

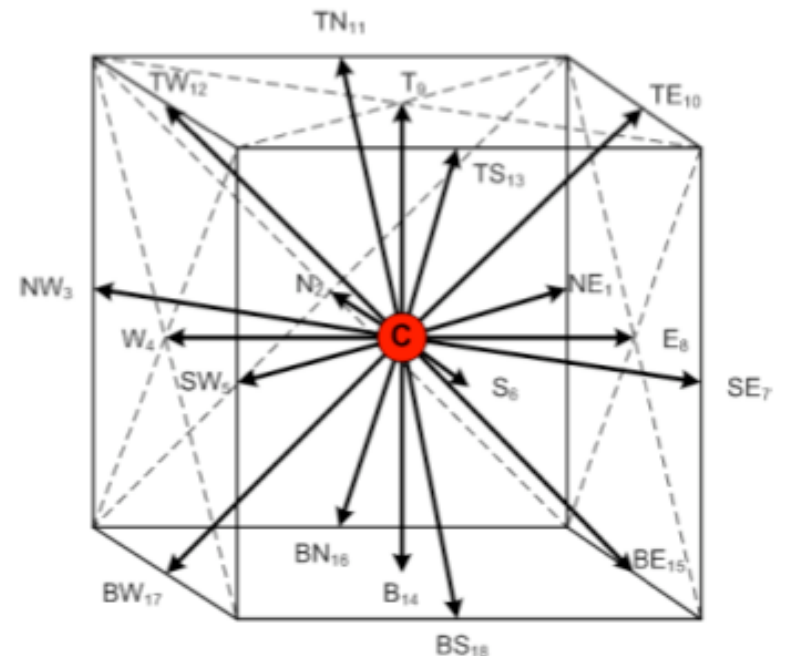
A full example: Lattice Boltzmann on CUDA

$$(\partial_t + \vec{v} \partial_x) f(\vec{x}, \vec{v}, t) = \omega(f^{eq} - f)(\vec{x}, \vec{v}, t)$$

$$f_i(\vec{x} + \vec{c}_i, t + 1) = (1 - \omega) f_i(\vec{x}, t) + \omega f_i^{eq}(\vec{x}, t)$$

$$f_i^{eq} = w_i \rho \left[1 + \frac{\vec{u} \cdot \vec{c}_i}{c^2} + \frac{\vec{u} \vec{u} : (\vec{c}_i \vec{c}_i - \vec{I})}{2c^4} \right]$$

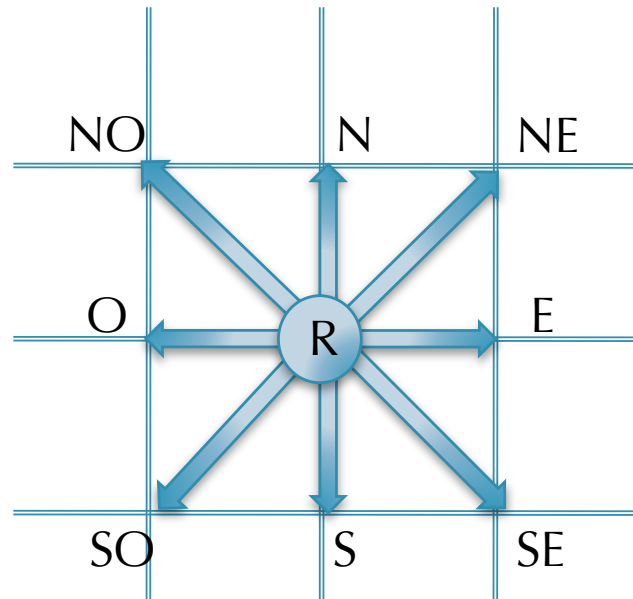
Navier-Stokes solver!



Lattice Boltzmann in 2D

$$f_i(\vec{x} + \vec{c}_i, t + 1) = (1 - \omega) f_i(\vec{x}, t) + \omega f_i^{eq}(\vec{x}, t)$$

