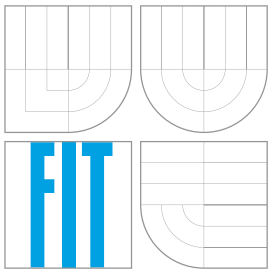
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

ACCELERATION OF LATTICE-BOLTZMANN ALGORITHMS FOR BLOODFLOW MODELING

AKCELERACE ALGORITMŮ LATTICE-BOLTZMANN PRO MODELOVÁNÍ TOKU KRVE V MOZKU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. RADMILA KOMPOVÁ

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. JIŘÍ JAROŠ, Ph.D.

BRNO 2016

Zadání diplomové práce/18589/2015/xkompo00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Kompová Radmila, Bc.**
Obor: Bioinformatika a biocomputing
Téma: **Akcelerace algoritmů Lattice-Boltzmann pro modelování toku krve v mozku**
Acceleration of Lattice-Boltzmann Algorithms for Bloodflow Modeling
Kategorie: Algoritmy a datové struktury

Pokyny:

1. Seznamte se s architekturou moderních vícejádrových procesorů. Zaměřte se především na organizaci vyrovnávacích pamětí a vektorových rozšíření typu AVX.
2. Prostudujte simulační techniku Lattice-Boltzmann určenou pro simulaci proudění kapalin a prostudujte její dostupné implementace. Zaměřte se především na simulační software HemeLB.
3. Analyzujte výkonnost současné verze HemeLB a identifikujte časově kritické části.
4. Navrhněte úpravy, které povedou ke zvýšení výkonnosti. Zaměřte se především na efektivní reprezentaci dat a vektorizaci výpočtu.
5. Navržené úpravy realizujte a změřte jejich dopad na výkonnost.
6. Zhodnoťte dosažené výsledky a diskutujte přínos práce.

Literatura:

- Wittmann, M., Zeiser, T., Hager, G., Wellein, G.: Comparison of different propagation steps for lattice Boltzmann methods, Computers & Mathematics with Applications, vol 65(6), s. 924-935, 2013.
- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Jaroš Jiří, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstract

This thesis aims to explore possible implementations and optimizations of the lattice - Boltzmann method. This method allows modeling of fluid flow using a simulation of fictive particles. The thesis focuses on possible improvements of the existing tool HemeLB which is designed and optimized for bloodflow modeling. Several vectorization and paralellization approaches that could be included in this tool are explored. An application focused on comparing chosen algorithms including optimizations for the lattice - Boltzmann method was implemented as a part of the thesis. A group of tests focused on comparing this algorithms according to performance, cache usage and overall memory usage was performed. The best performance achieved was 150 millions of lattice site updates per second.

Abstrakt

Tato práce se zabývá implementací a možnými optimalizacemi metody lattice - Boltzmann. Tato metoda umožňuje modelovat tok kapalin pomocí simulace pohybu fiktivních částic. Práce se zaměřuje na možná vylepšení existujícícho nástroje HemeLB, který se specializuje na simulaci proudění krve v mozku. V práci jsou mimo jiné zkoumány techniky vektorizace a paralelizace jejichž implementace by mohla pro tento nástroj být přínosná. Součástí práce je implementace aplikace srovnávající několik vybraných algoritmů pro metodu lattice - Boltzmann včetně jejich možných optimalizací. Zahrnuty jsou rovněž testy zaměřené na srovnání těchto algoritmů dle dosaženého výkonu, využití paměti cache a celkové spotřeby paměti. Nejlepší dosažený výkon byl 150 milionů aktualizovaných bodů mřížky za sekundu.

Keywords

Lattice-Boltzmann method, bloodflow modeling, HemeLB, algorithm acceleration, vectorization, parallelization, OpenMP

Klíčová slova

Lattice-Boltzmann metoda, modelování toku krve, HemeLB, akcelerace algoritmů, vektorizace, paralelizace, OpenMP

Reference

KOMPOVÁ, Radmila. *Acceleration of Lattice-Boltzmann Algorithms for Bloodflow Modeling.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Jaroš Jiří.

Acceleration of Lattice-Boltzmann Algorithms for Bloodflow Modeling

Declaration

I declare, that this thesis is my original work which I created under the leadership of Ing. Jiří Jaroš Ph.D.

.....
Radmila Kompová
May 23, 2016

Acknowledgements

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project „IT4Innovations National Supercomputing Center – LM2015070“

Contents

Chapter 1

Introduction

Fluid dynamics simulation has a wide range of possible applications in several areas including industry or medicine. Fluid simulation can be very useful in many cases but it still remains a computationally intensive task even with a constant increase of computers performance. One of the methods that can be used, is lattice - Boltzmann method. This method simulates the fluid flow in a discretized space represented by a grid consisting of individual lattice sites. The state of the fluid in the lattice site in the current time step is defined by a probabilitity that a virtual particle with a defined velocity currently resides in that lattice site.

In some applications, for example simulation of bloodflow for medical purposes, time of the simulation is crucial. Even when using multi - core architectures, the computational costs are still high, especially when modelling fluid flow in complex geometries. Therefore, it is important to decrease the simulation time by using a range of optimizations.

The goal of this thesis is to explore the possible optimizations of the lattice - Boltzmann method including the use of several algorithm modifications, addressing layouts, vectorization and parallelization. The performance will be measured on two supercomputers - Anselm (nodes with two 8 core Intel Sandy Bridge processors) and Salomon (nodes with two 12 core Intel Haswell processors).

Several possibilities of implementation and optimization of the lattice - Boltzmann method have been explored and described in the text. One of the existing tools, which specializes on bloodflow modeling, called HemeLB was also explored. An application focused on comparing several lattice - Boltzmann algorithms with various optimizations was implemented. The application was tested with a number of different settings and the algorithms were compared according to performance, cache usage and overall memory usage.

Chapter 2 covers a general introduction to the lattice - Boltzmann method, an overview of the basic boundary counditions (2.2) and the BGK collision operator (2.1). Chapter 3 focuses on the lattice - Boltzmann method implementation and describes the basic addressing layouts, the most common algorithms and the implemented application. The measurements comparing the application performance for different domain sizes and number of available processors are also a part of this chapter. Chapter 4 includes an overview of the available parallel computing techniques including SIMD, vectorization and OpenMP. Chapter 5 focuses on the description of the HemeLB and its capabilities. Chapter 6 describes implementation of the application focused on comparing several lattice - Boltzmann algorithms and its possible optimizations. Chapter 7 gives information about performed tests and comparison of the algorithms according to performance, cache usage and overall memory usage.

Chapter 2

Lattice - Boltzmann method

Lattice - Boltzmann method (LBM) is a method for fluid dynamics simulation. The simulated domain is discretized and represented by a grid consisting of lattice sites. To model the flow, the method uses virtual particles. The current state of the fluid in a particular lattice site is described by a particle distribution functions. The particle distribution functions express the possibility that a particle with a specific velocity resides in the lattice site. The LBM consists of two main parts, propagation and collision. During the collision step, the particle distribution functions are updated according to the collision operator used. In the propagation step, the values of the particle distribution functions are propagated to the neighbouring lattice sites.

From a historical perspective, lattice - Boltzmann method has been evolved from lattice - gas automata (LGA) [9]. LGA comprises a lattice, where the lattice sites can be in a certain number of different states. The various states express the number of the particles with certain velocities currently present in the lattice site. The states are boolean, meaning that there either is or is not a particle moving in each defined direction. The main difference is that LBM use discretized probability distribution functions for a particle in each lattice point so it does not clearly state that a particle either is or is not in the lattice site. It has been proved that the lattice - Boltzmann equation (2.3) can be directly derived from the continuous Boltzmann equation by discretizing this equation in both time and phase space in a specific way [14]. The Boltzmann equation (2.1) describes a change of a particle distribution function f in time t, ξ is the microscopic velocity, λ is the relaxation time due to collision, and g is the Boltzmann - Maxwellian distribution function.

$$\frac{\partial u}{\partial t} + \xi . \bigtriangledown f = -\frac{1}{\lambda}(f - g) \tag{2.1}$$

According to the type of the used lattice system, LBM can be characterized using a $D\alpha Q\beta$ notation. In this notation, α stands for the space dimension (usually two or three) and β for the number of particle distribution functions (PDFs). β is defined by number of links from each lattice site to its neighbour lattice sites. The considered lattice site itself is included in this number as well, therefore the number of the links to its neighbours is actually $\beta - 1$. The length of the links is typically either equal to the lattice spacing r (for the closest neighbour sites having a difference in only one coordinate in the Cartesian grid), $\sqrt{2}r$ for the diagonal links or $\sqrt{3}r$ for the more distant diagonal links. $D2Q9$ or $D3Q19$ belong to the most commonly used lattice models [26]. Picture 2.1 shows the links to the neighbouring lattice sites in the $D2Q9$ and $D3Q19$ models.

The LBM uses virtual particles that can have a certain number N of possible velocities

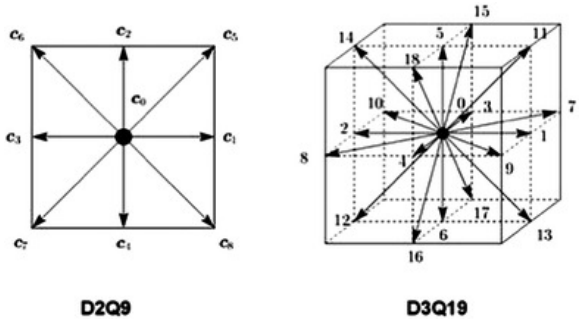


Figure 2.1: D2Q9 and D3Q19 models [15]

$c_i, i = 0, 1..N$ to describe the current state of the fluid at a specific time. The number N is defined by the number of links to the neigbors of the lattice site. The state of the fluid in the lattice point x at the time t is described by means of a distribution function $f_i(x,t)$. This function describes the probability that a particle with the velocity c_i is in the point x at the time t [20]. The set of possible velocities for a particle is determined by the chosen system of links to the neighbouring lattice sites. Formula 2.2 shows velocities for the $D2Q9$ model with 8 links to the neighbouring lattice sites [19].

$$c_\lambda = \begin{cases} (0,0) & \lambda = 0 \\ (r,0),(-r,0),(0,r),(0,-r) & \lambda = 1,2,3,4 \\ (r,r),(r,-r),(-r,r),(-r,-r) & \lambda = 5,6,7,8 \end{cases} \tag{2.2}$$

The lattice - Boltzmann equation (2.3) then describes how $f_i(x,t)$ is changed after a time step Δt, where Ω is the collision operator [20].

$$f_i(x + c_i\Delta t, t + \Delta t) = f_i(x,t) + \Omega(f_i(x,t)) \tag{2.3}$$

Several collision operators have been formulated, for example Bhatnagar–Gross–Krook (BGK), two relaxation time scheme (TRT), or multiple relaxation time scheme (MRT) [26]. The algorithm of the lattice - Boltzmann method can be divided to three main steps.

1. **Propagation:** In this step, values of the particle distribution functions f_i are shifted according to their velocities c_i to the corresponding adjacent lattice sites. This action can be described by equation 2.4.

$$f_i(x + c_i\Delta t, t + \Delta t) = f_i(x,t) \tag{2.4}$$

2. **Collision:** In the collision step, the values of the particle distribution functions are updated according to the collision operator used. The output ${f_i}^{out}$ of this step can be described by equation 2.5.

$${f_i}^{out} = {f_i}^{in} + \Omega(f_i(x,t)) \tag{2.5}$$

3. **Evaluation of macroscopic variables:** In this step, several macroscopic variables are computed and stored for an application in the next collision step. The fluid density can be computed from the particle distribution functions using formula 2.6.

$$\rho(x,t) = \sum_{i=0}^{N-1} f_i(x,t) \tag{2.6}$$

Macroscopic velocity is computed by using equation 2.7, where ρ is the fluid density in a particular lattice site. Depending on the collision operator used, there can be several other macroscopic variables that are computed in this step.

$$u(x,t) = \frac{1}{\rho} \sum_{i=0}^{N-1} f_i(x,t) c_i \tag{2.7}$$

Described steps do not have to be strictly separated and their order can also be modified. For example, evaluation of the macroscopic variables can be performed before the collision step. Frequently, evaluation of macroscopic variables is not even considered as a separated step and this computation is performed as a part of the collision step. The order of the propagation and collision step can also be switched or these steps can be merged into one single step. LBM algorithms can then be categorized as having either one - step or two - step approach.

2.1 BGK

One of the simplest collision operators which can be used for LBM is the BGK collision operator, where BGK stands for the abbreviation of names of its authors - Bhatnager, Gross and Krook. BGK can be described as a linear approximation of the collision integral from the Boltzmann equation, therefore determination of the post - collisional state of the fluid is simplified while the crucial constraints for collision operator are still fulfilled [11].

The first constraint demands the certainity of the conservation of the collisional invariants. The second constraint states that the system evolves towards the local equilibrium condition. The BGK collision operator can be described by formula 2.8 where f_i^{eq} stands for the local Maxwellian equilibrium distribution function and τ is the relaxation time constant [19].

$$\Omega(f_i(x,t) = \frac{1}{\tau}(f_i^{eq}(x,t) - f_i(x,t)) \tag{2.8}$$

Fulfillment of the second constraint is here ensured by the fact that the collision operator is applied to the PDF, which is then changed by an amount proportional to the difference between the value of this PDF and the local equilibrium [11]. Local equilibrium distribution function f_i^{eq} can be computed using equation 2.9, where w_i is determined by the system of links to the neighbouring lattice sites [9].

$$f_i^{eq}(x,t) = w_i \rho (1 + 3uc_i + \frac{9(uc_i)^2}{2} - \frac{3u^2}{2}) \tag{2.9}$$

The condition decribed by equation 2.10 has to be fulfilled for all models.

$$\sum_{i=0}^{N-1} w_i = 1 \tag{2.10}$$

In general it can be stated, that the farthest neighbouring lattice sites have the lowest weights assigned. In case of model $D2Q9$ the weights are expressed by formula 2.11.

$$w = \begin{cases} w_{i=0} & 4/9 \\ w_{i=1,2,3,4} & 1/9 \\ w_{i=5,6,7,8} & 1/36 \end{cases} \tag{2.11}$$

2.2 Boundary conditions

For simulation of fluids in real non - infinite environment, it is crucial to have a way in which the boundaries of the space are defined as one of the inputs to the LBM algorithm. The fluid dynamics near the specified boundaries then has to be handled in an appropriate way in the implemented algorithm. Therefore„ the boundaries are specified by boundary conditions. There are several boundary conditions that differ in complexity and accuracy. The simplest one is a bounce - back boundary condition that can have several variants. In all of them the collision process does not occur at the lattice sites within the solid boundary.

