Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Multi-GPGPU Cellular Automata Simulations using OpenCL

Sebastian Szkoda[a,c], Zbigniew Koza[a], Mateusz Tykierko[b,c]

[a] *sebastian.szkoda@ift.uni.wroc.pl Faculty of Physics and Astronomy, University of Wroclaw, Poland*
[b] *Institute of Computer Engineering, Control and Robotics, Wroclaw University of Technology, Poland*
[c] *Wrocław Centre for Networking and Supercomputing, Wroclaw University of Technology, Poland*

Abstract

The aim of this research it to examine the possibility of parallelizing the Frish-Hasslacher-Pomeau (FHP) model, a cellular automata algorithm for modeling fluid flow, on clusters of modern graphics processing units (GPUs). To this end an Open Computing Language (OpenCL) implementation for GPUs was written and compared with a previous, semi-automatic one based on the OpenACC compiler pragmas (S. Szkoda, Z. Koza, and M. Tykierko, Multi-GPGPU Cellular Automata Simulations using OpenACC, http://www.prace-project.eu/IMG/pdf/wp154.pdf). Both implementations were tested on up to 16 Fermi-class GPUs using the MPICH3 library for the inter-process communication. We found that for both of the multi-GPU implementations the weak scaling is practically linear for up to 16 devices, which suggests that the FHP model can be successfully run even on much larger clusters. Secondly, while the pragma-based OpenACC implementation is much easier to develop and maintain, it gives as good performance as the manually written OpenCL code.

1. Introduction

Frish-Hasslacher-Pomeau (FHP) model of fluid flow [1] is an important example of a cellular automaton, a broad class of numerical algorithms applicable in various areas of science and engineering [3], [4]. Our research on the FHP model carried out previously within PRACE-3IP [2] has shown promising results for OpenACC and CUDA ports on a single 8-GPU computing node. Therefore the main goal of the present work is to examine the possibility of porting the FHP algorithm to a cluster of GPU accelerators. From the two general-purpose computing environments available currently for GPUs, CUDA and OpenCL, we chose the latter one for its versatility, and compared the results with those obtained using another GPU programming model, pragma-based OpenACC, which is also versatile but much simpler to use.

2. Model

The FHP model of fluid flow is a cellular automaton in which particles forming the liquid are allowed to move only along the bonds of a triangular lattice [3], [4], hopping in discrete time steps Δt to one of the 6 neighboring lattice nodes. The exclusion rule asserts that only one particle is allowed to move along a given bond in a given direction. The particles can meet at lattice nodes and collide, exchanging the momentum and changing their velocities. The evolution from time t to $t + \Delta t$ takes part in two steps: propagation and scattering. In the propagation step each particle moves one node at a time in accordance to its current velocity. In the scattering step particles collide, changing their directions according to the rules imposed by the physical laws of mass and momentum conservation, see Fig. 1. If a particle hits an obstacle, it usually bounces back to mimic the no-slip boundary conditions typical of fluid flow simulations.

There are several versions of the model, each characterized by distinct collision rules. Here we consider the so called FHP I model, in which each lattice node can be occupied by up to six particles, one per direction represented by internode links (bonds). The velocities of the particles shall be denoted by v_i, $i = 0,\dots,6$.