$$f_i(\vec{x} + \vec{c}_i, t + 1) = f_i^*(\vec{x}, t)$$



D2Q9

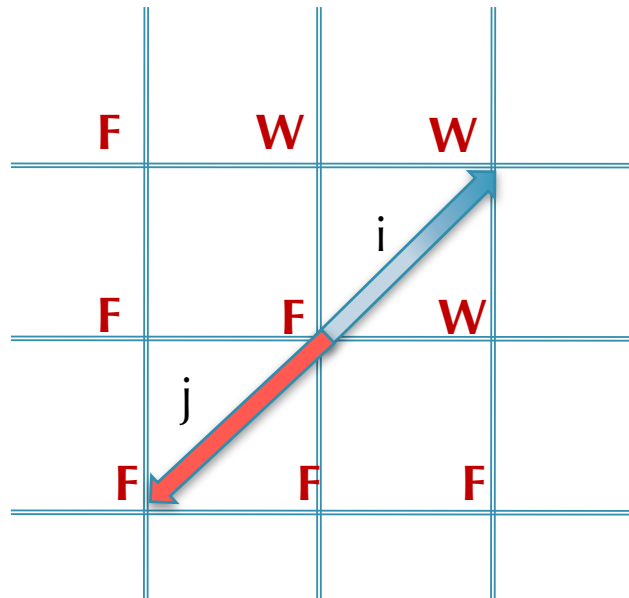
Mesh Node Tagging

F **W** **I** **O**

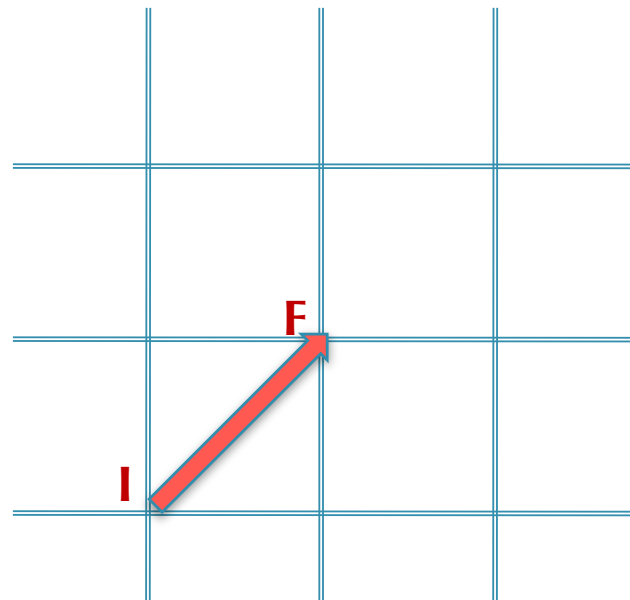
| | | | |
|----------|----------|----------|----------|
| W | W | W | O |
| W | F | F | O |
| F | F | F | W |
| I | F | F | W |

No-slip on Walls: Bounce Back

$$f_i(\vec{x} + \vec{c}_i, t + 1) = f_j(\vec{x} + \vec{c}_j, t)$$



Inlet/Outlet

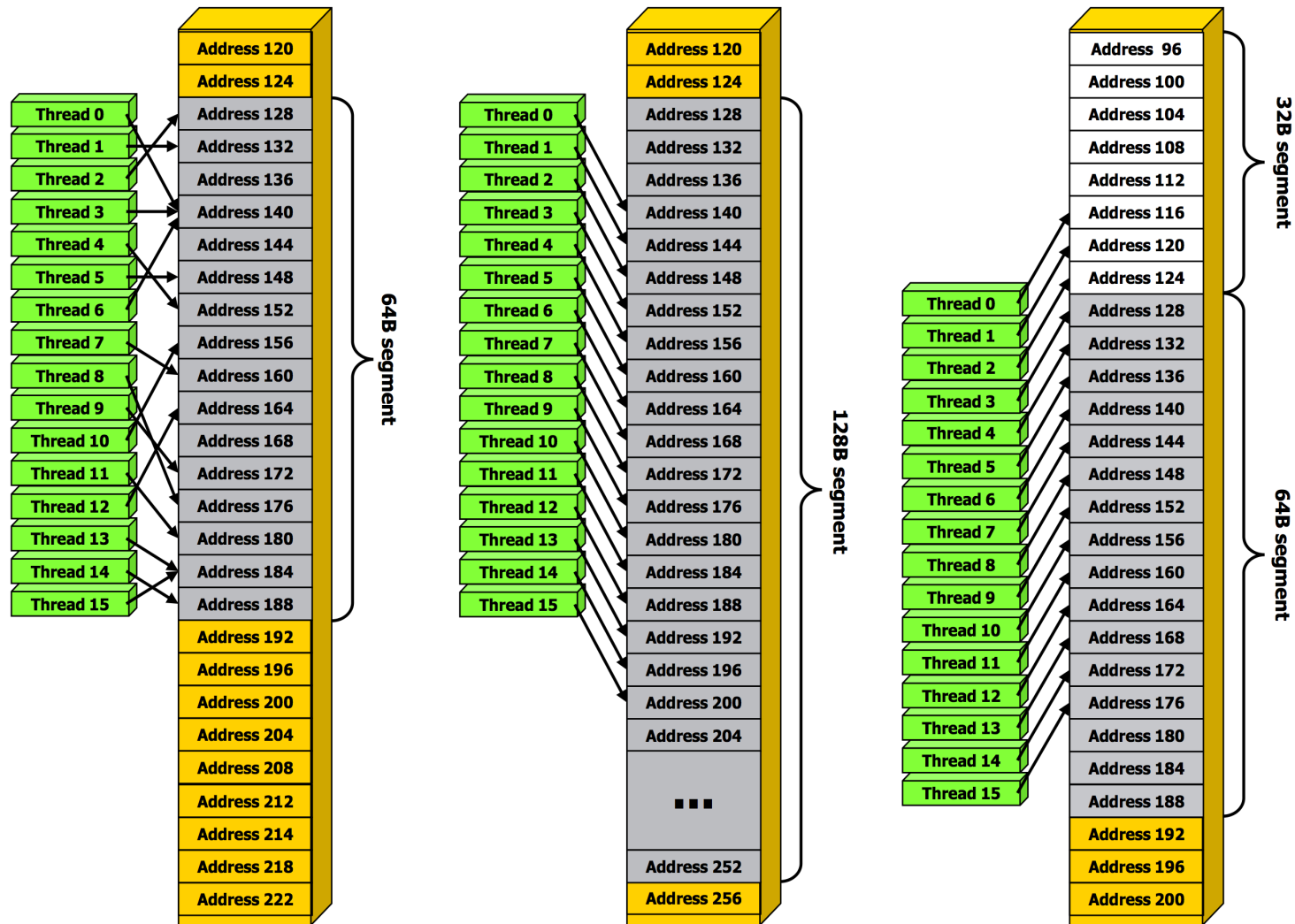


Uncoalesced accesses to GPU global memory

Memory accesses to global GPU memory should be properly aligned to guarantee highest possible bandwidth between memory and processors.

- ✓ In CUDA jargon, sub-optimal memory access patterns are labeled as *uncoalesced*.
- ✓ The definition of “optimal” access depends on the *capability* of the GPU.
 - ✓ GPU with capability up to 1.1 have very strict rules.
 - ✓ An uncoalesced memory access reduced the effective memory bandwidth of (about) one order of magnitude!
 - ✓ Modern GPU (like the T10) are much more *forgiving*...
 - ✓ Basically any access made by a *half-warp* (16 threads) within a segment of 128 bytes is carried out by means of a single transaction.

