Performance Model for a Cellular Automata Implementation on a GPU Cluster

Paul ALBUQUERQUE, Pierre KÜNZLI and Xavier MEYER
Institute for Informatics and Telecommunications (inIT)
University of Applied Sciences of Western Switzerland, 1202 Geneva, Switzerland

Abstract. In the past few years, GPUs have gained a lot of popularity as they offer an opportunity to accelerate many algorithms. In this paper we present a mono, a multi-GPU and GPU cluster implementation, based on CUDA, of a cellular automata. We derive a performance model and establish its accurateness.

Keywords. GPU cluster, CUDA, Parallel Programming, Performance Model, Cellular Automata

Introduction

The always increasing complexity and size of problems tackled by the industry drives the demand for more computational power. A few years back, in order to meet the demand for processing power, graphics card vendors made their *Graphics Processing Units* (GPU) available for general-purpose computing. Since then GPUs have gained a lot of popularity as they offer an opportunity to accelerate algorithms having an architecture well-adapted to the parallel GPU model. *Cellular Automata* (CA) fall into this category because of their intrinsic massive parallelism. Lattice Boltzmann methods, which were inspired by a particular class of CA, are computational fluid dynamics methods used for example in the simulation of blood flows. The heavy computations involved and their parallel nature make them ideal candidates for a GPU implementation. However, the subject is clearly not new ([1], [2], [3], [4]). The goal of this paper is to quantify the benefit that stems from parallelization of a CA on a GPU cluster. Towards this end, we propose a performance model for implementing a CA on a GPU and then extend it to the case of a GPU cluster. We validate it on a series of simulations. To focus on methodology, we chose a simple and well-known CA: the Game of Life. Our GPU implementation is based on the CUDA (Compute Unified Device Architecture [5])) technology of NVIDIA, and at the cluster level we use MPI (Message Passing Interface) to manage inter-GPU communications.

The paper is organized as follows. The next section covers the basics of GPU architecture and its peculiarities. In the following one, we focus on the GPU implementation and then explain how we split the CA domain to obtain the GPU cluster version. In the fourth section, we describe the performance models related to these implementations. Finally, we summarize the results obtained.

1. GPU Architecture: CUDA

We focus on CUDA, NVIDIA's GPU parallel computing architecture, which belongs to the *Single Instruction Multiple Data*[1] category. Its key feature is to exploit data parallelism and offer fast context switching for the threads. Recent generation GPUs display TFLOPS performances.

A GPU acts as a device controlled by the CPU. Both processing units share data via the GPU global memory (GRAM). If a computer contains several GPUs, each GPU must be managed by its own CPU thread. In a standard application, the CPU manages the data and the global algorithm, while the GPU does the computing[2]. Due to the nature of GPU architecture, conditional branching instructions are usually delegated to the CPU to avoid a breakdown on parallelism.

The GPU architecture is organized around two important notions: work decomposition and memory levels. Work decomposition describes how a computation can be decomposed using multiple threads. The parallel processing is related to the architecture of the computing units on a GPU. The latter contains several *Streaming Multiprocessors* (SM), each having a fixed number of processing units that run instructions in parallel from a single instruction unit. Thus, each thread on a multiprocessor must execute the same instruction to ensure true parallelism. The hierarchy of processing units leads to decomposing a computation over a multitude of threads, each of them doing mainly the same work. Consider for example the case of a matrix addition: **A=B+C**. Each thread could load a single element B_{ij} and C_{ij} from the matrices, sum both values and store the result into A_{ij}. Threads are grouped into blocks which will be dispatched to the SMs. Inside a block, threads are grouped into so-called *warps*. Blocks are further organized into a grid of blocks, so as to provide a unique ID for each thread. The ID of a thread consists of the position of the thread in its block and the position of the latter on the grid. This ID allows to access individual data during the computations.

There are different memory levels. A thread has its own registers on which the instructions are applied. Data accesses on registers are the fastest. At another level, each block has its own shared memory accessible only by its own threads. This memory is nearly as fast as the registers when correctly accessed. More importantly, shared memory is the safest and fastest way to communicate between threads. Since SMs are not synchronized instruction-wise, threads on different blocks cannot communicate directly. Finally, the GRAM is a global memory within the scope of all threads as well as the CPU. But access to this memory is costly since, in addition to the read/write instruction, it takes 400-600 cycles before the data is available from the GRAM.