2.2.1 Fullway bounce - back

The most straightforward variant of the bounce - back is a fullway bounce - back condition. In this case, the direction of the velocity is reversed for a particle, which is, according to its velocity, supposed to enter the lattice site within a solid boundary. Using this method, two time steps are needed for the particle to reach the boundary, flip direction of its velocity and go back to a regular lattice site. Implementation of this boundary condition is simple but it has been shown that the accuracy which can be achieved by this approach does not have to be sufficient in case that a high precision of the algorithm results is demanded [11].

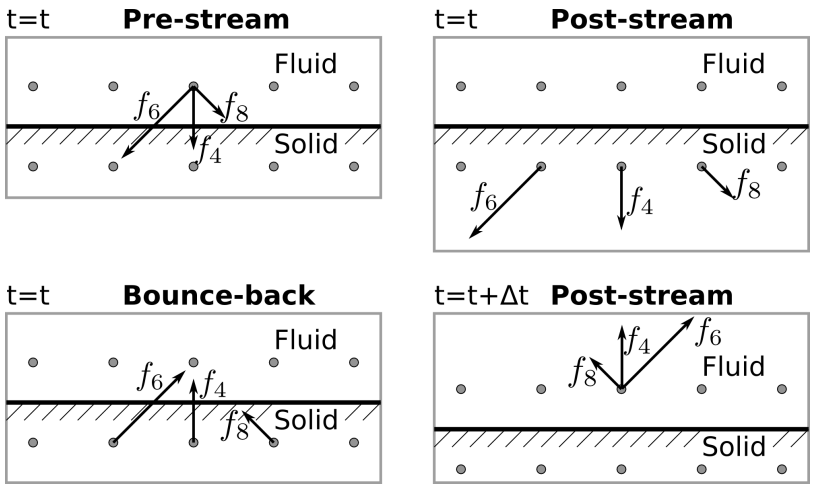


Figure 2.2: Halfway bounce - back [10]

2.2.2 Halfway bounce - back

Better accuracy has been observed for halfway bounce - back. In this case, the boundary is considered to be placed in the center between two rows of the lattice sites. Thanks to this modification, only one time step is needed for reversing the direction of the particle velocity in the corresponding direction [11]. Therefore, the boundary condition can be applied as a part of the propagation step, which makes it straightforward for integration. If a particle is about to enter a non - fluid lattice site during the propagation step, then instead of the standard propagation the PDFs are updated in a described way. The steps of the bounce - back rule application are demonstrated in figure 2.2.

Chapter 3

Implementation of the lattice - Boltzmann method

There are several variants how to implement LBM. Although the essential characteristics of this method remain the same, significant differences in performance and memory usage can be observed. No matter which particular variant is chosen, the main steps always have to be performed in some way. If the collision and propagation are fused, the algorithm is called one - step algorithm, otherwise they are referred to as two - step algorithm. The main challenge arising from the implementation of various optimizations of the basic algorithm is the data dependence between the adjacent lattice sites in the propagation step.

Another observed parameter is memory requirements. The main difference lies in the fact whether the memory to store values of the particle distribution functions has to be allocated only once or twice. Based on this condition, the algorithms are categorized as one - grid or two - grid ones. Each algorithm uses various number of additional variables but as the size of the domains modeled by LBM is usually large, these variables are usually not considered as significant for performance.

Regardless of the particular algorithm, data which has to be stored is determined by the essence of the LBM. The main structure is a grid containing individual lattice sites. The size and dimension of the grid are determined by several parameters, including the size of the particular domain and expected precision. For each lattice site, the values of the corresponding PDFs have to be stored. These requirements lead to the necessity of storing data in a generally multidimensional array.

The array can be either allocated directly as a multidimensional array or the multidimensional data can be mapped into one direction and stored in a one - dimensional array. In the latter case, an enumeration function has to be implemented to access the lattice sites [20]. This approach offers more flexibility in how exactly data will be stored, which can be convenient for improving memory and cache utilization.

Several data layouts for PDFs storage can be implemented depending on the exact variant of the implemented algorithm. The values of the PDFs are then accessed using access functions where a site index is used as a parameter. For example, data layouts which are optimized specifically for the collision and streaming steps have been developed. In the collision layout, the values of the PDFs are stored in a natural way subsequently to each other together for each lattice site. With the streaming layout, the values of the PDFs are grouped according to the direction of the velocity of each particle distribution function and these groups and values within each group are then stored together. Another interesting

data layout is a bundle layout, where values of the PDFs are sorted into several groups of the same size, but unlike the streaming layout, each group contains particles with more than one direction, which then leads to a lower number of groups [20].

3.1 Addressing layouts

The values of PDFs can be stored and accessed using several addressing layouts. Which layout is the best choice depends on several factors, including the algorithm used and the characteristics of the domain, especially the amount and position of solid lattice sites. The easiest solution is direct addressing. In this case the lattice site PDFs are accessed simply using an enumeration function. To distinguish between the fluid and solid lattice sites, another vector is required. During the propagation step, the information about neighbours of the currently processed lattice site is retrieved and the propagation is handled accordingly. When a direct addressing is used, a space for the solid lattice sites PDFs is also allocated.

The use of semi - direct addressing can decrease of memory requirements as only the space for fluid lattice sites PDFs is allocated. The drawback of this approach is that the PDFs can not be accessed directly using the enumeration function. Fortunately, the solution is simple, as an additional vector can be used to create an exclusive enumeration. To protect the consistency of the LBM algorithm, the exclusive enumeration has to preserve the original enumeration of lattice sites. For this reason, an exclusive enumeration index is stored in the positions of the fluid lattice sites in an additional vector. In the positions of the solid lattice sites, this information is stated for example by storing a predefined constant with a negative value [20].

When indirect addressing is used, no information is saved for the solid lattice sites. Even the additional vector containing the information about each lattice site is not used. The connection of the lattice sites and their positions can be stored in several ways depending on the used algorithm, for example the connectivity of the lattice sites can be handled in the preprocessing stage.

3.2 Algorithms

- **Two - step two - grid** Two - step two - grid is a naive and most straightforward implementation of LBM. In this case, the propagation and collision are treated as strictly separated steps. First, the collision is performed for each lattice site and the post - collision values of the PDFs are computed. In the propagation step, post - collision values are streamed to the adjacent lattice sites. For storing the post - collision values between the steps, the amount of memory allocated for the PDFs has to be doubled. Together with a high memory usage, the algorithm also performs several transfers between the memory locations in each iteration, which makes it very memory bandwidth demanding [26][20].

- **One - step two - grid** This implementation fuses the propagation and collision into one step. The variant that is referred to as the push scheme performs the collision before the propagation. In this case, the collision is performed for the lattice sites in the first grid structure and the post - collision values are immediately streamed to its neighbours and stored in the second grid structure [26]. The pull scheme performs the propagation before the collision. In this variant, the corresponding values from the

lattice sites adjacent to the site being currently processed are gathered, and the post-collision values are stored in the second grid structure at the position of this lattice site. After each algorithm step, the pointers to the first and second grid structure are swapped, which significantly decreases the memory traffic when compared to the previous algorithm.

- **Compressed grid (shift)** Unlike the previously mentioned algorithms, the compressed grid is a one-grid algorithm which implies that the required memory is almost halved. Only one grid structure has to be stored in the memory plus an additional row of lattice sites for each dimension. The data dependency between the adjacent lattice sites is removed using some additional lattice sites. These sites are used for the creation of an offset used for storing of the updated values of the PDFs so that the values of the PDFs, which will be further needed are not overwritten. The order in which the lattice sites are processed is different for even and odd time steps. In the odd steps the iteration over lattice sites takes place from the top right to the bottom left, in the even steps the direction is reversed [26][20]. Both the pull and the push scheme can be used.

- **Swap** the swap algorithm is also a one-grid algorithm and unlike the compressed grid it does not use any additional lattice sites. Data dependencies are handled by swapping half of the lattice site PDFs with the PDFs of its neighbour. Which PDFs are swapped depends on the used scheme, in both cases the iteration over the lattice sites is the same and takes place from the grid corner to the opposing one in the standard order of the lattice sites. If the push scheme is used, the PDFs of the lattice sites which have not been yet updated are stored in the opposite locations within the lattice site. When a lattice site is approached, the collision is performed and the post-collision values are saved within the lattice site in a standard way. The PDFs pointing after the collision to the adjacent lattice sites, which have already been visited in the current iteration are swapped with the neighbour PDFs pointing at them. In this way, the propagation is performed implicitly by gradual swapping of the PDFs [19].

  If the pull scheme is used, PDFs of the adjacent lattice sites, that have not yet been visited in the current iteration, have to first be swapped with the PDFs of the currently processed lattice site pointing at them. Then all PDFs that are needed for performing the collision are stored within the currently processed lattice site in the opposite locations. The post collision values are then written in their natural locations within the lattice site. A disadvantage of this algorithm is the necessity of special treatment of the lattice sites within the solid boundary.

3.3 Basic implementation

In order to show the basic concepts described and explore the possibilities for further work, a demo LBM application was created. The application was programmed in C++ and includes a basic implementation of the LBM algorithm. For the simplicity, the D2Q9 model and the two-step two-grid approach were used. The push scheme, the BGK collision operator and the halfway bounce-back boundary condition were used for the implementation. The application output in the form of an array of densities for each lattice site in each timestep is stored in a HDF5 file.

The grid is stored in a C - style three dimensional array and is accessed by direct addressing. The first dimension is the domain width, the second dimension corresponds to the domain height and the third dimension is used as a storage for the values of the PDFs. Definitions of the main data structures can be seen in the code example 3.1.

Pseudocode 3.1: Data structures used in LBM

```
int v[9][2];
double t[9];
double fin[NX][NY][9];
double fout[NX][NY][9];
```

The arrays `fin` and `fout` store the grid, the variables `NX` and `NY` express the size of the domain, the array `v` stores the velocities coefficients according to 2.2 where two values are stored for each PDF: the velocity in the `x` and `y` direction. The array `t` is used for storing weights of the neighbour lattice sites as stated in 2.11.

In the first step of the implemented algorithm, the arrays `v` and `t` together with other constants as the relaxation time constant are initialized. The initial velocity and density for the lattice sites are also set in this step. In the case that there are some non - fluid sites in the domain, the domain map is also initialized. The domain map is implemented as a two dimensional integer array with the size of the modeled domain. The values stored in this array are either 0 for the fluid sites or 1 for the non - fluid sites.

In the next step, the algorithm of the lattice - Boltzmann method is performed. For a specified number of iterations, collision and propagation are performed as can be expressed by the pseudocode 3.2 where `iter` is the number of the algorithm iterations and `collision()` and `propagation()` are methods.

Pseudocode 3.2: Loop of the LBM algorithm

```
for (int i = 0;i < iter; i++) {
    collision();
    propagation();
}
```

The collision method pseudocode can be seen in the code example 3.5. The first step of this method is computing the values of the macroscopic variables - the fluid density and the macroscopic velocity. The fluid density is computed according to equation 2.6, the particle velocity is computed using equation 2.7. A pseudocode of methods where both variables are computed can be seen in code examples 3.3 and 3.4.

Pseudocode 3.3: Density

```
double getDensity(int i, int j){
  double result=0.0;
  for (int k=0;k<9;k++)
  {
    result += fin[i][j][k];
  }
  return result;
}
```

Pseudocode 3.4: Macroscopic velocity

```
void getVelocity(int i, int j,
    double density, double *u) {
  for (int d=0;d<2;d++) {
    for (int k=0;k<9;k++)
    {
      u[d]+=fin[i][j][k]*v[k][d];
    }
    u[d]=u[d]/density;
  }
}
```

After the values of the macroscopic variables are evaluated, the collision can be performed. The value of the local equilibrium distribution function is evaluated and this value is then used to compute the new value of each PDF according to equation 2.8.

Pseudocode 3.5: Collision

```
void collision(){
  for (int i=0;i<NX;i++)
  {
    for (int j=0;j<NY;j++) {
      density=getDensity(i,j);
      getVelocity(i,j,density,u);
      u2 = u[0]*u[0]+u[1]*u[1];
      for (int k=0;k<9;k++) {
        for (d=0;d<dim;d++) {
          p+=v[k][d]*u[d];
        }
        feq = t[k]*density*(1.0 + 3*p + (9/2)*p*p/2 - (3/2)u2/2) ;
        fout[i][j][k] = fin[i][j][k] + 1/tau*(feq-fin[i][j][k]);
}}}}
```

The method performing propagtion also includes the bounce - back boundary condition described in 2.2.2. The pseudocode of the method can be seen in code example 3.6. The method `isAFluidSite` returns a boolean value according to whether the lattice site with the given coordinates lies within the solid boundary or not. Accordingly, the values of the PDFs of the currently processed lattice site are either streamed to the neighbouring lattice sites or the halfway bounce - back rule is applied. The array `oppositeOf` stores the coefficients of the opposite velocities. The code uses a periodic boundary condition meaning that when the particle is about to leave the domain at the bottom, it is propagated to the top of the domain instead.

Pseudocode 3.6: Propagation

```
void propagation(){
  for (int i=0;i<NX;i++) {
    for(int j=0;j<NY;j++) {
      for (k=0;k<9;k++) {
        ii=(i+v[k][0]+nx)%nx;
        jj=(j+v[k][1]+ny)%ny;
        if (isAFluidSite(ii,jj))
          fin[ii][jj][k]=fout[i][j][k];
        else
          fin[i][j][oppositeOf[k]]= fout[i][j][k];
}}}}
```

3.3.1 Measurements

A few basic optimizations have been implemented in the basic code. All arrays are aligned to 32 bytes to enable the vectorization of the code. To force the data alignment, the clauses `__declspec(align(32))` and `__assume_aligned(x, BYTE)` have been used. The propagation and collision method was parallelized using the `#pragma omp parallel for` pragma.

The pragma was placed above the most outer loop of both methods. Furthermore, the `#pragma omp simd` pragma was placed above the most inner loop of the collision method. Therefore,, this loop is vectorized. Vectorization was also performed in the method computing fluid density by using the `#pragma omp simd reduction` pragma. The dimension of the array `fin` and `fout` was also changed to 12 elements to enable vectorized processing.

Several measurements were performed to compare the basic and the vectorized version. All measurements were done on Anselm supercomputer (8 core Intel Xeon processors with the Sandy Bridge microarchitecture). For the measurements runs, the output of the application into a HDF5 file was not performed. Both versions of the code were run with the domain sizes $128 * 128$, $256 * 256$, $512 * 512$, $1024 * 1024$ and $2048 * 2048$. For each domain size, the optimized version was run with 1, 2, 4, 8 and 16 OpenMP threads. For each application run, the number of the lattice sites updates per second was computed. All the measurements have been performed ten times and an average value of this runs was computed. The results were then compared with the basic version anc can be seen in figure 3.1. On the graph, the number of the processors is on the X axis and millions of lattice site updates per second (MSUPS) is on the Y axis.

