation, the lattice itself is treated as both input and output to each iteration.

The memory architecture of the Epiphany system presents some challenges, since both the amount of local memory in each core as well as the external memory bandwidth are very much limited. All communication to the host needs to go through shared memory, which is external to the Epiphany system.

To avoid additional performance problems, the lattice is kept in local memory at all times and only copied to shared memory when necessary[1]. However, doing so severely restricts the achieveable lattice size.

Also, the numerical simulation accuracy will be limited, because the Epiphany architecture currently only supports single-precision floating-point operations natively.

[1]Depending on the simulation requirements, not all iteration results might be relevant, but – apart from benchmarking purposes – at least some will be.

The 32 KiB of local memory available to each eCore are divided into four independent banks, of which the first bank differs slightly by containing the interrupt vector table and startup code. According to Adapteva, code and data should be located in different banks for optimal performance[1]. Following this recommendation, the code is put into the first bank, leaving the remaining banks (24 KiB total) free for the block. Including any libraries, the code size is restricted to slightly less than 8 KiB in this scheme.

To achieve parallelization, the lattice will be divided into *blocks*, where each block is handled by a specific core. Thus, at a fixed block size, additional cores will not decrease the iteration time, but extend the lattice size instead.

A D*n*Q*m* lattice needs to store *m* floating-point values per node, which is 36 bytes in the D2Q9 model or 76 bytes in the D3Q19 model if using single-precision variables. Each block can only contain up to 682 D2Q9- or 323 D3Q19-nodes, and every core only holds a single block.

All nodes in a block can be classified as bulk nodes, if all of their neighbors are in the same block, or boundary nodes otherwise. Boundaries between blocks are called inner boundaries, and only their nodes require communication to other Epiphany cores. Outer boundaries define the edges of the simulation domain, and need special treatment (see chapter 2.2.2).

Together, these restrictions allow a small, but still reasonable implementation of the Lattice Boltzmann algorithm, while still providing relatively large lattices and adhering to Adapteva's guideline at the same time. Also, shared memory is only used to communicate results to the host, minimizing the required external memory bandwidth.

Splitting a two-dimensional lattice spatially in blocks, and mapping those onto the two-dimensional Epiphany mesh is straight-forward. In the three-dimensional case, both the shape and the location of blocks relative to their neighbors can be varied. However, because of the limited number of nodes per block and the small number of cores, the lattice will only be extended in two dimensions. Consequently, all inter-core communication will be next-neighbor only[2].

[2]Data destined for the host will still need to be routed through the mesh, though.

3.1 Data Layout

Each core contains a single *block* in local memory, which itself is a two- or three-dimensional array of nodes. Each node is an array of 9 or 19 floating-point values (see also fig. 2.2 and 2.3). This block is located at the beginning of bank 1, at address `0x2000`. Using a fixed address simplifies accessing nodes in other blocks, which is required for the inner boundaries. Generally, the indices used for coordinates are in reverse order, so that iterating over all nodes in x, y, z-order touches consecutive memory addresses.

The shared memory only contains a single fixed-size data structure (shown in fig. 3.1), which is used to exchange data between the Epiphany system and the host. Both the *timers* and *block* fields are two-dimensional arrays, providing each active core with its own external storage. The timers field is used for instrumentation only and usually contains timer values for different phases of the algorithm. The number of instrumentation entries (called `TIMERS`) is configurable, as are the number of nodes in a block (`NODES`) and the number of active cores (`CORES`). These values are checked at compile- or link-time, if possible.

Synchronization between the host and the Epiphany system is done through the *pollflag* and *iteration counter* fields. Since all cores run in lockstep, synchronizing a single core with the host is sufficient. Except for the *pollflag* field, which is used to synchronize to the host, data is always pushed to shared memory, never read.

field name	**offset (bytes)**	**size (bytes)**
pollflag	+0	4
padding	+4	4
iteration counter	+8	4
padding	+12	4
timers	+16	`CORES` * `TIMERS` * 4
padding		(`TIMERS` % 2) * 4
block		`CORES` * `NODES` * sizeof(node_t)

Figure 3.1: shared memory data structure
`NODES`, `TIMERS` and `CORES` are compile-time constants,
sizeof(node_t) equals 36 (D2Q9) or 76 (D3Q19) bytes

3.2 Algorithm

The functions *collision* and *stream* implement the main part of the algorithm and work on single nodes. These correspond to the collision and diffusion terms in equation 2.1. In a naive implementation, one needs two separate lattices, because when iterating over all nodes, values needed in a later iteration will be overwritten. By employing the "swap trick"[15] (swapping each value with its opposite in both steps) this is avoided. This mode of operation will be called *collision-streaming* approach later.

Still, all nodes have to be touched twice per iteration. To save local memory bandwidth, the optional *bulk* function combines both the collision and streaming steps of the algorithm, accessing each node only once per iteration. However, this is not possible for boundary nodes, which always use the collision and streaming functions. This mode of operation is called *boundary-bulk* approach and benefits from larger block sizes.

Not all iteration results are required, depending on the simulation. If they are, each core copies its block from local memory to shared (external) memory, from where the host can access it. A flag in shared memory is used to synchronize communication between the Epiphany cores and the host, and the barriers provided by the Epiphany SDK synchronize all cores. All cores run in lockstep, sharing the iteration and current phase in processing. After the last iteration has been done, the host is flagged and all cores go idle.

To ensure correctness of the algorithm, it has been implemented with the open-source CFD solver *Palabos*[9] as reference.

3.3 Host Code

The host code runs on on the ARM cores inside the Linux operating system and controls the Epiphany system. After starting the cores, the host busy-waits on the polling flag in shared memory. Unfortunately, the Epiphany SDK currently does not provide a more efficient way of monitoring the cores.