A GPU architecture uses pipelines to improve performances. There is a fixed depth (D) pipeline depending on the number of processors per SM, because a *warp* consists of D batches whose threads run concurrently on the SM. Another pipeline is used to hide the latency due to GRAM accesses. The scheduler swaps *warps* waiting for data, with *warps* ready to be executed. The more warps, and thus threads, per block, the more efficiently this latency can be hidden.

[1]Referred to as *Single Instruction Multiple Threads* by NVIDIA.

[2]As in a master-slave paradigm.

2. Implementations

2.1. The Game of Life

A CA consists of a number of cells which are spatially organized according to some geometry. Each cell is characterized by an internal state belonging to a finite set. The geometry also defines the notion of a neighborhood of a cell. The state of a cell evolves through time by applying an evolution rule. This rule computes the new state of a cell as a function of its current state and those of its neighbors.

We now consider Conway's famous Game of Life. The CA domain is a square grid of 2-state cells (dead=0 or alive=1), with periodic boundary conditions. The neighborhood of a cell consists of its 8 adjacent cells (north, south, east, west plus diagonals). The evolution rule asserts that, at the next generation, a cell having

- exactly 3 living neighbors, will become or remain alive;
- exactly 2 living neighbors, has its state unchanged;
- less than 2 or more than 3 living neighbors, will die or remain dead.

We consider a standard and simple sequential implementation of the Game of Life, which will serve as a basis for its parallelization (see listing 1).

Listing 1 Sequential implementation of the Game of Life

```
void nextGenSeq(int* current, int* future, int nrow, int ncol) {
   for (int i=0,j=0; i<nrow,j<ncol; i++,j++) {
      int nb = current[north(i,j)]+...+current[southeast(i,j)];
      if (nb == 3) future[index(i,j)] = 1; //alive
      else if (nb<2 || nb>3) future[index(i,j)] = 0; //dead
   ...
}
...
for (int nbGen=0; nbGen<maxGen; nbGen++) {
   nextGenSeq(current,future,nrow,ncol);
   swap(current,future);
}
```

2.2. Single GPU

Initially, the whole domain is copied from the RAM to the GRAM. Then the same kernel is called every generation, and during its execution, each CUDA thread updates one cell. Since threads cannot be synchronized across different blocks, the generation loop must be placed outside the kernel. Data remains in GRAM between kernel calls, and will thus be available at the next kernel call (listing 2).

A block consists of $NTX \cdot NTY$ threads[3], while the dimension of the grid is $((ncol + NTX - 1)/NTX) \times ((nrow + NTY - 1)/NTY)$. A thread computes its cell (x, y)-coordinates as $x = blockIdx.x \cdot NTX + threadIdx.x$ and $y = \ldots$

[3] NTX, NTY: block dimensions in x, y; $NTX = NTY = 16$ was chosen empirically; $ncol, nrow$ = number of columns, rows of the domain

Listing 2 Host code for the CUDA implementation of the Game of Life

```
int* current,future;
cudaMalloc((void**)&current,nrow*ncol);
cudaMemcpy(current,currentHost,nrow*ncol,cudaMemcpyHostToDevice);
dim3 blocks((ncol+NTX-1)/NTX,(nrow+NTY-1)/NTY),threads(NTX,NTY);
for (int nbGen=0; nbGen<maxGen; nbGen++) {
   nextGenKernel<<<blocks,threads>>>(current,future,nrow,ncol);
   cudaThreadSynchronize();
   ...
}
```

To avoid reading the 8 neighbors of a cell from GRAM, the necessary data is loaded into the shared memory. The domain is thus subdivided into as many subdomains as there are blocks of threads. The shared memory being local to each block, the neighborhood data must also be loaded (fig. 1). Note that some cells are neighbors to many blocks. Once the data is loaded into shared memory, threads require no further reading from GRAM. The usage of the shared memory of the blocks decreases the number of GRAM accesses.

