Vectorization of Cellular Automaton-Based Labeling of 3-D Binary Lattices

Peter Zinterhof

Abstract Labeling connected components in binary lattices is a basic function in image processing with applications in a range of fields, such as robotic vision, machine learning, and even computational fluid dynamics (CFD, percolation theory). While standard algorithms often employ recursive designs that seem ill-suited for parallel execution as well as being prone to excessive memory consumption and even stack-overflows, the described new algorithm is based on a cellular automaton (CA) that is immune against these drawbacks. Furthermore, being an inherently parallel system in itself, the CA also promises speedup and scalability on vector supercomputers as well as on current accelerators, such as GPGPU and Xeon PHI.

1 Introduction

Labeling connected components in binary lattices is a basic function in image processing with applications in a range of fields, such as robotic vision, machine learning, and even computational fluid dynamics (CFD, percolation theory). While standard algorithms often employ recursive designs that seem ill-suited for parallel execution as well as being prone to excessive memory consumption and even stack-overflows, the described new algorithm is based on a cellular automaton (CA) that is immune against these drawbacks. Furthermore, being an inherently parallel system in itself, the CA also promises speedup and scalability on vector supercomputers as well as on current accelerators, such as GPGPU and Xeon PHI.

The discussed algorithm for finding connected components within 3-dimensional lattices is based on a Cellular Automaton (CA) [1] which is a classic and well-studied tool in computer science.

In general, Cellular Automata operate on a set of cells (e.g. pixels or data items) which are manipulated according to a set of transition rules, which can be applied to all cells sequentially or in parallel. The manipulation is repeated iteratively until

P. Zinterhof (✉)
Department of Computer Science, University of Salzburg, Jakob-Haringer-Str. 5, 5020 Salzburg, Austria
e-mail: peter@zinterhof.com

M.M. Resch et al. (eds.), *Sustained Simulation Performance 2017*,
DOI 10.1007/978-3-319-66896-3_6

Table 1 Execution times (ms) of dense CA of varying dimensions

CA dim	GTX680	NEC ACE-SX	Tesla P100	Xeon E1620	Xeon Phi 5110
128	0.25	0.57	0.26	11.37	n/a
256	1.86	3.71	0.69	68.85	n/a
384	6.53	13.37	5.8	189.88	n/a
512	15.55	31.81	5.8	399.86	n/a
704	49.55	79.03	13.98	772.18	42.44

some stopping criterion has been reached, for instance an equilibrium condition in which no further changes do occur or some runtime constraint has been met.

Among a series of convenient characteristics of Cellular Automata we want to emphasize their decent memory requirements which in most cases will be fixed during runtime and proportional to the number of cells while being agnostic to cell states. Also, updating cells in a CA is an operation that shows very high degrees of data-locality, which by itself can be regarded as an important prerequisite in the context of implementations for massively parallel and even distributed systems.

We consider these properties to be quite an advantage over recursive algorithms for finding connected components, which display patterns of memory consumption that are related both to the number and states of lattice cells. This makes CA-based computation of connected components an attractive choice for tightly memory restricted computer systems, in some cases probably even the only viable choice. Additionally, two very important advantages of the proposed algorithm can be named by the homogeneity of computational intensity within the lattice of cells, and the high regularity of memory access patterns during the iterations of the algorithm. Both specifics lend themselves well to high performance implementations on parallel systems.

As the set of transition rules can be applied to all cells of the lattice in parallel, the computational core of the algorithm is inherently parallel, too.

Our main contribution is given by the definition and discussion of vectorized and parallel implementations of the basic CA-algorithm on a variety of recent vector- and parallel compute architectures (Table 1).

2 Related Work

This work is based on the important paper by Stamatovic and Trobec [4] which introduces a new method of computing connected components in binary lattices by application of the well-known theory of cellular automata[1] in a new way. Compared to [6] we concentrate on the 3-D case instead of the 2-D case of input data.

[1]A comprehensive introduction to CA theory can be found in [2].

Table 2 Typical memory requirements (GB) for Matlab function 'bwconncomp' and reported CA-based implementations

CA dim	Matlab	Cellular automaton
256	1.83	0.13
512	11	1.0
768	29	3.6

The considerably higher RAM-requirements for Matlab's 'bwconncomp' function may also lead to swapping on some systems and CA dimensions, which will not occur in the CA-based counterpart

Stamatovic and Trobec [4] also covers the 3-D case but puts more emphasis on the discussion of algorithmic details and the general proof-of-concept by displaying implementations in Matlab and NetLogo while this contribution is focused on various aspects of high-performance implementations on parallel hardware. The well-known Matlab software environment also offers some built-in function 'bwconncomp' which is capable of computing connected components within multi-dimensional arrays. Despite being convenient to use, Matlab's implementation falls short with regards to performance, memory requirements (Table 2) and potential use of accelerators, such as GPUs.

Other related work includes the class of stencil-based algorithms, which are not widely regarded as siblings of CA theory but the field of numerical analysis. To name just a few, stencil-based algorithms are applied in areas such as computational fluid dynamics (CFD), Lattice-Boltzmann computations, Jacobi solvers, Simulation of heat-transfer (e.g. convection) and image processing. Due to the importance of stencils in computer science[2] there have been many approaches to improve computational performance of the core algorithm by means of vectorization and parallelization [3]. Initially, these approaches were mostly based on optimization of some given algorithm on some specific target system. This exhibited limitations both to portability and usability, as forthcoming developments in parallel and distributed systems technology or additional requirements on the algorithmic level implied deep changes to the initially optimum code bases and implementation details.