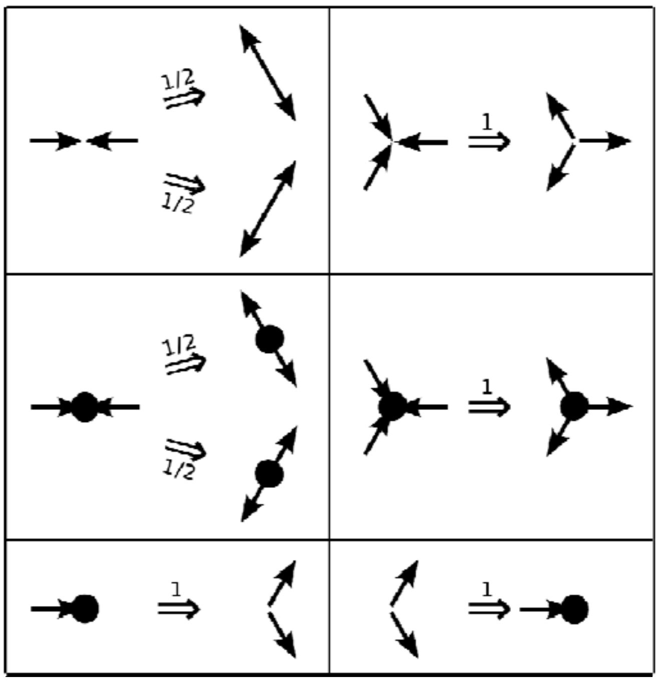

Figure 1 The FHP I and FHP II collision rules. The states before and after the collision are visualized in the first and second column, respectively. The first row shows FHP I collision rules, and the whole table presents the FHP II collision rules.

3. FHP with a multi-spin technique

In the multi-spin coding approach particle velocities at each lattice node can be represented by single bits and stored in independent arrays x_i, $i=1,\dots,6$. Each array stores information about the presence (bit set to 1) or absence (bit set to 0) of the particle in a particular direction. The lattice is mapped into a large parallelogram, Figure *2*.

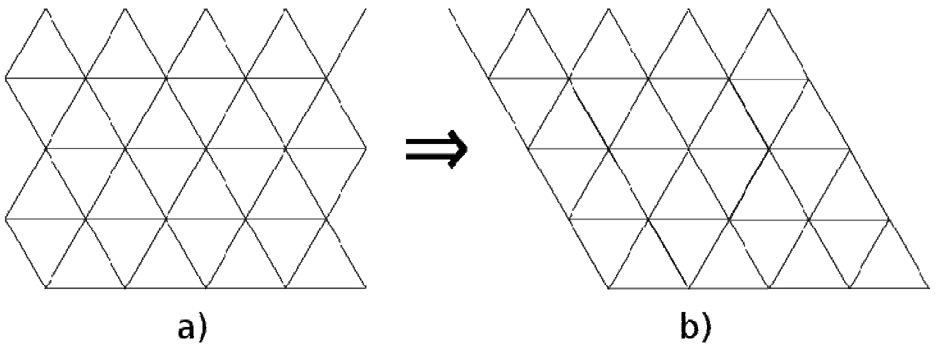

Figure 2 A triangular lattice (a) mapped onto parallelogram (b).

For effective scattering, instruction-level and data-level parallelism of the FHP model can be used. Such an approach involves dividing the data into separate bits and distributing them appropriately in computer words. Then it is possible, through logical and arithmetic bitwise operations, to execute more than one operation per processor clock. For the propagation step to be performed efficiently, a special ordering of bits representing particles in lattice nodes in arrays x_i must be applied. The first N/B nodes are assigned in the first bits of subsequent words, then next N/B nodes are assigned to the second bits of consecutive words, and in such manner repeated B times, where B is the maximum computer word length. Most often B equals 64, being the length of the *long long integer* type. The aim of such a data distribution is to perform the particle movement through a simple assignment of words.

Besides the six velocity arrays, one still needs an array holding the placement of obstacles and an array of random bits. The array of random bits used in the scattering step to mimic the stochastic choice of the scattering pattern for the cases where a given collision can result in two collisions with probability 1/2, Fig 1.

4. Implementations

In this report we compare two implementations of the FHP I model, an OpenCL-based and an OpenACC-based [7], the latter described in detail in [2]. Both of them use the multi-spin coding technique.

4.1. OpenCL implementation

OpenCL [6] is a free programming standard for general purpose parallel programming, unifying code development techniques for CPUs, GPUs and other processors. It is designed to take the full advantage of heterogeneous processing platforms, such as multi-core CPUs and GPUs, including also DSP's, the Cell processor etc. OpenCL consists of a compiler-independent runtime API and the OpenCL C language. The runtime API is meant to be used by the flow control program written in C or C++ to execute computing kernels written in OpenCL C. OpenCL C is almost the standard C language that enables writing computing kernels which are loaded into flow control program during runtime. This feature enables lunching the same binary on various hardware platforms, swapping only the computing kernels in such way that they fit the best accelerator available in the computational environment.