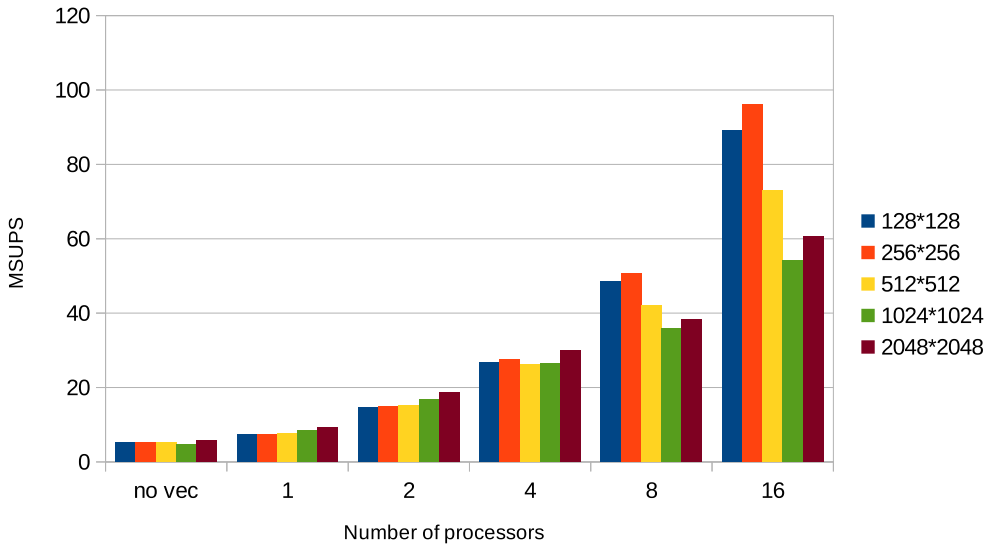


Figure 3.1: Performance of the basic version (labeled as no vec) and the optimized version on a different number of processors

As could be expected, the performance of the application is the worst for the non-optimized version. The performance of the optimized version is increasing with the increasing number of processors. The differences in the performance between the different domain sizes are in most cases not very significant. The big difference in performance between the runs with the two smallest grid sizes on 8 and 16 CPUs is most likely caused by fitting the small size of the domain and therefore its fitting into cache.

Chapter 4

Parallel computing

Parallelism can be implemented on several levels. Bit - level parallelism allows the processor to work simultaneously on larger data and includes for example SIMD approach. Instruction - level parallelism uses instructions pipelining and is used on superscalar processors. Thread - level parallelism includes multithreading techniques available in several programming languages and multiprocessing techniques that can be implemented using OpenMP. This text will mainly focus on bit - level parallelism in the form of SIMD and parallelization using OpenMP.

4.1 SIMD

Single Instruction Multiple Data (SIMD) architectures can be described as a system of processors controlled by a central unit where all currently working processors perform the same instruction at the moment. Therefore, the SIMD allows the programmer to implement data parallelism in a standardized way using defined pragmas and clauses. A scheme of this technique is depicted in picture 4.1. Even though the SIMD architectures can offer several advantages, they can not be always considered as the best and universal solution for all computing tasks. For the possibility of exploitation of the available features, it may be necessary to change or adjust the used algorithms in the particular application.

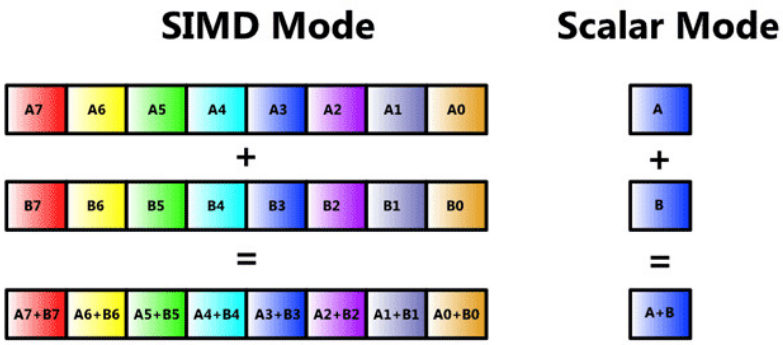


Figure 4.1: SIMD versus scalar operations [17]

The SIMD features are currently available on the most of the contemporary computer architectures as an instruction set extension. To enable the use of the SIMD on standard

processors, a technique called SWAR is used. The abbreviation SWAR stands for SIMD Within A Register and describes that the SIMD operations are performed within a group of processor registers so that no special architecture is required. The most known SWAR extensions on the x86 processors include MMX, SSE and AVX. The text will further focus on the SIMD extensions available on the current Intel processors.

The first SIMD extension available was MMX, which supported only integer operations and used the FPU registers and so it was difficult to simultaneously work with the floating point numbers and to use SIMD instuctions within the same application. The next introduced SIMD extension was SSE, which unlike the MMX used new 128 - bit registers called XMM0 - XMM7 designated for the use with SIMD instructions and supporting floating point operations. Later, the SSE2, SSE3 and SSE4 were introduced with the support of both integer and floating point operations, larger registers and new instructions [5]. The newest introduced SIMD extension is Intel Advanced Vector Extensions (AVX). AVX is designed to extend the existing SIMD by offering several new features. One of the main inovations is an increased size of the used registers to 256 bits. The overview of possible data types which can be used by AVX and SSE instructions can be seen in figure 4.2.

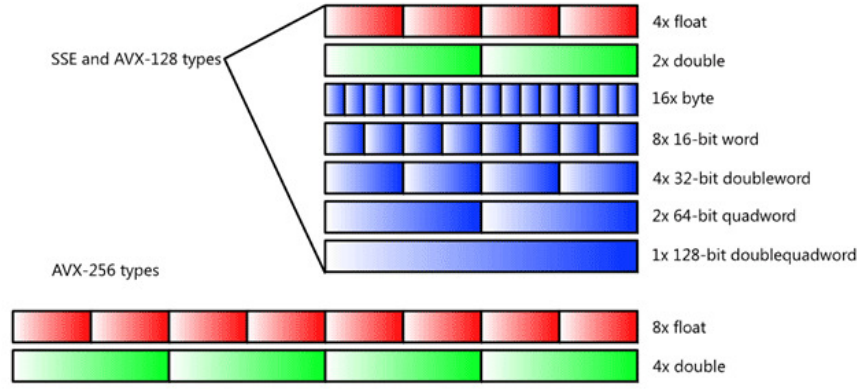


Figure 4.2: SSE and AVX data types overview [17]

4.2 AVX

The hardware support of the AVX includes 16 registers called YMM0 - YMM15 and one control and status register called MXSCR [18]. The YMM registers are aliased over the old XMM registers and therefore offer backward compatibility for all versions of the SSE instructions. When using the SSE instructions, the required registers are simply mapped to the lower half of the corresponding YMM registers. Even though this feature is supported, it is not recommended to often switch between the old legacy instructions and the new AVX instructions as it can negatively influence the optimal throughput of the application.

AVX allows the use of several data types in several sizes. The number of the values of a particular data type which can be processed by one SIMD instruction depends on the size of the used data type, for example an instruction can be available to either eight floats or four doubles. The new instructions use a two or three byte prefix called VEX prefix. This prefix also allows the use of a three operand syntax, which is another highlite of the AVX. With the three operand syntax it is possible to preform non - destructive operations that

do not overwrite the source operand so its value can be further used without the need of storing it in a separate register before performing the operation. A few instruction even allow the use of four operands.

In comparison to the SSE instructions, the requirements for the data alignment can be considered as less strict in AVX. While for the SSE the data alignment was required for most instructions, some new instructions also allow an unaligned access even though the performance can decrease. That is why memory alignment is still strongly recommended as a good practice even though in some cases is not mandatory [18].

4.3 Vectorization

There are several ways in which the vectorization of the code using the SIMD instructions can be performed. The easiest way is to simply rely on the compiler to perform an automatic vectorization. With this approach, it is necessary to follow several rules of writing the code. For example it is necessary that the data are correctly aligned and there is no data dependance between the steps that are likely to be vectorized. As it can be seen, this approach might not be optimal for most applications where the vectorization is required. The following approaches to vectorization of the code differ in the amount of control which the programmer has over the code on one side and simplicity of the solution on the other side.

To ensure that the code will be compiled using the SIMD instructions, it is possible to use auto - vectorization hints which will be recognized by the compiler, for example the `#pragma ivdep` directive [25]. This directive will make the compiler assume that there is no data dependency between iterations of the following loop and so the code can be vectorized. Even though this information is passed and recognized by the compiler, the dependencies will still be checked and when the compiler assumes that a data dependency exists, the code will still not be vectorized. The vectorization might not also be performed because the compiler might not detect that the vectorization of the loop would increase the performance. In this case, the `#pragma vector always` can be used to force the vectorization.

Another way of enforcing the vectorization is using the `#pragma simd` statement that enforces the loop vectorization. Unlike the previous techniques, it can also be used in case that there are proven data dependencies detected by the compiler [25]. Under these circumstances, the code under the influence of the `#pragma simd` has to handle the data dependencies itself to prevent generating of incorrect code. The statement also supports several optional clauses that can be used for example for a specification of the vector length or data type.

Using vector intrinsics can be also used for enforcing the code vectorization. Intrinsics are functions which can be called directly from C/C++ code after the appropriate header file is included. By using intrinsics, it can be directly controlled to which instructions the code will be compiled. This can be also done by including the instructions as inline assembly code.

4.4 Data alignment

Data alignment is a method used to force the compiler to store data object on a specific byte positions in the memory. By specifying the suitable byte boundaries for data, it is possible to increase the efficiency of memory operations because processors are designed to

move data efficiently from the specific byte boundaries (for example 32 byte). Thus the compiler is able to produce more optimized code when memory operations work with the aligned data. By default, the compiler can not assume that the data are aligned without being explicitly informed by the programmer. Therefore, it is important to not only align the data but also tell the compiler to assume the data to be aligned.

For static arrays, the data alignment can be implemented using the `__declspec(align (BYTE))` directive. For dynamically allocated arrays, the alignment can be ensured using the `_mm_malloc()` function. The `__assume_aligned(x, BYTE)` clause then has to be used to inform the compiler that the data have been aligned. Considering multidimensional arrays, it is also beneficial to ensure that the row length of the matrix is padded out to be a multiple of the chosen byte boundary. This step can then lead to significant performance increase.

To indicate that all arrays within a particular loop are aligned, the `#pragma vector aligned` directive can be used. This pragma applies only to the immediately following loop and can be useful especially when the code is assumed to be vectorized by the compiler. When using the pragma, the programmer is fully responsible for ensuring that the data alignment was performed, otherwise segmentation faults can occur [16].

In general, there are two basic memory layouts that can be used for storing multiple values into memory. The most straightforward variant is using an Array of Structures (AoS) which is implemented simply as an array where the array elements are structures. When accessing one field of the structure across the array elements to fetch the subsequent values within a loop, expensive memory gather - scatter operations have to be performed. This fact can increase the latency as well as the bandwidth usage and can also make it difficult for the compiler to vectorize the loop.

Structure of Arrays (SoA) can prevent expensive memory operations because it keeps separate arrays for each structure field. Thus it allows a continuous memory access when the fields have to be accessed within a loop. Using the SoA can lead to better code vectorization and increasing of the SIMD instructions efficiency. When accessing all fields within the same structure is also needed, it can be convenient to combine both variants and implement an Array Of Structure Of Arrays (AoSoA) which combines the two previously mentioned approaches [6].

4.5 OpenMP

OpenMP (Open Multi - Processing) is an application programming interface which provides support for creating multi - threaded applications. It is a set of compiler pragmas, directives, function calls, and environment variables that explicitly inform the compiler where and in which way multiple threads should be used [12]. Therefore, the OpenMP offers a multiplatform way of parallelizing the source code without the need of using the explicit threading properties of the programming language. It also determines how many threads should be created, how to synchronize them and it also destroys them when they are not needed any more. However, many of these properties can also be explicitly set by the programmer when needed.

All the OpenMP pragmas start with the `#pragma omp` followed by some of the clauses. For example a parallelization of a for loop can be implemented using the `#pragma omp parallel for`. When this pragma is processed by the compiler, the work done inside the loop is distributed among the threads created by an OpenMP. For this approach, the term work - sharing is used in the OpenMP terminology. For the loop parallelization possibility,

it is necessary for the loop to fullfill several requirements, for example a specific form of the comparison operation when evaluating the loop condition. Therefore, it might be needed to rewrite the loop that is likely to be parallelized to follow the given restrictions [12].

When creating multithreaded applications, it is necessary to avoid race condition for instance when all the threads are supposed to use the same variable. Several techniques can be used to prevent the race condition, for example the variable can be declared to be private for each thread by using the clause `#pragma omp parallel for private(VARIABLE)`. The data dependency between the loop iterations can also cause significant problems with parallelization.

By default, the OpenMP assumes that all loop iterations take the same amount of time. If it is not true, it is possible to use a clause to specify how exactly the relevant loop should be parallelized. The clause has the form `#pragma omp parallel for schedule(kind [, chunk size])` and allows to force the parallelization scheduling performation in the defined way and thus improve the load balancing. Since OpenMP4.0, it is also possible to use the API for vectorization. The vectorization of the loop can be enforced by using the `#pragma omp simd` before the loop [23]. This pragma can especially be useful when the auto-vectorization fails even after using the auto-vectorization hints.

The technique called SIMD-enabled functions can also be used to ensure that the code will be vectorized. This approach makes it possible to define functions for vectorization when called from within a vectorized loop or with an array notation of the arguments [24]. The SIMD-enabled functions concept allows to use standard scalar syntax to define the function and then enables its vectorization by using predefined pragmas. The `#pragma omp declare simd` pragma has to be placed before the function definition. When the function is called later from a loop with the `#pragma omp simd` pragma, the compiler uses a vectorized version of the method. The function can also still be normally used in a scalar form. There are also `linear` and `uniform` clauses which have to be used when only some of the function parameters are supposed to be in a vector form.

Chapter 5

HemeLB

HemeLB is a tool developed at the Centre for Computational Science at the University College London. The tool is designed for predicting the fluid flow in complex geometries. It is a massively parallel framework that is aimed to be used especially for blood flow modeling. One of the goals is to offer a toolkit that provides assistance to surgeons for example when treating patients with intracranial aneurysms. Therefore, a high precision as well as a reasonable computational time is demanded.

One part of the framework is the HemeLB Setup Tool that allows to load angiographic data from the patients and to choose the domain to be simulated using a graphical user interface. The HemeLB Steering Client allows to connect remotely to the simulation and to perform runtime visualization as well as the possibility of steering the simulation. The visualization is implemented by a parallelised ray - tracing algorithm [13]. The snapshots from the visualization can be seen in figure 5.1. The core of the HemeLB is written in C++ and consists of an implementation of the lattice - Boltzmann method optimized for modeling the fluid flow in sparse geometries.