Each time the host is flagged, it copies the shared memory to a local variable and resets the flag immediately afterwards, allowing the cores to continue with the next iteration. Helper functions are available to write the normalized particle densities or velocities to grayscale PNM image files, or dump the raw data into ASCII files. Additionally, it is possible to automatically generate a GIF or MP4 animation from these images.

4 Results

In order to analyze the suitability of the Epiphany system, some experiments using the Lattice Boltzmann implementation have been conducted and analyzed in detail.

Timing results were obtained by having test points in the Epiphany code, in which an event timer is stopped, read, reset and restarted. Each test point incurs an overhead of about 100 clock cycles, and since all experiments used only few (up to ten) test points per iteration, the instrumentation influence of less than 1% was considered negligible.

Both the collision-streaming and the boundary-bulk approaches have been tested. Although the former turns out to be slower in all cases, it carries out the calculation and communication phases separately, providing more insight. However, this does not concern the communication with the host, which is handled separately.

Each test was run for 1000 iterations, with test points after different iteration phases. Each iteration was measured separately, generally resulting in very consistent behaviour (usually less than 1% deviation). Diagrams show averages and – where applicable – standard deviations. The very first iteration, which suffers from some startup delays, is not averaged to avoid bias.

All experiments used the lid cavity test case and either the two-dimensional D2Q9 or the three-dimensional D3Q19 lattices. The primary two-dimensional experiment used a block size of 24x24 and all 16 cores, which were both successively varied. In three dimensions, only two block sizes (3x35x3 and 7x6x7) were used. Additionally, the influence of compiler optimization on code size and execution speed was investigated.

When starting the host program, all cores are reset and loaded sequentially with their code from the host. They immediately start with their own initialization, until they run into their very first barrier and wait there for all other cores to reach that point, too. As it turns out (Fig. 4.1), loading cores[1] with 8 KiB of code takes about 3.8 million (±126.000) clock cycles, or about 90 ms at 700 MHz. The startup delay for the last core is only 390 clock cycles, providing an estimate for the minimal overhead of using a barrier.

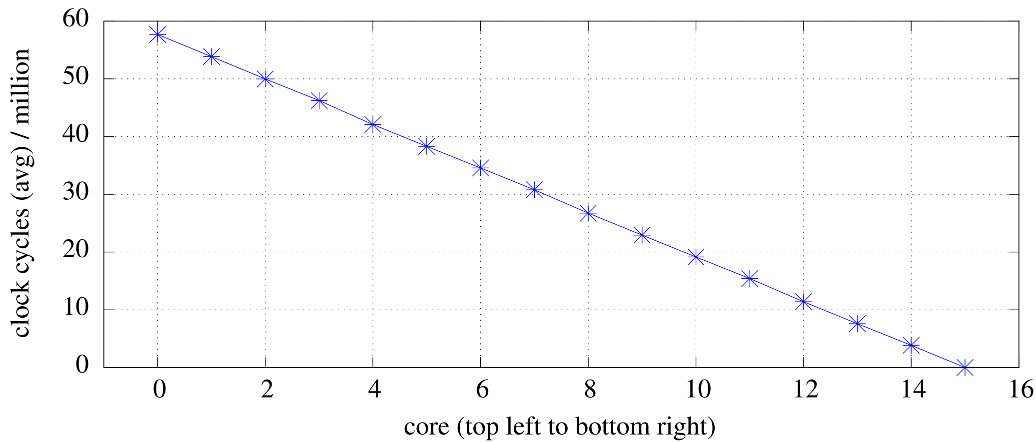


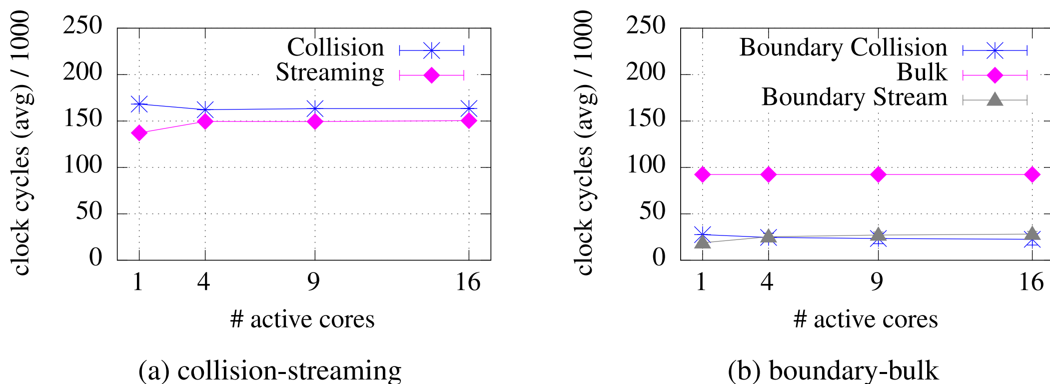
Figure 4.1: Total accumulated startup delay, per core

4.1 Computation

The Lattice Boltzmann algorithm is designed to scale extremely well, which is also true for the Epiphany implementation. Since parallelization is done in space by extending the lattice, the number of nodes in the lattice increases linearly with the number of cores used for the algorithm.

[1]using the e_load() function from the SDK

In the two-dimensional case, varying the number of cores has no effect on the collision phase of the algorithm, while the performance of the streaming phase decreases slightly as soon as multiple cores are involved (Fig. 4.2a). The boundary-bulk approach (Fig. 4.2b) is about twice as fast as the naive approach, although the streaming step – for the boundary only – is still slightly slower with multiple cores. Given the increased work done per iteration, scalability of the algorithm on the Epiphany architecture is excellent.