In this programming model all computing devices (e.g., CPU cores, GPUs) are treated as co-processors running kernels asynchronously to the work flow control program. In the case of the FHP implementation, the main task of the control program is to transfer the data to the GPU, launch GPU computational kernels in an appropriate order, and read the results from the GPUs. The main application was written in C++ and compiled with the GCC 4.7.2 compiler set, while the computational part uses the OpenCL 1.1 implementation from Nvidia

SDK 5.5.22. Nvidia's OpenCL implementation was chosen as it seems to be the most appropriate for their devices.

In our OpenCL implementation of the FHP model, the CPU subroutines presented in PRACE-3IP [2] were reimplemented as OpenCL computing kernels. Kernel implementations differs slightly from the Nvidia CUDA one described in [2]. However, the host code necessary to prepare the computations on the GPU is definitely longer and more complicated in the case of the OpenCL, which is the price of OpenCL's versatility.

In the collision kernel (Listing 1) the local (shared) memory was used to optimize the multiple memory reads of arrays x_i and *nob*. The value 32 for the local workgroup size gives the best performance results. In the remaining kernels the use of the local memory wouldn't be as profitable, as they use only simple memory assignments. The multi-GPU communication was realized through the MPICH3 implementation of the Message Passing Interface (MPI).

Before the simulation begins, the system is initialized on N computing systems where N is the number of GPUs, with the first H/N rows assigned to the first GPU, the next H/N rows to second GPU and so on. Each GPU is attached to one MPI process which creates one context and one command queue. The communication is provided by the non-blocking send and receive calls. The N-th process sends the first row of particles heading towards 0 and 1 directions to the one with the previous rank, and the last row of particles with velocities 3 and 4 to the next process. At the same time, it receives the appropriate rows from the next and previous processes. Due to the impermeable boundary conditions on the top and bottom edges of the system, the first and last processes communicate only with one neighboring process, either the next or the previous one.

To optimize inter-process communication, before the sending process begins, the first row of arrays holding particles with velocities pointing in directions 0 and 1 are aggregated into one buffer to perform one send operation on larger data array. The same applies to the last rows of directions 3 and 4. Particles that have just been sent to another process have to be cleared in the current one. In the multi-spin coding implementation, particles in each direction are stored in different arrays so they will not be overwritten by incoming data. The row clearing was implemented by device's internal memory copying of the previously prepared zero-filled buffer. This operation is done during non-blocking MPI communication. This kind of optimization brings a noticeable benefit only when we consider strong-scaling calculations.