Uncoalesced accesses to GPU global memory



Memory alignment

High memory bandwidth:

1. access device memory by one block as a continuous, aligned access
2. each thread id N should access element N at address

$$\mathbf{BaseAddress + sizeof(type)*N}$$

with

sizeof(type)=4, 8, 16

BaseAddress aligned to $16 * \text{sizeof(type)}$ bytes

Any address of a variable in global memory is always aligned in this way.

*A simple example: $M_{ij}=0.5*M_{ij}$*

```
// allocation of host memory
float* fH = (float*) malloc
              (mem_size_Mat);

// initialize host memory
for(y=0 ; y< ny ; y++){
    for(x=0 ; x< nx ; x++){
        k = nx*y+x;
        fH[k]=1.0;
    }
}

// allocate device memory
cudaMalloc((void**) &f0, mem_size_Mat);
cudaMalloc((void**) &f1, mem_size_Mat);
// copy host memory to device
cudaMemcpy(f0, fH, mem_size_Mat,
           cudaMemcpyHostToDevice);
// setup execution parameters
dim3 threads(num_threads, 1, 1);
dim3 grid(nx/num_threads, ny);
//Execute the kernel
Kernel<<< grid, threads >>>
    ( nx, f0,f1);
...
```

The device code (kernel) is very simple and reads as follows:

```
_global_ void Kernel(int nx,float* f0,
                    float* f1)
{
    // number of threads
    int num_threads = blockDim.x;
    // Thread index
    int tx = threadIdx.x;
    // Block index x
    int bx = blockIdx.x;
    // x-Index
    int x = tx + bx*num_threads;
    // Block index y = y-Index
    int y = blockIdx.y;
    // f0[k]:Load data from device memory
    // f1[k]:Write data to device memory
    int k = nx*y + x;
    f1[k]=0.5*f0[k];
}
```

With this setup a performance of 72 GB/s is achieved corresponding to 70% of the theoretical maximum bandwidth.

Setting the stage

Allocate $2 \times 8=18$ arrays for double buffers
(post-collisional and post-streaming ops).

Avoid using two separate steps:

$$1\ R + 1\ W + 1\ R + 1\ W \rightarrow 1\ R + 1\ W\ (\text{FUSED})$$

Let n_x, n_y to be multiples of 16 (relax constraints by padding)

Arrays allocated with an offset of $\text{startoff}=16$ in y -direction

Kernel: LBCollProp

Collision and propagation of the fluid and bounce-back

Layout:

A block is

(num_threads, 1, 1)

Grid of blocks is

(nx/num_threads, ny)

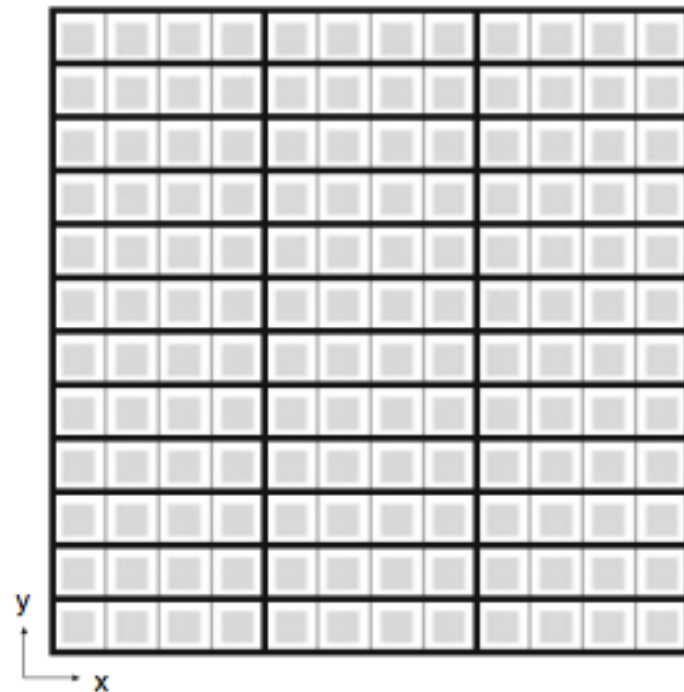


Fig. 2 Grid configuration for LBCollProp for a 12×12 matrix and 4 threads: grid configuration is represented by *black lines* (12×4), the thread partition within one block is represented by *gray lines*

Kernel: LBExchange

Synchronize the populations at the block borders

Layout:

A block is
(num_threads, 1, 1)

Grid of blocks is
(1, ny/num_threads)

(elements in rows independent)

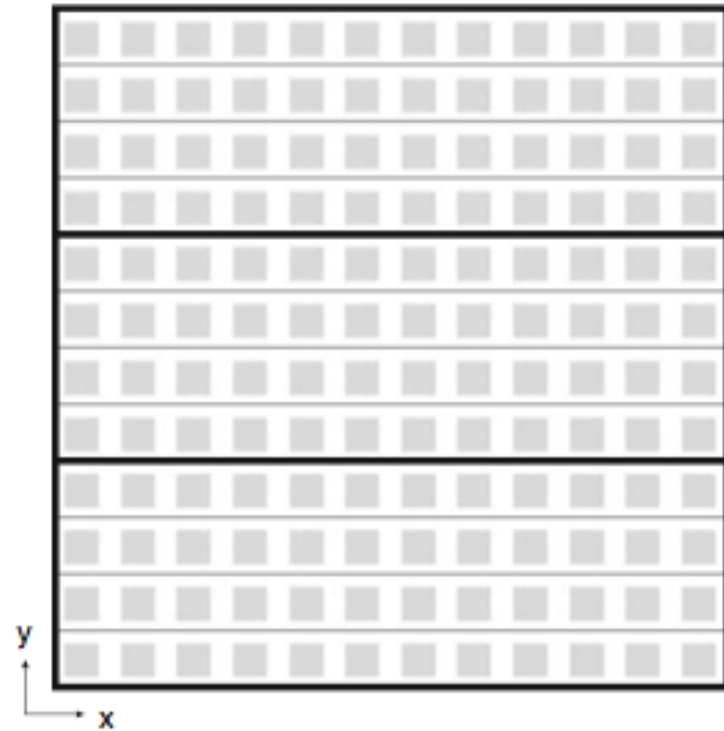


Fig. 3 Grid configuration for LBExchange and LBBC for a 12×12 matrix and 4 threads: grid configuration is represented by *black lines* (3×1), the thread partition within one block is represented by *gray lines*

Kernel: LBBC

Set the populations at the Inlet and Outlet

Layout:

same as before....