(a) collision-streaming

(b) boundary-bulk

Figure 4.2: Calculation times (2D, block size 24x24)

For 3D simulations, two different block sizes were studied, but because of code size limitations (see below), only collision-streaming results could be obtained. A block size of 7x6x7 (315 nodes) approximates a cube and maximizes the number of bulk nodes, while a block size of 3x35x3 (294 nodes) maximizes the number of boundary nodes within the limits of hard- and software. Again, the collision times are independent of the number of active cores, while the streaming step takes a small performance hit when going from one to four cores, but stays constant afterwards (Fig. 4.3). Even in 3D, scalability on the Epiphany architecture is excellent.

It also turns out that the time spent on computing the collisions time is independent of the shape of the blocks (although a 7x6x7 block contains fewer nodes than a 3x35x3 block, thus requiring less time to compute). On the other hand, the streaming step suffers less with cubic blocks, since there is less boundary. It should be noted that the 3D lattice is only extended in two dimensions, so all communication is strictly next-neighbor.

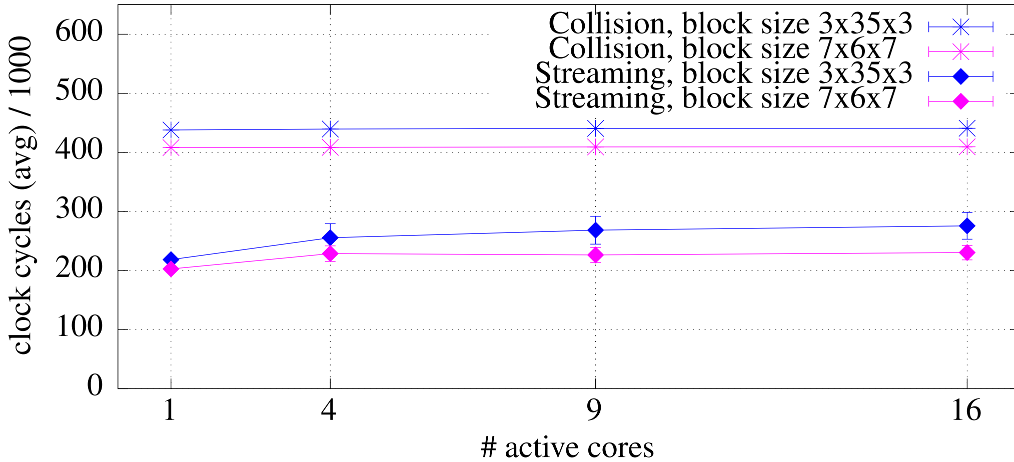


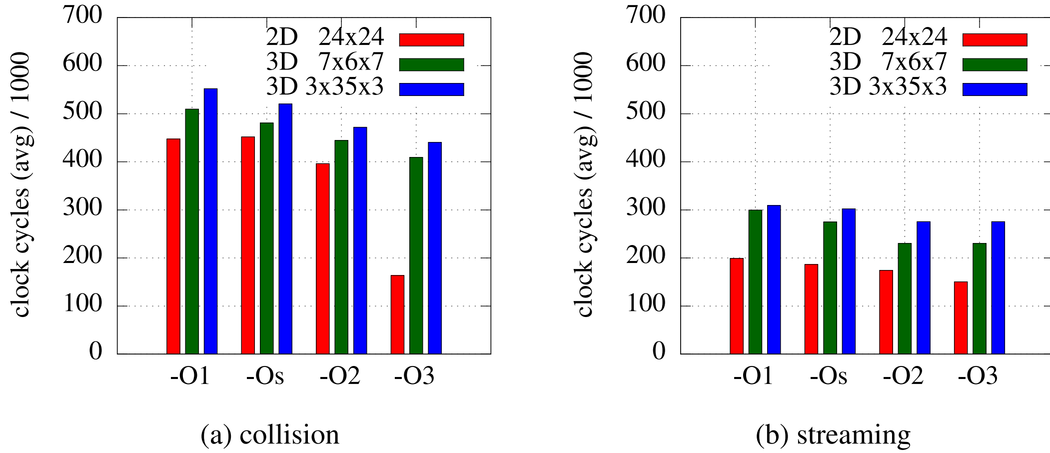
Figure 4.3: Calculation times (3D, varying block sizes)

The compiler optimization level influences both code size and performance. Especially in the 3D cases, the 8 KiB code size limitation turned out to be challenging. Figure 4.4 shows the code sizes in bytes for different compiler settings and implementations. Rows marked with *bulk* contain the additional code for the boundary-bulk approach, and rows marked with *nolibc* have been compiled without the standard C library. Entries marked in bold exceed the available code space.

Implementation	`-O0`	`-O1`	`-Os`	`-O2`	`-O3`
2D	**10212**	6348	6260	6268	7020
2D-bulk	**10604**	6548	6484	6492	7244
3D	**44568**	**9244**	**9436**	**8892**	**9524**
3D-nolibc	**43088**	7732	7548	7276	7960
3D-nolibc-bulk	**43816**	8060	7884	7596	**8244**

Figure 4.4: Code sizes in bytes, different optimization levels,
bold entries exceed the available 8 KiB code space

Figure 4.5 shows the influence of compiler optimization to the performance of the algorithm in the collision-streaming implementation. Generally, higher optimization levels result in slightly faster code, but do not produce huge performance increases – with a single exception. Going from `-O2` to `-O3` is extremely beneficial to the 2D collision step, where a 12% increase in code size results in a 2.4x speedup. However, this difference was not investigated further.