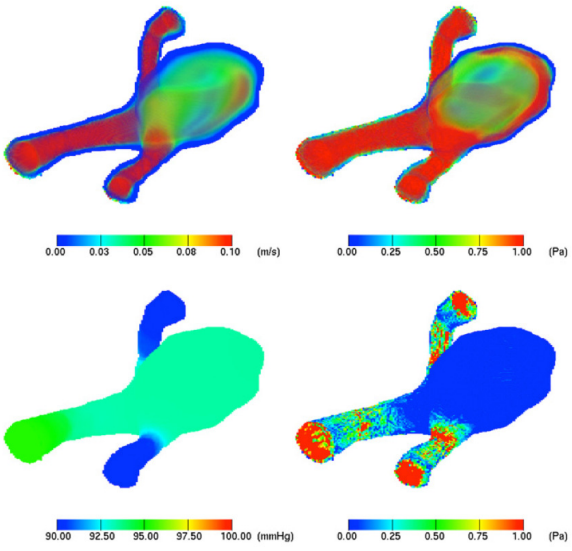


Figure 5.1: Snapshots of the simulation results: velocity field (top left), stress field (top right), wall pressure (bottom left), and wall shear stress (bottom right) [2]

The HemeLB simulation core uses the $D3Q15$ lattice-Boltzmann model with the BGK collision operator. Several boundary conditions have been implemented in the application. The biggest problem with choosing the right one is finding the right tradeoff between the complexity and thus the computational costs and the accuracy of the condition. A new boundary condition based on extrapolation was implemented in the core. This method is confined by pressure and no-slip boundaries and has been observed to have the first-order accuracy.

As HemeLB is designed to be used for sparse systems, it is important to use the data structure that minimizes the necessity of storing information connected to the solid lattice sites while ensuring a good data locality for the fluid lattice sites. The used approach splits the grid into a two level representation - when there are any fluid cells present for a particular coarse grid cell, then this cell is decomposed into a finer one. More precisely, the HemeLB uses a two-level grid where each cell of a coarse grid, called block, is decomposed into the chosen number of cells in the second level grid for each direction.

The example of a two-grid representation of a simple bifurcation for a block size of 4 cells can be seen in picture 5.2, where the lattice sites are depicted by squares with solid edges. Only the data associated with the non-solid blocks and the solid ones represented with thin dashed edges are allocated in the memory. The block size of 8 cells has been identified as optimal to ensure a good cache use within the blocks while minimizing the data exchange with the neighbouring blocks [21].

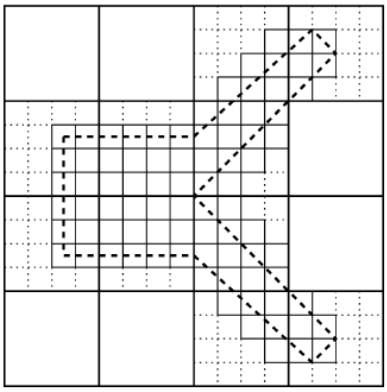


Figure 5.2: Two-level grid representation in HemeLB [22]

As a consequence of using the two-level grid representation, a mapping between the coordinates of a lattice site in the single level representation and those ones in the two-level representation is implemented. The essence of this mapping is not complicated and consists of a few multiplications and additions for the conversion of the indexes, but for a large domain sizes it can be too computationally extensive to preform this mapping during each propagation step. Therefore,, a look-up table containing the precomputed indices was implemented.

In the HemeLB implementation, the multidimensional data are stored in 1D arrays. Several implementations of the simulation can be used. For storing information about the non-fluid sites in the domain, a precomputed array is available. Considering the two-level grid approach, it is important to identify the neighbouring lattice site effectively. This is achieved by a construction of the neighbour map designed as a 1D look-up table. The

look - up table stores $8^3 * 15$ for the block size of 8 cells and the $D3Q15$ model.

This array stores the increments in coordinates for accessing the neighbouring blocks and their lattice sites. It is possible to avoid the use of the precomputed array for storing information about the domain boundary conditions by extending the look - up table to store the precomputed coordinates of a neighbouring block for each block separately. When a neighbour lattice site resides in the wall, this fact can then be stated by storing a special constant in the appropriate position. This implementation omits the array with boundary information at the expense of increased memory consumption caused by the space needed for the extended look - up table.

To enable parallelization that is needed for modeling of large domains, it is necessary to perform a domain decomposition. The domain decomposition has to ensure good workload and communication balancing during the parallel execution. The domain decomposition algorithm is based on a graph growing partitioning algorithm.

After the domain decomposition is done, the blocks are assigned to the corresponding processors. As the communication between the neighbouring processor sub - domains has to be performed during each propagation step, the amount of information that has to be exchanged needs to be minimized. This is achieved by precomputing buffers of indices that allow to avoid several computational operations connected to indexing the neighbour lattice sites on different processors.

HemeLB can also be used for cross - site simulation which allows to perform simulation of a very large domain on different machines [21]. In this case, the domain decomposition has to take into consideration the processors locality as the communications between the processors on different machines take significantly more time. Therefore,, the algorithm has to be able to deal with the high latency and low memory bandwidth when inter - machine communication has to be performed. This is achieved by using MPI non - blocking communications. All communications needed for each iteration (for example exchanges required for the LB algorithm, steering and visualizations) are bundled into a single batch of messages. As a result of this approach, each iteration requires only one MPI synchronization point [13].

The output of the simulation are values of the effective pressure, velocity and von Mises stress flow for each lattice site. HemeLB also offers the possibility of checkpointing, during which the whole system is dumped. The stored values can then be used for continuing the simulation after the crash without the necessity of starting it from the beginning. Checkpointing also allows to monitor the results and to change the configuration parameters during the simulation accordingly [13].

Chapter 6

Implementation of the application

After exploring the current HemeLB capabilities, a decision was made to aim the thesis to compare several LBM algorithms with the use of vectorization and parallelization. The algorithms were supposed to be compared not only according to their performance but also on the basis of memory and cache utilization.

The final application was created based on the basic implementation described in chapter 3.3. The application has a command line interface. One of the biggest changes that were implemented was adjusting the code to support 3D simulations. Therefore, the application supports D3Q15 and D3Q19 models. The following three algorithms for the LBM simulation were implemented: two-step two-grid, one-step two-grid and shift algorithm. Based on the performance results, the shift was proven to be the fastest one and therefore it was chosen for further optimization. The optimized version of this algorithm is the part of the final application as a fourth variant of the LBM algorithm.

In the basic implementation, the arrays used for storing the values of the PDFs were allocated as static 3D arrays. This approach had to be changed for the possibility of supporting bigger sizes of the modeled domain. All arrays are therefore allocated dynamically. This approach ensures that the user is informed at the start of the application if the requested amount of memory can not be allocated. To provide a good vectorization support, the arrays are allocated as aligned to 32 bytes using function `_mm_malloc`. An example of allocation of an aligned array is shown in code example 6.1.

Pseudocode 6.1: Allocation of an aligned array

```
fin = (T *) _mm_malloc(size*sizeof(T), 32);
__assume_aligned((T*)fin, 32);
```

With this approach, an array is allocated as a 1D array while an abstraction of a 4D array is needed for the simulation of a 3D domain (one dimension has to be added for PDFs). This requirement was solved by implementing a mapping function that maps 4D coordinates to a flat 1D array. The source code of the mapping function can be seen in code example 6.2 where `nx`, `ny` and `nz` stand for the domain sizes in each dimension.

Pseudocode 6.2: Method for mapping 4D to 1D coordinates

```
inline int LBM<T>::mapp(int i, int j, int k, int l) {
  return ny*nz*neighbor*i + nz*neighbor*j + neighbor*k + l;
}
```

Along with the change in the number of dimensions of the arrays for storing values of the PDFs, the arrays for storing velocity coefficients (`v`) and the weights of the PDFs (`t`)

had to be changed as well. Both arrays are also allocated dynamically with as 1D arrays aligned to 32 bytes. The array that is storing velocity coefficients has three values for each PDF (velocity in x, y and z directions). Therefore, a mapping function which maps 2D coordinates (PDF, dimension) to a 1D array was created to access these values. The source code of the mapping function is shown in code example 6.3.

Pseudocode 6.3: Method for mapping from 2D to 1D coordinates

```
inline int LBM<T>::mapVelocity(int i, int j) {
  return i*dim + j;
}
```

The array for storing macroscopic fluid density after each time step and the domain map are 3D arrays which are also allocated dynamically as aligned 1D arrays. Therefore, another mapping function to map 3D coordinates to a 1D array was needed in this case. As the LBM algorithm requires frequent accesses to all arrays, all mapping functions were declared as `inline` methods to ensure the least possible negative influence on the application performance. The source code of the method used for mapping can be seen in code example 6.4.

Pseudocode 6.4: Method for mapping from 3D to 1D coordinates

```
inline int LBM<T>::mapp(int i, int j, int k) {
  return ny*nz*i + nz*j + k;
}
```

The application prints several statics and information about the application run to the standard output. The first measured value is the time spent on performing the chosen algorithm. The time is measured using the module `std::chrono::steady_clock` from the standard `C++` library. The time point when the algorithm run was started is saved and after the required number of iterations is completed, another time point is saved. The elapsed time can then be computed simply as the difference between these two values. The performance of the algorithm is measured in millions of lattice site updates per second (MSUPS). This value is computed from the elapsed time and the total number of the lattice sites updates. The number of the lattice site updates is computed during the intialization as a multiplication of the number of fluid lattice sites and the number of iterations.

More detailed statistics are saved to a file when the corresponding parameter is used. These statistics include L2 and L3 cache accesses and misses, branches prediction information and processor cycles usage. This values are measured using PAPI (Performance Application Programming Interface) [7]. Another measured statistic is memory usage which is measured using function `getrusage`. This function is available in the header file `sys/resource.h`.

The computations included in the LBM algorithms can be performed either with `float` or `double` data type. This possibility is enabled with the use of C++ templates in the source code. The user can determine which data type will be used by choosing the corresponding parameter. The application source codes include a Makefile which ensures using all the required compiler and linker flags. The Makefile is designed to be used with the Intel C++ Compiler (`icpc`) [3]. The used compiler flags include `-O3` to ensure the maximum level of optimizations and `-std=c++11` to allow using of the features included in the C++11 standard. The flags `-lpapi` and `-openmp` ensure the correct linking of the PAPI and OpenMP libraries. The parameter `-xhost` enables usage of the AVX instructions needed for vectorization.

The whole application consists of three files. The implemented LBM algorithms including the corresponding initialization routines are part of a class in the file `lbm-demo.cc` with the header file `lbm-demo.h`. This file also comprises the definition of constants needed for the computation. The application is run from the file `main.cc`, where the command line parameters are parsed and the class is instantiated and initialized. The methods implemented in this file also perform several performance measurements and print the statistics about the application run both to the standard output and to a file.

The application always prints information about the current run to the standard output. The printed information consists of the LBM model, the number of PDFs, the used algorithm, the domain sizes, the number of iterations, the number of OpenMP threads, the total number of lattice site updates, MSUPS and the elapsed time. An example of the application output is shown in example 6.5. The results can be also saved to a file, if corresponding parameters are used, as described in section 6.1.

Pseudocode 6.5: Example of the application standard output

```
D3Q15 shift 120x130x140 iterations: 100 cores: 24
total site updates: 214935000 time: 4.30476 MSUPS: 49.7124
```

6.1 Application parameters

The application has to be run with several parameters that specify the domain, the used algorithm and the application output. The first three parameters `-x`, `-y` and `-z`, all followed by a value, are used to describe domain sizes in three dimensions. The domain sizes have to be positive integer values. The maximal supported size of a domain depends on the amount of memory available on the computer where the application is run. The next parameter is the number of PDFs for one lattice site. If the parameter `-n` is used, then the D3Q19 model is used, otherwise the model is set to D3Q15. The next parameter `-i`, followed by a value, is used for the number of the iterations of the LBM algorithm. This value has to be a positive integer.

Parameter `-a`, followed by a value, specifies the algorithm that will be used. This algorithm can be either two-step two-grid, one-step two-grid, shift or optimized shift. The parameter value has to be a number from 0 to 3, the numbers correspond to the algorithms in the order given in the previous sentence. Parameter `-f` specifies that the application will perform the computations using variables with `float` data type. If this parameter is not used, `double` variables are used instead. Parameter `-o` switches on the output to a HDF5 file. The output file is always named `density.h5` and is stored in the application folder. The file includes values of fluid density for all lattice site after each time step.

Parameters `-m`, `-c` and `-b` define the type of statistic that will be saved. If none of these parameters is used, no statistics are saved. As the statistics are measured using PAPI, only one statistic can be saved within one application run, because the number of counters which PAPI is using is limited. The first statistic describes cache use and can be switched on by using the parameter `-m`. This statistic shows the number of L2 and L3 cache misses and cache accesses. These values can then be used to compute the cache miss or hit rate.

The second statistic, that can be chosen, gives the information about cycles that have been used on computations and cycles in which no instructions could be completed. The statistic can be switched on by using parameter `-c`. It includes the number of cycles in which

the processor was stalled waiting for any resource, cycles stalled waiting for memory writes and total cycles completed. Based on these values, it is possible to compute the percentage of cycles where the processor was waiting. PAPI supports the use of more counters for statistics which can measure processor cycles effectiveness but most of these counters were unfortunately not available in the version of the Intel compiler that was currently available.

The last statistic, which is performed when parameter `-b` is used, gives an information about branches execution and prediction. The statistic includes the number of branches mispredicted and the number of branches correctly predicted. These values can then be used to compute the percentage of mispredicted branches. All statistics require setting an environment variable for PAPI before the application is run. The command for setting this variable for each statistic is shown in the code example 6.6. The first line is for cache statistic, the second line for a cycles statistic and the last line is for a branches statistic.

Pseudocode 6.6: Environment variables for PAPI

```
export PAPI_EVENTS='PAPI_L3_TCM|PAPI_L3_TCA|PAPI_L2_TCM|PAPI_L2_TCA'
export PAPI_EVENTS='PAPI_MEM_WCY|PAPI_RES_STL|PAPI_TOT_CYC'
export PAPI_EVENTS='PAPI_BR_MSP|PAPI_BR_PRC'
```

All statistics are saved to a file. It is possible to specify a file name by using a non-option argument. The file name has to be the last argument. If no file name is given, the chosen statistic is saved to a file named `res`, which is created in the application folder. If the file already exists, the values from the current run are appended to the end, each application run on a new line. Therefore, the file can be easily used to save the data from multiple runs. The values are delimited by the character `|` which ensures good possibilities for parsing by a script. The formatting of the line is shown in example 6.7. Parameter `-h` prints help with a descrtiption of possible parameters and their usage.

