Vectorization of Cellular Automaton-based labeling of connected components in 3D binary lattices

Peter Zinterhof

CHPC / Dept. of CS / University of Salzburg

outline:

- motivation for finding connected components
- Cellular Automaton(CA)-based approach
- GPU implementation
- NEC SX ACE implementation
- optimization
- outlook

general motivation:

- basic function in image processing (3D case → series of images)
- compute spanning tree in transport process simulations

my personal motivation:

- chance to put high-performance hardware into action
- create first accelerator-based implementation

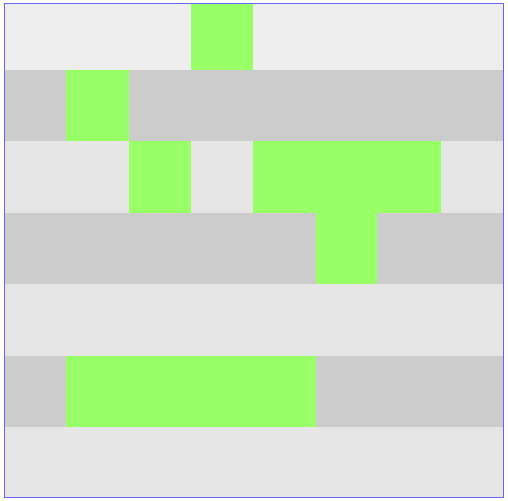
2D: OpenCV library: - fast

3D: Matlab function *bwconncomp:* - *CPU-only*
- *memory intensive*
- *not scalable*

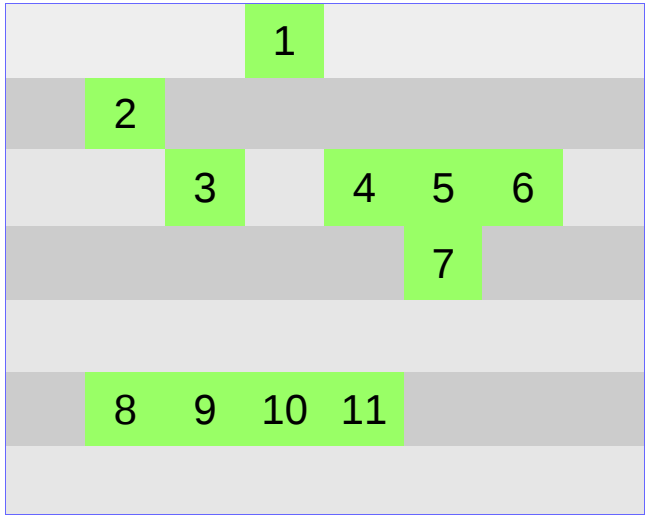
Cellular Automaton-based approach* :

- modest (fixed) memory consumption
- high data-locality
- high inherent parallelism

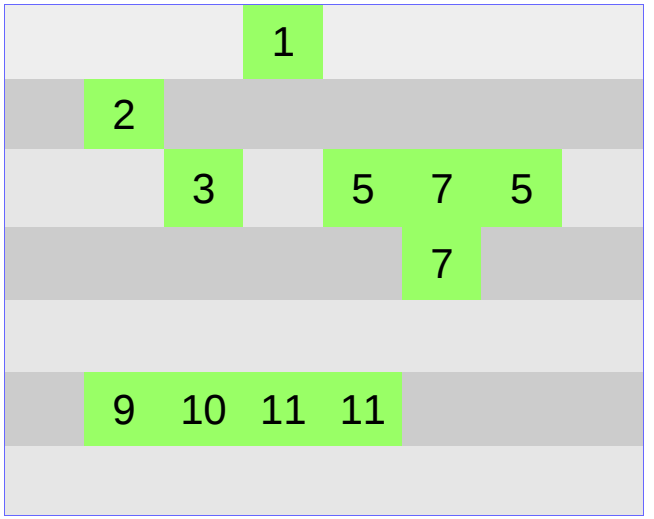
*as described by B. Stamatovic & R. Trobec