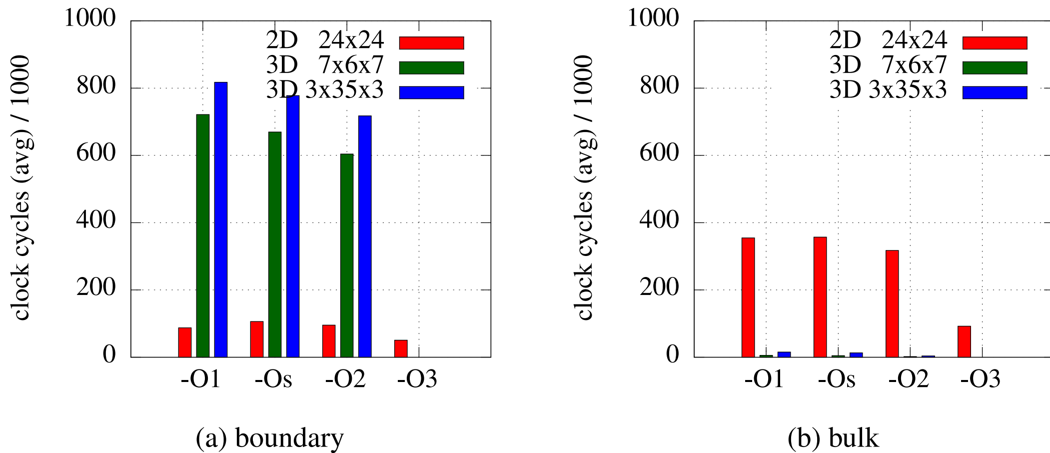


(a) collision

(b) streaming

Figure 4.5: Compiler optimization level influence (collision-streaming)

Looking at the boundary-bulk approach (Fig. 4.6), the highest optimization level gained large performance increases for the 2D simulations, with speedups of 1.9x for the boundary and 3.4x for the bulk computations (2.9x total). However, due to code size limitations (see above), no comparable results exist for the 3D simulations. Generally, the 3D implementations only see a small performance improvement from this approach (11% for 7x6x7 blocks, and 3.5% for 3x35x3 blocks at the `-O2` optimization level compared to the collision-streaming approach). This bulk-based optimization is rendered somewhat inefficient with the smaller block sizes and more involved boundary conditions compared to the 2D case.



(a) boundary

(b) bulk

Figure 4.6: Compiler optimization level influence (boundary-bulk)

At 576 nodes per block (24x24) and optimization level `-O3`, the current implementation needs about 164 000 clock cycles to do the two-dimensional collision steps, or about 285 clock cycles per node. These cycles are used to execute 135 IALU- and 73 FPU-instructions in average, resulting in about 4300 collisions per second. Only 26% of all clock cycles are used for FPU instructions (73% if counting all instructions). In theory, perfect scheduling could at most quadruple the collision throughput[2].

Maximum performance of the two-dimensional simulation has been achieved with the boundary-bulk approach, resulting in about 2.8 MLU/s (millions of lattice node updates per second) for a 24x24 block size on a single core. Together, all cores are able to achieve 45 MLU/s. According to [4], an AMD Opteron at 2.2 GHz reaches about 7 MLU/s using a single thread, although with double precision.

In the D3Q19 models however, the per-core Epiphany performance drops down to about 0.34 MLU/s (boundary-bulk, 7x6x7 block size, `-O2`). If using all 16 cores, 5.4 MLU/s can be obtained, which is about the single-core performance obtained in 2005[27]. In contrast, an NVidia Tesla was shown to achieve up to 650 MLU/s in 2012[11] using the same lattice model at single-precision accuracy.

It should be noted that all comparison results have been made with much larger lattices and included memory bandwidth limitations, while the Epiphany results presented here only used very small lattices stored in on-chip memory.

Since the collision times depend linearly on the number of nodes in each block, the number of iterations per second for a given lattice size can be increased at the cost of requiring more cores. Reaching a 1000x1000 lattice already requires a 42x42 mesh of Epiphany cores (see fig. 4.7), which would require a multi-chip solution and using the slower inter-chip communication paths. To the author's knowledge, no such configurations have been studied yet.

[2] However, in that case, the iteration would be limited by the streaming phase.

block size	collisions per second	cores required (1000x1000 lattice)
24x24	4300	42x42
12x12	17000	84x84
8x8	38000	125x125
6x6	64500	167x167

