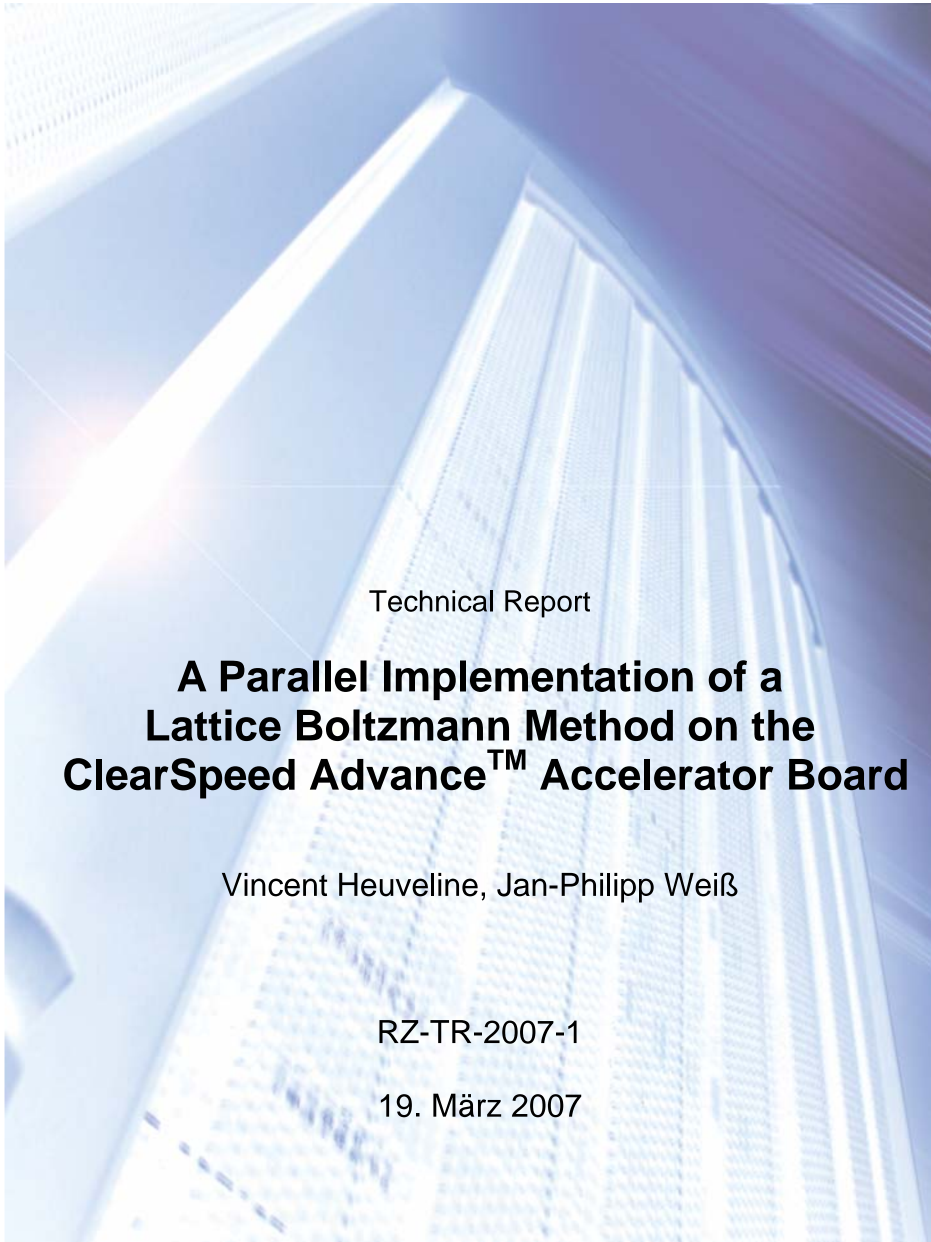
Karlsruhe Institute of Technology



Technical Report

A Parallel Implementation of a Lattice Boltzmann Method on the ClearSpeed Advance^{TM} Accelerator Board

Vincent Heuveline, Jan-Philipp Weiß

RZ-TR-2007-1

19. März 2007

SCC
Steinbuch Centre for Computing

A PARALLEL IMPLEMENTATION OF A LATTICE BOLTZMANN METHOD ON THE CLEARSPEED ADVANCE^{TM}ACCELERATOR BOARD

VINCENT HEUVELINE[1], JAN-PHILIPP WEISS[1]

ABSTRACT. Coprocessor and multicore technologies represent the current main development stream for compute server architectures in high performance computing. The primary challenge relies on the ability to exploit the associated computing power for highly CPU time-consuming applications in scientific computing. In this paper, we analyze specific methods adapted to the ClearSpeed Advance^{TM}accelerator board for the numerical solution of problems in computational fluid dynamics (CFD) by means of the lattice Boltzmann method. In this context, the main emphasis is given to the evaluation of this new technology with respect to sustained performance and efficiency. The ClearSpeed Advance^{TM}accelerator board is a PCI-X card equipped with two CSX600 processors, where each one holds 96 processing element cores. Each processing element handles 64-bit IEEE 754 floating point operations with double precision, which makes it attractive for applications in numerical simulation. The considered parallelization paradigm involves new concepts related to the distinction between `poly` and `mono` variables. As a model problem for our examination of the ClearSpeed Advance^{TM}accelerator board, we consider the simulation of fluid flow in a cuboid, known as lid driven cavity. An adequate parallel version of the lattice Boltzmann method is applied. Lattice Boltzmann methods are known to be perfectly suited for parallel architectures with high computing power due to the locality of the involved interactions. However, in the considered application, the solution process relies on a huge amount of data which needs to propagate along the underlying mesh. This fact, which is prototypical for this type of problem, shows up the bottleneck of the current internal communication bandwidth of the accelerator board.