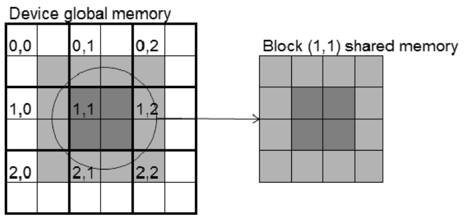


Figure 1. Loading data into shared memory: in dark grey the data to be updated by the block and in light grey the neighboring data required

Also note that data is stored optimally in GRAM regarding data access. GPUs are able to retrieve more than a single data in one instruction whenever the addresses are coalesced[4]. For example, a streaming multiprocessor is able to read single precision variables for up to 16 threads in a single instruction, if the addresses are coalesced. Such accesses substantially increase the global memory throughput.

2.3. Multi-GPU

We consider the case where a host manages many GPUs (as for example with a GPU NVIDIA Tesla S1070). The host should have as many cores as GPUs.

We use the same algorithm as in the previous section. However, some communications must be added to synchronize the GPUs. We split the domain horizontally across the GPUs, since this is simple, efficient, and reduces communications.

After initializing the domain, the CPU creates threads to manage the GPUs. Each CPU thread initializes its GPU and loads a subdomain in GRAM, before

[4]CUDA concept for successive data addresses.

Listing 3 CUDA kernel using block shared memory for the Game of Life

```
 __global__ void nextGenKernelShared(int* current, int* future,
                                   int nrow, int ncol) {
   int myX = blockIdx.x*NTX+threadIdx.x, myY = ...;
   int myPos = ncol*myY+myX;
   __shared__ int sharedArray[NTX+2][NTY+2],ymin,ymax,xmin,xmax;
   ...
   //thread position in block shared memory
   int myXs = threadIdx.x+1, myYs = threadIdx.y+1;
   ...
   __syncthreads();
   if (threadIdx.y == 0) //loading neighborhood
      sharedArray[myXs][myYs-1] = current[ncol*ymin+myX];
   ...
   if (threadIdx.x == 0 && threadIdx.y == 0) {
      int subX = myPos+NTX-ncol, subY = myPos+NTY-nrow;
      ...
      sharedArray[NTX-subX][NTY-subY] = current[ncol*ymax+xmax];
   }
   //loading remaining data, each thread loads one element
   if (myX<ncol && myY<nrow) {
      sharedArray[myXs][myYs] = current[myPos];
      __syncthreads();
      //number of neighbors alive computed from block shared memory
      int nb = sharedArray[myXs-1][myYs]+sharedArr[myXs][myYs-1]+..
      if (nb == 3) future[myPos] = 1;
      if (nb<2 || nb>3) future[myPos] = 0;
      if (nb == 2) future[myPos] = sharedArray[myXs][myYs];
   }
}
```

starting the generations. The communication between the GPUs is handled by the CPU threads via memory copying in the CPU main memory (fig. 2, left).

GPUs cannot communicate directly: communications take place through the CPU threads. This can either be done using POSIX or MPI[5]. The CPU threads communicate using a shared data structure stored in the CPU memory, and POSIX barriers permit to remain synchronized at each generation.

2.4. GPU Cluster

A MPI program consists of a number of processes (`nbPE`) assigned to computing nodes. Each process has an identifier (`myPE`). As in the multi-GPU case, the domain is split horizontally. Process 0 initializes the domain and scatters it across all processes. Then each process computes the identifier of its below and above

[5]Message Passing Interface

neighbor processes. Every generation, processes exchange missing boundary cells (fig. 2, right) via calls to the function `MPI_Sendrecv()` (listing 4).