Figure 4.7: block size influence, ignoring multi-chip overhead, 700 MHz

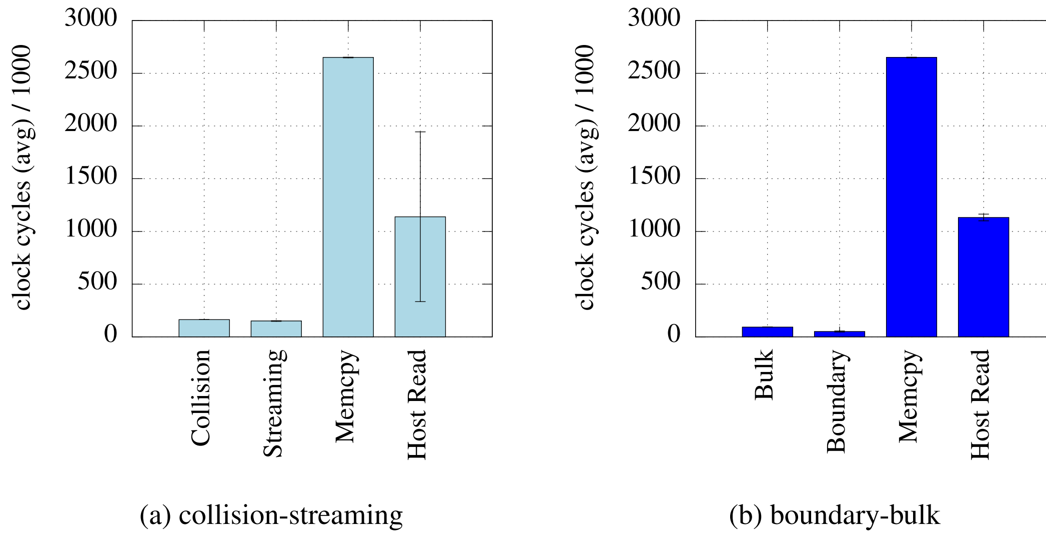
4.2 Shared Memory Access

Except for benchmarking purposes, the results of a simulation are very much relevant as well, often including the results of some or all intermediate steps. In the current implementation, the analysis of the results is left to the host application. To get the data to the host, each core pushes its own block to shared memory, and the host application reads the data from there. Since shared memory is external to the Epiphany (on the Parallella board, it is actually a part of the host RAM), accessing it is very costly[3].

Figure 4.8 puts the 2D calculation times in relation to the time needed to copy the data to the host (called *Memcpy* in all figures). Copying a two-dimensional 96x96 lattice to the host is about 18.5x slower than actually computing one iteration using the boundary-bulk approach. However, this number goes down when comparing against the slower three-dimensional computations.

The host needs to read the lattice from shared memory as well, which is about twice as fast. In cases where not all iteration results are required, multiple buffers could be kept in shared memory to hide this overhead completely. However, this approach breaks down when data is in average produced faster than the host can consume (or store) it. The limited size of local memory prevents similar optimizations in each core, though. In any case, the current implementation does not take advantage of double-buffering. Also, although the host reading time could be overlapped with the next iterations' calculation time, it is not done.

[3]Especially executing code from shared memory should be avoided at all costs. While developing, some math routines accidentally ended up in shared memory, leading to a decreased speed of execution: The attempt was cancelled after 25 minutes had passed without the first iteration finishing. After fixing the problem, iterations took less than a second each.



(a) collision-streaming

(b) boundary-bulk

Figure 4.8: Clock cycles per algorithm step (2D, 16 cores, 24x24 blocks)

When including this Memcpy time in the scalability graph, a total linear dependency becomes obvious, as can be seen in Figure 4.9. This linearity is based on the fact that additional cores increase the number of nodes, and thus the size of the lattice increases linearly as well.

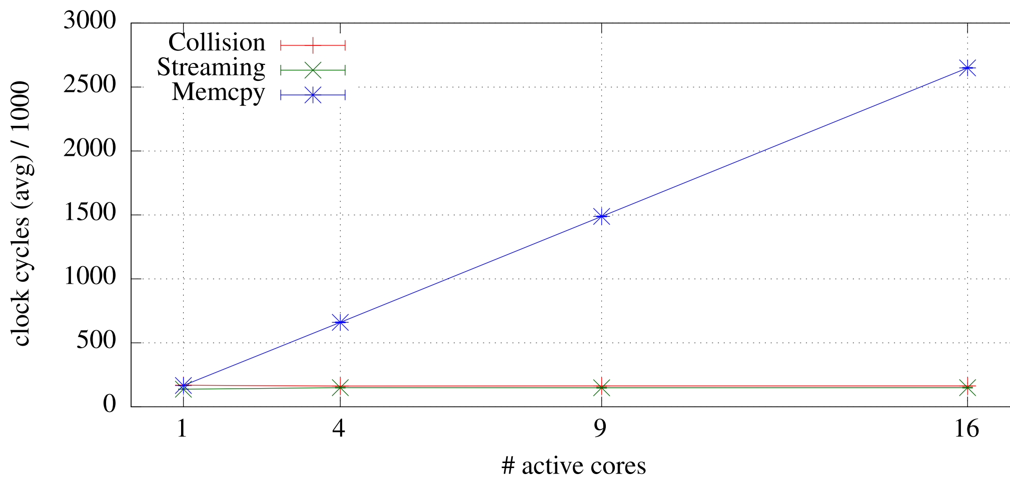


Figure 4.9: Algorithm scalability (2D, block size 24x24)

If the lattice size is kept constant when adding more cores, i.e. decreasing the block size, the time spent copying the results to shared memory becomes constant as well (Fig. 4.9), showing that the memory copy time only depends on size of the data.

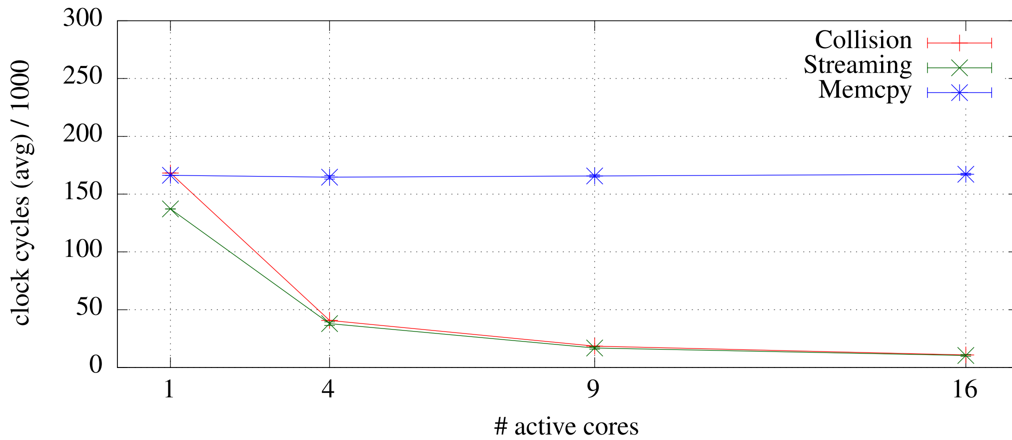


Figure 4.10: Algorithm scalability (2D, lattice size 24x24)

The time spent copying the fixed amount of data to shared memory differs quite a lot between cores[4]. On the Parallella board, shared memory is logically located east of the southernmost core (see also chapter 2.3.1), which is always among the first cores to finish copying. However, the behaviour shown in Figure 4.11 has not been studied in more detail, since the throughput is independent of the number of active cores.