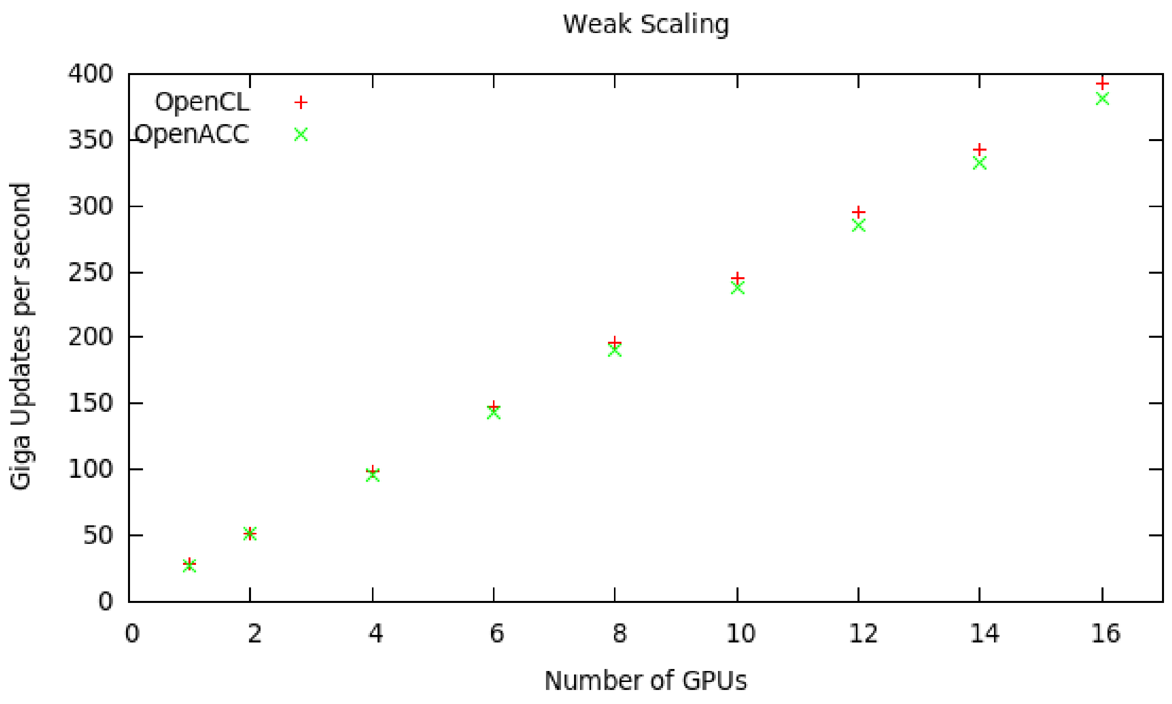

Figure 3 The weak scaling on 8 nodes with 2 NVIDIA M2050 cards.

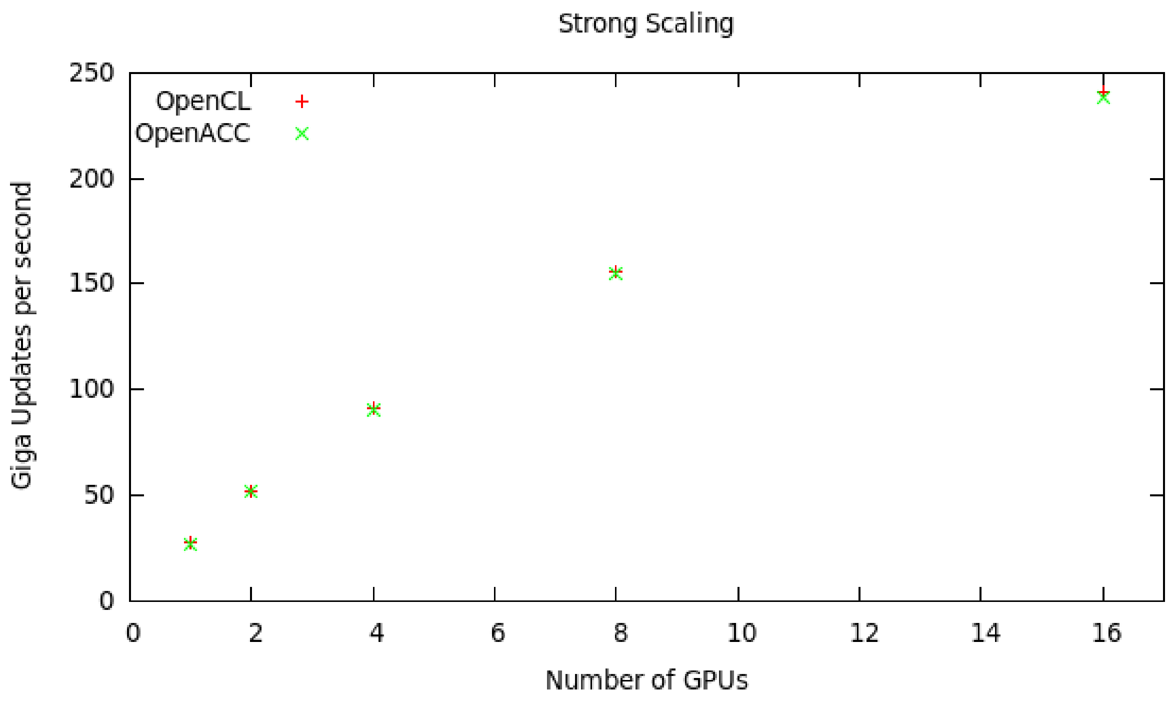

Figure 4 The strong scaling on 8 nodes with 2 NVIDIA M2050 cards.