Input: binary dataset



setup phase: apply labels

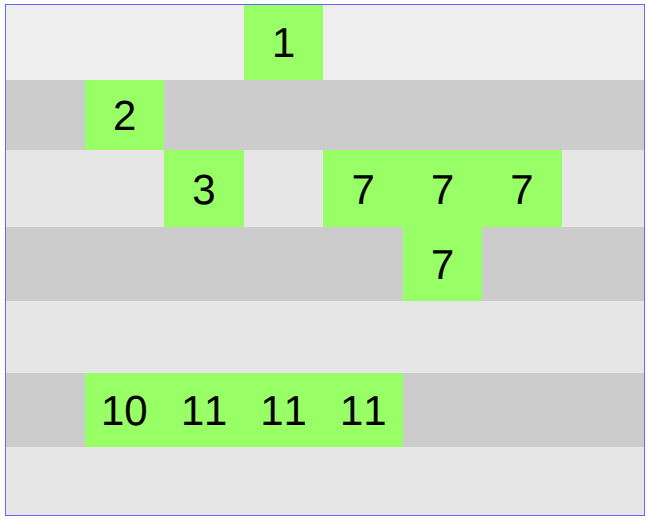


coloration: iteration 1

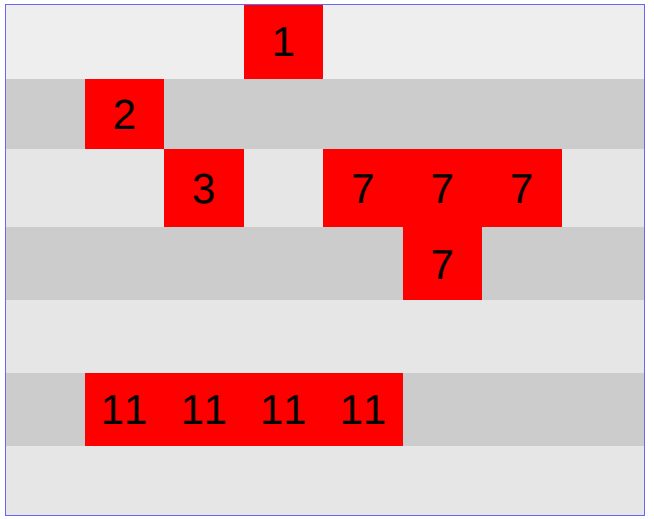
Update rule:

cell(x,y)=

max(4_neighborhood)

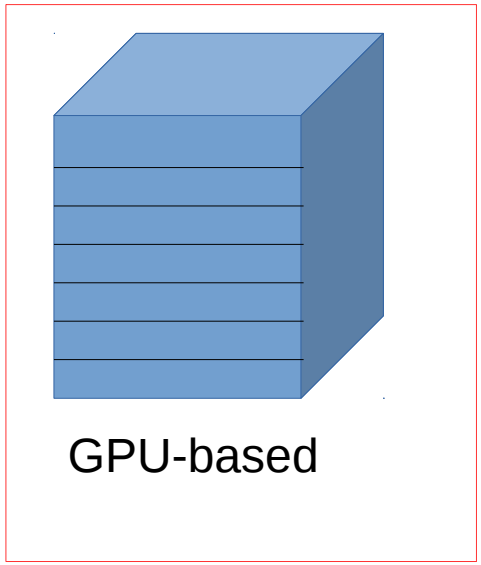


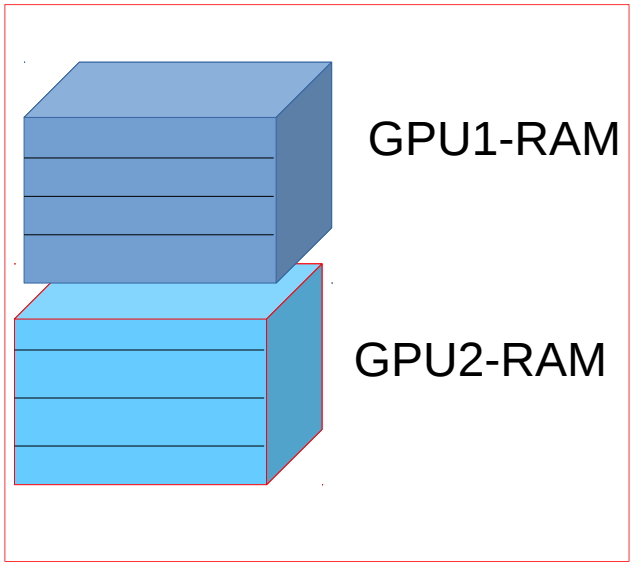
coloration: iteration 2

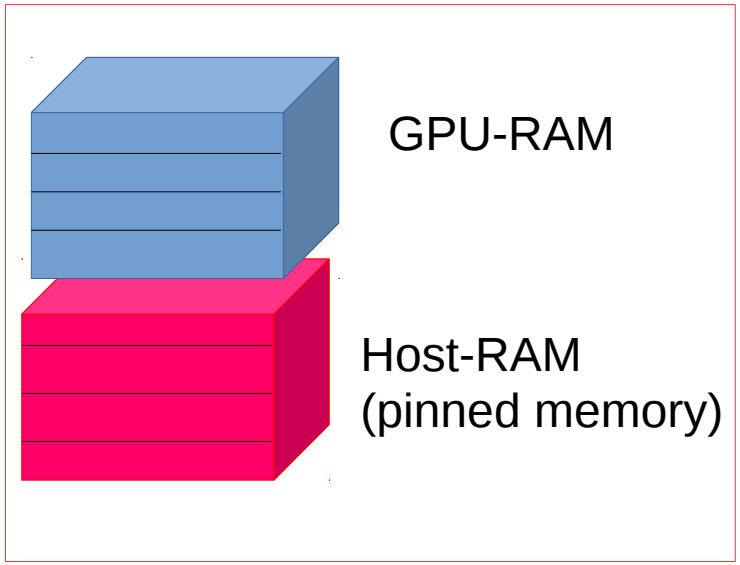


Equilibrium → solution

Implementations:







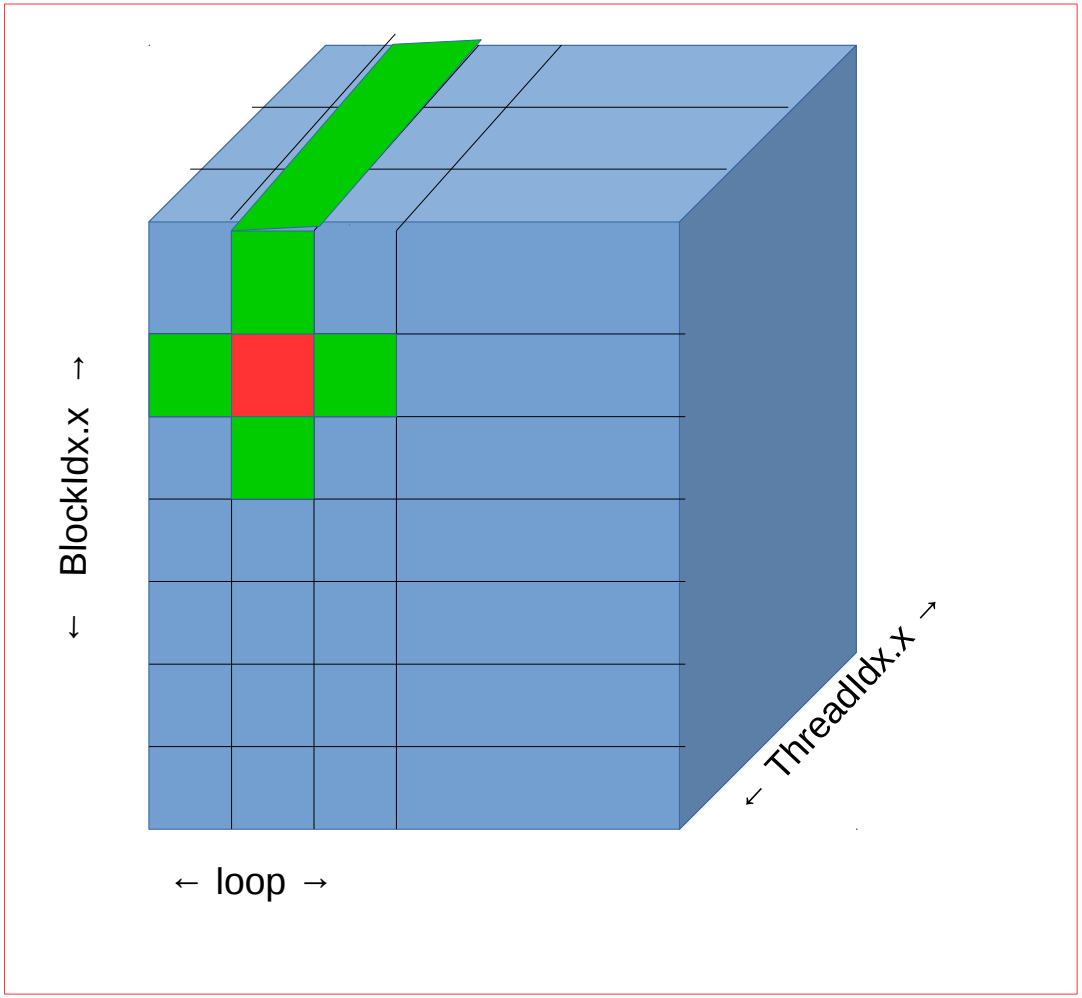
single GPU

dual GPU

single GPU + host memory

CUDA kernel: data layout

loop over slices
    column-wise read
    update cells



use shared-mem to achieve memory coallescing and re-use of column data: 'software controlled cache'