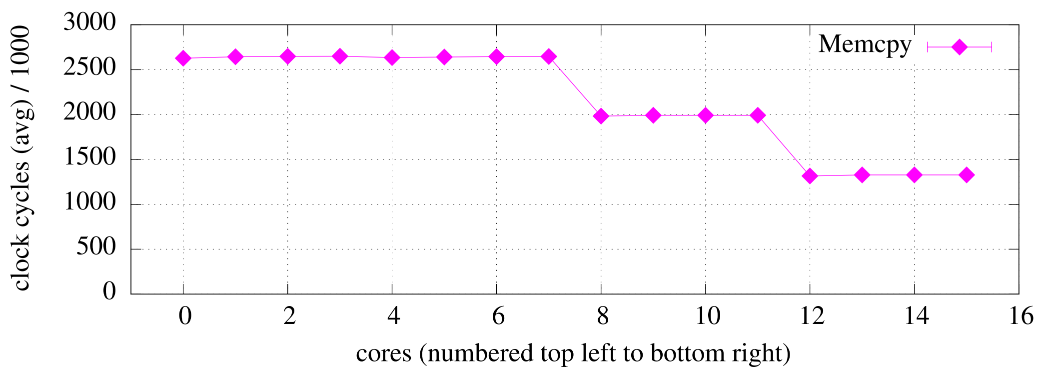


Figure 4.11: Shared memory copy time

[4]So far, the Memcpy times included the synchronization time as well, hiding these differences. However, the times measured for each core show almost no variation and are consistent between runs.

Using the D2Q9 model, the 96x96 lattice contains 324 KiB, or 20736 bytes per 24x24 block. The measurements show that it takes about 2.6 millions of clock cycles to copy the data to shared memory, which at 700 MHz thus exhibits a bandwidth of about 85 MiB/s when using the memcpy-function supplied by the Epiphany SDK. A smaller function, which copies each byte separately, is used in the three-dimensional cases and showed a throughput of 38 MiB/s only.

According to Andreas Olofsson of Adapteva[5], the maximum bandwidth is 600 MiB/s (at 600 MHz), or 200 MiB/s if not using the optimal access pattern, although the used FPGA logic limits the obtainable bandwidth further.

Using the 324 KiB lattice as an example, the current implementation only allows transmission of about 270 results per second to the host. Even though there is still room for improvement, transmitting 1900 lattices per second even saturates the theoretical shared memory bandwidth completely. However, since calculations happen in-place and double-buffering in each core is infeasible, overlaying computation and transmission times is not possible.

Also, since the available bandwidth is already used, it is not possible to store the lattice externally, which would be the only way to increase lattice sizes beyond the limitation imposed by local memory size.

[5]From a post in the Parallella forum written in march 2014, available at `http://forums.parallella.org/viewtopic.php?f=23&t=993&p=6302#p6319`

5 Conclusion

The goal of the master thesis was to explore the efficiency of the Lattice Boltzmann algorithm on Adapteva's Epiphany platform. It has been shown that although the algorithm scales extremely well, both performance and usability are limited by the memory system.

In this implementation, each core calculates the algorithm on a small part of the lattice (block). Due to the scarce local memory, only tiny blocks can be stored in each core, rendering some optimizations impossible or inefficient, especially when using more advanced lattice models. However, the computation throughput in the two-dimensional case looks mildly promising.

Unfortunately, the limited external memory bandwidth makes storing even parts of the lattice in the larger, shared memory infeasible, preventing lattice sizes beyond the local memory size. Already at a few thousand nodes, the time needed to copy the lattice to shared memory overshadows the computation times. Consequently, extending the lattice size is only possible through increasing the number of cores. Since the number of processing units per chip currently is limited, even moderate lattice sizes require a multi-chip solution, which involves the slower inter-chip communication links.

Current generations of the Epiphany architecture do not have native double-precision floating-point support, although this support is likely to appear in the future.

It can be concluded that the Epiphany architecture in its current state is not able to run the Lattice Boltzmann algorithm efficiently unless the problem size is very small and only few iteration results are required.

6 Future Work

The current implementation hard limits the code size per core to slightly less than 8 KiB in order to maximize the number of nodes in local memory. Consequently, several trade-offs were necessary to keep below that limit, especially in the 3D case. Given more room for code space at the expense of block size, further optimizations are possible.

Functionally, the implementation uses older and simpler boundary conditions instead of more accurate, but larger (in terms of code size) ones. Implementing the results of current research (e.g. [13]) should provide improvements to the simulation results. Also, it might be worth investigating the overhead of using double-precision calculations.

This work focused only on a single-chip Epiphany system. It is possible to both extend the number of cores by adding more chips to form a single, larger Epiphany mesh, or to use clustering techniques using multiple Parallella boards. Both approaches might be able to alleviate some of the limitations encountered, although the bandwidth limitations will provide challenges in getting the data in and out of the system efficiently.

The memory handling could be improved. Caching the inner boundaries and using the DMA engines for communication might improve performance. Also, it should be possible to – at least in part – overlap shared memory accesses with the calculation times to utilize the Epiphany further. Further low-level code optimizations (e.g. inlining, loop unrolling) might be beneficial as well. Since all of these examples require additional memory, they will increase performance at the cost of reducing block size.