Pseudocode 6.7: Statistics file formatting

```
model | algorithm | x | y | z | iterations | time | MSUPS | threads
    count | statistic type | statistics | memory usage
```

Field `statistic type` describes the statistic that was preformed, which can be useful in case that the file is processed automatically. The field has the value of 0 for a cache statistic, 1 for a cycles statistic and 2 for a branches statistic. The field statistics contains the values of the chosen statistic again delimited by the character `|`. The order of values for each statistic is shown in example 6.8. When the option `-n` is used, and therefore no special statistic is produced, the output is saved to a file only when a filename is specified. In this case, the field `statistics` is not present in the output data and the field statistic type has the value of 3.

Pseudocode 6.8: Fields in statistics

```
//cache
L3 cache misses | L3 cache accesses | L2 cache misses | L2 cache
    accesses
//cycles
cycles stalled waiting for memory writes | cycles stalled | total cycles
     count
//branches
branches mispredicted | branches correctly predicted
```

The order of parameters is arbitrary but parameters for the domain sizes, the number of iterations and the algorithm always have to be used. An example of running an application with the domain size of 64x64x64, 100 iterations, shift algorithm, `float` variables and statistics describing cache misses which are saved in a file named `stats` is shown in code example 6.9.

Pseudocode 6.9: Example of running the application

```
./lbm-demo -x 64 -y 64 -z 64 -i 100 -a 2 -f -m stats
```

6.2 Initialization

At the start of the application, several variables need to be initialized according to the entered command line parameters. The required parameters include the size of the simulated domain in a 3D space, the type of the LBM model (D3Q15 or D3Q19), the number of iterations, the selected LBM algorithm and optionally a parameter for measuring and saving detailed statistics and a parameter for selecting the data type for computations (`float` or `double`).

The size of arrays, needed for storing the values of the PDFs, is determined on the basis of the size of the domain and the chosen algorithm. For two-step two-grid and one-step two-grid algorithms, the size is computed from the following equation $size = x * y * z * n$, where x, y and z stand for the sizes of the modeled domain in each dimension and n stands for the number of neighbor lattice sites (15 or 19). For shift algorithm, the array has an additional row of cells in each dimension. Therefore, the size of the array is computed as follows: $size = (x + 1) * (y + 1) * (z + 1) * n$. When shift algorithm or optimized shift algorithm is used, only one array for storing the values of the PDFs is allocated (`fin`), the other algorithms require two arrays (`fin` and `fout`). The arrays are initialized with an initial density. The initial value of each PDF is computed as a multiplication of the initial density and the weight of the particular PDF.

The information about which nodes of the simulated domain are fluid and which ones lay within the solid boundary is stored in an array. The array has a size equal to the size of the domain and the information is stored as an `integer` constant - a zero values means a fluid node, a values of one means a solid node. This array is initialized before running the selected algorithm. The application also uses an array for storing the densities at each lattice site after one step of the LBM algorithm. If the corresponding parameter is used when running the application, the content of this array is saved in a HDF5 file after each time step. All described arrays are initialized to default values at the start of the application. The initialization is performed in parallel to ensure NUMA first touch policy.

After the initialization, the selected LBM algorithm is performed for the chosen number of iterations. The method implementing the particular algorithm is called using a function pointer which is set during the initialization. This is possible because all the implemented algorithms have the same method signatures.

6.3 Two-step two-grid

This algorithm is based on the basic LBM implementation described in section 3.3. The run of the algorithm is divided into two parts - collision and propagation. In the collision part, the lattice sites in the array `fin` are accessed using three nested cycles, one for each

dimension. The cycles are executed in parallel, using an OpenMP pragma. Both parts are performed only for fluid sites, therefore this fact is checked for each lattice site before the computations are started. The pseudocode of the loops for accessing lattice sites in a 3D space is shown in code example 6.10.

Pseudocode 6.10: Loops iterating through the modelled domain

```
#pragma omp parallel for reduction
for (int i=0;i<nx;i++) {
  for(int j=0;j<ny;j++) {
    for(int k=0;k<nz;k++) {
      if(isAFluidSite(i,j,k)) {
        //collision and/or propagation executed here
      }
} } }
```

To allow better compiler optimizations, the variables were declared as `const` whenever it was possible. The macroscopic variables are computed in the first part of the collision method. Fluid density in the particular lattice site is computed in the method `getDensity`. The principle of the computation is the same as in the basic implementation but the method has been optimized using vectorization. Because each call of the mapping function for translating the coordinates from a 4D space to a 1D array requires a significant number of arithmetic operations, the address of the first PDF for the currently processed lattice site is saved and the other PDFs are accessed on the basis of this value.

This is possible because the mapping ensures that the values of the PDFs for one lattice site are saved subsequently in a row. This approach also allows better vectorization because the values are accessed continuously and not using a mapping function that would require generating gather instructions. The vectorization is performed using an OpenMP pragma. Because the density is computed as the sum of all values of the PDFs for one lattice site, the vectorization has to be performed as a reduction. The source code of the method can be seen in code example 6.11.

Pseudocode 6.11: Optimized density

```
T LBM<T>::getDensity(const int i, const int j, const int k) {
  T result = 0.0;
  T const *finRow = &fin[mapp(i,j,k,0)];

  #pragma omp simd reduction (+:result)
  for (int l = 0; l < neighbor; l++) {
    result += finRow[l];
  }
  return result;
}
```

The second macroscopic variable that has to be computed for each lattice site is the fluid velocity. This value is computed in the method `getVelocity`. As in the case of the fluid density, this method is an optimized variant of the method used in the basic implementation. To avoid calling the mapping function more often than necessary and also to allow a better vectorization, the values of the PDFs are again accessed in a row based on the address of the first PDF. The method is optimized with the use of vectorization, more precisely a reduction as in the previous case. The constant `1.0` is casted to the currently

used data type to avoid a run - time type casting. The source code of the method is shown in code example 6.12.

Pseudocode 6.12: Optimized macroscopic velocity

```
void LBM<T>::getVelocity(const int i, const int j, const int k, const T
    density, T *u) {
  for (int d = 0; d < dim; d++) {
    T uTmp = 0.0;
    T const *finRow = &fin[mapp(i,j,k,0)];

    #pragma omp simd reduction (+:uTmp)
    for (int l = 0; l < neighbor; l++) {
      uTmp += finRow[l]*v[mapVelocity(l,d)];
    }
    u[d] = uTmp * (((T)1.0)/density);
  }
}
```

After the macroscopic variables are computed, the collision itself is performed. This computation requires looping through all PDFs for the currently processed lattice site. In each loop iteration, a local equilibrium distribution function is evaluated. Then the new value of the PDF is computed as the difference between the equilibrium value and the current value of the PDF, multiplied by the constant `omega`. The code of the described loop can be seen in code example 6.13.

Pseudocode 6.13: Collision

```
#pragma omp simd
for (int l=0;l<neighbor;l++) {
  const T p = v[mapVelocity(l,0)]*u[0] + v[mapVelocity(l,1)]*u[1] + v[
      mapVelocity(l,2)]*u[2];
  const T feq = t[l]*density*((T)1.0 + p/cs2 + p*p/(((T) 2.0)*cs2*cs2) -
       u2/(((T) 2.0)*cs2));
  foutRow[l] = finRow[l] + omega*(feq-finRow[l]);
}
densities[mapp(i,j,k)] = density;
```

The constant `omega` stands for a reciprocal value of the relaxation time constant. The constant `cs2` is the speed of sound. The loop was vectorized using the `omp simd` pragma. The array `u` stores the values of fluid velocities in each dimension computed in the method `getVelocity`. The variable `u2` is used for the sum of squares of the macroscopic velocities for each dimension as can be seen in code example 6.14. This value is the same for all PDFs within one lattice site, therefore this value is precomputed and saved in this way. After the computation is performed for all the PDFs for one lattice sites, the macroscopic fluid density is saved to an array called `densities`. This array is then saved to the `HDF5` file after each time step.

Pseudocode 6.14: Sum of fluid velocity squares

```
const T u2 = u[0]*u[0]+u[1]*u[1]+u[2]*u[2];
```

In the propagation part, the new values of the PDFs are distributed to the neighbor lattice sites. The bounce - back boundary condition is a part of the propagation method. There

are only minor differences between the propagation method in the basic implementation described in the section 3.3 and the method in the final application. One of the differences is adjustment the method for a 3D space. Another difference is the usage of mapping functions to access the values of the PDFs and also the velocity coefficients. To improve the performance, the loops accessing the lattice sites are parallelized using OpenMP. The source code of the propagation method is shown in code example 6.15.

Pseudocode 6.15: Propagation

```
for (int l=0;l<neighbor;l++) {
  const int ii = (i + v[mapVelocity(l,0)] + nx) % nx;
  const int jj = (j + v[mapVelocity(l,1)] + ny) % ny;
  const int kk = (k + v[mapVelocity(l,2)] + nz) % nz;

  if (isAFluidSite(ii,jj,kk))
    fin[mapp(ii,jj,kk,l)] = fout[mapp(i,j,k,l)];
  else {
    const int opposite = oppositeOf[l];
    fin[mapp(i,j,k,opposite)] = fout[mapp(i,j,k,l)];
  }
}
```

Figure 6.1 shows a graph describing time spent in particular parts of the algorithm. The graph was generated by an analysis in Intel VTune Amplifier [4]. It can be seen, that a significant amount of time is spent in the collision and propagation methods. As these are the main parts of the algorithm, this fact would not be a problem. There is however also quite a lot of time spent in the mapping funcion, method computing fluid velocity and method checking if the currently processed lattice site is a fluid site. This results are partly caused by looping through the lattice sites twice, once in the collision and once in the propagation part.

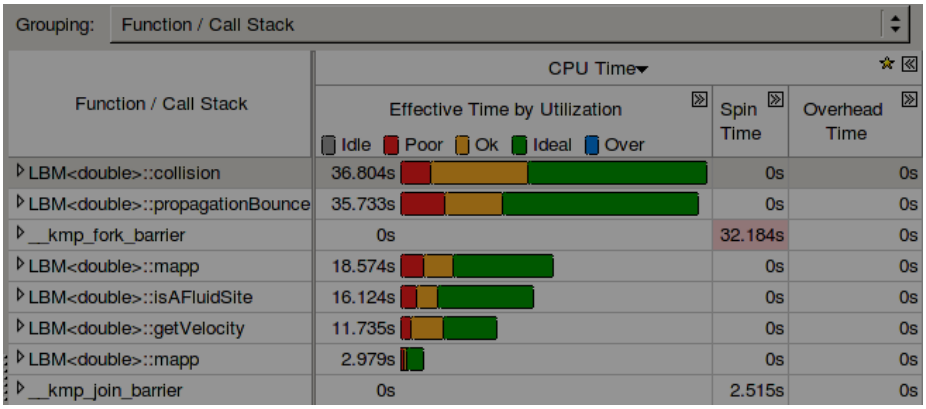


Figure 6.1: Two-step two-grid in Intel VTune Amplifier

6.4 One - step two - grid

The one - step two - grid algorithm performs propagation and collision within the same loop through the lattice sites in the modeled domain. The collision if performed in the same way as in two - step two - grid algorithm but newly computed values of the PDFs are immediately propagated to the corresponding neighbor lattice sites. Current values of the PDFs are read from the array `fin` and updated values are streamed to the array `fout`. After the collision and propagation are done for all the lattice sites, the pointers to `fin` and `fout` are swapped which ensures that the updated PDFs' values are used in the next time step.

The methods computing macroscopic variables are the same as in the previous algorithm, the differences are only in the most inner loop going through all PDFs for the particular lattice site. The pseudocode of the loop is shown in code example 6.16. The loop is optimized by vectorization using OpenMP as in the previous case.

Pseudocode 6.16: One-step two-grid inner loop

```
#pragma omp simd
for (int l = 0; l < neighbor; l++) {
  //computation of the local equilibrium distribution function (feq)
  //computation of the neighbour lattice sites indexes (ii,jj,kk)
  ...
  if (isAFluidSite(ii,jj,kk))
    fout[mapp(ii,jj,kk,l)] = finRow[l] + omega*(feq-finRow[l]);
  else {
    const int opposite = oppositeOf[l];
    fout[mapp(i,j,k,opposite)] = finRow[l] + omega*(feq-finRow[l]);
} }
```

As can be seen in figure 6.2, performing only one loop through the lattice sites contributes to a visible improvement in the processor usage. The most of the time is still spent in the method performing collision and propagation but there is a significant decreasing of time spent in the mapping function and the method checking if a lattice site is a fluid site.

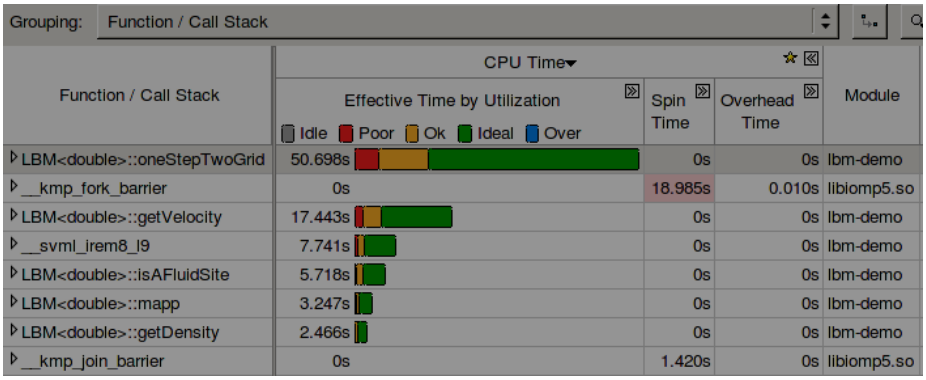


Figure 6.2: One - step two - grid in Intel VTune Amplifier

6.5 Shift

The biggest difference between the shift algorithm and the previously described algorithms lies in a significantly lower memory usage. The shift algorithm uses only one array for storing the values of the PDFs, therefore only the array `fin` is used and the array `fout` is not allocated at all. As already mentioned in section 6.2, in this case the size of the array is one row of cells bigger in each direction. This change of size is necessary to ensure that the values of the PDFs, which will be further needed, are not overwritten by newly computed values.

This condition is accomplished by using a different offset for positions in the array in odd and even time steps. The order in which the lattice sites are processed also differs. The iteration over the array is performed as three nested loops that are again executed in parallel by using an OpenMP pragma. In even time steps, the lattice sites are accessed starting from the maximum value of all three coordinates lowered by one $(nx-1, ny-1, nz-1)$ and going to zero. A pseudocode of the loop for an even time step is shown in code example 6.17. The offset for storing new PDFs' values is $+1$. In odd time steps, the array is accessed with coordinates going from $(1, 1, 1)$ to the maximal values. In this case, the offset for storing the new PDFs is -1. The pseudocode can be seen in code example 6.18.