Various forms of code generators and definition languages (for more information also see the interesting work on the pochoir stencil compiler [5]) for stencil computations have been described, which essentially introduce some kind of abstraction layer between actual compute hardware and the mathematical definition of stencils. Due to ever increasing complexity in compute hardware, namely increasing number of memory levels that operate at different speeds and latencies, and increasing numbers of cores per system, the generation of high performance code is now a task that seems to overwhelm not only most human software developers but also many standard code generators. To alleviate this rather undesirable situation, special auto-tuning frameworks have been proposed. These frameworks aim to sift through the

[2]The large-scale research project 'Exastencils' (http://www.exastencils.org/) is also to be mentioned in this context.

enormous number parameters found in the implementation of some stencil code and find optimal settings automatically without requiring much domain-specific expertise by the user.

Despite this intriguing corpus of related work, we found very little support for the kind of mathematical operations that the proposed CA-based algorithm is based on. Also, most work on stencil-based algorithms are based on regular and dense datasets, while our approach complements computation of dense datasets by some sparse formulation of the CA update routines. To the best of our knowledge, we will give the first report on the application of this CA-based algorithm in the context of sparse lattices on parallel hardware.

3 Algorithm

The basic algorithm for computing a 3-dimensional Cellular Automaton for finding connected components in a lattice follows [6]. The main algorithmic steps show great similarity to a stencil computation, in which floating point data is exchanged by integer data and the numerical summation of stencil pixels is replaced by the computation of the maximum value within the stencil pattern.

Also, we restrict the ‘observed neighborhood’ of each cell to the Von-Neumann neighborhood, which is defined as a center- or ‘host’-cell and two directly adjacent neighbor-cells for each dimension. Following Trobec and Stamatovic [6] the lattice boundary cells are fixed during cell updates. By employing this fixed boundary condition the resulting code complexity can be reduced, which is an advantage on most of the projected target platforms that we will consider in the following section.

On entry, the binary lattice describes a distribution of ‘background’ and ‘foreground’ values only. The task of finding connected components is accomplished by a short initialization phase, along with the actual Cellular Automaton update phase, which by itself is an iterative process.

3.1 Initialization

During initialization the binary lattice data will be transformed into an initial configuration or ‘coloring’ in which each foreground pixel will be given a unique index value or ‘color’ that will enable a clear distinction of pixels inside the lattice. For performance reasons we refrain from the initial coloring scheme described by Trobec and Stamatovic [6], which takes into account so-called ‘corner pixels’ for setting up the initial lattice values. Instead we choose a strictly monotonous series of cardinal values that are attached distinctively to any binary foreground pixel. Albeit this approach is inherently sequential (Algorithm 1), we found it of sufficient speed. Alternatively, initial coloring can also be achieved by choosing the positional data of each lattice point, which is given by the distinct tuple of (x, y, z)-values from which some distinct cardinal value can readily be derived in parallel execution mode.[3] It is essential to have boundary cells being initialized to 'background' states. This can be accomplished either in the initial binary lattice data or in initialization step.

3.2 Cellular Automaton Update

The update rule for the lattice (CA1) is applied to each cell that has neither been labeled as 'background' nor is a member of the boundary of the lattice. Each iteration of the Cellular Automaton takes CA1, the current state of the CA, and yields a new lattice CA2 in which the states of non-background and non-boundary cells have been updated. The update rule is given by the maximum function, applied to the current host cell C and its 6-neighborhood of surrounding cells. As each to be updated host cell C might also be a neighbor cell, this newly computed cell state must not be stored at the current lattice position but in a disjoint memory location in state array CA2, hence the transform $CA1- > CA2$. Algorithm 3 gives an outline of the update process in pseudo code.

Algorithm 1 CAinitialize

```
1: procedure CA_INITIALIZE(input: binary_lattice, output: CA)
2:     color ← 0
3:     for all cells in binary_lattice do
4:         if cell then
5:             color ← color + 1
6:             CA.cell.color ← color
7:             return(CA)
```

```
1: procedure MAXIMUM_NEIGHBOR(input: cell, output: color)
2:     color ← cell.color
3:     color ← max (color, cell.up)
4:     color ← max (color, cell.down)
5:     color ← max (color, cell.left)
6:     color ← max (color, cell.right)
7:     color ← max (color, cell.before)
8:     color ← max (color, cell.after)
```

[3]Let's consider a cubic lattice of dimension N. For any lattice element at some position (x, y, z) the unique positional information can be used to derive an initial coloring $Color = (((z * N) + y) * N + x$.

Algorithm 2 Termination check

```
1: procedure TERMINATION(input: CA1, input: CA2, output: bool)
2:     for all cells in CA1 do
3:         if (Ca1.cell.color NOT CA2.cell.color) then return False
       return True
```

Algorithm 3 Cellular automaton update

```
1: procedure CA_UPDATE(input: CA1, output: CA2)
2:     for all cells in CA1 do
3:         if cell NOT (background OR boundary) then
4:             CA2.cell.color ← maximum_neighbor (CA1.cell)
```

3.2.1 Maximum Operator

Despite being a very basic operation, computing the maximum pixel values of the surrounding neighborhood of each cell constitutes the main part of the Cellular Automaton which usually will take most of the total runtime of the proposed algorithm on any given hardware platform. Hence, we aim to support high levels of performance not only by applying proper platform-specific code optimizations (see Sect. 4), but also by choosing hardware-friendly operations in this most crucial algorithmic core operation.

Obviously, the straight forward solution for computing the numeric maximum of two pixel values involves some branch-instruction. Probably all recent high-performance CPU-hardware offer intricate and even online performance optimization techniques, such as instruction reordering, branch prediction, and speculative execution. Along with hierarchical multi-level caching memory CPUs are mostly capable of executing branching operators without suffering from significant performance penalties. The situation is quite different on many modern accelerator-based hardware platforms, which usually offer higher compute core counts at the cost of reduced core complexity. Our rationale here is to avoid branching operations to a high degree, as these operations tend to stall the stream of instructions on GPGPU-hardware, which diminishes overall throughput. Also, on hardware that supports true vector-processing[4] such as the high-performance computing platform NEC ACE SX, a steady stream of branch-less instructions promises to be beneficial towards our goal of high computational throughput.