A Appendix

The two- and the three-dimensional implementations of the Lattice Boltzmann algorithm are independent and although they are very similar in structure, they don't share any code. Each of those implementations contains two applications, one running on the host and one running on all Epiphany cores.

The source code to both implementations is split into four parts each: The build system, a shared header file, a host application and a single Epiphany application, which is run on all cores simultaneously.

A.1 Build System

To build both the host and Epiphany applications together, a reasonably generic `Makefile` is provided. The first part of it specifies important aspects of the building process, which compilers to use, additional parameters for compilation and linking, as well as file and folder names. Only these should need system-specific customization, while the remainder makes up the targets and rules supported.

Generally, variable names starting with the letters *H* and *E* concern the host system and the Epiphany, respectively.

Five directory names are to be specified: *HSRC*, *ESRC*, *HDEST*, *EDEST* and *DEST*. These specify the locations of source files and intermediate output files, split between host and Epiphany code (which must not be shared between both architectures) as well as the final target location, which is used for both.

Only a single host application can be built per project. It is constructed from all object files specified in the *HOBJS* variable and named in *HAPP*. The object files themselves refer to corresponding C files in the host source directory.

It is possible to build multiple Epiphany applications, which are named SREC-files in *EAPPS*, but they may only consist of a single unique C source file each. Sharing code between multiple applications is possible by specifying additional object files in the *ECOMMON* variable, which are then linked into all Epiphany applications. This allows easy integration of shared communication code into the unique kernels.

The Makefile contains the targets *host*, *target*, and *all* to build the host application, Epiphany applications, or both. For convenience, the targets *run* (build all, then run host application), *clean* (clean all intermediate and final output files), and *help* (show target list and explanations) are also provided. If the environment is set up correctly (cf. [3]), executing *make run* should be enough to get started.

A.2 Shared Header File

The file `shared.h` is the only file used by both the host and the Epiphany compiler, and contains global configuration values and data structures. It is necessary that all data structures described in this file have the same in-memory data layout even across architectures, so they should be marked with an explicit alignment.

Here, the number of cores to use (in two dimensions) is specified, as well as the number of nodes stored in each core (in two or three dimensions). Additionally, the maximum number of timing values used for measurements is given here.

Every node is stored as an array of 9 floating-point values for the D2Q9 lattice model, and 19 floating-point values for the D3Q19 model, called *node_t*, and combined into a two- or three-dimensional array *block_t*.

The shared memory structure *shm_t* consists of a pollflag field, an iteration counter, an array of timing measurements and an array of blocks. Both arrays reproduce the two-dimensional structure of the Epiphany mesh, and each core only writes to their own entries. Scalar entries are only written by a single, designated core.

A.3 Epiphany

Each Epiphany core only handles a part of the lattice, called a *block*, which is configurable in size. Blocks may not be larger than the available memory in each core, and each dimension has a minimum size of three. Where possible, the configuration values are sanity-checked at compile or link time.

The functions *init*, *collision*, *stream* and *bulk* are implemented in `d2q9.c` or `d3q19.c` and contain the algorithm itself. The API to those functions is defined in an accompanying header file.

To provide sensible initial values to the algorithm, the *init* function initializes all nodes at their equilibrium with a uniform density. Since memory management for the data banks is handled manually, the values would come up undefined otherwise (they are not implicitly zeroed). In real applications, the initial values should probably be read from shared memory instead, leading to a larger start-up time.

Zou/He boundary conditions and the BGK operator are implemented in the *collision* function. The top boundary($y = 0$ in 2D, or $z = 0$ in 3D) can be assigned a velocity vector. All other boundaries implement no-slip walls with zero velocity, and corners are assigned to one of the adjacent boundaries, to save code space. Particle probability densities are swapped with their opposite directions. This step represents the particle-particle interactions, and can also introduce external forces on the top boundary.

The *stream* function represents the diffusion of particles in space. Inner borders (between blocks) are transparently extended in the *xy*-plane in 2D, or the *xz*-plane in 3D by using the shared address space on the Epiphany. In this function, particle properties are again swapped with their opposite directions. Doing these swaps allows running the algorithm in-place, which saves 50% of the data size compared to a naive implementation.

Finally, the *bulk* function is a faster variant combining the functionality of both the *collide* and *stream* functions, which only iterates through the block once[15]. To allow parallel execution, all boundary nodes (including the inner boundaries) need to be processes separately using the two functions described above. Consequently, this function does not need special-casing of boundaries.

The main logic of a core is implemented in `main.c`, which ties everything together. Each core contains a single *block_t*, which is manually allocated at the fixed address `0x2000`. Dummy variables of 8192 bytes each are put into each data bank to make the linker avoid them. This simple solution has two side effects: The linker will complain loudly if the code does not fit the first bank, and the SREC file will contain 24 KiB of unnecessary zeros, which are filtered as part of the build process.

Also, each core puts exactly one single variable in shared memory (of type *shm_t*, so that the address of it is known to the host. Since shared memory is shared among all cores, each core only accesses its own part of the structure to avoid collisions; shared fields are handled by the top-left core. The corresponding section is also filtered from the SREC file when building to avoid overwriting shared memory when loading cores.

At startup, every core fetches its own coordinates and writes them into global variables, which are used by the algorithm to differentiate inner from outer borders, initializes its block, and runs a predefined number of simulation iterations. Barriers, as provided by the Epiphany SDK, are used to synchronize all cores when needed, and time intervals are measured and stored in a local array.