1. Introduction

High performance computing (HPC) has reached a level of maturity and has become an integral part of day-to-day research activities in a diversity of scientific disciplines. As generally known, computing power alone does not guarantee an optimal throughput with respect to the scientific results. In addition to hardware related topics, high performance computing relies on an interdisciplinary expertise that encompasses computer architecture, programming models, multilevel parallelization, scalable software design, application software as well as scientific computing. The emergence of new coprocessor and multicore technologies leads to new challenges in the development of adequate numerical methods and software which can take advantage of the associated tremendous compute power in scientific computing. The focus of this paper is the evaluation of the ClearSpeed Advance^{TM}accelerator board for the solution of problems in computational fluid dynamics (CFD) considering a lattice Boltzmann (LB) method (see e.g. [11, 2, 12] and references therein). We examine the benefits and restrictions in that field of application and evaluate this new coprocessor technology with respect to sustained performance and efficiency.
The ClearSpeed Advance^{TM}accelerator board is a PCI-X card equipped with two CSX600 processors. The ClearSpeed CSX600 chip is a radical example of energy-efficient multicore processor design with 96 processing element cores running at 250 MHz. The ClearSpeed chip is a coprocessor and depends on a general purpose processor for the host. Each processing element (PE) handles 64-bit floating point operations with double precision following the IEEE 754 standards. The result is a chip that is dedicated to applications in scientific computing.
Many technologies support the data parallelism required to accelerate floating-point operations. Graphics processing units (GPUs) have evolved into attractive hardware platforms to accelerate general purpose floating-point calculations. Focused on the demand of the gaming market, this

technology offers high floating-point processing performance and huge on-chip memory bandwidth at comparatively low costs. GPUs provide however quasi-IEEE 32-bit single floating precision which is not sufficient for many applications in numerical simulation. Furthermore, using GPUs for scientific computing requires usually a reformulation of the underlying algorithms to the data-stream based programming model. Field programmable gate arrays (FPGAs) offer a different alternative to accelerate floating point applications. They rely on the design of specific integrated circuits and are inexpensive. However, their current performance for double precision remains limited. An exhaustive survey of currently existing technologies and projects related to coprocessor and multicore platforms go beyond the scope of this paper. The quoted examples however show that the available technologies have a high potential in HPC applications that are well matched to their specifications.
In this paper, we analyze specific methods adapted to the ClearSpeed Advance$^{\mathrm{TM}}$accelerator board for the solution of problems in computational fluid dynamics (CFD) by means of the lattice Boltzmann method. Lattice Boltzmann methods are numerical schemes for the discrete solution of Navier-Stokes like equations. These methods rely on the description of particles on a mesoscopic scale. Lattice Boltzmann methods are perfectly suited for parallelization since they do not involve any global operation, not even solving matrix systems or performing matrix multiplications. All interactions are strictly local.
This article is organized as follows. In Section 2, a short description of the architecture of the ClearSpeed Advance$^{\mathrm{TM}}$accelerator board is presented. Section 3 is dedicated to the parallelization paradigm on the ClearSpeed board. A technical synopsis of lattice Boltzmann methods for flow problems is depicted in Section 4. The developed parallel numerical scheme adapted to the coprocessor board is outlined in Section 5. In Section 6, the proposed numerical schemes are validated by means of numerical tests leading to an evaluation of the performance and efficiency of the coprocessor board. An outlook and concluding remarks are the object of Section 7.

2. Architecture

The ClearSpeed Advance$^{\mathrm{TM}}$accelerator board is a PCI-X card equipped with two ClearSpeed CSX600 coprocessors. A view of the board is shown in Figure 1. Almost every standalone compute server or node of a cluster providing PCI slots can be extended by several of such boards. Each CSX600 itself is endowed with 512 Mbytes of DDR2 DRAM local memory and 96 processing element cores. ClearSpeed's CSX600 is an embedded scalable power-efficient data-parallel coprocessor that handles 64-bit IEEE 754 conform floating point operations with double precision. It is advertised to provide 25 GFLOPS of sustained double precision floating point performance for Level 3 BLAS DGEMM matrix-matrix multiplication. The CSX600 processor dissipates an average of 10 Watts. It is a system-on-a-chip (SoC), based around the combination of ClearSpeed's multi-threaded array processor (MTAP).
The two CSX600s and a further FPGA as host interface are daisy-chained together via high speed bridges. The FPGA provides an efficient memory architecture between the CSX600 processors and the host's memory system. The control unit dispatches instructions to the execution units. The programmer only has to provide a single instruction stream. Each instruction is sent to one of the execution units. One part of the execution unit is formed by the `mono` execution unit. The `mono` execution unit has to process `mono` data, i.e. non-parallel data, and it has to handle the program flow control (e.g. branching, thread-switching).
The 96 processing elements (PEs) form the `poly` execution unit, that has to treat `poly` data, i.e. parallel data. It can be considered as a Single Instruction Multiple Data (SIMD) processor. Each PE has its own local memory of 6 Kbytes and a dual 64-bit FPU. It is further endowed with its own ALU, integer MAC, registers and I/O. The PEs operate at a clock speed of 250 MHz. The aggregate bandwidth of all PEs is specified to be 96 Gbytes/s. In Figure 2, the control unit, the `mono` execution unit, the `poly` execution unit and the in-between communication are depicted.
For some applications, the ClearSpeed board operates at the standard math library level. The Advance board works by offloading compute intensive math library routines called by applications

FIGURE 1. ClearSpeed Advance^{TM}accelerator PCI-X board.

Advance^{TM}board	CSX600 processor	Processing element
2 CSX600 processors	10 W max power consumption	250 MHz clock speed
25 W max power consumption	25 GFLOPS for DGEMM	6 KBytes local memory
50 GFLOPS for DGEMM	96 processing elements	1 Gbyte/s bandwidth
1 Gbyte memory	512 Mbytes memory	
3.2 Gbyte/s external bandwidth	96 Gbyte/s bandwidth	

TABLE 1. Performance and features of the Advance^{TM}board; SDK release 2.23.