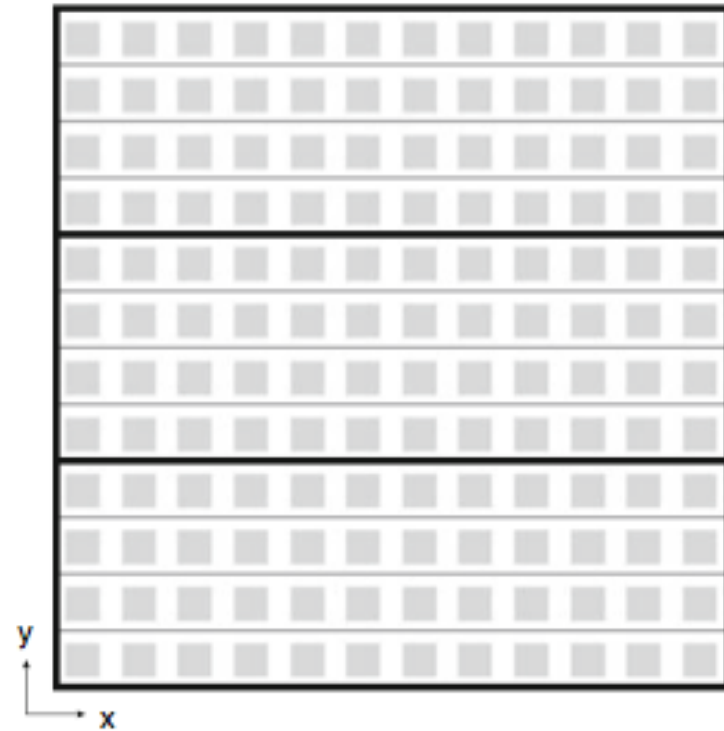


Fig. 3 Grid configuration for LBExchange and LBBC for a 12×12 matrix and 4 threads: grid configuration is represented by *black lines* (3×1), the thread partition within one block is represented by *gray lines*

Main loop

```
...
//allocate  fr0,fe0,fn0,fw0,fs0,fne0,fnw0,fsw0,fse0
//and  fr1,fe1,fn1,fw1,fs1,fne1,fnw1,fsw1,fse1
...
dim3 threads(num_threads, 1, 1);
dim3 grid(nx/num_threads, ny);
dim3 grid1(1, ny/num_threads);
```

```
...
for(t=0;t<=tend;t++)
{
    //Set Pointers
    if(t%2==0)
    {
        frOld=fr0;feOld=fe0;fnOld=fn0;...
        frNew=fr1;feNew=fe1;fnNew=fn1;...
    }
    else{
        frOld=fr1;feOld=fe1;fnOld=fn1;...
        frNew=fr0;feNew=fe0;fnNew=fn0;...
    }

    //collision and propagation
    LBCollProp<<< grid, threads >>> (nx, ny,
        startoff, s, geoD,
        frOld, feOld, fnOld, fwOld, fsOld,
        fneOld, fnwOld, fswOld, fseOld,
        frNew, feNew, fnNew, fwNew, fsNew,
        fneNew, fnwNew, fswNew, fseNew);

    //synchronize distributions across
    // thread blocks
    LBExchange<<< grid1, threads >>> (nx, ny,
        startoff, nx/num_threads, feNew, fwNew,
        fneNew, fnwNew, fswNew, fseNew);

    //impose boundary conditions on
    //in- and outlet
    LBBC<<< grid1, threads >>> ( nx, ny,
        startoff, geoD,
        frNew, feNew, fnNew, fwNew, fsNew,
        fneNew, fnwNew, fswNew, fseNew);

    //Postprocess results
    ...
}
}
```

Kernel: LBCollProp

R, N and S populations directly written in device memory since they are aligned at $16 * \text{sizeof}(\text{type})$ bytes

W, SW, NW, E, SE, NE **misaligned!**

Writing directly into dev memory would degrade bandwidth 100 GB/s → 10 GB/s

TRICK:

Use shared memory to :

1. propagate without a shift
2. write back from shared to device memory
3. populations leaving the border reinjected at the opposite side of the block
(shared mem is only available within a block)

Kernel: LBCollProp (1)

```
__global__ void LBCollProp(int nx, int ny,
int startoff, float4 s, unsigned int* geoD,
float* fr0, float* fe0, float* fn0,
float* fw0, float* fs0, float* fne0,
float* fnw0, float* fsw0, float* fse0,
float* fr1, float* fe1, float* fn1,
float* fw1, float* fs1, float* fne1,
float* fnw1, float* fsw1, float* fse1)
{
    // number of threads
    int num_threads = blockDim.x;

    // local thread index
    int tx = threadIdx.x;

    // Block index in x
    int bx = blockIdx.x;

    // Block index in y
    int by = blockIdx.y;

    // Global x-Index
    int xStart = tx + bx*num_threads;

    // Global y-Index
    int yStart = by + startoff;

    // Index k in 1D-arrays
    int k = nx*yStart+xStart;

    //Shared memory for propagation
    __shared__ float F_OUT_E[THREAD_NUM];
    __shared__ float F_OUT_W[THREAD_NUM];
    __shared__ float F_OUT_NE[THREAD_NUM];
    __shared__ float F_OUT_NW[THREAD_NUM];
    __shared__ float F_OUT_SW[THREAD_NUM];
    __shared__ float F_OUT_SE[THREAD_NUM];

    ...
    //load fr0[k], fe0[k], fn0[k], fw0[k], fs0[k],
    //fne0[k], fnw0[k], fsw0[k], fse0[k]
    //to local variables F_IN_R, F_IN_E, F_IN_N,
    //F_IN_W, F_IN_S, F_IN_NE, F_IN_NW, F_IN_SW, F_IN_SE
    ...
}
```