Listing 4 Domain boundary exchange

```
int belowPE = (myPE+1)%nbPE, abovePE = (nbPE+myPE-1)%nbPE;
for (int nbGen=0; nbGen<maxGen; nbGen++) {
   MPI_Sendrecv(topRow,belowPE,bottomRow,abovePE);
   MPI_Sendrecv(bottomRow,abovePE,topRow,belowPE);
   nextGenMpi(arrayLocal,newGen,upRow,downRow,nrowLocal,ncolLocal);
   ...
}
```

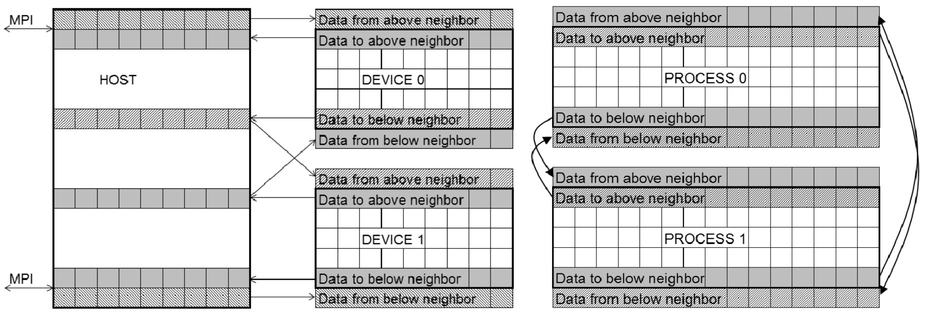


Figure 2. Left: data exchange between host and devices using `cudaMemCpy()`. Right: illustration for a domain distributed over 2 processes

3. Performance Model

3.1. Modelling a Kernel

A CUDA kernel performance model must take into account the GPU architecture (section 1) which decomposes computations into threads, warps and blocks. One must also pay attention to GRAM accesses and to how the pipelines reduce the associated latency. Our model is inspired by [6], but we should also mention [7].

The total work C_T done by a thread depends on the number of GRAM accesses and arithmetic instructions. In CUDA, a single precision add costs 4 cycles and a division 36 cycles, while a GRAM access amounts to a 4 cycle instruction (read/write) followed by a non-blocking latency of 400-600 cycles. The latter may be used to hide arithmetic instructions that are concurrent to memory accesses.

We now determine how many threads are run by a processor. A kernel decomposes into blocks of threads. Thus each streaming multiprocessor (SM) has to execute N_B blocks. Each of these blocks has itself N_W warps consisting of N_T threads. A SM has N_P processors which execute batches of threads in a pipeline of depth D. Thus the total work C_{Proc} done by a processor is given by

$$C_{Proc} = N_B \cdot N_W \cdot N_T \cdot C_T \cdot \frac{1}{N_P \cdot D} \tag{1}$$

Finally, the execution time of a kernel is $T_{Kernel} = T_{launch} + C_{Proc}/freq$ where $freq$ is the processor frequency and T_{launch} the launch time of the kernel.

The number of blocks per SM is $N_B = N_{cells}/(NTX \cdot NTY \cdot N_{SM})$ where N_{SM} is the number of SM (recall that $NTX \cdot NTY$ is the size of a block). Since the number of warps $N_W = NTX \cdot NTY/N_T$, we obtain $C_{Proc} = \dfrac{N_{cells} \cdot C_T}{N_{SM} \cdot N_P \cdot D}$. Note that N_{SM}, N_P, D, N_W are GPU characteristics.
The values of C_T and T_{launch} will be inferred from simulations.

3.2. Single GPU and GPU Cluster Performance Model

The single GPU implementation requires CPU-GPU memory transfers only at initialization and to gather the data after the final generation. It is not useful to include these in the model. Denoting by g the number of generations and $freq$ the processor frequency, the execution time on a single GPU is thus given by

$$T_{GPU} = g \cdot T_{kernel} = g \left(\frac{N_{cells} \cdot C_T}{N_{SM} \cdot N_P \cdot D \cdot freq} + T_{launch} \right) \tag{2}$$

On a GPU cluster, the domain is split across the GPUs and the local domain thus has a size N_{cells}/N_{GPU} where N_{GPU} is the number of GPUs in the cluster. Hence, the total work C_{Proc} is divided by N_{GPU}. Every generation, it is also necessary to transfer the boundaries of the local domains from the GPU to the CPU memory, then across the network and back from the CPU to the GPU memory. The transfer times depend on the latencies and the bandwiths of the PCIe bus and the network. Therefore, we must add to T_{kernel} a transfer time $T_{trans} = T_{lat} + N_{boundary}/bw$ where T_{lat} is the sum of latencies and bw the equivalent bandwith. The time T_{trans} will also be inferred from simulations.