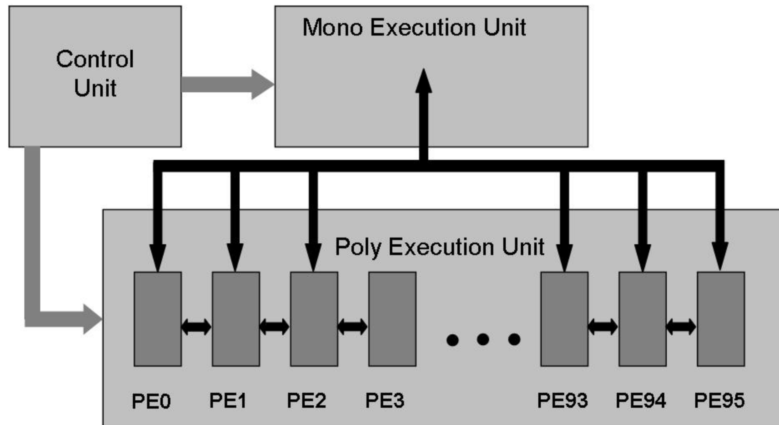


FIGURE 2. Control unit, `mono` execution unit and `poly` execution unit.

running on the host processor at the expense of additional communication. However, there's no control, which process or computation of the application is loaded onto the Advance^{TM}board.
When a call is made by an application to a ClearSpeed supported standard math library, it is intercepted by CSXL, ClearSpeed's accelerated math library, which calculates if the function call is worth off-loading. When it is, the CSXL transfers the required data to the board in order to

compute the function. The answer is calculated on the board and the results are read back into host memory before returning to the application.
This offloading feature is restricted to some selected routines like DGEMM or fast Fourier transformation. Further routines are expected to succeed. Supported application softwares are MATLAB and MATHEMATICA. Other specialized algorithms or routines have to be user coded by employing ClearSpeed's software development kit (SDK). The basic ingredient is ClearSpeed's programming language C^n, which is an extension to ANSI C especially adapted to the architectural requirements. Standard libraries are optimized for the MTAP architecture. The provided debugger is based on gdb. A macro assembler assembles the code generated by the compiler or hand-written assembler source. The Linpack benchmark was used to pronounce the performance gain by using several ClearSpeed Advance™boards in a cluster. Further information on the hardware and the associated software can be found in the documentation [8, 9, 7]. A report on the acceleration of the Intel MKL is provided in [5].
For our applications, the ClearSpeed board was built into a compute server with two 2.66 GHz Intel Xeon 5150 Woodcrest dual-core processors. In the overall of this paper, the considered software release is SDK 2.23.

3. The programming concept

The adoption of the ClearSpeed Advance™accelerator board requires new parallel programming concepts introduced by ClearSpeed's programming language C^n. Well-established programming concepts like MPI [3] or OpenMP [1] are not supported. The main difference is the distinction between `poly` and `mono` variables. The `mono` variables are held in the `mono` execution unit, the `poly` variables are dedicated to the `poly` execution unit; confer Figure 2.
Variables in `mono` space are equivalent to common C variables with one instance dedicated to sequential programming structure. Variables in `poly` space have an instance on each processing element (PE). These variables enable parallel data processing. Like in classical C, data administration is handled via pointers. C pointers have to be seen as `mono` pointers to `mono` data. Additionally, `poly` pointers to `mono` data are now considered, that is, there is a pointer variable on each PE that holds the address of (different) `mono` variables, e.g. elements of an array in `mono` space; see Figure 3 for a visualization. Furthermore, `mono` pointers to `poly` data hold the address of a `poly` variable; see Figure 4. These pointers have to be employed for the exchange of data between `mono` space and `poly` space. Unfortunately, `poly` pointers to `mono` data do not allow for a direct dereferencing. The library function `memcpym2p` has to be used for the data transfer instead. Data transfer from `poly` to `mono` involves the library function `memcpyp2m`.
For an efficient program flow asynchronous communication is a vital ingredient. The overlap of read and write processes renders a pronounced speed-up. In practice, manual double buffering controlled by semaphores has to be performed. However, double buffering decreases the size of available memory on the PEs by a factor of two.
Program control structures like loops or `if`-statements have to be divided into `poly` instructions or `mono` instructions depending on the control variables. As in standard C, in `mono` loops, the `mono` control variable decides whether the loop is performed or not and in a `mono if`-statement only one of the branches is executed whereas the other one is skipped. In `poly` loops or `poly` conditionals, a decision is taken, whether the corresponding PE is enabled or disabled. Disabled PEs do not take part at the evaluation of the given instructions and stay idle. As a consequence, all of the branches have to be run through, since some PEs may require execution. This means, that no `poly` branches can be skipped. A loss in time performance is the result, especially if the PEs have to perform different jobs. In our model problem, boundary nodes have to be treated in different ways than fluid nodes. Different boundary situations like different geometries (surface, edges, corners,...) or different boundary conditions require further differentiation. Disabled PEs have to wait for the execution of code by the enabled PEs. This kind of sequential branching has to be seen as a disadvantage in comparison to MPI methods. In `poly` loops, the loop is performed as long as the loop condition is true on any of the PEs. This may result in inevitable idle time. Special attention

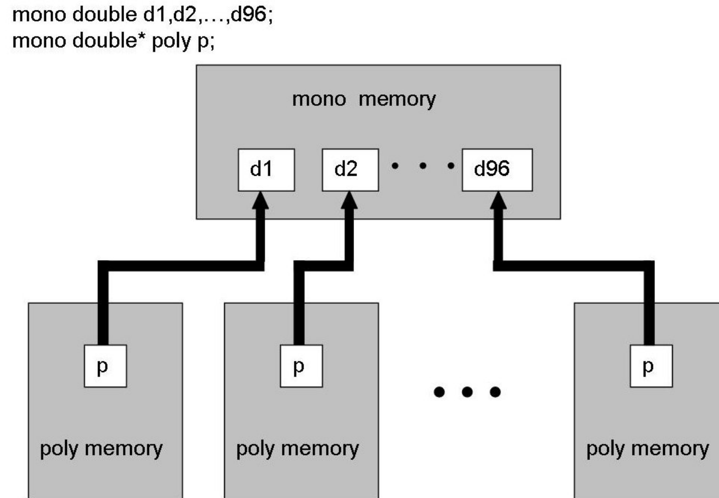


FIGURE 3. `poly` pointers to `mono` variables.