[4]While all modern CPUs do actually support high-throughput instructions that operate on short vectors of data elements (e.g. SSE, AVX, Altivec, etc.), we want to make the distinction against pipelined vector processing, which is capable of processing vectors of arbitrary length while also employing a richer set of instructions compared to standard x86-based processors.

```
#define MAX(a,b)     (((a)>(b))?(a):(b))
```

Fig. 1 Branch-based maximum operator

```
#define MAX(a,b)     (a-((a-b)&(a-b)>>31))
```

Fig. 2 Closed-form maximum operator

3.2.2 Branch-Based Maximum Operator

The definition depicted in Fig. 1 constitutes a classic macro of the C language, which translates into efficient code on modern CPU-hardware, such as Intel x86 or IBM Power architectures.

3.2.3 Closed-Form Maximum Operator

Figure 2 constitutes the closed-form macro[5] for computing the maximum of two signed integer values (int32 data type). It involves no branching operation, but basic arithmetic and bit-wise operations only. Due to the absence of branch operators, this function incurs no warp-divergence on CUDA-enabled devices and promises benefits on any in-order execution compute platform.

Finally, a host-based driver routine (Algorithm 4) is used to orchestrate the series of compute and termination criterion (Algorithm 2) functions that resemble the Cellular Automaton.

Considering the GPU implementation, updates of the CA and corresponding termination checks will exclusively be accomplished in GPU RAM.

Algorithm 4 Driver

```
1: procedure DRIVER(input: binary lattice, output: CA1)
2:     $CA1 \leftarrow$ initial coloring (binary lattice)
3:     $CA2 \leftarrow CA1$
4:     repeat
5:         $CA\_update(CA1, CA2)$
6:         $CA\_update(CA2, CA1)$
7:     until termination
```

[5] As proposed by Holger Berger of NEC Germany.

4 Implementation

Our baseline implementation consists of OpenMP-enhanced x86-code (C language), from which several code-branches for GPU (nVidia CUDA), multi-GPU, Intel Xeon Phi, and NEC ACE code have been derived. These approaches support dense data sets, which are stored as a standard array in C. Depending on density and distribution of non-background pixels in a given data set, we find that an alternate, sparse representation of Cellular Automaton data offers performance benefits, albeit at the cost of some increase in memory usage.

For performance reasons we decided to employ a granularity of two updates per termination check. The main advantage of this design decision is the potential for omitting any swapping operations on input and output state arrays, such as described in [3]. By switching input- and output parameters between two consecutive calls to the update function, state arrays CA1 and CA2 serve both as input and output array. The frequency of calls to the termination criterion function is also reduced as a consequence, which is preferable for reasons of performance but in general will also lead to one potentially superfluous update operation in the last phase of the algorithm as it reaches the equilibrium state of the CA.

4.1 Dense Data Representation

4.2 OpenMP-Code

OpenMP is the industry-standard for task-level parallelization on multi-processor systems. It allows for convenient development cycles, which usually start from a sequential code base. By incrementally adding parallel constructs to the code, the execution speed will be enhanced and the software will be enabled to utilize all available system resources. Fortunately, NEC is offering their own high-performance implementation of the OpenMP runtime and compiler environment so that porting efforts starting from the x86-based code base prove to be a rather straight-forward process. Consequently, the resulting source codes both for x86- and NEC ACE-SX systems do look very similar and we only want to give a glimpse on the differences.[6] Porting the core update function to the Xeon Phi (KNC) processor follows Intel's standard programming model called 'function offloading'. In this model the Intel C-compiler is guided by means of a few directives to generate parallel OpenMP-based code for the Xeon Phi-accelerator as well as the necessary staging of function data (e.g. state arrays CA1, CA2). Figure 3 displays the x86-based update routine.

[6]The NEC implementation of OpenMP offers pragma based hints to the compiler which signify independence of nested loops, such as loops 'row' and 'col' in Fig. 3. By adding '#pragma cdir nodep' to the inner loops, the compiler is set to optimize in more aggressive way.

```
void update_CPU (int N, int *input, int *output)
{
int row, col, slice;
int cell;

#pragma omp parallel for private (row, col, cell)
for (slice = 1; slice < SDIM -1; slice++)
  {
  for (row = 1; row < SDIM -1; row++)
    {
    for (col = 1; col < SDIM -1; col++)
      {
      if (input[slice * N * N + (row * N) + col] != 0)
        {
        cell = input[slice * N * N + ((row) * N) + col];
        cell = Max(cell, input[slice * N * N + ((row-1) * N) + col]);
        cell = Max(cell, input[slice * N * N + (row * N) + col -1]);
        cell = Max(cell, input[slice * N * N + (row * N) + col+1]);
        cell = Max(cell, input[slice * N * N + ((row+1) * N) + col]);
        cell = Max(cell, input[(slice-1) * N * N + (row * N) + col]);
        cell = Max(cell, input[(slice+1) * N * N + (row * N) + col]);
        output[slice * N * N + (row * N) + col] = cell;
        }
      }
}}}
```

Fig. 3 OpenMP code: parallel update of CA cells

4.3 CUDA Implementation

For achieving high computational performance in the Cellular Automaton kernel we find two very relevant design decisions. First, data decomposition has to fit the memory subsystem of the GPU hardware. This essentially boils down to proper memory coalescing, which is a standard technique of forcing adjacent CUDA cores access adjacent memory locations in parallel. We aim to achieve high memory throughput on the GPU by assigning an appropriate number of CUDA threads to the computation of the inner-most loop ('columns') (4), which as a result displays the necessary memory access patterns. In other words, the inner-most loop is squashed altogether and being replaced by an appropriate number of CUDA threads that operate in parallel. This limits the maximum dimension of the CA to the maximum number of CUDA threads per CUDA block. For current generation NVIDIA-hardware this amounts to 1024 threads, hence a maximum dimension of 1024^3 cell elements[7] is being supported by our current GPU-implementation.