The execution time $T_{cluster} = g \cdot (T_{kernel} + T_{trans})$ on a GPU cluster is thus

$$T_{cluster} = g \left(\frac{N_{cells} \cdot C_T}{N_{GPU} \cdot N_{SM} \cdot N_P \cdot D \cdot freq} + T_{launch} + T_{trans} \right) \tag{3}$$

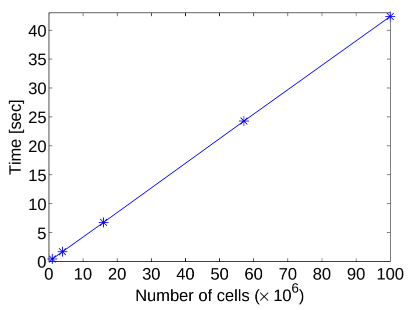
4. Measurements and Validation of the Model

Our test environment is a GPU cluster with $N_{GPU} = 10$. Each GPU (a NVIDIA GTX285) has 4GB GDDR3 of GRAM, $N_{SM} = 30$ streaming multiprocessors, each holding $N_P = 8$ processors ($freq = 1.48[GHz]$). The pipeline depth is $D = 4$.

All our simulations ran over a 1000 generations. We determined $C_T = 605$ and $T_{launch} = 2.66 \cdot 10^{-5}[s]$ by running simulations on a single GPU where we measured the execution time as a function of the domain size (fig. 3, left).

Sequential simulations on the host CPU and a domain of size 7560×7560 appeared to be a 100 times slower than on the GPU. This is only indicative since our sequential implementation is not optimized. One must be cautious with these comparisons, since the architectures are different.

We then measured timings and speedup on a GPU cluster (fig. 3, right). The bandwith contribution is negligible because the amount of data transfered is small. So the main contribution to T_{trans} is $T_{lat} = 5.3 \cdot 10^{-4} [s]$ (determined empirically). Correlation between our performance model and measurements was above 99%.



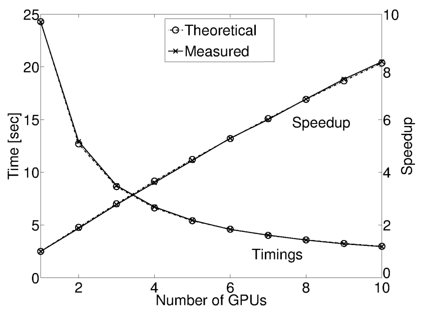


Figure 3. Left: single GPU execution time vs. domain size over a 1000 generations. Right: timings and speedup on a 10 GPU cluster for a domain size of 7560×7560 over a 1000 generations.

5. Conclusion

In this paper, we presented a performance model for a single GPU and a GPU cluster implementation of a CA. This model was validated on a series of simulations. Our focus was on methodology since the performance model is simple and straightforward. Even though it is quite obvious that CA are well-adapted to GPUs, the emphasis here was to provide tools to quantify this fact and predict what acceleration one may expect from a parallelization on a GPU cluster.

References

[1] L. Zaloudek et al. GPU Accelerators for Evolvable Cellular Automata. In *Computation World*, (2009), 533–537.

[2] J. Zhou et al. Multiple-GPUs Algorithm for Lattice Boltzmann Method. In *Int'l Symp. on Information Science and Engineering*, (2008), 793–796.

[3] M. Bernaschi et al. A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. *Concurrency Computations: Practical Experiments*, **22** (2010), 1–14.

[4] Y. Zhao. Lattice Boltzmann based PDE solver on the GPU. *The Visual Computer: International Journal of Computer Graphics*, **24**(5) (2008), 323–333.

[5] NVIDIA. *CUDA Compute Unified Device Architecture : Programming Guide*, 2008.

[6] K. Kothapalli et al. A Performance Prediction Model for the CUDA GPGPU Platform. Technical report, Int'l Inst. of Information Technology, Hyderabad, 2009.

[7] D. Schaa D. and D.R. Kaeli Exploring the multiple-GPU design space. In *Int'l Parallel and Distributed Processing Symp.*, (2009), 1–12.