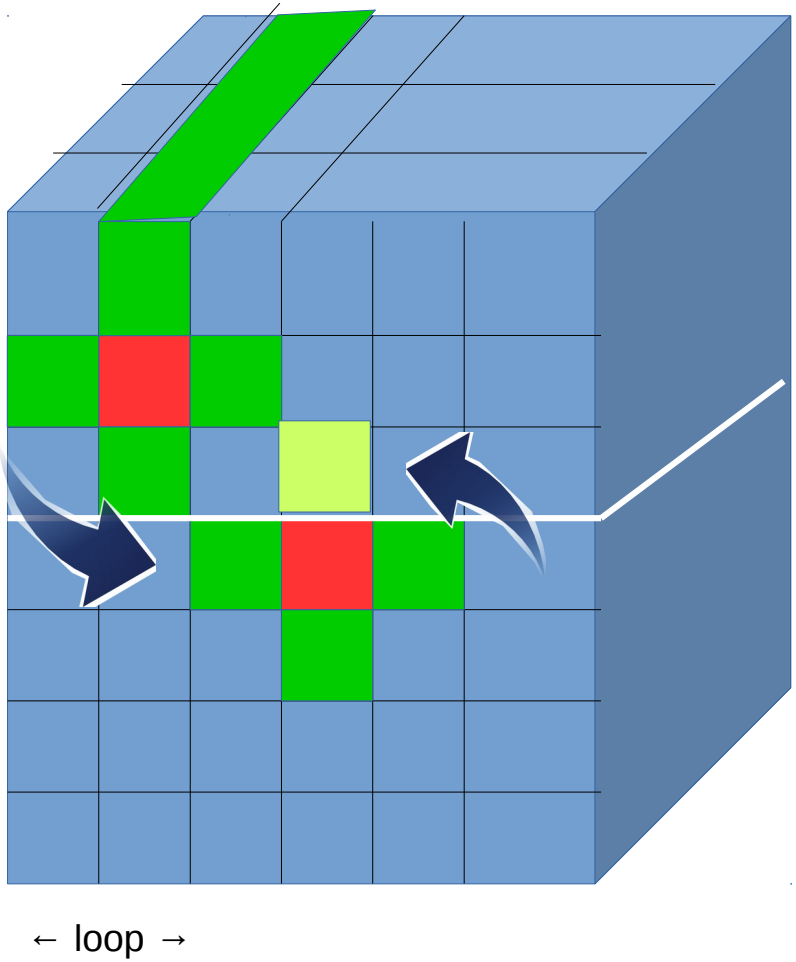
```
__global__ void  sweep_cached  (int N, const int * __restrict__ input,  int * __restrict__ output)
{
int slice = blockIdx.x;   int col = threadIdx.x;
int row;   int max;

if ( (slice>0) && (slice < N-1) ) {                                  // exclude 'south' and 'north' slices
  for (row = 1; row < (N-1); row++)  {                               //  exclude border rows
   if ((threadIdx.x > 0)&&(threadIdx.x < N-1)) {                     // exclude border pixels
      if (input[slice * N * N + row * N + threadIdx.x] != 0)         // update this cell
      {
       max = input[slice * N * N + (row-1) * N + threadIdx.x]>input[slice * N * N + row * N + threadIdx.x] ?
             input[slice * N * N + (row-1) * N + threadIdx.x] : input[slice * N * N + row * N + threadIdx.x];

       max = input[slice*N*N +row*N+threadIdx.x-1]>max ? input[slice*N*N + row*N + threadIdx.x-1] : max;
     ...
       max=input[(slice+1)*N*N+row*N+threadIdx.x]>max ? input[(slice+1)*N*N+row*N+threadIdx.x] : max;

    output[slice*N*N + row*N+ threadIdx.x] = input[(slice+1) *N*N + row*N + threadIdx.x]>max ?
          input[(slice+1) * N * N + row * N + threadIdx.x] : max;
      }  // if update
   }  /if  border
  } // for row
 }  // if slice
}
```

case 2: dual-GPU → unified virtual addressing (UVA)

kernel gets two pointers

“G1”, “G2”