[7]While this may seem to be a limiting factor in the application of the kernel for large CAs, it should be stated that the corresponding amount of necessary GPU-memory quickly fills the available on-chip resources of the accelerator, which might be the main limitation towards employing larger datasets.

The second design decision of the CUDA kernel involves the method of parallelizing the two outer loops (‘rows’, ‘planes’) of the kernel. Since no memory-coalescing issues have to be taken into account at this level, we enjoy freedom to employ loops, a CUDA grid-based decomposition, or some mixture of both design models. Relying on for-loops only puts high computational pressure on each CUDA thread and—probably even more important—hampers the inherent capability of the GPU in hiding latencies of memory accesses by employing large numbers of active CUDA blocks and threads. Also, due to the very high core counts (e.g. 3584 cores on recent PASCAL-cards) of modern high performance hardware some loop-only based approach would severely limit the degree of parallelism.[8]

Historically, CUDA-enabled devices offered little or no L2-cache memory, but some fast ‘shared memory’ or scratch-pad memory on-chip. By resorting to software-controlled caching mechanisms this lack of hardware-controlled L2-cache could in general be alleviated by clever kernel design. Recent generations of CUDA-hardware do offer improvements both in terms of size and levels of control of cache memory. Nevertheless, there is no easy way to decide whether to just rely on hardware-controlled L2-caching mechanisms or to exert explicit control over memory access by resorting to ‘old-style’ programming techniques.

In order to achieve maximum performance we apply explicit cache control by way of memory-coalescing during column-reads (function ‘read_column’ in Fig. 4) in conjunction with implicit, hardware-based cache control. As can also be seen in Fig. 4, the number of column-reads can be diminished for all row iterations except for the first one. This is accomplished by explicitly copying column data that is already present in the shared memory segment ‘local’ following the direction of the loop. Hence, for each new iteration in which the stencil-column is being moved towards the last row of the current slice (z-plane of the CA), three instead of five column-reads are sufficient, which marks a 40% reduction of memory transfers.

4.3.1 Termination Criterion

Checking the termination criterion in parallel is based on finding any discrepancy between state arrays CA1 and CA2. Depending on the dimensions of the computed CA, this involves transfer of data on the order of multiple GiB, which makes repeated checks prone to becoming a major bottleneck of the algorithm. We therefore aim for an early termination of the check routine itself, which requires fast synchronization of collaborating GPU-threads. As outlined in code ‘termination’, read access to both state arrays CA and CA2 is being coalesced by enabling threads to work in lockstep on properly aligned data. Discrepancies within two given data columns lead to immediate termination of those threads, that spotted some discrepancy. Equally important, discrepancies will raise some global flag, which

[8]Employing 3584 cores to a CA-kernel of dimension 1024^3 would yield hardware utilization rates below of 29%.

```
__global__ void update_GPU ( const int * __restrict__ input,
                             int * __restrict__ output)
{
__shared__ int local[6][N];    // N = dimension of CA
int slice = blockIdx.x;
int row, col = threadIdx.x;
int max;

if ((slice>0) {

for (row = 1; row < (N-1); row++)
   {
    __syncthreads();
   if (row==1)  // read full stencil
     {
     readcolumn (&local[0][0], &input[slice*N*N+(row-1)*N]);
     readcolumn (&local[1][0], &input[slice*N*N+row*N]);
     readcolumn (&local[2][0], &input[slice*N*N+(row+1)*N]);

     readcolumn (&local[3][0], &input[(slice-1)*N*N+row*N]);
     readcolumn (&local[4][0], &input[(slice+1)*N*N+row*N]);
     local[5][threadIdx.x]=local[1][threadIdx.x];//output line
     }
    else      // read partial stencil with reuse of recent data
     {
     readcolumn (&local[3][0], &input[(slice-1)*N*N+row*N]);
     readcolumn (&local[4][0], &input[(slice+1)*N*N+row*N]);
     local[0][threadIdx.x] = local[1][threadIdx.x]; // reuse
     local[1][threadIdx.x] = local[2][threadIdx.x]; // reuse
     readcolumn (&local[2][0], &input[slice*N*N +(row+1)*N]);

     local[5][threadIdx.x]=local[1][threadIdx.x];//output line
     }

   __syncthreads(); // wait for data to have arrived
    if ((threadIdx.x > 0)&&(threadIdx.x < N-1)) {
      if (local[1][threadIdx.x] != 0)
      {
      max=local[0][threadIdx.x]>local[1][threadIdx.x] ? _
             local[0][threadIdx.x] : local[1][threadIdx.x];
      max=local[1][threadIdx.x-1]>max?local[1][threadIdx.x-1]:max;
      max=local[1][threadIdx.x+1]>max?local[1][threadIdx.x+1]:max;
      max=local[2][threadIdx.x]>max ? local[2][threadIdx.x] : max;
      max=local[3][threadIdx.x]>max ? local[3][threadIdx.x] : max;
      max=local[4][threadIdx.x]>max ? local[4][threadIdx.x] : max;

      local[5][threadIdx.x] = local[4][threadIdx.x]>max ? _
                              local[4][threadIdx.x] : max;
      }
    }
    // store resulting column in global output array
    writecolumn (&local[5][0], &output[slice * N * N + row * N]);
   }
  }  // slice
}
```

Fig. 4 CUDA code: parallel evaluation of termination criterion

will prevent any not-yet active CUDA block from entering the checking routines. It has to be noted that thread termination within some active CUDA block will only affect remaining threads of that block. This might be regarded to be sub-optimal, but experiments with an increased level of synchronization at Warp-level exhibited inferior overall performance.

Please also note the absence of any explicit synchronization construct in the above implementation of non-blocking synchronization.