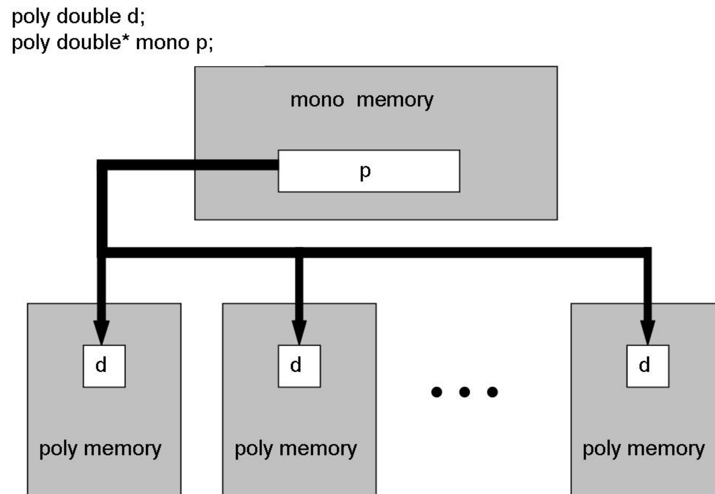


FIGURE 4. `mono` pointers to `poly` variables.

has to be taken to the fact, that `mono` code is always executed in `poly` conditionals or `poly` loops, even if the branches are not met by any of the PEs. Unexpected side effects may happen.

4. Lattice Boltzmann methods

Lattice Boltzmann (LB) methods allow to simulate the dynamics of particle distribution functions in phase space. These techniques find their application, among others, in the simulation of kinetic equations for fluids. In this paper, we consider a LB method for the numerical solution of the instationary quasi-incompressible Navier-Stokes equations. An exhaustive derivation of LB schemes can be found in [11, 2]. Our goal in this section is to introduce the associated concepts and define the model problem considered in the sequel of this paper.
The derivation of the LB models relies on a statistical description of gases where the main object of interest is the molecular probability distribution $f(\mathbf{x}, \mathbf{v}, t)$ which depends on the position $\mathbf{x}$, the

velocity $\mathbf{v}$ and time t. In LB models the velocity space is discretized, that is, the particles are only allowed to move along certain directions in the spatial grid. For our test problem, we consider the three-dimensional D3Q19 model with 19 velocities $\mathbf{c}_i$, $i = 0, \ldots, 18$, that are chosen to fit into the regular cubic grid. Each grid node has 26 direct neighbors, where only the 8 neighbors along the spatial diagonals are neglected. The velocity $\mathbf{c}_0$ is chosen to be zero. Furthermore, in the LB method the collision process between moving particles is simplified. The collision operator is chosen as the BGK relaxation type term and it describes the deviation from an equilibrium state. In each grid node, physics is described by 19 mesoscopic distribution functions, each one corresponding to one of the discrete velocities. Macroscopic values, that is, the values of interest like the density ρ or the fluid velocity $\mathbf{v}$ are gained by averaging in the velocity space, that is,

$$\rho^k(\mathbf{x}) := \sum_{i=0}^{18} f_i^k(\mathbf{x}), \tag{4.1}$$

$$\mathbf{u}^k(\mathbf{x}) := \frac{1}{\rho^k(\mathbf{x})} \sum_{i=0}^{18} \mathbf{c}_i f_i^k(\mathbf{x}), \tag{4.2}$$

where $f_i^k(\mathbf{x})$ describes the discrete counterpart of the distribution function $f(\mathbf{x}, \mathbf{v}, t)$. The upper index k refers to the discrete time variable and the lower index i specifies the discrete velocity variable. The discrete functions are defined and evaluated in the discrete grid points only. This scheme is an explicit scheme, where the nonlinearity is fully included in the evaluation of the local equilibrium distribution $F_{i,\mathrm{eq}}^k$ defined by

$$F_{i,\mathrm{eq}}^k(\mathbf{x}) := w_i \rho^k(\mathbf{x}) \left(1 + 3\mathbf{c}_i \cdot \mathbf{u}^k(\mathbf{x}) + \frac{9}{2}(\mathbf{c}_i \cdot \mathbf{u}^k(\mathbf{x}))^2 - \frac{3}{2}\mathbf{u}^k(\mathbf{x}) \cdot \mathbf{u}^k(\mathbf{x})\right) \tag{4.3}$$

with the weights $w_i \in \{1/3, 1/9, 1/36\}$. The BGK lattice Boltzmann algorithm for the D3Q19 model can be written as

$$f_i^{k+1}(\mathbf{x} + \tau \mathbf{c}_i) = f_i^k(\mathbf{x}) + \omega \left(F_{i,\mathrm{eq}}^k(\mathbf{x}) - f_i^k(\mathbf{x})\right) \quad \text{for } i = 0, \ldots, 18.$$

The relaxation parameter ω is interpreted as the collision frequency. It depends on the viscosity of the fluid. One lattice Boltzmann step can be split into the collision step

$$\widetilde{f}_i^k(\mathbf{x}) = f_i^k(\mathbf{x}) + \omega \left(F_{i,\mathrm{eq}}^k(\mathbf{x}) - f_i^k(\mathbf{x})\right) \quad \text{for } i = 0, \ldots, 18, \tag{4.4}$$

and the propagation step

$$f_i^{k+1}(\mathbf{x} + \tau \mathbf{c}_i) = \widetilde{f}_i^k(\mathbf{x}) \quad \text{for } i = 0, \ldots, 18. \tag{4.5}$$

The collision step is strictly local. Only the distribution functions of the same node are encountered. On the other hand, the propagation step does not involve any kind of computation. Only an exchange and update of data takes place.

Lattice Boltzmann methods strongly depend on the scaling of mesh space h, time step τ and the relaxation parameter ω. To attain the Navier-Stokes limit with prescribed viscosity, the coupling $\tau \sim h^2$ has to be chosen.