at intersecting planes columns-wise reads on G2 and G1



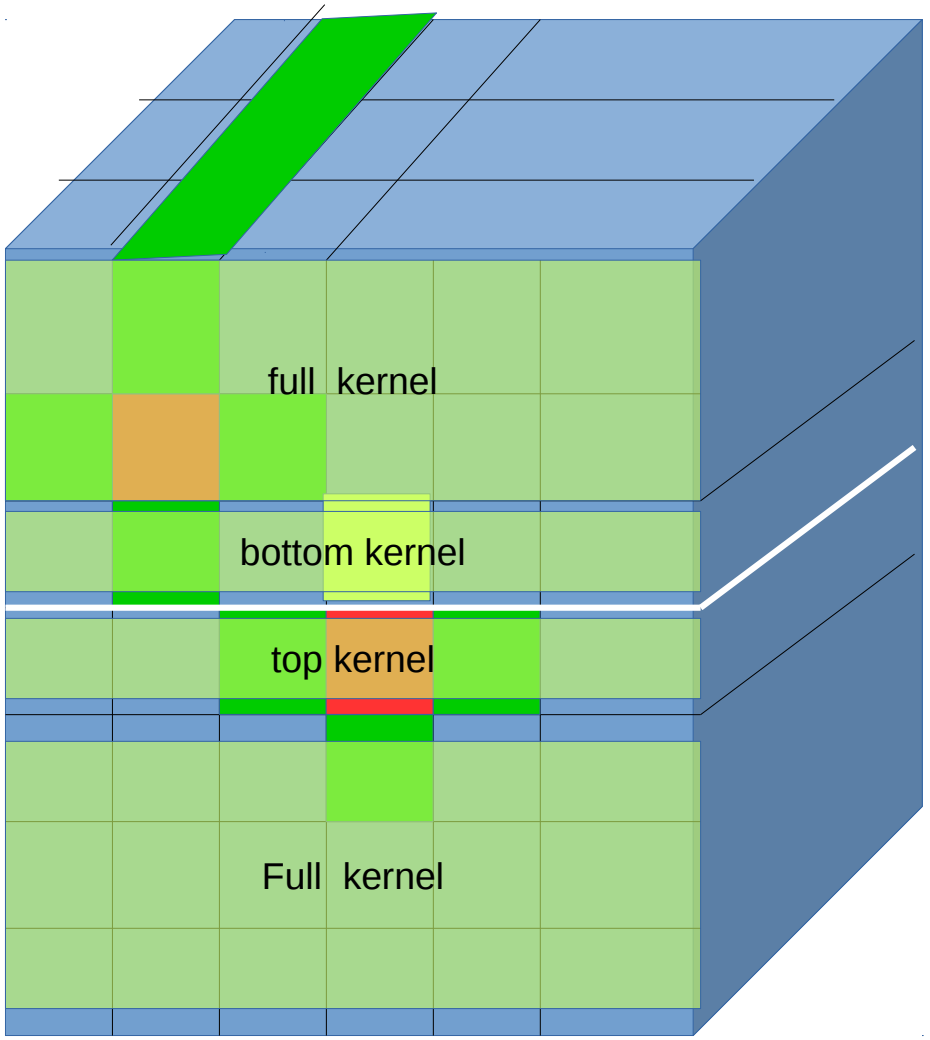
Dual-GPU: Optimization

specialized kernels 'top' & 'bottom'

→ no branching in full kernel

*int * inputGPU1 vs. int *inputGPU2*

→ no branching in top&bottom, too!



Further optimizations: Conventional GPU wisdom uncovered

cw: MP-shared memory is beneficial for re-using data and coallescing memory access WRONG