Interestingly enough, efficient parallelization of the termination criterion proves to be much harder in OpenMP than in CUDA, due to the absence of premature termination of parallel for-loops in OpenMP. As a consequence, OpenMP code will be forced to sift through both state arrays CA1 and CA2 in total, even when differences should have been spotted during the first few comparisons. Albeit explicit and more complex task-based implementations within OpenMP would have been possible, we instead opt for a sequential version. Due to the simplicity of the core operator (check for equality of two cell states) the resulting code will operate close to the saturation level of the memory system, which can be taken as an argument against parallelization of this code section in the first place (Fig. 5).

```
__global__ void termination (const int N,
                 const int * inputA, const int * inputB,
                 unsigned char * __restrict__ activity)

{
__shared__ int local[2][SDIM];

int slice = blockIdx.x;
int row;
int col = threadIdx.x;
int max;
int p;
unsigned char flag=0;

if ((slice>0)&&(slice <SDIM-1)) {
  if (activity[0]==0) {
   for (row = 1; row < (N-1); row++)
     {
     readcolumn (&local[0][0], &inputA[slice * N * N + row * N]);
     readcolumn (&local[1][0], &inputB[slice * N * N + row * N]);

     if (local[0][threadIdx.x] != local[1][threadIdx.x])
        {
        row=N;       // terminate thread
        activity[0]=1;  // raise global termination flag
        }
     }
   } // if activity==0
  }  // if slice > 0
}
```

Fig. 5 CUDA code: parallel evaluation of termination criterion

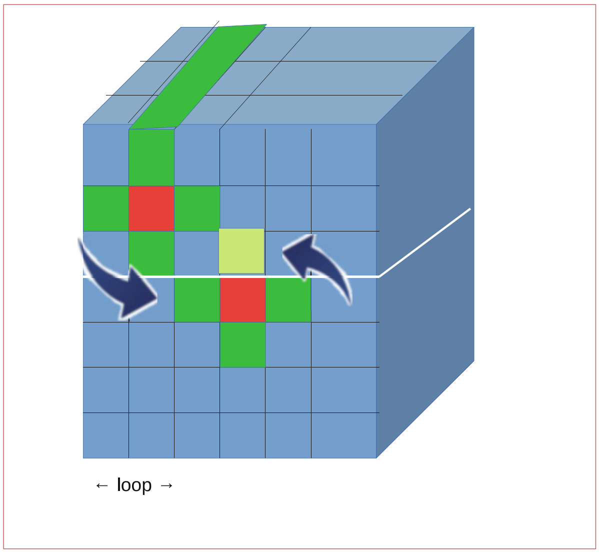

Fig. 6 In dual-GPU environments, state arrays are partitioned into two even portions with each portion being stored locally on one of the participating GPUs. Both GPUs may access state arrays of the corresponding partner GPU by means of unified virtual addressing (UVA) mode

4.4 Multi-GPU Computation

Figure 6 depicts the basic layout of CA data in dual-GPU setups. Each GPU is enabled to access ghost-cells (cells that are read but never being written to) that physically reside in the partner GPU's local memory by means of unified virtual addressing (UVA) mode. However, the CUDA kernel is repeatedly forced to decide whether a certain column of data is available locally or whether it has to be fetched via the UVA mechanism. This decision adds to an increase in code complexity[9] and potentially also harms the overall throughput of the kernel. By introducing two separate kernels that are specialized for operation in the areas of ghost cells that emerge at the lower and upper borders of their data partition, we aim to alleviate this performance bottleneck. The performance numbers reported on in the following section are based on this improved multi-kernel model.

[9]While code complexity is not regarded an issue on standard CPU-based systems, it certainly can lead to an inflation of the size of the binary executable, which in extreme cases can result in non-executable kernels.

4.5 Sparse Data Representation

For easier usage we provide a method for generating some sparse representation of any given data set out of its initially dense, array-based representation. For the discussed dense 3D datasets this method builds a vector V of tuples (x, y, z) with each tuple designating the coordinates x, y, and z of a distinct pixel in the dense data set. Hence, the size of vector V directly corresponds to the number of relevant pixels, that is, pixels not belonging to boundaries or background. By cycling over the z, y, and x-planes of the dense dataset in that order, we ensure the tuples of resulting vector V to be ordered in a way that proves to be cache- and memory-page friendly during the following cell update.

Note that vector V is merely an index of foreground-pixels, actual CA state information will still be managed in the form of the dense systems CA1 and CA2 (Sect. 4.1).

Updating the CA state arrays CA1 and CA2 can now be pinpointed to the exact locations of foreground pixels, but comes at the cost of additional memory access for dereferencing the corresponding tuple in V. Parallelization of cell updates is a straight-forward process being accomplished at the level of the tuple vector V, which is a densely packed dataset that lends itself well to OpenMP-, CUDA-, and vector processing approaches. Nevertheless, at this point the general choice of dense versus sparse data representation has to be grounded on heuristics.

4.5.1 OpenMP-Code

The structure of nested loops in the dense case (as shown in Fig. 3) is replaced by a single loop (Fig. 7) that operates on the vector V of tuples.

5 Simulation Results

The simulation code for the CA has been run under benchmarking conditions on a set of systems with the intention of giving ‘the bigger picture’ on what performance levels are to be expected on recent high-performance compute hardware. With the notable exception of the NEC ACE-SX system, the employed hardware belongs to the class of so-called accelerators that—probably also due to its potential performance and affordability—seems to be attaining more attention both in science and engineering for quite some time now. **Table 3 gives an overview of obtained speedups of the investigated compute architectures over the baseline system Intel Xeon 1620 (quad core)**. Again, we want to stress the fact that these

```
void update_CPU_sparse (int N, int *input, aPixel *V,
                        unsigned int pixels, int *output)
{
int row, col, slice;
int cell;
unsigned int nr;

#pragma omp parallel for private (slice, row, col, cell)
for (nr =0; nr < pixels; nr++)
   {
   slice = V[nr].z;
     row = V[nr].y;
     col = V[nr].x;

   cell = input[slice * N * N + ((row) * N) + col];
   cell = Max(cell, input[slice * N * N + ((row-1) * N) + col]);
   cell = Max(cell, input[slice * N * N + (row * N) + col-1]);
   cell = Max(cell, input[slice * N * N + (row * N) + col+1]);
   cell = Max(cell, input[slice * N * N + ((row+1) * N) + col]);
   cell = Max(cell, input[(slice-1) * N * N + (row * N) + col]);
   cell = Max(cell, input[(slice+1) * N * N + (row * N) + col]);
   output[slice * N * N + (row * N) + col] = cell;
   }
}
```