Kernel: LBCollProp (2)

```
if(geoD[k] == GEO_FLUID)
{
    //collision:
    //modify F_IN_R,F_IN_E,...,F_IN_SE
    ...
}
else if(geoD[k] == GEO_SOLID)
{
    //bounce back:
    //modify F_IN_R,F_IN_E,...,F_IN_SE
    ...
}

//Propagation using shared memory for
//distributions having a shift
//in east or west direction
if(tx==0)
{
    F_OUT_E [tx+1]=F_IN_E;
    F_OUT_NE[tx+1]=F_IN_NE;
    F_OUT_SE[tx+1]=F_IN_SE;

    //store distribution leaving
    //the domain across the west border
    F_OUT_W [num_threads-1]=F_IN_W;
    F_OUT_NW[num_threads-1]=F_IN_NW;
    F_OUT_SW[num_threads-1]=F_IN_SW;
}

else if(tx==num_threads-1)
{
    //store distribution leaving
    //the domain across the east border
    F_OUT_E [0]=F_IN_E;
    F_OUT_NE[0]=F_IN_NE;
    F_OUT_SE[0]=F_IN_SE;

    F_OUT_W [tx-1]=F_IN_W;
    F_OUT_NW[tx-1]=F_IN_NW;
    F_OUT_SW[tx-1]=F_IN_SW;
}
else{
    F_OUT_E [tx+1]=F_IN_E;
    F_OUT_NE[tx+1]=F_IN_NE;
    F_OUT_SE[tx+1]=F_IN_SE;
    F_OUT_W [tx-1]=F_IN_W;
    F_OUT_NW[tx-1]=F_IN_NW;
    F_OUT_SW[tx-1]=F_IN_SW;
}
```

Kernel: LBCollProp (3)

```
// synchronize threads
__syncthreads();

// write data back to device
frl[k]=F_IN_R;
fel[k]=F_OUT_E[tx];
fwl[k]=F_OUT_W[tx];

k = nx*(yStart+1) + xStart;
fnl[k]=F_IN_N;
fnel[k]=F_OUT_NE[tx];
fnwl[k]=F_OUT_NW[tx];

k = nx*(yStart-1) + xStart;
fsl[k]=F_IN_S;
fswl[k]=F_OUT_SW[tx];
fsel[k]=F_OUT_SE[tx];
```

Kernel: LBExchange

```
__global__ void LBExchange(int nx, int ny,
int Startoff, int nbx, float* fel, float* fw1,
float* fnel, float* fnw1, float* fsw1, float* fsel)
{
    //number of elements of one thread block in LBCollProp
    int num_threads = THREAD_NUM;

    //number of threads in this function
    int num_threadsl = blockDim.x;

    // Block index y
    int by = blockIdx.y;

    // local thread index
    int tx = threadIdx.x;

    int bx;
    int xStart, yStart;
    int xStartW, xTargetW;
    int xStartE, xTargetE;
    int kStartW, kTargetW;
    int kStartE, kTargetE;
```

```
    for(bx=0; bx<nbx ;bx++)
    {
        xStart = bx*num_threads;
        xStartW = xStart+2*num_threads-1;
        xTargetW = xStartW-num_threads;
        yStart = by*num_threadsl + Startoff + tx;
        kStartW = nx*yStart+xStartW;
        kTargetW = nx*yStart+xTargetW;

        fw1[kTargetW] = fw1[kStartW];
        fnw1[kTargetW] = fnw1[kStartW];
        fsw1[kTargetW] = fsw1[kStartW];
    }

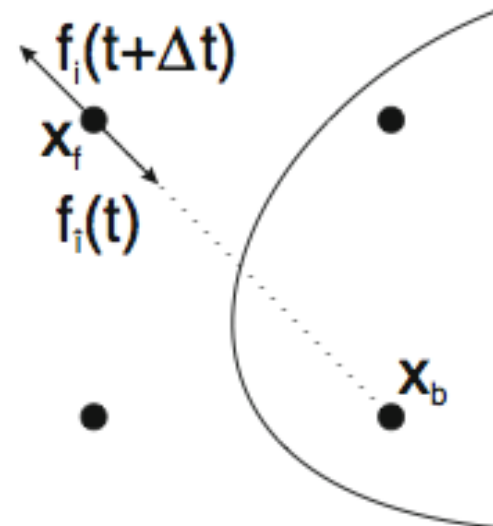
    for(bx=nbx-1; bx>=0 ;bx--)
    {
        xStart = bx*num_threads;
        xStartE = xStart;
        xTargetE = xStartE+num_threads;
        yStart = by*num_threadsl + Startoff + tx;
        kStartE = nx*yStart+xStartE;
        kTargetE = nx*yStart+xTargetE;

        fel[kTargetE] = fel[kStartE];
        fnel[kTargetE] = fnel[kStartE];
        fsel[kTargetE] = fsel[kStartE];
    }
}
```