If the result is to be sent to the host, which does not necessarily happen after each iteration, each core copies its block and timer values to shared memory, the top-left core writes the iteration counter, sets the polling flag and waits until the host tells it to continue. Again, barriers are used to synchronize. After all iterations are done, the host is again flagged, and all cores go idle.

The more complicated algorithm in the 3D case results in larger code, which exceeds the size of bank 0. Since the C library is not used, except for *memcpy* and *memset*, the 3D subproject does not link against it. The missing functions, as well as some symbols required by the Epiphany startup code, are provided by an extra C file. As a consequence, *main* should never return.

A.4 Host

The host program running on the Linux kernel on the ARM processor, is responsible for handling the Epiphany system, as well as reading and interpreting the data produced by the lattice Boltzmann algorithm.

After resetting and initializing the Epiphany system, opening a workgroup and allocating (and initializing) some shared memory, all reserved cores are loaded with the same SREC file. Since the current Epiphany SDK does not allow signalling the host from Epiphany code, the host polls the first field of the shared memory structure for changes. If the Epiphany signals being done, the host program does some cleanup and ends.

Otherwise, an *shm_t*-sized chunk of memory is copied from the shared memory into a variable and the flag is immediately reset. While the Epiphany calculates the next time step(s), the host program then handles the data it just received.

It is possible to write the normalized particle densities or velocities into grayscaled PNM image files, or to write the raw data and timer values into ASCII text files, depending on the use case. Also, the generated images may be converted to a GIF animation of MP4 video file using external applications (ImageMagick[14] or ffmpeg[8], respectively).

B References

[1] Adapteva. Epiphany Architecture Reference. http://adapteva.com/docs/epiphany_arch_ref.pdf, 2014. Rev 14.03.11.

[2] Adapteva. Epiphany E16G3 Datasheet. http://adapteva.com/docs/e16g301_datasheet.pdf, 2014. Rev 14.03.11.

[3] Adapteva. Epiphany SDK Reference. http://adapteva.com/docs/epiphany_sdk_ref.pdf, 2014. Rev. 5.13.09.10.

[4] Alexander Dreweke. Implementation and Optimization of the Lattice Boltzmann Method for the Jackal DSM System. Bachelor thesis, Friedrich-Alexander-Universität, Erlangen-Nürnberg, 2005.

[5] S. Chen and G. D. Doolen. Lattice Boltzmann method for fluid flows. *Annual review of fluid mechanics*, 30(1):329–364, 1998.

[6] Clay Mathematics Institute. Millennium Problems. http://www.claymath.org/millennium-problems.

[7] G. Crimi, F. Mantovani, M. Pivanti, S. F. Schifano, and R. Tripiccione. Early experience on porting and running a lattice boltzmann code on the xeon-phi co-processor. *Procedia Computer Science*, 18:551–560, 2013.

[8] FFmpeg. FFmpeg. http://www.ffmpeg.org/.

[9] FlowKit.com. Palabos Home. http://www.palabos.org/.

[10] FlowKit.com. Palabos LBM Wiki » models:bc. http://wiki.palabos.org/models:bc.

[11] J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein. Performance engineering for the lattice Boltzmann method on GPGPUs: Architectural requirements and performance results. *Computers & Fluids*, 80:276–282, 2013.

[12] X. He and Q. Zou. Analysis and boundary condition of the lattice Boltzmann BGK model with two velocity components. *arXiv preprint comp-gas/9507002*, 1995.

[13] C.-F. Ho, C. Chang, K.-H. Lin, and C.-A. Lin. Consistent boundary conditions for 2D and 3D lattice Boltzmann simulations. *Computer Modeling in Engineering and Sciences (CMES)*, 44(2):137, 2009.

[14] ImageMagick Studio LLC. ImageMagick: Convert, Edit, Or Compose Bitmap Images. http://www.imagemagick.org/.

[15] J. Latt. Technical report: How to implement your DdQq dynamics with only q variables per node (instead of 2q). Technical report, Tufts University, 2007.

[16] W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19(7-8):444–456, 2003.

[17] R. Mei, W. Shyy, D. Yu, and L.-S. Luo. Lattice Boltzmann method for 3-D flows with curved boundary. *Journal of Computational Physics*, 161(2):680–699, 2000.

[18] K.-i. Nomura, S. W. de Leeuw, R. K. Kalia, A. Nakano, L. Peng, R. Seymour, L. Yang, and P. Vashishta. Parallel lattice boltzmann flow simulation on a lowcost playstation 3 cluster. *International Journal of Computer Science*, 2008.

[19] Parallella. Parallella | Supercomputing for Everyone. http://www.parallella.org.

[20] Parallella. Parallella Reference Manual. http://www.parallella.org/docs/parallella_manual.pdf. Rev. 13.11.25, preliminary.

[21] L. Peng, K.-i. Nomura, T. Oyakawa, R. K. Kalia, A. Nakano, and P. Vashishta. Parallel lattice Boltzmann flow simulation on emerging multi-core platforms. In *Euro-Par 2008–Parallel Processing*, pages 763–777. Springer, 2008.

[22] Red Hat. The Newlib Homepage. `http://sourceware.org/newlib/`.

[23] S. Savas, E. Gebrewahid, Z. Ul-Abdin, T. Nordström, and M. Yang. An Evaluation of Code Generation of Dataflow Languages on Manycore Architectures. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug. 2014.

[24] M. Sukop. *Lattice Boltzmann modeling an introduction for geoscientists and engineers*. Springer, Berlin New York, 2006.

[25] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39, 2010.

[26] J. Ward-Smith. *Mechanics of Fluids, Eighth Edition*. CRC Press, 2005.

[27] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*, 35(8):910–919, 2006.

[28] Q. Zou and X. He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids (1994-present)*, 9(6):1591–1598, 1997.