→ better: just rely on L2 caches

cw: avoid loops in CUDA-kernels

WRONG

→ better: profile block-based decomposition vs. loops

cw: GPUs have tremendous hp but suffer badly from branches

- Maximum-operator: substitute branches by bit-wise operations

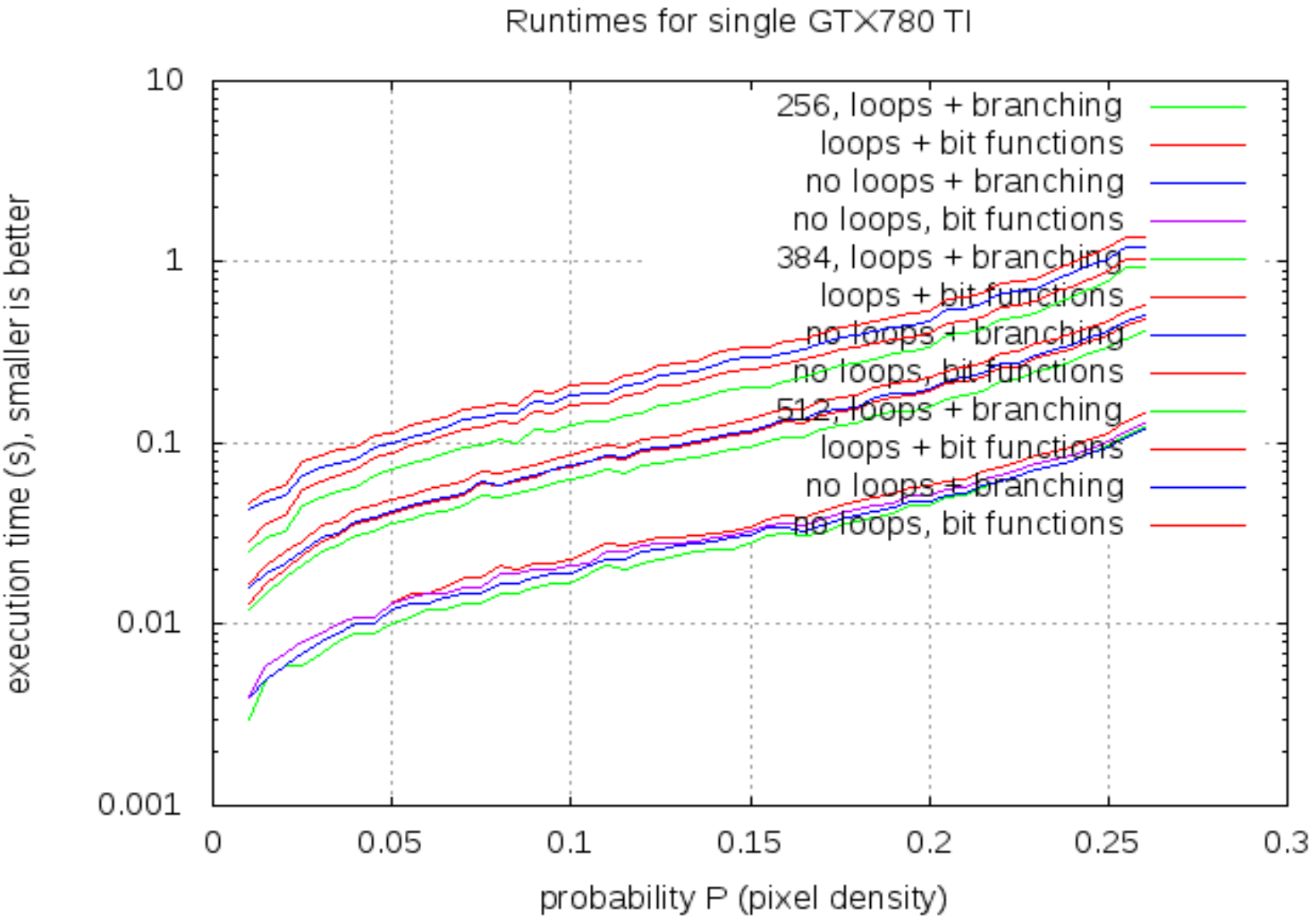
WRONG

max (a,b): a -= b;
a &= (~a) >> 31;
a += b;

closed form: #define Max(a,b) (a-((a-b)&(a-b)>>31))

(credits: Holger Berger, NEC)

→ better: *max = input[pos_1] > max ? input[pos_1] : max;*

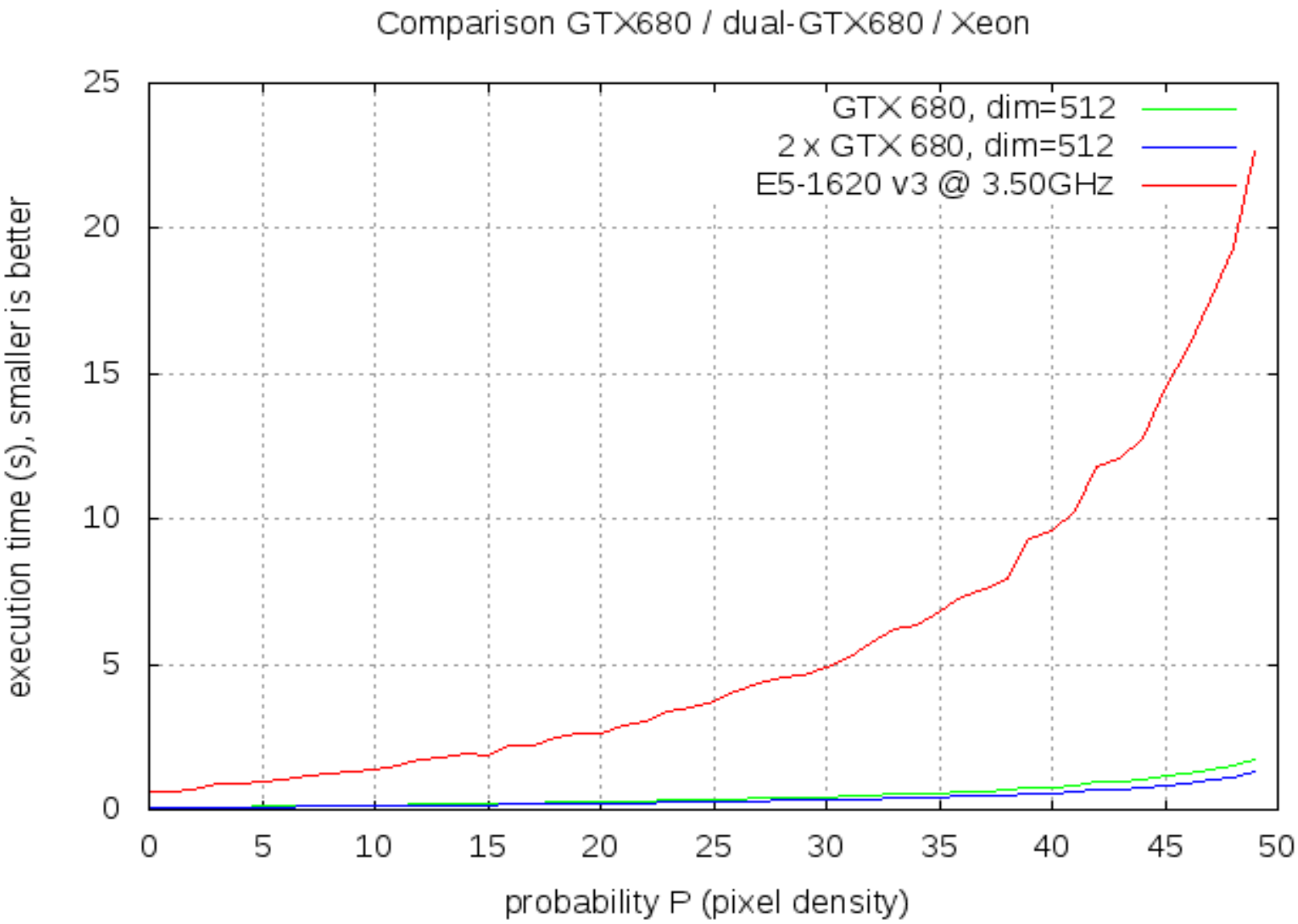

Runtimes for single GTX780 TI
execution time (s), smaller is better
probability P (pixel density)
10
1
0.1
0.01
0.001
0
0.05
0.1
0.15
0.2
0.25
0.3
256, loops + branching
loops + bit functions
no loops + branching
no loops, bit functions
384, loops + branching
loops + bit functions
no loops + branching
no loops, bit functions
512, loops + branching
loops + bit functions
no loops + branching
no loops, bit functions