Pseudocode 6.17: Shift - even time step

```
for (int i=nx-1; i >= 0; i--) {
  for (int j=ny-1; j >= 0; j--) {
    for (int k=nz-1; k >= 0; k--) {
}}}
```

Pseudocode 6.18: Shift - odd time step

```
for (int i=1; i <= nx; i++) {
  for (int j=1; j <= ny; j++) {
    for (int k=1; k <= nz; k++) {
}}}
```

As only one array is used, the collision and propagation are performed within one iteration over the lattice sites as in the case of the one-step two-grid algorithm. The use of only one array leads to decreasing the memory requirements almost by half. It also ensures better cache utilization an decreasing the number of cache misses. The macroscopic density and velocity are computed using the same methods as in the previously described algorithms. The inner loop accessing the PDFs' values and computing their new values was again optimized using vectorization.

Because the coordinates to the array differ in even and odd time steps, the values of the PDFs are not propagated over the border of the domain with the use of modulo operator as in the previous algorithms. Instead, the computed coordinates are checked and then propagated only if they fit within the domain borders. A pseudocode of the propagation part with this change implemented is shown in code example 6.19.

Pseudocode 6.19: Propagation in the shift algorithm

```
const int ii = i + v[mapVelocity(l,0)];
const int jj = j + v[mapVelocity(l,1)];
const int kk = k + v[mapVelocity(l,2)];
```

```
if (ii >= 0 && ii < nx && jj >= 0 && jj < ny && kk >= 0 && kk < nz &&
    isAFluidSite(ii,jj,kk))
  fin[mapp(ii+1,jj+1,kk+1,l)] = finRow[l] + omega*(feq-finRow[l]);
else
  fin[mapp(i+1,j+1,k+1,oppositeOf[l])] = finRow[l] + omega*(feq-finRow[l
      ]);
```

Figure 6.3 shows profiling of the algorithm in Intel VTune Amplifier. In comparison with the results of the previous algorithms (figures 6.1 and 6.2), it can be seen that there is a significant improvement in time spent in the mapping function and the method used for checking if a particular lattice site is a fluid site.

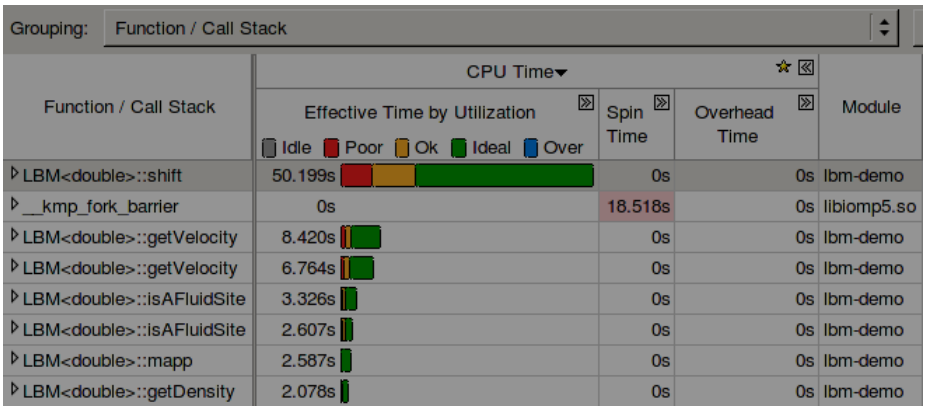


Figure 6.3: Shift algorithm in Intel VTune Amplifier

6.6 Optimized shift

Based on several measurements and profiling, the shift algorithm was determined the one with the highest performance. Therefore, it was chosen for further optimization. The application was profiled using Intel VTune Amplifier [4]. According to the results, a significant amount of time was spent in the mapping method and the methods computing the macroscopic density and velocity.

Therefore, the first matter on which the improvement effort was focused was the mapping method used for transforming coordinates in 4D space to 1D space. Because the mapping method was used several times for each lattice site in each time step, high performance of this method was critical. As the mapping method consists only of one line and was already declared as inline, there was no more space for optimization within the method. Therefore, it was necessary to try to minimize the number of calls of the method as much as possible. This effort can also be beneficial for vectorization, as accessing the arrays using a mapping function can negatively influence the vectorization effectiveness, because it leads to the use of gather and scatter instructions. This is the reason why some of the attempts to minimize the use of the mapping method were already performed as described in section 6.3.

Even though this approach led to significant improvements, the mapping function for getting the address of the first PDF for the currently processed lattice site was still called several times during each time step. Two calls were used in the methods computing the fluid

density and velocity and one was used before computing the values of the local equilibrium and the new PDFs values. One call was also necessary for transforming the 3D coordinates to 1D space used for checking if the currently processed lattice site is a fluid site. The partial solution of this problem could be calling the mapping method only once and passing the address to the array as a parameter to the methods which are computing fluid density and velocity.

There is however one even better solution that allows calling only the method mapping 3D to 1D once and then getting an index to a 4D array based on the computed value. This approach is possible because there is always a fixed number of PDFs (15 or 19) for each lattice site and all PDFs for one lattice site are stored subsequently in a row. Therefore, it is possible to get the index to 4D array simply by multiplying the index for the 3D array by the number of the PDFs. Then the array `fin` for the particular lattice site can be accessed as shown in code example 6.20. The index stored in the variable `ind` is then also used for accessing the array used for storing the density of each lattice site.

Pseudocode 6.20: Using 3D mapping method for 4D mapping

```
const int ind = mapp(i,j,k);
T const *finRow = &fin[ind*neighbor];
```

The optimization efforts were focused on computations of the macroscopic fluid density and velocity. As can be seen in code examples 6.11 and 6.12, both methods perform a loop through all the PDFs of the particular lattice site to access its values. Therefore, it is possible to perform the necessary computations within one loop. This optimization leads to a significant decreasing of the number of the loops needed for computing the mascroscopic variables. In the original method computing the macroscopic velocity, it was necessary to loop through the PDFs once for every dimension plus once for computing the fluid density. As the macroscopic velocity was computed in two nested loops, this approach required unrolling of the outer loop. As the final application is designed for 3D domains, this could be easily done manually. However, if there were a requirement to support 2D also, it would be necessary to solve this issue for example by creating an overloaded method for 2D.

The loop can be vectorized by the OpenMP SIMD pragma. Because it is necessary to use a reduction, the temporary values of the macroscopic velocity, which were used within the loop, were created as `float` or `double` variables, instead of an array as in the original method. With this approach, all three variables (one for each dimension), together with the density variable, could be specified in the reduction clause of the pragma. After the loop, it was necessary to multiply each temporary value of velocity by the reciprocal value of computed fluid density to get the final macroscopic velocity. The pseudocode of the optimized computation of macroscopic variables is shown in code example 6.21.

Pseudocode 6.21: Optimized computation of the macroscopic variables

```
#pragma omp simd reduction (+:density,uTmp0,uTmp1,uTmp2)
for (int l=0;l<neighbor;l++) {
  density += finRow[l];
  uTmp0 += finRow[l]*v[mapVelocity(l,0)];
  uTmp1 += finRow[l]*v[mapVelocity(l,1)];
  uTmp2 += finRow[l]*v[mapVelocity(l,2)];
}
uTmp0 *= ((T)1.0)/density;
```

As the access to the array containing the velocity coefficients is performed through a mapping function, the compiler has to generate gather instructions to access these values to make vectorization possible. This fact can be checked in the compiler vectorization report. Because the array is accessed three times in each loop iteration, the use of such amount of gather instructions could possibly have a negative influence on the application performance. Therefore, a solution that does not require the use of a mapping function was suggested.

The original array containing velocity coefficients was arranged as a 2D array with three values (one for each dimension) for each PDF. The values for one PDF were stored subsequently. To avoid the need for a mapping function, the array was divided into three arrays, each of them storing the values for all the PDFs for one dimension. This means that the velocity coefficients for one PDF were stored at the same position in each array. After this change, the loop can be adjusted as shown in code example 6.22.

Pseudocode 6.22: Computation of the macroscopic variables without mapping function

```
#pragma omp simd reduction (+:density,uTmp0,uTmp1,uTmp2)
for (int l=0;l<neighbor;l++) {
  density += finRow[l];
  uTmp0 += finRow[l]*vx[l];
  uTmp1 += finRow[l]*vy[l];
  uTmp2 += finRow[l]*vz[l];
}
```

The original array `v` had `int` data type. As the array `fin` (accessed through the pointer `finRow`) has `float` or `double` data type, it was necessary to perform several type converts during the computation. The need for the type converts can cost significant amount of time and can also lead to a worse use of the vectorization unit. To avoid this situation, the arrays `vx`, `vy` and `vz` were declared to have the data type determined by the used template. Therefore, the type converts at run time are no longer required.

After the macroscopic variables are computed, the collision and propagation can be performed. Both computations are performed within one loop through the PDFs of the current lattice site as it is done with basic shift algorithm that is described in section 6.5. The velocity coefficients are needed also for computations of the local equilibrium as can be seen in code example 6.13. To avoid the use of mapping methods, the use of the array `v` can be again replaced with the use of three new arrays in the same way as during the computation of macroscopic variables.

The velocity coefficients are also used in the computation of coordinates of the neighbor lattice sites during propagation. In this case, the situation is more complicated and the accesses to the array `v` through the mapping function can not be simply replaced with the new arrays as in the previous cases. This situation happens because the result of the computation are coordinates which are supposed to be `int` data type so it is possible to use them in an already defined mapping method further in the propagation part.

The solution could be found in performing type converts but as this operation can have negative influence on the performance of the vectorization, a different approach was chosen. Three new arrays `vxi`, `vyi` and `vzi` were defined, containing the same values as the arrays `vx`, `vy` and `vz` but with `int` data type. With this solution, both the use of mapping function and type converts were avoided. It is however necessary to allocate and initialize three new arrays. Considering the size of each array will contain only 15 or 19 elements, this requirement was evaluated as not having a significant influence on the total application space requirements. The coordinates of the neighbor lattice sites, to which the new values

of the PDFs will be propagated, are then computed as shown in code example 6.23.

Pseudocode 6.23: Computation of the neighbour lattice sites indexes

```
const int ii = i + vxi[l];
const int jj = j + vyi[l];
const int kk = k + vzi[l];
```

Figure 6.4 shows results of profiling the algorithm in Intel VTune Amplifier. In comparison with the results for the shift algorithm (figure 6.3), there was significantly less time spent in the mapping function. The time spent by computation of macroscopic variables (fluid density and velocity) is in this case included in the method `optShift` which is handling also collision and propagation.

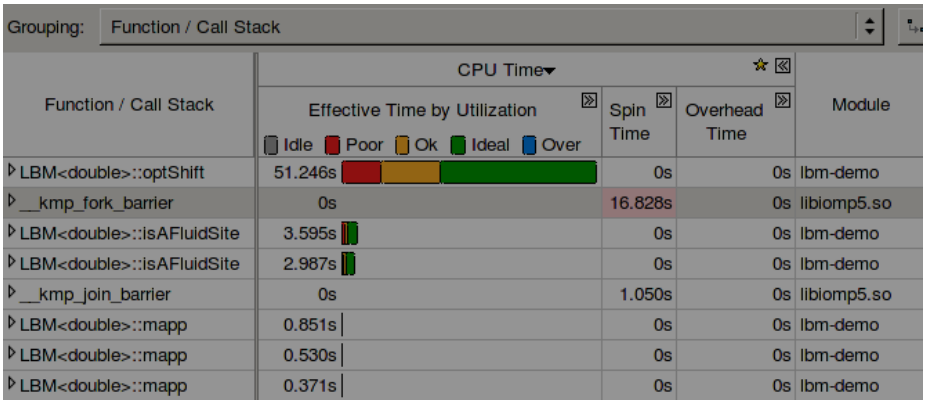


Figure 6.4: Optimized shift algorithm in Intel VTune Amplifier

6.7 Structure of arrays

The previously described algorithms work with Array of Structures (AoS). Despite this memory layout being more straightforward, it is not always necessarily the best choice in connection to performance. Structure of Arrays (SoA) can in some cases lead to a better cache utilization and can also be more suitable for vectorization. For more details on the memory layouts, see chapter 4.4.

There was an attempt to implement the SoA memory layout in the application and use it with adjusted versions of already implemented algorithms. As the memory layout is only an abstraction over the application data, no big changes in the structure of the application were needed. Crucial change was an implementation of a new mapping function which would reflect the new data layout. The source code of the mapping function for transforming coordinates in a 4D space to a 1D space can be seen in code example 6.24. This function was then used for replacement of the mapping function shown in code example 6.2. As the SoA was used only for arrays storing values of the PDFs, other mapping functions did not have to be changed.

Pseudocode 6.24: Mapping function for the SoA memory layout

```
inline int LBM<T>::mappSoA(int i, int j, int k, int l) {
  return ny*nz*i + nz*j + k + nx*ny*nz*l; }
```

With the SoA memory layout, it was necessary to change the places where the code is supposed to be vectorized because the PDFs for one lattice site were not stored subsequently as with the AoS. Therefore, vectorization of the loops through the PDFs of one lattice site would lead to a high number of cache misses because these values were stored far from each other. Based on this assumption, it was decided to try to vectorize the whole loop over the lattice sites. With this approach, several lattice sites could have been processed at once.

The implementation of this approach did not involve big changes in the source code, basically only moving the OpenMP pragmas to a different place. To make vectorization easier and more straightforward ,the three loops (showed in 6.10) over the lattice sites were replaced by a single loop. This action was possible because the SoA memory layout allows quite easy computation of indexes to the array which stores PDFs. The loop is shown in code example 6.25.

Pseudocode 6.25: Loop over the lattice sites with the SoA memory layout

```
const int loop = nx*ny*nz*neighbor;
const int n = neighbor;

#pragma omp parallel for simd
for (int i=0;i<size;i++)
{
  for (int l=0;l<neighbor;l++) {
    const T currentPdf = fin[i+l*size];
    //collision or computation of macroscopic variables performed here
  }
}
```

The mentioned techniques could be easily applied to any of the implemented algorithms. As this approach was supposed to have a potential of improvement of cache utilization and overall performance, the described changes were implemented to the optimized shift algorithm. Despite the expectations, this variant showed to have even lower performance than the two-step algorithm. According to this result it is obvious that there was some problem with vectorization of the code. As was stated in the vectorization report, the code was vectorized and there were also no errors or complaints from the compiler, however the vectorization appeared to be very ineffective.

Because the vectorization report did not provide more information, it is difficult to identify the exact cause of this situation. For example a problem with unrolling the inner loops or assumed data dependance between some parts of the algorithm could belong to possible reasons. Identifying the exact cause would involve watching a number of processor counters together with a deep analysis of assembly code produced by the compiler. These actions would probably exceed the scope of this thesis, therefore the SoA memory layout was not included into the final application.