Special care has to be taken at the boundaries. Each node gets its information from the neighboring nodes. At the boundaries, these neighboring nodes are partially missing. The inflow values are undetermined and the macroscopic density and velocity cannot be determined for the computation of the equilibrium function. In the case of solid walls, bounce-back boundary conditions are commonly used to overcome this difficulty. The outflow values to non-existent neighbors are directly used to assign the opponent inflow values. At parts of the boundary where the velocity is prescribed, the corresponding equilibrium function is directly employed to assign the updated distribution functions.

For the correct choice of the boundary conditions, lattice Boltzmann methods are known to be second order accurate in space and first order accurate in time. Since an accurate approximation is not in our scope - we only want to measure the performance of the method on this specific

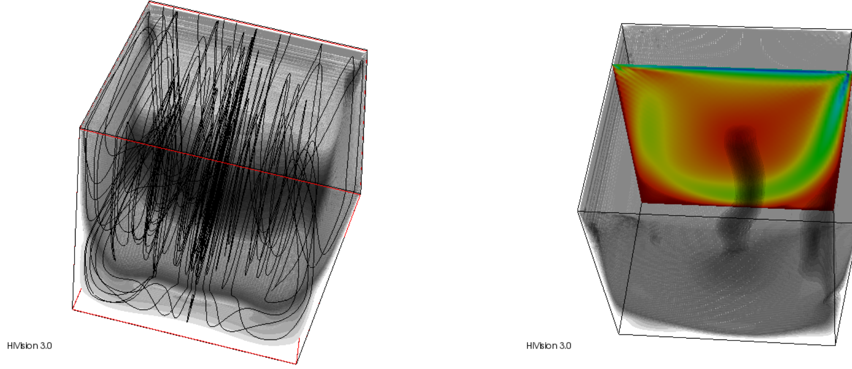


FIGURE 5. Lid driven cavity for $Re = 1000$: (left) plot of the streamlines; (right) main vortex.

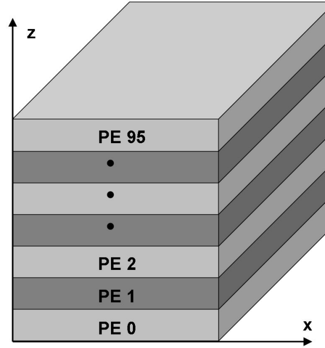


FIGURE 6. Partitioning of the computational domain: layers in z-direction.

architecture - we restrict ourselves to a simplified version of bounce-back boundary conditions at the solid walls.
The problem considered is a three-dimensional viscous flow in a cavity. An incompressible fluid is bounded by a cubic enclosure and the flow is driven by a uniform translation of the top surface. Numerical methods are often tested and evaluated on this cavity flow due to the rich vortex phenomena occurring at many scales depending on the Reynolds number Re (see [4] and references therein). In the overall computation in this paper we assume $Re = 1000$. Plots of the fluid flow are depicted in Figure 5.

5. Algorithmic aspects

5.1. **The applied parallel concept.** Lattice Boltzmann methods are perfectly suited for parallelization. The algorithm does not involve any global operation, not even solving matrix systems or performing matrix multiplications. All interactions are strictly local. In the collision step, the updated distribution functions of one node are computed by involving only the distribution functions of the same node. In the propagation step, each node has to exchange its distribution functions with 18 of its 26 neighboring nodes. These facts naturally imply domain decomposition methods for parallelization. Each PE is assigned to a number of nodes. In our case, we use an allocation in layers into z-direction; see Figure 6. This partitioning is surely non-optimal in the sense of minimizing interfaces. It allows however to simplify the treatment of the coupling between the subdomains which is more convenient for the analysis of the performance results.

The first approach for the parallelization relies on the concepts considered for distributed memory platforms. Each PE holds a layer in z-direction with approximately the same number of nodes. Communication with other PEs has only to be done at the upper and lower boundaries of the corresponding layer. Since all the PEs are aligned sequentially with respect to the underlying geometry, each PE has to communicate with its direct neighbors. Hence, the `swazzle` routines of ClearSpeed's programming language C^n can be applied that allow for fast communication between neighboring PEs. This kind of communication is indicated in Figure 2 by the arrows between the PEs. Note that there is no possibility for direct communication and data exchange between arbitrary PEs. In the non-neighboring case, data has to be exchanged via the `mono` domain. With respect to this constraint, this distributed memory approach fails, since there is not enough memory to hold enough data on the PEs. Each PE can only hold data for 40 nodes (40*19*8 bytes $<$ 6 Kbytes). This limited capacity is by far not enough for realistic simulations. In the case of asynchronous communication with double buffering, only 20 nodes can be stored per PE.
The second approach is based on a shared memory interpretation. The whole data are stored globally in the `mono` domain. Each PE is responsible for its own portion of data. For the collision and propagation step the corresponding data are loaded onto the PEs in the `poly` domain, processed and written back to the `mono` domain. As a consequence, additional time has to be spent for data transfer from `poly` to `mono` and back in each time step.
Our test code involves only one of the CSX600 processors of the board. To integrate the host processor or the second CSX600 processor, a common interface has to established by using assembler code on both devices. All statements on the performance of the architecture in this article apply to one CSX600 chip and its direct linkage to the operating system, employing ClearSpeed's programming language C^n provided by the software development kit.