5. Results

Using 8 professional HPC computing nodes with two Nvidia M2050 units each, we could go up to 400 GUps (giga lattice updates per second). The result is one order of magnitude faster than for the single GPU implementation, which still is 4 times faster than the fastest CPU implementation (OpenMP + SSE) on a computing node with Intel Xeon X5670, considering weak scaling. The results for the weak and strong scaling for up to 16 GPUs are shown in Figure 3 and Figure 4, respectively. The weak scaling is almost linear for both implementations. Figure 3 and Figure 4 also presents results for an alternative OpenACC, Multi-GPU implementation, presented in the previous work [2]. Performance results of both implementations are equal, but it is important to remark that the multi-spin implementation considered here is perfectly suited for SIMD devices which allows semi-automatic OpenACC porting to be equally efficient as the hand-written code. Probably such a situation is rather rare.

6. Listings

```
#define TYPE unsigned long int
#define BS 32 //Local work size
#define NEXTY(cw,ccw,same,oop)\
    ((( cw )  & ( cang ))+(( ccw ) & ( caang ))+\
    (( same ) & ( ncol ))+(( oop ) & ( ~tnob[i] )))
TYPE col,ncol,cang,caang;
__kernel void sub_step_1(...)
{
  __local TYPE t1[BS],t2[BS],t3[BS],t4[BS];
  __local TYPE t5[BS],t6[BS],tnob[BS];
  unsigned id = get_global_id(0);
  unsigned i  = get_local_id(0);
  t1[i]=x1[id]; t2[i]=x2[id];
  t3[i]=x3[id]; t4[i]=x4[id];
  t5[i]=x5[id]; t6[i]=x6[id]; tnob[i]=nob[id];
  ncol =(\
        (t1[i]^t4[i])|(t2[i]^t5[i])|\
        (t3[i]^t6[i])\
        )&\(\
        (t1[i]^t3[i])|(t3[i]^t5[i])|\
        (t2[i]^t4[i])|(t4[i]^t6[i])\
        )&(tnob[i]);

  col     = (~ncol) & ( tnob[i]);
  cang    = (  col) & ( ang[i]);
  caang   = (  col) & (~ang[i]);
  ang[i]  = (  col) ^ ( ang[i]);

  y1[i] = NEXTY(t2[i],t6[i],t1[i],t4[i]);
  y2[i] = NEXTY(t3[i],t1[i],t2[i],t5[i]);
  y3[i] = NEXTY(t4[i],t2[i],t3[i],t6[i]);
  y4[i] = NEXTY(t5[i],t3[i],t4[i],t1[i]);
  y5[i] = NEXTY(t6[i],t4[i],t5[i],t2[i]);
  y6[i] = NEXTY(t1[i],t5[i],t6[i],t3[i]);
}
```

Listing 1 Collision step OpenCL kernel..

7. Acknowledgments

This work SS This work was financially supported by the PRACE-2IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493. This work was financially supported by the Polish Ministry of Science and Higher Education (funds for R\&D in 2013-2014). This research was supported in part by PL-Grid Infrastructure. Calculations were carried out in Wroclaw Centre for Networking and Supercomputing (http://www.wcss.wroc.pl).

References

[1] Frisch, U., Hasslacher, B., and Pomeau, Y. (1986) Lattice gas automata for the Navier-Stokes equation. Phys. Rev. Lett., 56, 1505.
[2] S. Szkoda, Z.Koza, M.Tykierko (2014), Multi-GPGPU Cellular Automata Simulations using OpenACC, http://www.prace-project.eu/IMG/pdf/wp154.pdf
[3] Wolfram, S. (2002) A new kind of science. Wolfram Media, Champaign.
[4] Chopard, B. and Droz, M. (1998) Cellular automata modelling of physical systems. Cambridge Univ. Press, Cambridge.
[5] Kohring, G. (1992) The cellular automata approach to simulating fluid flows in porous media. Physica A, 186, 97–108.
[6] Khronos OpenCL Working Group (2011) The OpenCL Specification.
[7] Portland Group (2010) PGI Fortran & C Accelerator Programming Model new features ver. 1.3.
[8] Johnson, M. G. B., Playne, D. P., and Hawick, K. A. (2010) Data-parallelism and GPUs for lattice gas fluid simulations. Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA10), Las Vegas, USA, 12-15 July, pp. 210–216. CSREA. PDP4521.