Chapter 7

Testing

The application was tested on two supercomputers: Anselm and Salomon. Both computers have nodes with a x86-64 architecture and an operating system compatible with the RedHat Linux family. Each node of the Anselm cluster has 16 cores, divided into two 8 core Intel Sandy Bridge E5-2665, processors working on 2.4GHz frequency. Each node has 64 GB RAM and a local hard drive. Salomon has nodes with 24 cores consisting of two 12 core Intel Xeon E5-2680v3 processors running on 2.5GHz frequency. Each node has 128 GB RAM. In all performed tests, a whole node was always allocated even if not all processors were in use in the particular test to avoid interfering with other applications or users. The highest currently available version of the Intel Compiler `icpc` was `16.0.1` on both clusters.

All the results were always computed as average values from multiple runs. To ensure the placement of one OpenMP thread to one core and to avoid the possibility of moving the threads between the cores at run time, the commands shown in 7.1 were always executed before running a test. It was also necessary to load the modules needed for running the application: modules `intel`, `PAPI` and `HDF5`. The application was always compiled with the highest available version of the compiler.

Several tests were performed with the pursuit to compare the application performance with different algorithms, various domains and different number of cores. The performance was measured in MSUPS (millions of lattice site updates per second), which belongs to the standard measurements of the LBM algorithms. Together with the performance, various variants were also compared based on the statistics described in section 6.1 - L2 and L3 cache miss rate, usage of processor cycles and branches prediction. The application results for each test were saved to a file which was further processed by a `python` script. The script was also used to create the graphs using a library `matplotlib`.

Execept the performance, the different algorithms and other application settings were also compared based on the L2 and L3 cache miss rate and percentage of cycles where processor was stalled waiting. The amount of mispredicted branches was also computed, however the values were very low (under 1% for all run) therefore this comparison is not included.

Pseudocode 7.1: Setting of the OpenMP variables

```
export OMP_PLACES=cores
export OMP_PROC_BIND=true
```

The domain sizes that were used in tests were 128∗128∗128 and 64∗64∗64. This domain sizes are further referred as the bigger and smaller domain size. This domain sizes are always marked in the legeng of graphs with testing results. On Anselm, the tests were run on 1, 2,

Table 7.1: Amount of memory in MB used for different application settings

	double		**float**	
algorithm/domain size	**small**	**big**	**small**	**big**
two step/one - step (D3Q15)	67	508	36	260
two step/one - step (D3Q19)	83	636	44	324
shift/optimized shift (D3Q15)	37	270	21	141
shift/optimized shift (D3Q19)	45	334	25	173

4, 8 and 16 cores. On Salomon, the number of cores was the same with the addition of 24 cores. For all tests within one domain size, the same domain map was used. This domain map was created using a cube of non - fluid sites with the size $(nx/4)*(ny/4)*(nz/4)$ where nx, ny and nz are domain sizes in each dimension. This cube was always placed to the centre of domain, the rest of domain was filled with fluid sites. One row of the lattice sites was also initialized to have slightly higher initial velocity to initiate moving of the fluid in the modelled domain.

The four implemented LBM algorithms were tested with the use of both `double` and `float` variables. Each variant was run with 100 algorithm iterations. Table 7 shows the amount of memory in MB used by each of the algorithms with different domain sizes, data types and models. The amount of memory needed for two - step and one - step algorithms is the same, the same situation applies to shift and optimized shift algorithms. The amount of memory was equal for different numbers of cores and there were also no differences between Anselm and Salomon.

7.1 Performance for different algorithms

Several tests were performed to compare the performance of different algorithms. As could be expected, the performance was always better when the test was run on a higher number of cores. When the number of cores was doubled, the performance was usually doubled as well. This behavior is illustrated in figure 7.1. This figure shows the comparison of performance measured in MSUPS for two different domain sizes and two LBM models with the shift algorithm using `float` data type. The same behavior was observed for all other algorithms and domain sizes on Salomon. In this particular case, the speedup ratio was around 20 with efficiency 80% for 24 cores. There were no big differences in this statistics between different application settings.

Generaly it can be stated, that the optimized shift algorithm had the biggest performance with all configurations. The second best algorithm was the shift algorithm, followed by the one - step two - grid algorithm. The two - step two - grid algorithm showed to have the worst performance with all configurations. These results were expected considering the design of the algorithms.

However it can not be stated that doubling the number of cores always doubles performance. This outcome could be caused by multiple reasons, including OpenMP overhead, worse cache utilization and also necessary communication between different processor sockets. Therefore, if the application was run on even higher number of cores, the described trend of increasing performance would start to disappear. The signs of this situation can be observed on Anselm when using 16 cores as can be seen in figure 7.2. This figure shows the performance of the one - step two - grid algorithm for two different domain sizes and two

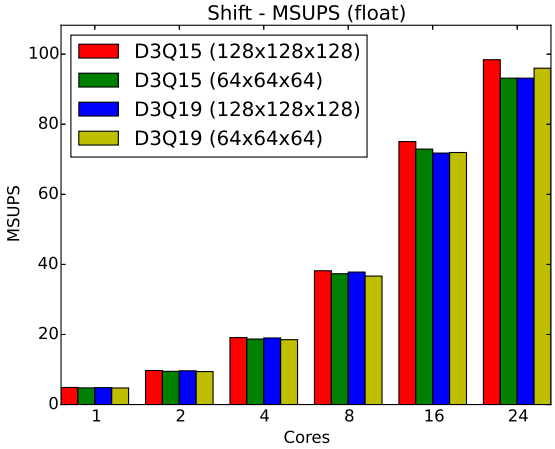


Figure 7.1: Performance of the shift algorithm on Salomon

LBM models with the `double` data type. As was previously stated, it is possible to see that the performance increase is approximately doubled for all the numbers of cores except 16 cores. In this case the speedup ratio was around 12 and efficiency 75% for 16 cores.

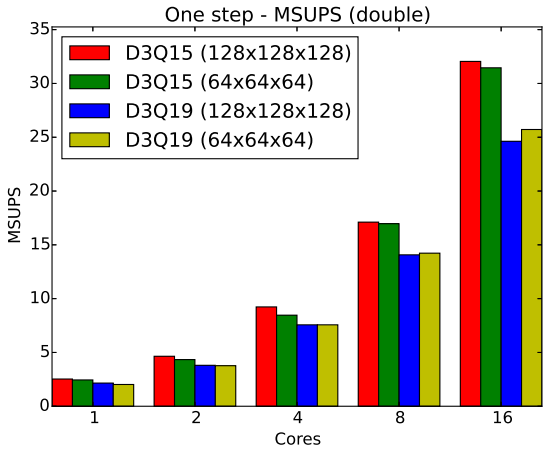


Figure 7.2: Performance of the one-step algorithm on Anselm

The performance was always better when using the `float` data type (command line parameter `-f`). This behavior was observed on both supercomputers with all algorithms. These results are primarily caused by better utilization of the vectorization unit. When the computations are performed using the `float` data type, 8 values can be processed at the same time within one instruction while for the `double` data type it is only 4 of them. Another point that can contribute to a better performance with `float` is the fact that the size of the values of this data type is smaller. Therefore, more values fit into a cache which can lead to a lower number of cache misses. As the result, the difference is bigger in the algorithms or domain sizes that have lower numbers of cache misses or a better performance in general.

The comparison of performance with `float` and `double` can be seen in figure 7.3. This figure shows the performance of all algorithms with different domain sizes and the model D3Q19 on 16 cores on Anselm. The difference in performance is the most obvious with the optimized shift algorithm. This result is probably caused by the overall high performance of this algorithm together with the lowest number of cache misses.

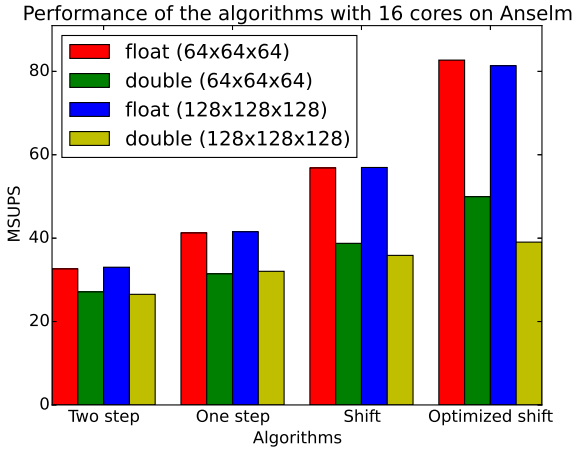


Figure 7.3: Performance of all algorithms with float/double and 16 cores on Anselm

Another interesting observation is the difference in performance between D3Q15 and D3Q19 models. This behavior was observed for all numbers of cores but it was more significant when the number of cores was 16 or 24. This trend was observed on both clusters, it was however more significant on Salomon. When using `float` data type, the performance was usually nearly the same for both models. In some cases the D3Q15 model had a slightly higher performance but the difference was within the range of a few percent. The situation was however different when the `double` data type was used. Here the difference in performance between the two models appeared to be more significant. The D3Q15 model had in most cases about 10-20% higher performance than D3Q19. The described behavior can be seen in figure 7.4, which shows the performance of the shift algorithm on Salomon with `double`, when compared with figure 7.1 which shows the same test run with `float`.

An exception was the optimized shift algorithm when running on 24 cores (16 on Anselm) with which the D3Q19 model showed to have higher performance on the smaller domain size than D3Q15 on the bigger domain size. This even occured on both Anselm and Salomon. This effect can result from a combination of several causes, including very high number of cache misses or relatively small domain size. This could lead to an increase of the influence of OpenMP overhead on performance. It is also necessary to note, that the performance results on high number of cores, especially 24 cores on Salomon, had most differences between the runs with the same settings. Even though the performance results were always computed as an average value from multiple runs, these differences could have some negative influence on the measurement accuracy in these cases.

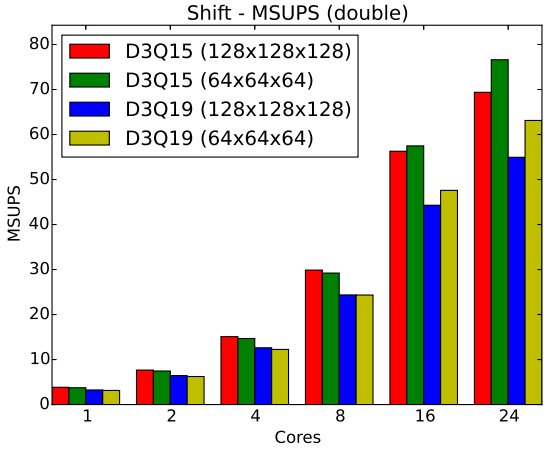


Figure 7.4: Performance of the shift algorithm on Salomon

7.2 Performance on Anselm and Salomon

The best performance of 155 MSUPS (78 GFLOPS) was achieved on Salomon with the configuration consisting of the optimized shift algorithm, D3Q19 model, 24 cores, smaller domain size and `float` data type. These results were closely followed by the same configuration with D3Q15 model with 149 MSUPS (61 GFLOPS) and D3Q15 on the bigger domain size with 144 MSUPS (59 GFLOPS). On Anselm, the best performance of 82 MSUPS (34 GFLOPS) was achieved also with the optimized shift algorithm and `float` data type, D3Q15 model and the smaller domain size on 16 cores. This result was followed by 81 MSUPS (33 GFLOPS) with the same settings and the bigger domain size and 70 MSUPS (36 GFLOPS) with D3Q19 model and the smaller domain size. The comparison of the best achieved result for all algorithms on 16 cores for Anselm and Salomon is depicted in figure 7.5.

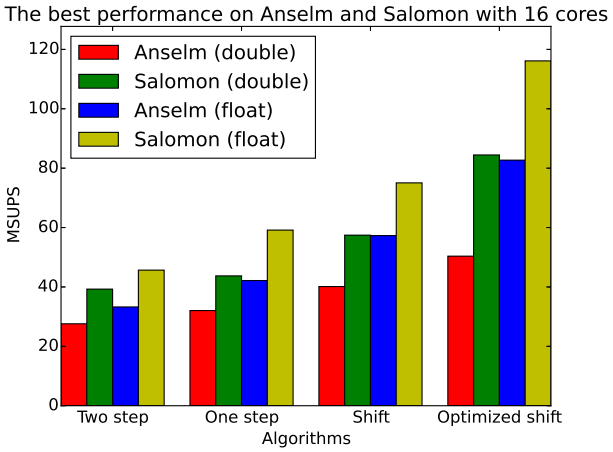


Figure 7.5: The best performance achieved with each algorithm on Anselm and Salomon

There can be several reasons why the performance with the same application settings is worse on Anselm than on Salomon. The first reason is that the compute nodes on Anselm have older processors that do not support AVX 2.0 as do the processors on Salomon. Therefore, the code has a potential to be better vectorized on Salomon. The processors on Ansels also work on a slightly lower frequency - 2.4 GHz (up to 3.1 GHz when using Turbo Boost) while on Salomon it is 2.5 GHz (up to 3.3 GHz when using Turbo Boost) [1][8]. There is also a difference in the available processor cache, this fact is further discussed in section 7.3.

7.3 Cache miss rate for different algorithms

Several tests were performed to determine the L2 and L3 cache miss rate. These values were computed on the basis of the values of total cache accesses and numbers of cache misses which can be obtained by running the application with parameter `-m`. There is a difference between the cache memory on Anselm and Salomon. The compute nodes on Anselm have 256 KB L2 cache per core and 20 MB L3 cache per processor [1]. The processors on Salomon use 30 MB Intel Smart Cache that is supposed to allow a better sharing of data between the cores [8].

Generally it can be said that the L3 cache miss rate was getting worse with an increasing number of cores. Especially with 16 and 24 cores the number of cache misses was very high, usually over 50% with the bigger domain size. In most cases the number of cache misses was also higer when using the bigger domain size. This effect was however especially significant with two - step and one - step algorithms. With shift and optimized shift algorithm the difference was not that significant.

The cache miss rate was very low (around 5%) for the one - step and two - step algorithms on the smaller domain size with `float`. The most probable reason is that almost all the PDFs could fit into L3 cache and therefore there was a very little number of L3 cache misses. The results for the one - step algorithm with `float` on Salomon are shown in figure 7.6, the percentage is expressed as decimal numbers. The results for the two - step algorithm were very similar, with the bigger domain the cache miss rate was in most cases about 10% higher. The difference in cache miss rate between the bigger and smaller domain size was significantly smaller for the tests which were run with `double`.

Interesting was the comparison with the shift and optimized shift algorithms that showed significantly higher cache miss rate with the same appliction parameters. Here the cache miss rate was in most cases approximately 30% for both domain sizes. The results for the shift algorithm with `float` can be seen in figure 7.7. The results for the optimized shift algorithm showed to be very similar with slightly lower cache miss rate for some of the configurations. It is however important to mention that the number of the L3 cache accesses for these two algorithms was almost half of the number for the one - step and two - step algorithms. The same situation applies to the overall amount of memory used by each of the algorithms. A higher percentage of cache misses could be also caused by a different behavior of prefetcher when more advanced optimizations are present in the code.