SX-ACE: observations

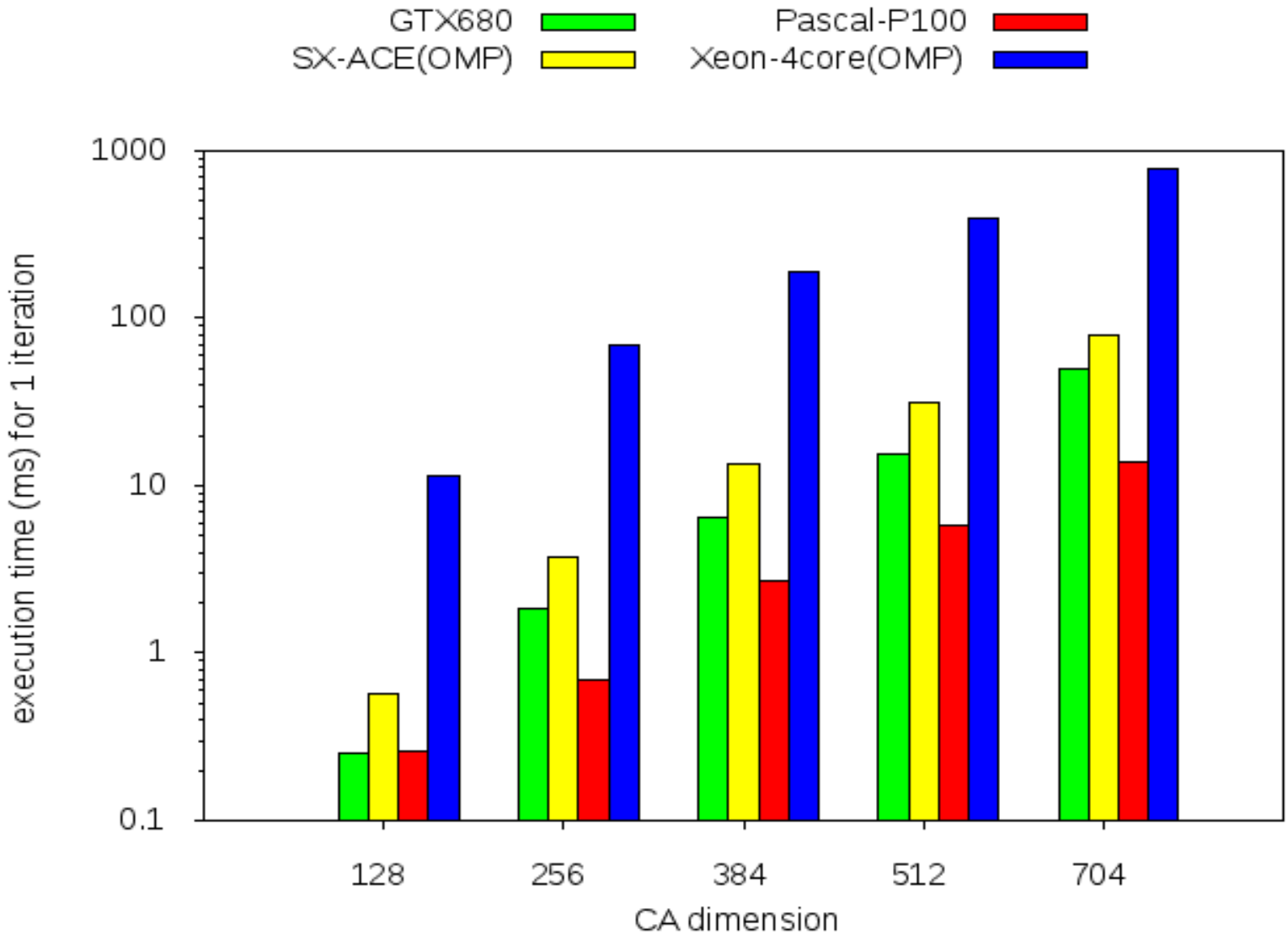
"branching / bitwise ops. / OpenMP"

code	Execution (ms)	speedup
Sequential, branching	107.9	1.00
Sequential, bitwise ops.	233.11 (!)	0.46 x
Parallel, branching	81.74	1.00
Parallel, bitwise ops.	79.03	1.034 x

On Tesla P100 (dim=1024) bitwise ops. Yield ~ 3 % performance increase.