Obstacles & suspended bodies

$$\mathbf{F}_k(t + \Delta t/2) = -2 \frac{\Delta x^2 l_z}{\Delta t} \mathbf{e}_i f_i(t + \Delta t, \mathbf{x}_f),$$

$$\mathbf{F} = \sum_{k \in \mathcal{C}} \mathbf{F}_k + \mathbf{F}_{\text{body}}$$

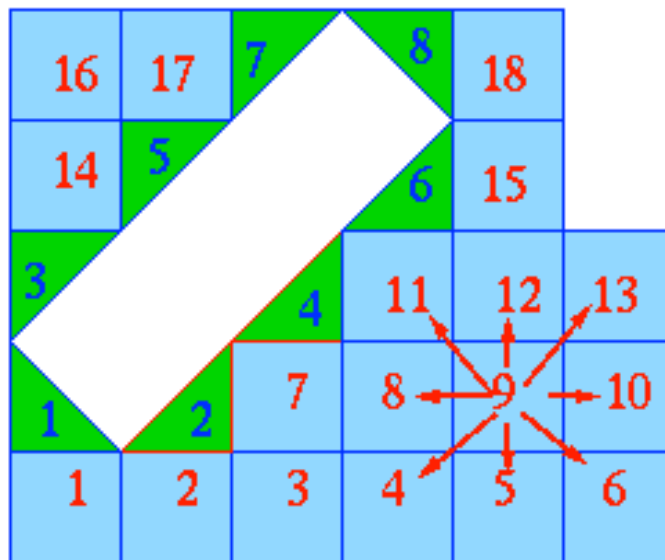


Strategy does not change!

Sparse Mesh: Indirect addressing

“Sparse” matrix representation with indirect addressing

- ✓ Like in linear algebra, only data really useful for the calculation stored in memory



Advantages:

- ✓ Only *fluid*, *wall*, *inlet*, *outlet* nodes need to be stored

- ✓ Homogeneous nodes can be stored contiguously

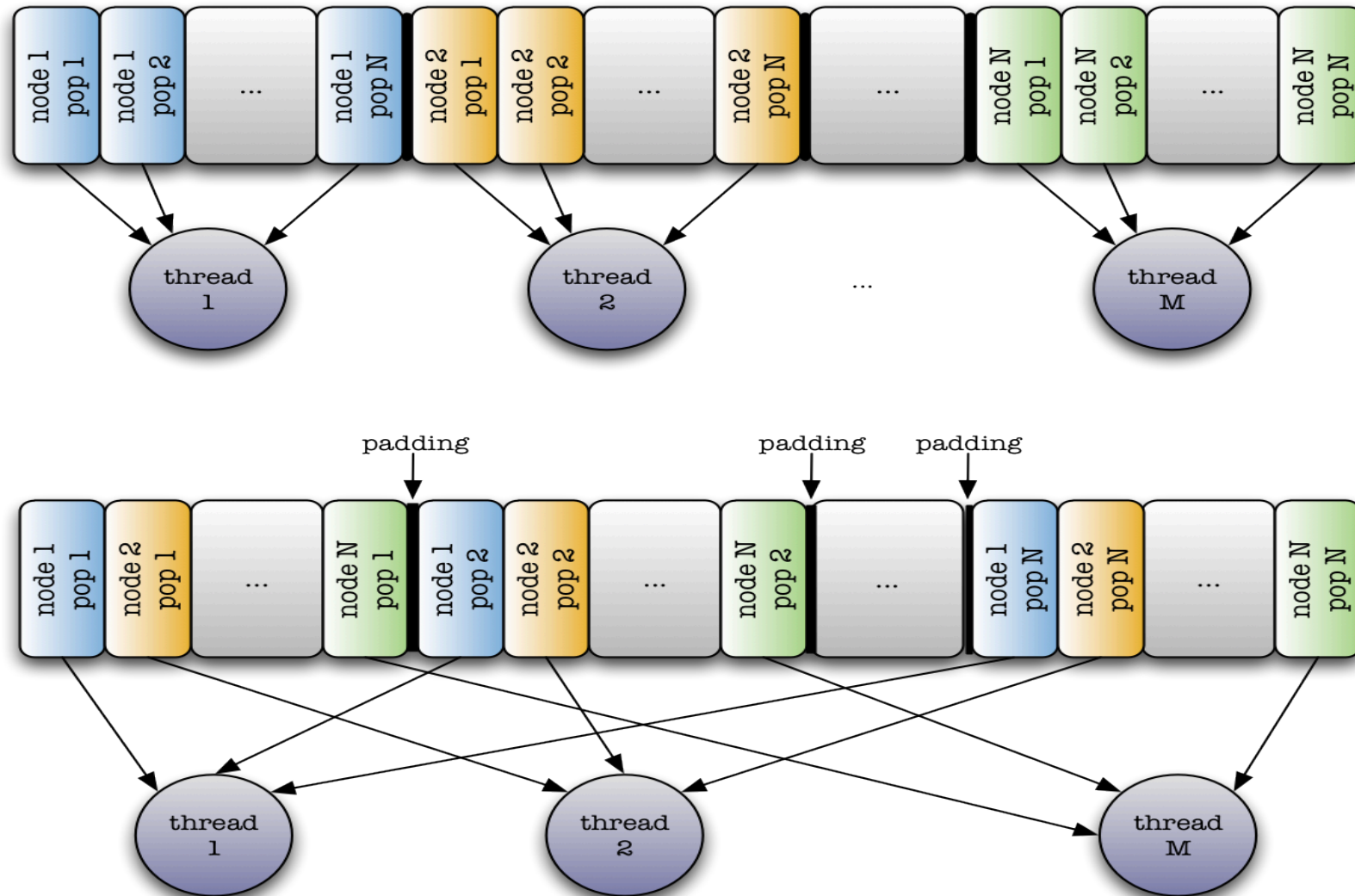
Drawbacks:

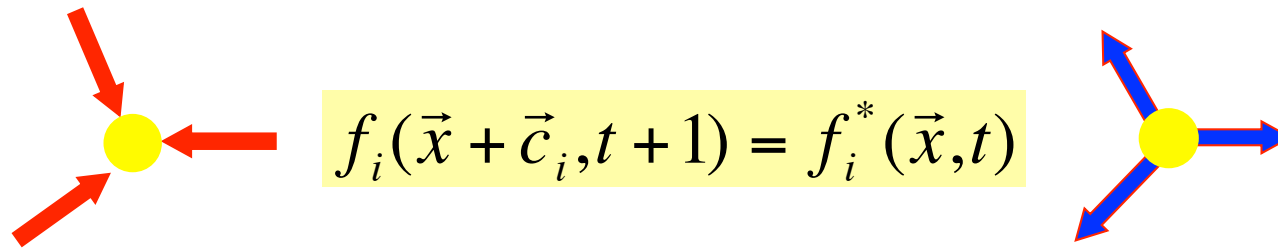
- ✓ Target locations of the *stream* phase not easily computed

- ✓ **Connectivity** matrix required.

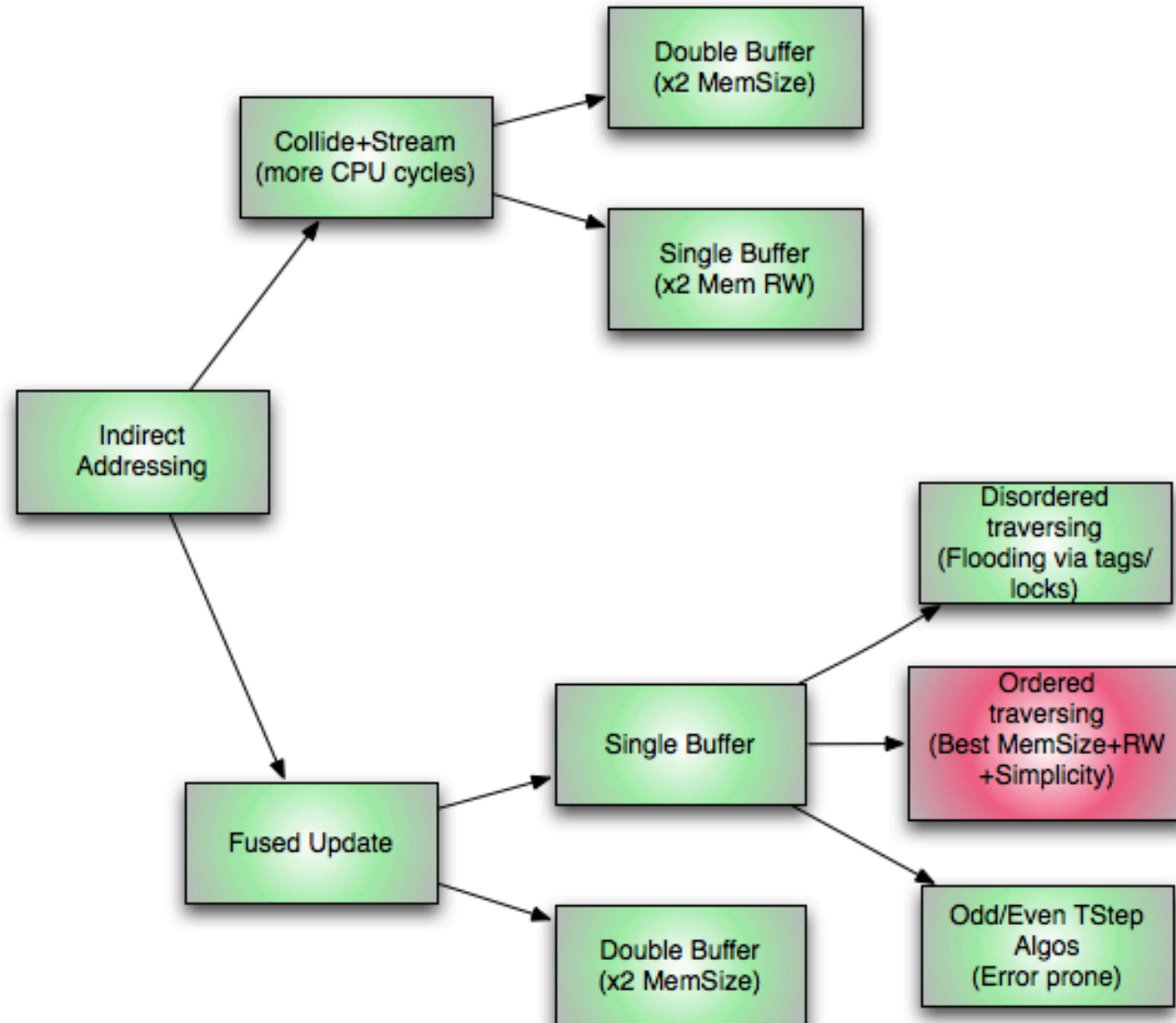
| | | | | | | | | |
|---|---|---|---|---|----|----|----|----|
| 9 | 4 | 5 | 6 | 8 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|----|----|----|----|