All described patterns applied also to the results on Anselm. One small difference was a slightly bigger cache miss rate for the smaller domain size (around 10% for the one - step algorithm). Generaly there were also smaller differences between cache miss rate for the smaller and bigger domain size when the one - step and two - step algorithms were used with `double`.

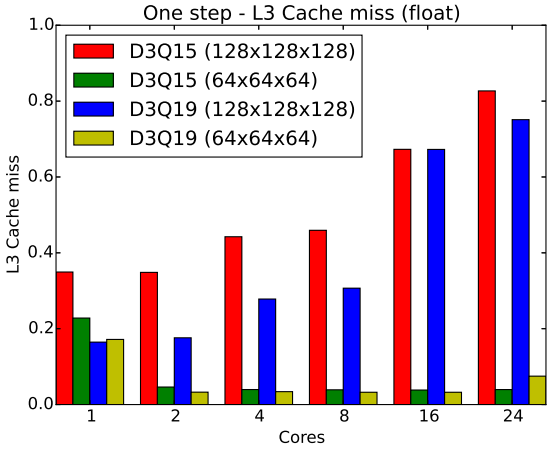


Figure 7.6: L3 cache miss rate for one-step two-grid algorithm on Salomon

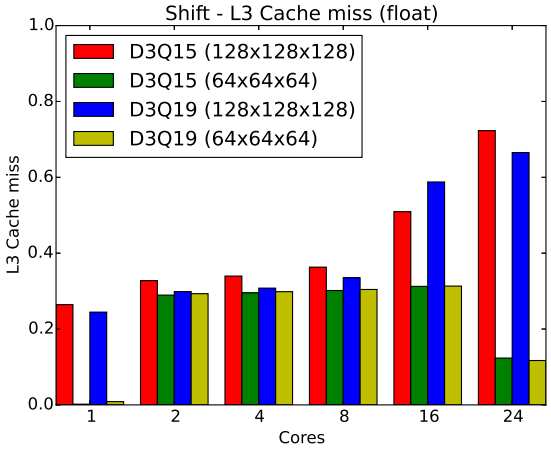


Figure 7.7: L3 cache miss rate for shift algorithm on Salomon

The L2 cache miss rate was relatively low for all algorithms. The miss rate showed to be almost independent of the number of cores. There were also not observed any significant differences between the results on Anselm and Salomon. The used LBM model and domain size also appeared to have a very little impact on the result. When run with `float` variables, the miss rate was slightly lower in most cases. The highest cache miss rates were achieved with two-step and one-step algorithms. This result is not surprising because these two algorithms use two arrays for storing the PDFs and therefore they use almost twice as much memory as the other ones. The L2 cache miss rate for these two algorithms was of 5-10%. The value for the shift and optimized shift algorithms was of 2-5%.

7.4 Processor cycles usage

The tests aimed to determine the effectivity of the processor cycles usage were performed only on Salomon as the required counters were not available in the PAPI version which was available on Anselm. The information on the numberof cycles during which the processor was stalled can be obtained by running the application with parameter `-c`. The values can be then used to compute the percentage of cycles in which the processor was stalled waiting for any resource and of those ones in which the processor was stalled waiting for memory writes.

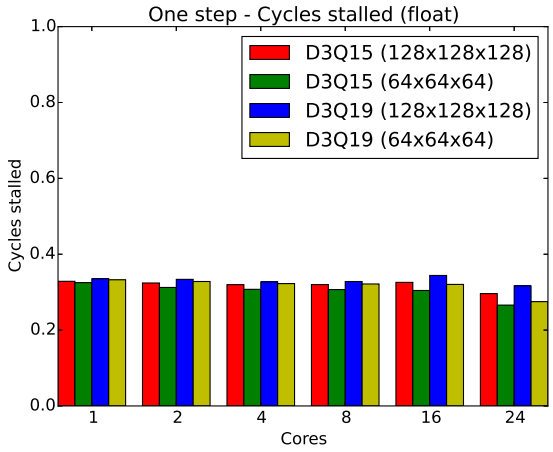


Figure 7.8: Cycles stalled for one-step two-grid algorithm

Based on the results it can be said that the amount of cycles stalled for any reason was always significantly lower when using `float` variables. This effect can be caused by a smaller amount of memory needed and therefore a better cache utilization and better vectorization. There were no significant differences in the results with different domain sizes and LBM models. For the one-step algorithm, the values were around 50% when run with `double` and 35% with `float`. These results can be seen in figure 7.8, the percentage is expressed as decimal numbers. For the shift algorithm, average values were around 40% for `double` and 25% for `float`.

The results of the two-step and optimized shift algorithms showed to be quite interesting. The two-step algorithm achieved around 40% stalled cycles for `double` and approximately 15% when using `float`. Especially the results for `float` can be considered as quite surprising as this algorithm showed to have the lowest performance. It could probably be caused by the fact, that this algorithm as the only one saves the new values of the PDFs during propagation sequentially. It is however difficult to determine the definitive reason as there were almost no more counters for processor cycles usage available. On the contrary the results for the optimized shift algorithm which had the best performance of all were significantly worse than had been expected. The average amount of cycles stalled was 50% with `double` and 35% with `float`. This effect was most likely caused by some of the implemented optimization, it is however quite difficult to identify the exact cause.

No great differences were found between different algorithms and domain sizes considering the amount of cycles stalled waiting for memory writes. The results were in most cases lower than 3% for all the tests. The only difference was observed when an algorithm

with the bigger domain size was run on 16 and 24 cores. In this case the average results were about 5-10%, closer to the lower boundary when using `float`.

Chapter 8

Conclusion

The thesis was focused on the lattice - Boltzmann method (LBM) used for fluid flow modelling. The goal was to explore possible implementations, suggest and implement optimizations and compare these implementations on the basis of several measurements including performance and cache utilization. The work was aimed to have a potential to contribute to an improvement of the tool HemeLB which is used for blood flow modelling.

The thesis offers a general overview of the LBM describing historical origins and crucial equations including the information about how these can be derived. The further focus is on the fundamental parts of the method - evaluation of macroscopic variables, collision and propagation. An overview of one of the most common collision operators called BGK is included. An important part is also a description of the most used LBM models and boundary conditions, with the focus on a bounce - back boundary condition.

After describing the fundamentals of the LBM, the text further focuses on possibilities of the LBM implementations. In this part of the thesis, several data and addressing layouts are introduced and compared. The most important part is a description and comparison of several algorithms which are further used in the thesis. This part includes a description of the basic implementation of a demo LBM application. As one of the goals of the thesis was to suggest possible optimizations, one chapter is focused on parallel computing including tools and methods that can be used for this purpose. The chapter describes available techniques for performing parallelization and vectorization of the code as OpenMP. One chapter is also devoted to an overview of the HemeLB.

On the basis of the HemeLB analysis and the explored LBM possibilities it was decided to create an application which includes an implementation of several algorithms together with several possible optimizations. The application was implemented in C++ and contains implementations of three chosen LBM algorithms: two - step two - grid, one - step two - grid and shift algorithm. Based on the performance results, the shift algorithm was chosen for further optimization. This optimized version of the algorithm is a part of the application as the fourth variant of the LBM algorithm. The application includes the possibilities of comparison of the implemented algorithms according to the performance, the L2 and L3 cache miss rate and the percentage of cycles in which processor was stalled.

The application was tested with several application settings on the Anselm and Salomon supercomputers. All algorithms and application settings were compared according to the performance, cache usage, overall memory requirements and processor cycles effectiveness. The results were presented together with possible explanations of the observed behavior. The best performance was achieved with the optimized shift algorithm when run on 24 cores on Salomon. The performance in this particular case was 155 MSUPS (78 GFLOPS)

which makes this algorithm approximately 3x faster than the most simple two - step two - grid algorithm. The best performance achieved on Anselm was 82 MSUPS (34 GFLOPS) with the optimized shift algorithms on 16 cores. Therefore, the shift algorithm can be recommended as convenient and effective LBM implementation, considering also its relatively low memory requirements and quite straightforward implementation.

The parallel efficiency was 70 - 80% for all algorithms on both clusters. The peak performance of one node on Salomon is 460 GFLOPS so the best algorithm achieves around 5% of this amount. The quite low number is caused mostly by quite high L3 cache miss rate and also quite high number of stalled processor cycles.

There are several possible suggestions on the further development of the application. One of the possibilities is an implementation of more LBM algorithms to get more options for comparing and choosing the most suitable one. Another way could be focusing on one or several of the already implemented algorithms and performing deeper analysis of the algorithms efficiency including identifying its weak points. With the focus on these places, the algorithms could be further optimized for example with the use of intrinsic functions. A potential of improvement could also lay in an implementation of Structure of Arrays memory layout or indirect addressing. The quite high number of cache misses could be improved by implementing some variant of loop blocking.

Bibliography

[1] Anselm cluster documentation. [online][cit. 2016-05-05]. URL: https://docs.it4i.cz/anselm-cluster-documentation/compute-nodes.

[2] HemeLB. [image][online][cit. 2015-12-23]. URL: http://www.2020science.net/software/hemelb.

[3] Intel c++ compilers. [online][cit. 2016-11-05]. URL: https://software.intel.com/en-us/c-compilers.

[4] Intel vtune amplifier. [online][cit. 2016-11-05]. URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[5] Intel® 64 and IA-32 architectures optimization reference manual. [online][cit. 2015-12-8]. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[6] Memory layout transformations. [online][cit. 2015-12-8]. URL: https://software.intel.com/en-us/articles/memory-layout-transformations.

[7] Papi. [online][cit. 2016-11-05]. URL: http://icl.cs.utk.edu/papi/.

[8] Salomon cluster documentation. [online][cit. 2016-05-05]. URL: https://docs.it4i.cz/salomon/compute-nodes.

[9] Dragos B Chirila. Introduction to lattice Boltzmann methods, 2010. [online][cit. 2015-11-08]. URL: http://www.dis.uniroma1.it/~pellegrini/project-assignment/cpp/pdf/boltzmann-lattice-2.pdf.

[10] Nicolas Delbosc. Real-time fluid simulation. [image][online][cit. 2016-01-09]. URL: http://www.efm.leeds.ac.uk/~mnnd/web_presentation/villanova_2015/.

[11] Alexandre Dupuis. *From a lattice Boltzmann model to a parallel and reusable implementation of a virtual river*. PhD thesis, University of Geneva, 2002. [online][cit. 2015-11-15]. URL: http://cui.unige.ch/~chopard/CA/aDupuisPhD.pdf.

[12] Richard Gerber. Getting started with OpenMP*. 2012-06-07. [online][cit. 2015-12-8]. URL: https://software.intel.com/en-us/articles/getting-started-with-openmp.

[13] Derek Groen, James Hetherington, Hywel B. Carver, Rupert W. Nash, Miguel O. Bernabeu, and Peter V. Coveney. Analysing and modelling the performance of the HemeLB lattice Boltzmann simulation environment. *Journal of Computational*

Science, 4(5):412 – 422, 2013. [online][cit. 2015-12-23]. URL: http://www.sciencedirect.com/science/article/pii/S1877750313000240.

[14] Xiaoyi He and Li-Shi Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811, 1997. [online][cit. 2015-11-15]. URL: http://link.aps.org/pdf/10.1103/PhysRevE.56.6811.

[15] Sasidhar Kondaraju, Hassan Farhat, and Joon Sang Lee. Study of aggregational characteristics of emulsions on their rheological properties using the lattice Boltzmann approach. *Soft Matter*, 8:1374–1384, 2012. [image][online][cit. 2016-01-09]. URL: http://dx.doi.org/10.1039/C1SM06193C.

[16] Rakesh Krishnaiyer. Data alignment to assist vectorization. 2013-09-07. [online][cit. 2015-12-8]. URL: https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.

[17] Chris Lomont. Introduction to Intel® advanced vector extensions, 2011-06-21. [image][online][cit. 2015-12-8]. URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf.

[18] Chris Lomont. Introduction to Intel® advanced vector extensions. 2011-06-21. [online][cit. 2015-12-8]. URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf.

[19] Keijo Mattila, Jari Hyväluoma, Tuomo Rossi, Mats Aspnäs, and Jan Westerholm. An efficient swap algorithm for the lattice Boltzmann method. *Computer Physics Communications*, 176(3):200 – 210, 2007. [online][cit. 2015-11-08]. URL: http://www.sciencedirect.com/science/article/pii/S0010465506003663.

[20] Keijo Mattila, Jari Hyväluoma, Jussi Timonen, and Tuomo Rossi. Comparison of implementations of the lattice Boltzmann method. *Computers & Mathematics with Applications*, 55(7):1514 – 1524, 2008. [online][cit. 2015-11-08]. URL: http://www.sciencedirect.com/science/article/pii/S0898122107006232.

[21] M.D. Mazzeo and P.V. Coveney. HemeLB: A high performance parallel lattice Boltzmann code for large scale fluid flow in complex geometries, 2008. [online][cit. 2015-12-23]. URL: http://www.sciencedirect.com/science/article/pii/S0010465508000805.

[22] M.D. Mazzeo and P.V. Coveney. HemeLB: A high performance parallel lattice Boltzmann code for large scale fluid flow in complex geometries. *Computer Physics Communications*, 178(12):894 – 914, 2008. [image][online][cit. 2015-12-23]. URL: http://www.sciencedirect.com/science/article/pii/S0010465508000805.

[23] Anoop Madhusoodhanan Prabha. Enabling SIMD in program using OpenMP 4.0. 2013-12-02. [online][cit. 2015-12-8]. URL: https://software.intel.com/en-us/articles/enabling-simd-in-program-using-openmp40.

[24] Anoop Madhusoodhanan Prabha and Bob Chesebrough. Performance essentials using OpenMP 4.0 vectorization with C/C ++. [online][cit. 2015-12-8]. URL: https://software.intel.com/sites/default/files/managed/37/df/OpenMP4-performance-essentials-using-vectorization.pdf.

[25] Mark Sabahi. A guide to vectorization with Intel® C++ compilers. 2012-04-27. [online][cit. 2015-12-8]. URL: https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf.

[26] Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. Comparison of different propagation steps for lattice Boltzmann methods. *Computers & Mathematics with Applications*, 65(6):924 – 935, 2013. Mesoscopic Methods in Engineering and Science. URL: http://www.sciencedirect.com/science/article/pii/S0898122112003835.

Appendices

List of Appendices

Appendix A

Content of the CD

The CD which is attached to the thesis includes the following data:

- **sources/:** Source codes of the implemented application including Makefile
- **README:** File containing necessary information for running the aplication
- **text/:** LaTeX source files of the text of the thesis
- **thesis.pdf:** The text of the thesis in PDF format
- **scripts/:** Scripts for Anselm and Salomon which were used for testing and comparing algorithms