Comparison GTX680 / dual-GTX680 / Xeon
GTX 680, dim=512
2 x GTX 680, dim=512
E5-1620 v3 @ 3.50GHz
execution time (s), smaller is better
probability P (pixel density)
0
5
10
15
20
25
30
35
40
45
50

benchmark single iteration:



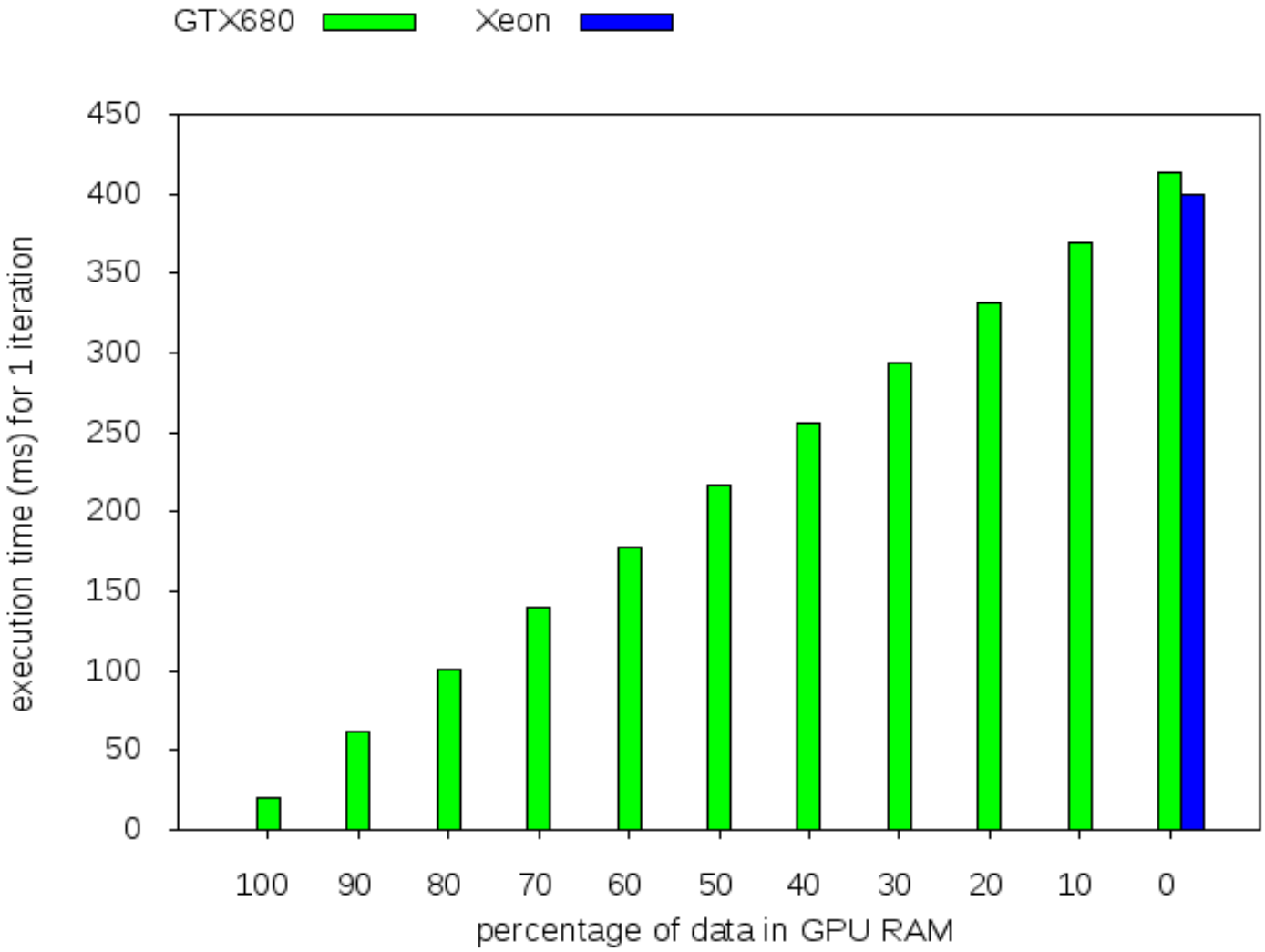
Performance-overview: speedup

dim	GTX 680	SX-ACE	Pascal	Xeon Phi	Xeon
128	45 x	19.9 x	43.7 x		1
256	37 x	18.5 x	99.7 x		1
512	25.7 x	12.5 x	68.9 x		1
704	15.5 x	9.7 x	55.2 x	18.1 x	1

Execution times (ms):

dim	GTX 680	SX-ACE	Pascal	Xeon Phi	Xeon
128	0.25	0.57	0.26	n/a	11.37
256	1.86	3.71	0.69	n/a	68.85
512	15.55	31.81	5.8	n/a	399.86
704	49.55	79.03	13.98	42.44	772.18

Mixing GPU / CPU RAM: out of core execution



Conclusions:

CA-based approach marks viable solution
- speedup on accelerator- and “real” vector hardware.
Matlab (Xeon) : 56.000 ms
CA on P100 : 271 ms (> 206 x speedup)

SX ACE – surprisingly high performance from single node, despite short vector length

outlook: - target n-dim case
- use NVIDIA Pascal page-faulting mechanism for larger problem sizes

Thank you !