5.2. **Implementation issues.** For an insight to the chosen algorithm, we take a closer look to some programming details. There are several ways to arrange the data. In the case of lattice Boltzmann methods one has to treat four-dimensional data: three dimensions in space (x, y and z) and one dimension in velocity, where the index i ranges from 0 to 18. Since the outer `mono` loops are coded in terms of the `mono` space variables x and y, we use the indexing $f[x][y][z][i]$. For the domain decomposition in z-direction there are local z-dependencies on each PE. On the PEs, we run a `poly` loop from z_{PE} to $z_{PE} + z_{PEsize}$, where $z_{PEsize} + 1$ is the thickness of the layer, that has to be processed by the corresponding PE. For the data transfer from `mono` to `poly`, we use `poly` pointers to `mono` data, that is, on each PE there is a pointer to $f[x][y] + z_{PE}$, which then represents a two-dimensional array.
When using static arrays, these arrays are de facto sequentially aligned one-dimensional arrays. In order to access the elements, these pointers have to be dereferenced only once. In the case of two-dimensional dynamic arrays, double dereferencing is necessary, which leads to further communication. Since a direct dereferencing of these `poly` to `mono` pointers is not possible, the library function `memcpym2p` has to be called. In the collision step, the data of one node can be copied in blocks. In the propagation step, the data of the neighboring nodes has to be made available additionally. Hence, the immediate copying of a bulk of nodes is preferred, but the PEs cannot hold more than 20 nodes at the same time. In a cuboid of $2 \times 2 \times 4$ nodes, none of the nodes can access all its neighbors within the same block. Since we end up with additional algorithmic difficulties at the bulk's boundary, we split collision and propagation step. As result, we have to perform two global loops which gives a time loss by a factor of two. This has to be kept in mind when measuring the performance.
Typically, more than 10^6 time steps have to be computed and more than 99.9% of the total computing time is spent in the time loop. Each time step consists of the collision step and the propagation step. The corresponding program flow of the time loop is depicted in Table 2.

5.2.1. *The collision step.* In order to process the data in the collision step, we define a $2{\times}19$ array in the `poly` domain for the 19 distribution functions of one single node. The first index with size 2 is chosen due to double buffering by active and background buffer.

Time loop: $\sim 10^6$ steps for $(t = 0;\ t \leq t_{max};\ t = t + \Delta t)$	
	Collision step: ~ 350 floating point ops per node $\sim 10^9$ nodes, 19 distribution functions per node
	Propagation step: no computations, 9 swaps per node $\sim 10^9$ nodes, 19 distribution functions per node

Table 2. Time loop with collision and propagation.

In two outer `mono` loops we iterate along the grid in x- and y-direction. These loops are supplemented by an inner `poly` loop in z-direction. On each PE, there is a `poly` pointer to `mono` data, pointing to $f[x][y][z]$. In the interacting active and background processes, the 19 distribution functions of one selected node are copied from the four-dimensional global `mono` array to the local (double buffered) one-dimensional `poly` array, using the asynchronously working library function `async_memcpym2p`. This transfer can be executed in blocks. In the next step, the data is processed and prepared for propagation.

In the fluid nodes, the distribution functions are employed to determine the local density and velocity with respect to (4.1) and (4.2). According to (4.3), the local equilibrium distribution is computed. With the relaxation process in (4.4), the computational part is finished. The data are written back to the `mono` domain via the asynchronous library function `async_memcpyp2m`. This step includes a subtlety: the processed data are not written back to their origin; the values of $\widetilde{f}_i$ are written to the storage space of the distribution functions belonging to the opponent discrete velocities. This preparation of the data enables the propagation step to be performed with 9 additional swaps per node without an intermediate storage of the data. This process is described in more detail in the documentation of the OpenLB project [6, 10]. The initiation and termination of the read and write processes are controlled via semaphores.

The boundary nodes at the lid have to be treated in a slightly different way. The necessary distinction is realized via a `poly if` conditional. The affected PE has to perform different instructions. Since both branches of the `poly` conditionals are performed sequentially, this ends up with a loss of performance by a factor of approximately two. The only difference in the branches is the disabling and enabling of the corresponding PEs. By a tricky application of the bounce-back boundary conditions, the boundary nodes at the solid walls can be treated in the same way as the fluid nodes.

In the following code fragment the described procedure is listed. For simplicity and facility of inspection, we abandon asynchronous communication and double buffering. The distribution functions are stored in the global array `double f[max_x][max_y][max_z][19]`. The velocity boundary values at the lid are provided by `double u_0[max_x][max_y]`. The constants w_i, $i = 0, \ldots, 18$, and the grid velocities $\mathbf{c}_i$, $i = 0, \ldots, 18$, are stored in `double weights[19]` and `int c[57]`.

```
mono int x,y,i;           /* mono loop variables */
poly int z;               /* poly loop variable  */
mono double* poly fpp;    /* poly pointer to mono data */
poly double fp[19];       /* poly array for distribution functions of one node */
poly double f_eq;         /* equilibrium distribution */
poly double rho;          /* density */
poly double ux,uy,uz;     /* x,y,z-velocity */
poly double sq, c_u;      /* auxiliary variables */
for(x=1;x<max_x-1;x++){   /* mono loop over fluid nodes* /
```

```
    for(y=1;y<max_y-1;y++) { /* mono loop over fluid nodes */
        for(z=z_PE; z<=z_PE+z_PE_size; z++) { /* local poly loop on each PE */
            fpp=f[x][y][z];  /* set poly pointer to mono data */
            memcpym2p(fp, fpp, 19*sizeof(double)); /* copy data to poly */
            rho = calc_rho(fp); /* calculate local density */
            if(z<max_z-1) {  /* poly conditional: fluid node */
                ux = calc_ux(rho,fp); /* calculate local x-velocity */
                uy = calc_uy(rho,fp); /* calculate local y-velocity */
                uz = calc_uz(rho,fp); /* calculate local z-velocity */
            }
            else if(z==max_z-1) { /* poly conditional: lid node */
                ux = u_0[x][y];  /* set local x-velocity */
                uy = 0.0;
                uz = 0.0;
            }
            /* compute local equilibrium */
            sq = 1.5 * (ux*ux + uy*uy + uz*uz);
            for(i=0;i<19;i++){
                c_u = c[i*3]*ux + c[i*3+1]*uy + c[i*3+2]*uz;
                f_eq = rho * weight[i] * (1. + 3.*c_u +4.5*c_u*c_u - sq);
                if(z<max_z-1)   /* poly conditional: fluid node */
                    /* update distribution functions */
                    fp[i] = fp[i] + w*(f_eq - fp[i]);
            }
            memcpyp2m(fpp,    fp,      sizeof(double)); /* copy data */
            memcpyp2m(fpp+1,  fp+10, 9*sizeof(double)); /* back to mono */
            memcpyp2m(fpp+10, fp+1,  9*sizeof(double)); /* and swap */
        }
    }
}
```

In each fluid node, approximately 350 floating point operations have to be performed. The maximum number of fluid nodes that can be handled in the CSX600's memory with 512 Mbytes are 3.5×10^6. Hence, more than one billion floating point operations are required per time step. Assuming the advertised performance of 25 GFLOPS, the computing time of the collision step would last around 0.05 seconds. The practical result is around 1.75 seconds. Hence, more than 97% of the time is spent in communication.