Fig. 7 OpenMP code: evaluation of sparse state array

Table 3 Speedup overview (based on results presented in Table 1)

CA dim	GTX680	NEC ACE-SX	Tesla P100	Xeon E1620	Xeon Phi 5120
128	45×	19.9×	43.7×	1×	n/a
256	37×	18.5×	99.7×	1×	n/a
512	25.7×	12.5×	68.9×	1×	n/a
704	15.5×	9.7×	55.2×	1×	18.1×

systems stem from different cycles in hardware development, so the interpretation of reported numbers should take that into account, too. The Pascal-based Tesla P100 offers extreme levels of performance with runners-up found in the GTX680 GPU and the Xeon Phi. In light of the high thread-counts of these three platforms which range from several hundreds up to more than 3.500 threads the performance of the quad core vector processor ACE-SX is quite astonishing (Fig. 8).[10]

Figure 9 depicts the relation of execution times for dense and sparse codes on two hardware platforms, one GPU and one node of the NEC ACE-SX vector

[10]Even more so when we want to put this into relation with the age of the hardware concept of this generation of the NEC processor, that apparently goes back at least 5 years from the time of this report.

```
__global__ void sparse_update (unsigned int number,
                               const aPixel* __restrict__ pixel_list ,
                               const int * __restrict__ input ,
                               int * __restrict__ output)
{
int max, iter , x,y,z;
unsigned int pos;

pos = (THREADS * Block_LOOPS)*blockIdx.x+threadIdx.x;

for (iter = 0; iter < Block_LOOPS; iter++)
 {
 if (pos < number)
  {
  z=(int)pixel_list[pos].z;
  y=(int)pixel_list[pos].y;
  x=(int)pixel_list[pos].x;

  max=input[z*N*N+(y-1)*N+x] > input[z*N*N+y*N+x] ? _
       input[z*N*N+(y-1)*N+x] : input[z*N*N+y*N+x];
  max=input[z*N*N+y*N+x-1]>max ? input[z*N*N+y*N+x-1]:max;
  max=input[z*N*N+y*N+x+1]>max ? input[z*N*N+y*N+x+1]:max;
  max=input[z*N*N+(y+1)*N+x]>max ? input[z*N*N+(y+1)*N+x]:max;
  max=input[(z-1)*N*N+y*N+x]>max ? input[(z-1)*N*N+y*N+x]:max;

  output[z*N*N+y*N+x] = input[(z+1)*N*N+y*N+x]>max ? _
                         input[(z+1)*N*N+y*N+x]:max;
  pos += (THREADS);
  } // pos < number
 }
}
```

Fig. 8 CUDA code: evaluation of sparse state array

processor system, respectively.[11] Both platforms deliver very stable execution times for updates of the dense CA. As expected, execution times in the sparse formulation of the CA update compare favorably to their dense counterparts for highly sparse systems (e.g. upwards of 95% of background pixels). As can be seen, the GPU system performs in robust way after reaching saturation at sparsity levels in the range of 10–15%, whereas the vector processor ACE-SX seems to struggle with the increased level of scattered memory access. It has to be noted that ACE-SX reaches the cut-off point at which the sparse code no longer prevails over the dense code at a later stage than the GPU platform. Hence, sparse formulation of updates is beneficial for a wider range of densities in a given CA on ACE than it is for the GPU.

[11]The results that we report here have to be taken with some caution, as the ACE-SX and GTX 1080Ti belong to rather different eras of their respective development time.

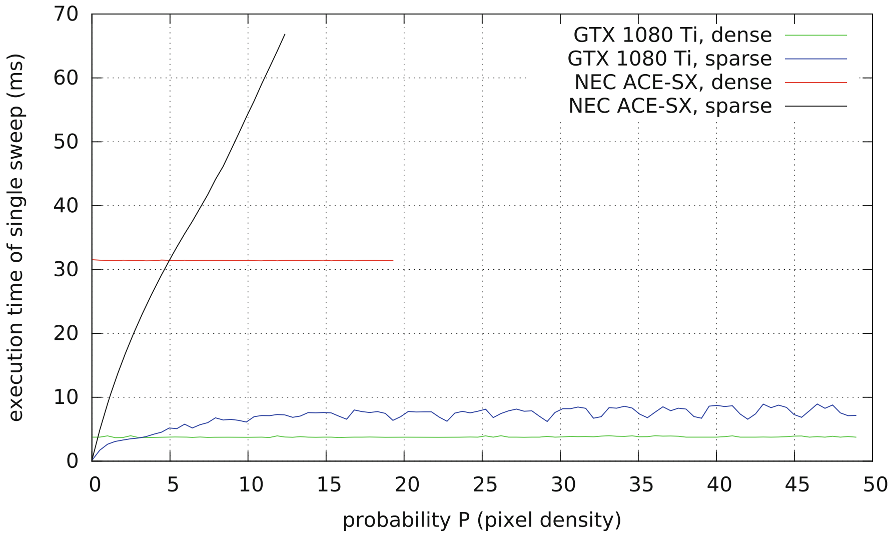

Fig. 9 Comparison of NVIDIA GTX1080 Ti GPU with NEC ACE-SX for dense and sparse systems of dimension 512

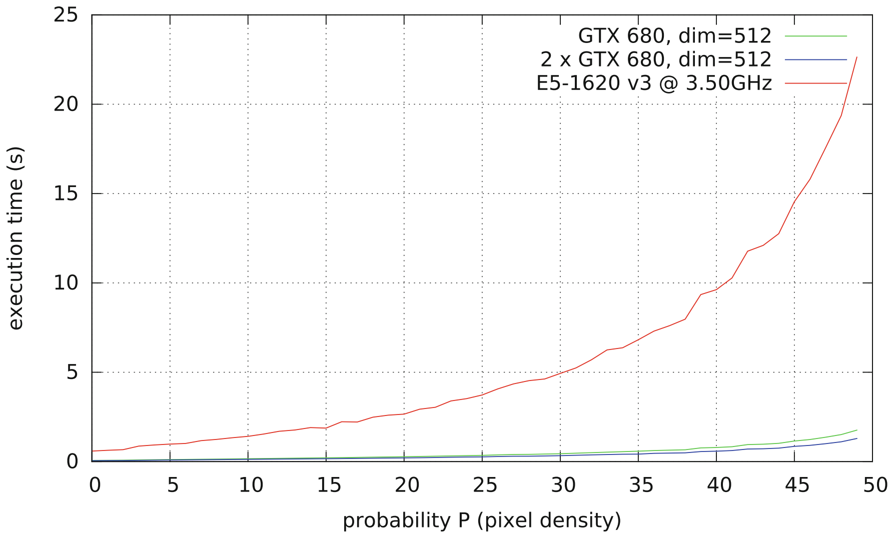

Fig. 10 Relation of runtime and pixel probability (density of CA) for single-, dual-GPU and quad-core CPU systems

As shown in Fig. 10, both GPU setups (single GTX 680, dual GTX 680) exhibit performance levels that are robust against increased levels of pixel densities of the simulated CA. On the contrary, the Xeon E5-1620-based system is able to maintain relatively low turn-around times for low pixel densities, but falls short for densities beyond 20% when execution times do increase substantially. As both the amount of memory and access patterns are fixed for all cases depicted in Fig. 10 variations of execution times have to be attributed to differences of the computational workload, which obviously is directly proportional to pixel densities in the CA. In this particular case, the observed speedup of GPUs over the CPU counterpart is not so much a result of some higher level in memory throughput on the GPU system, but it is a consequence of the usually much lower core count on the CPU. Unfortunately, speedup for the dual-GPU setup is limited to 1.4× due to UVA-related PCI transactions that result from access to ghost cells on the corresponding partner GPU. For the sake of completeness, we also want to report on the observed performance on the above-mentioned CPU system both for Matlab and the discussed CA-based algorithm. On average, Matlab's bwconncomp function will take 12 s (wall time) in a 3-d lattice of dimension=512, while the CA-based algorithm shows a cutoff at pixel density of some 40% beyond which execution times will mostly exceed that of the corresponding Matlab function. In a CPU-only setting, the new algorithm usually exhibits performance levels that are superior to Matlab for density levels below 40%.

The optimized checking of the termination criterion yields stable and much reduced turn-around times or single iterations up to some 70% of the total runtime of the CA. Figure 11 depicts the expected increase of execution times towards the end of the simulation of the CA, in which the CA states are beginning to settle and remaining activity within the state array CA1 requires an increasing effort to detect.

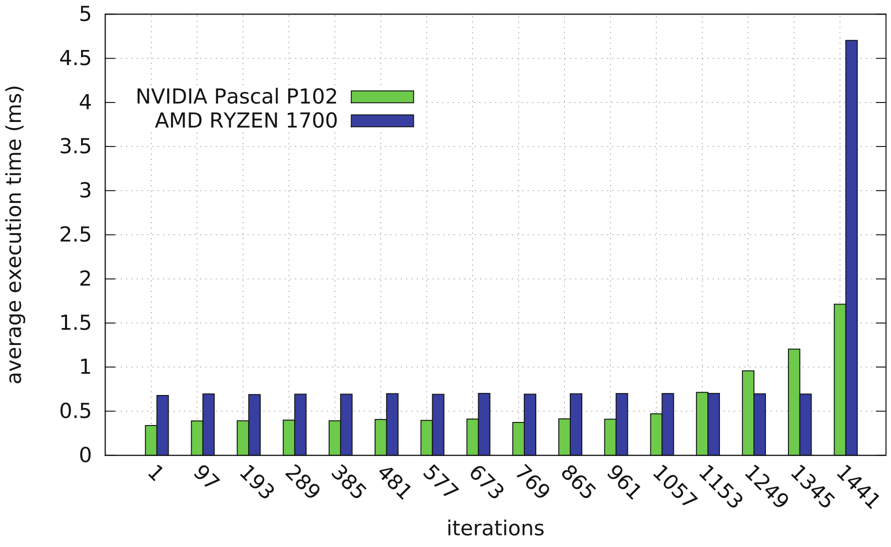

Fig. 11 Execution timings of termination checks (based on code given in Fig. 5)

5.1 Maximum Operator

Our initial motivation for conducting experiments with an alternative, closed-form maximum operator has been driven by the typical hardware characteristics of modern GPUs, that on one hand do offer numerous numbers of powerful CUDA cores, but on the other hand suffer from branching-incurred penalties (e.g. warp-diversion). Unfortunately, the increased computational complexity of the closed-form operator cannot be overcome by the sheer power of the GPU system.