5.2.2. *The propagation step.* In the propagation step according to (4.5), there are no computations involved. Parallelization does only come into play, when copying the data to the PEs and swapping them in the write process to the `mono` domain. In this point, the available computing power plays absolutely no role. As an alternative, one can think of swapping in a sequential loop in the `mono` domain.
For each of the fluid nodes, 9 distribution functions have to be swapped with the neighbors in selected directions. Then in total all distribution functions are swapped and the propagation step is completed. We have to distinguish three different cases: fluid nodes, boundary nodes at the lid and boundary nodes at the bottom. The remaining boundary nodes at the sides of the cuboid can be treated in the same manner as the fluid nodes. Hence, we end up with three branches of the `poly if` conditional, which results in a loss of performance by approximately a factor of three.
The main disadvantage in comparison to the collision step is, that the distribution functions of the neighbors cannot be copied in blocks. This results in a further deterioration.

	Grid size $135 \times 135 \times 192$			Grid size $190 \times 190 \times 96$		
	Startup	Total	only Col.	Startup	Total	only Col.
Advance^{TM}board	19.0s	7494s	1844s	18.8s	8289s	1740s
Woodcrest	1.1s	5396s	2842s	1.1s	5377s	2814s
Desktop	1.5s	5626s	3454s	1.4s	5955s	3716s
Laptop	2.2s	7686s	4912s	2.2s	7686s	4861s

Table 3. Computing times for 1000 time steps on different grids and machines.

6. Numerical and performance results

In this section, the results of the performance tests are presented. Since the memory on the CSX600 processor is 512 Mbytes, the storage of approximately 3.5×10^6 grid points is possible. We consider cuboids with $135 \times 135 \times 192$ and $190 \times 190 \times 96$ nodes (135*135*192*19*8 bytes < 512 Mbytes, 190*190*96*19*8 bytes < 512 Mbytes). In these settings, each PE has to treat 135*135*2 or 190*190 nodes in each time step. For the evaluation of the performance of the parallel code, the results are compared with a sequential implementation. The sequential code is tested on a Dell 3.4 GHz Intel Pentium 4 dual-core Linux desktop, a Lenovo X60s 1.66 GHz Intel Centrino Duo dual-core Linux laptop and the Woodcrest compute server with two 2.66 GHz Intel Xeon 5150 dual-core processors running Fedora Core 5. We either consider the total computing times, as well as the computing times, when the propagation step is skipped and only the collision step is performed. The latter case does not make sense from a computational point of view, but it allows to focus on the more efficient part of the parallel implementation.

As fundamental result, the computing times of the parallel implementation on the ClearSpeed Advance^{TM}board cannot compete with the sequential code versions, except for the case when disregarding the propagation step. In Table 3, the corresponding results are listed.

As estimated in Section 5.2.1, the worse performance of the Advance^{TM}board has to be attributed to the time-wasting streaming of data from the `mono` to the `poly` domain and back. Due to the limited memory of 6 Kbytes on the PEs, there is no chance for an improvement. The asynchronous communication via `async_memcpym2p` and `async_memcpyp2m` turns out to be the bottleneck of the whole story. In the case of considering the collision step only, the computational power of the Advance^{TM}board has a considerable impact. In this case, clear advantages can be gained.

The delays are pronounced in the propagation step, where a blockwise copy of the data of neighboring nodes is impossible due to the position of data in the global array. A sequential execution of the propagation step in the `mono` domain leads to further deteriorations of the performance.

In Table 4 we investigate the performance for different problem sizes. We find that only one fourth of the total computing time is spent in the collision step, although this step involves all the computations. In the propagation step, the same number of read and write processes have to be performed. However, the data cannot be copied in blocks. This is the crucial point for performance interference.

In a further test, we want to assess the internal speed-up of the Advance^{TM}board. For this reason, only parts of the PEs of the CSX600 are enabled. In the extreme case, only one PE has to process all data. The corresponding results on a grid of $50 \times 50 \times 192$ nodes and 100 time steps are displayed in Table 5. The speed-up is computed by the formula $T(1)/T(p)$, where $T(p)$ denotes the computing time on p PEs. The efficiency $T(1)/(pT(p))$ of this test is plotted in Figure 7.

The execution of sequential code in the `mono` domain is quite slow. The idea of performing the propagation step without copying the data to the `poly` domain and writing back has to be discarded. The computing time for 100 time steps on a grid of $50 \times 50 \times 96$ nodes is extended to 1230 seconds, when implementing the propagation step in a sequential way. The full parallel version only takes 57 seconds, whereas the sequential version on the Linux desktop takes 36 seconds.

		100 time steps		1000 time steps	
Grid size	Startup	Total	only Col.	Total	only Col.
$20 \times 20 \times 192$	0.5s	16s	4s	159s	34s
$35 \times 35 \times 192$	1.3s	50s	13s	496s	114s
$50 \times 50 \times 192$	2.6s	105s	26s	1016s	240s
$75 \times 75 \times 192$	5.9s	238s	61s	2300s	556s
$100 \times 100 \times 192$	10.4s	424s	109s	4102s	1002s
$120 \times 120 \times 192$	15.0s	611s	158s	5917s	1452s
$135 \times 135 \times 192$	19.0s	774s	201s	7494s	1844s

TABLE 4. Computing times on the Advance™board for different problem sizes.

	Grid size $50 \times 50 \times 192$			
#PEs	Total	Speedup	only Col.	Speedup
1	8188s		1815s	
2	4160s	1.97	910s	1.99
4	2058s	3.98	457s	3.97
8	1038s	7.89	231s	7.86
12	697s	11.75	155s	11.71
16	538s	15.22	118s	15.38
24	358s	22.87	81s	22.41
32	274s	29.88	62s	29.27
48	189s	43.32	44s	41.25
64	147s	55.70	35s	51.85
96	105s	77.98	26s	69.81

TABLE 5. Internal speed-up on the Advance™board, 100 time steps.