As displayed in Fig. 12 an approach employing the branch-based maximum-operator (Fig. 1) in conjunction with a loop-based computation of the row elements of the CA state array performs best on the NVIDIA GTX 780 Ti hardware. Depending on the dimensions of the CA the next best option for dimension=256 is stated by a CUDA grid-based parallelization of the row elements and for dimension=512 the second best option is given by a combination of loops and closed-form maximum operator. This difference can be attributed to the increased overhead that is introduced by the large numbers of CUDA blocks in dimension=512 which amount to $512^2 = 262{,}144$. Apparently, instantiation of such large numbers of CUDA-blocks happens to be a rather expensive task in itself. Based on this observation a second important finding follows, which can be stated as the occasional necessity

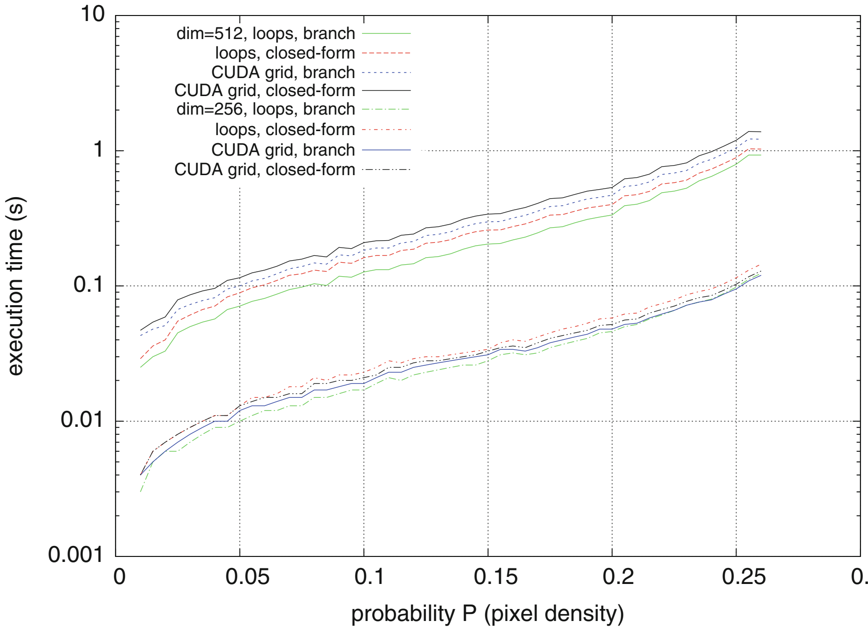

Fig. 12 NVIDIA GTX 780Ti: various update kernels based on loops vs. CUDA grid and branching vs. closed-form maximum operators

Table 4 Performance and scalability on NEC ACE-SX

Max operator	CPU core(s)	Execution (ms)	Speedup	Global speedup
branch-based	1	107.9	1.0	1.0
closed-form	1	233.11	0.46×	0.46×
branch-based	4 (OpenMP)	81.74	1.0	1.32×
closed-form	4 (OpenMP)	79.03	1.034×	1.37×

for conventional loop-structures on GPUs, even when CUDA allows for a clean and lean formulation of an algorithm entirely devoid of loops.

On NEC ACE-SX an ambivalent situation prevails when the closed-form maximum operator is applied. This system consists of a number of distributed cluster-nodes, that communicate via a high-speed network and message-passing style communication primitives. Each cluster-node comprises four distinct vector processing cores, that allow for very convenient and tightly coupled OpenMP-based parallelization as well as vectorization within the cores. Our experimental setup concentrates on a single such node, hence we may apply vectorization and shared-memory parallelization, respectively. In the single core variant the update of the state array suffers from severe performance penalties (see Table 4, rows 1 and 2) when the standard branch-based maximum operator (Fig. 1) is replaced by its closed-form counterpart (Fig. 2). However, in the parallel 4-core setup a modest speedup of some 32% is gained for the branch-based code, while the closed-form operator yields 37% overall speedup against the fastest sequential code. We do not regard this speedup in the range of a mere 3% to be essential, but it nevertheless seems noteworthy that the closed-form operator is moving from the position of the slowest version (sequential case) right to the fastest version (parallel case). Unfortunately, we do not have the resources to finally explain this result here, but it can be suspected that the sequential code already operates close to the limit that is imposed by the memory subsystem of the ACE-SX node. As the closed-form operator sports higher computational complexity than the branch-based operator, it benefits very clearly from the introduction of three additional cores in the parallel run, hence the considerable speedup in this case.

6 Conclusions

To the best of our knowledge we have reported on the first implementations of a Cellular Automaton-based algorithm for the computation of connected components in 3-d binary lattices on vector-processing hardware, as the NEC ACE-SX system and various accelerator-style hardware, such as nVidia-CUDA- and Intel Xeon Phi equipped systems. The algorithm proves to be very suitable to all of these platforms, and high levels of performance have been gained due to the massive amounts of parallelism that is inherent to this class of algorithms. Even the baseline code that we benchmarked on some modest mainstream Intel Xeon CPU offers certain performance benefits in comparison to the corresponding function that is provided with the popular Matlab environment. The reported performance numbers clearly indicate potential benefits from experimenting with different algorithmic solutions, even within the same general platforms (e.g. GPU).

Acknowledgements We would like to sincerely thank Professor Michael M. Resch and the whole team of HLRS for their valuable support, continued guidance and discussions, and provision of systems.

References

1. Datta, K., et al.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (2008)
2. Hoekstra, A.G., Kroc, J., Sloot, P.M.A.: Simulating Complex Systems by Cellular Automata. Springer, Berlin (2010)
3. Holewinski, J., Pouchet, L.-N., Sadayappan, P.: High-performance Code Generation for Stencil Computations on GPU Architectures. ACM, New York (2012). doi:10.1145/2304576.2304619
4. Stamatovic, B., Trobec, R.: Cellular automata labeling of connected components in n-dimensional binary lattices. J. Supercomput. **72**(11), 4221–4232 (2016). doi:10.1007/s11227-016-1761-4
5. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.-K., Leiserson, C.E., The Pochoir Stencil Compiler. ACM, New York (2011). doi:10.1145/1989493.1989508
6. Trobec, R., Stamatovic, B.: Analysis and classification of flow-carrying backbones in two-dimensional lattices. Adv. Eng. Softw. **103**, 38–45 (2015)