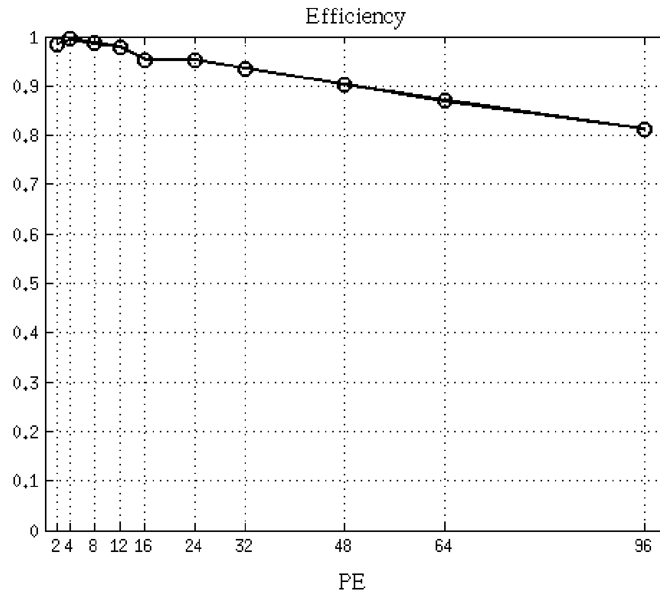


FIGURE 7. Efficiency on the Advance™board.

In the considered software release (SDK 2.23), there are no functions like `scanf` implemented for data input. File handling via `fstream`, `fopen` or `fprintf` is not yet supported. The only possibility left is output to the screen, which is accompanied with significant delays. Writing 4.6×10^5 double values (i.e. all distribution functions on a $5 \times 5 \times 96$ node grid) to a file by pipelining the screen output takes approximately 43 seconds. On the Linux desktop, the same task is completed within 0.05 seconds. Furthermore, no routines for time measurement were available in the SDK. Hence, it was a difficult task to rate the performance of the utilized routines and functions. The upcoming software releases include a visual profiling tool that allows for a detailed performance analysis.

7. Outlook

The ClearSpeed Advance^{TM}accelerator board is an innovative product, that makes high performance computing available on standalone workstations and aims at improving the performance of existing supercomputers. Despite the fact that not all functions and routines are implemented or tuned to performance at the current stage, this emerging technology has reached a stable state towards the use in numerical simulation.
The application of the ClearSpeed Advance^{TM}accelerator board to our test problem demonstrates the promising capabilities of this technology in scientific computing. However, our examination showed up some insufficiencies that should be fixed by future extensions in technical equipment and programming support.
The major constraint is the limited memory space on the processing elements, that prevents from working in a distributed memory sense. As a direct consequence, a huge amount of data has to be streamed from the `mono` domain to the `poly` domain and back. Due to the unsatisfactory performance of the data transfer, a large impact on the computing time has to be encountered. Improvements in the bandwidth by means of software or hardware adjustments could be a remedy to overcome the faced restrictions.
Another current handicap is related to the tedious linkage to the host processor or the secondary CSX600 processor. Via assembler code for both devices a common interface has to be defined for the exchange of data and communication. More user friendly solutions would be helpful.
Despite some deficiencies in the initial setup of the ClearSpeed Advance^{TM}accelerator board, this paper clearly shows the high potential of the underlying architecture and technology in the highly CPU-time demanding area of flow simulation.

8. Acknowledgements

The authors would like to thank ClearSpeed Technology for the kind disposal of the ClearSpeed Advance^{TM}accelerator board during the evaluation period from October 2006 to February 2007.

All mentioned products and brand names are trademarks or registered trademarks of their respective owners.

References

[1] R. Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann, San Francisco, 2001.
[2] B. Chopard and M. Droz. *Cellular automata modeling of physical systems*. Cambridge University Press, Cambridge, 1998.
[3] W. D. Gropp, E. Lusk, and A. Skjellum. *Using MPI : portable parallel programming with the message-passing interface (2nd ed.)*. MIT press, Cambridge, 1999.
[4] J.-L. Guermond, C. Migeon, G. Pineau, and L. Quartapelle. Start-up flows in a three-dimensional rectangular driven cavity of aspect ratio 1:1:2 at $Re = 1000$. *J. Fluid Mech.*, 450:169–199, 2002.
[5] J. L. Gustafson and B. S. Greer. ClearSpeed whitepaper: accelerating the Intel Math Kernel Library. *http://www.clearspeed.com/docs/resources/ClearSpeed_Intel_Whitepaper_Feb07.pdf*.
[6] V. Heuveline and J. Latt. The OpenLB project: an open source and object oriented implementation of lattice Boltzmann methods. *Int. J. Modern Physics C*, to appear.
[7] ClearSpeed Technology plc. ClearSpeed Software Description. *https://support.clearspeed.com/documents/*, 2007.

[8] ClearSpeed Technology plc. ClearSpeed whitepaper: CSX processor architecture. *http://www.clearspeed.com/docs/resources/*, 2007.
[9] ClearSpeed Technology plc. CSX600 Runtime Software User Guide. *https://support.clearspeed.com/documents/*, 2007.
[10] OpenLB project. *http://openlb.org*.
[11] M. C. Sukop and D. T. Thorne. *Lattice Boltzmann modeling*. Springer, 2006.
[12] J.-P. Weiß. *Numerical analysis of lattice Boltzmann methods for the heat equation on a bounded interval*. Universitätsverlag Karlsruhe, Karlsruhe, 2006.

[1] Numerik auf Höchstleistungsrechnern, Steinbuch Centre for Computing (SCC), University of Karlsruhe, Germany.
E-mail address: vincent.heuveline@rz.uni-karlsruhe.de, jan-philipp.weiss@rz.uni-karlsruhe.de
URL: http://numhpc.rz.uni-karlsruhe.de