Different strategies for the LB update

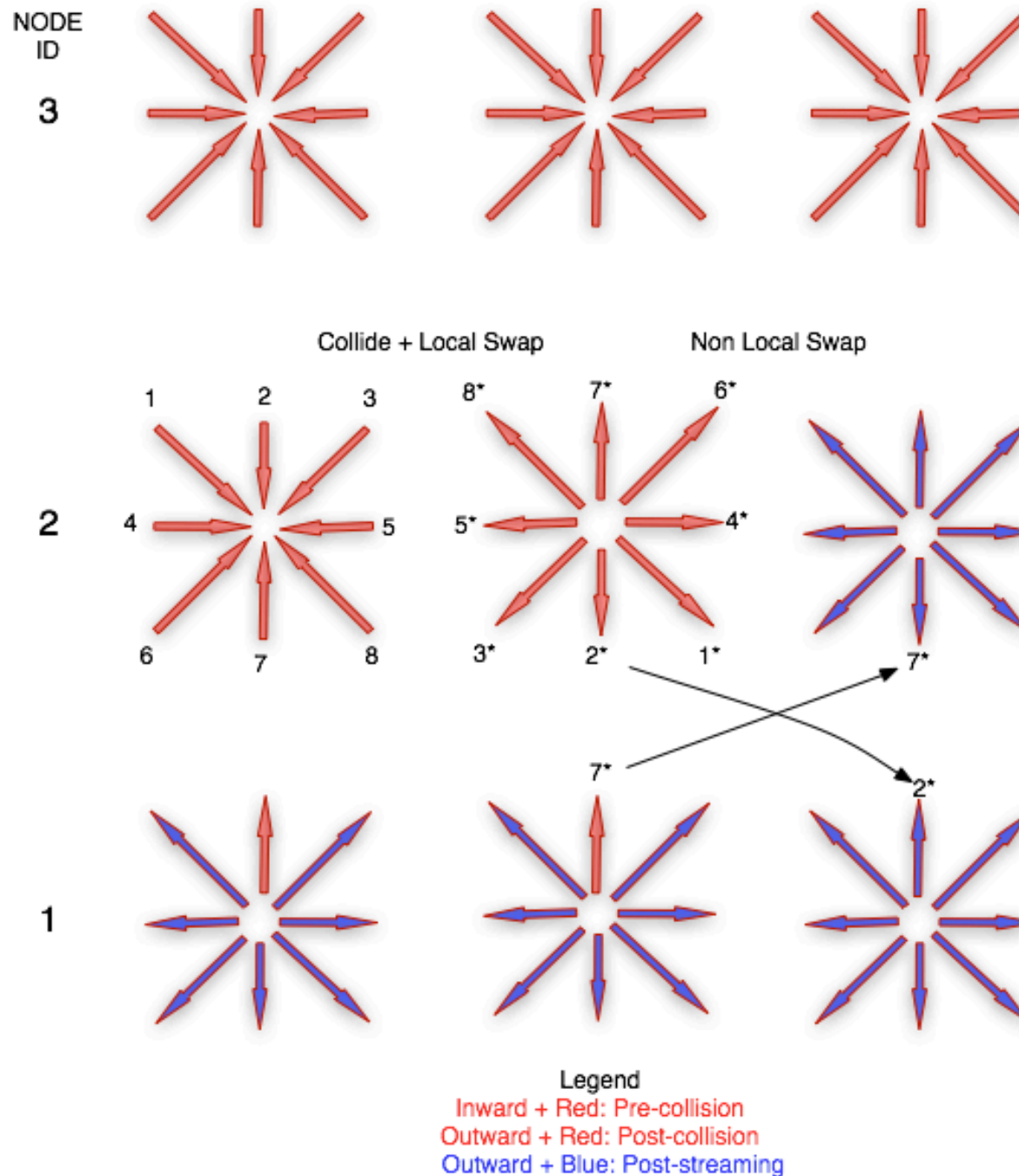




$$f_i(\vec{x} + \vec{c}_i, t + 1) = f_i^*(\vec{x}, t)$$

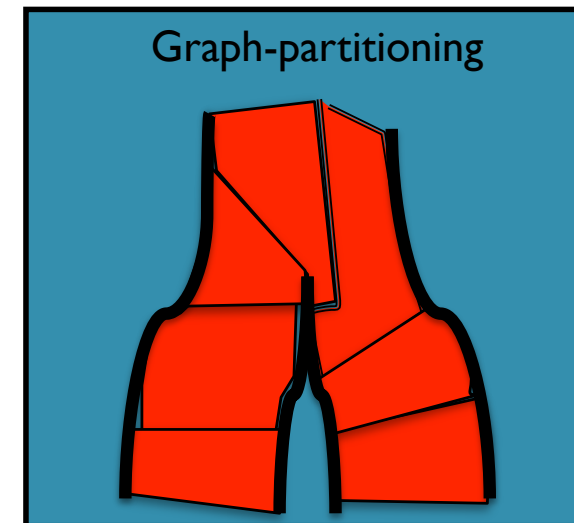
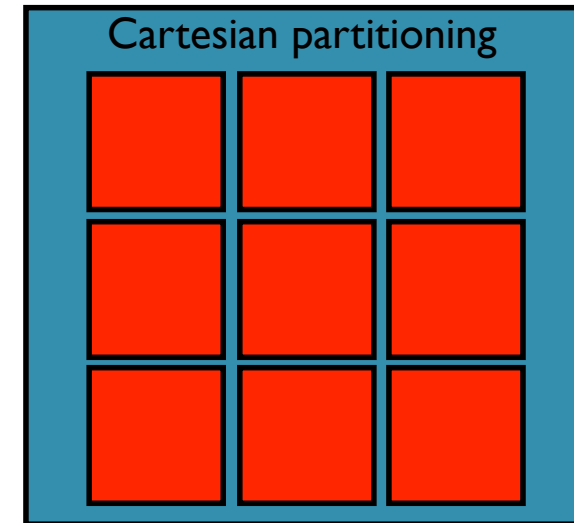


Indirect addressing and Swap Trick



Multi-GPU: Domain Decomposition

- ✓ Domain decomposition by cartesian or graph-based (multilevel k-way) partitioning.
- ✓ Indirect addressing for very sparse domains.
- ✓ Each domain assigned to a CPU and GPU



Results (1)

Three sets of tests:

1. GPU-only tests to spot possible bottlenecks
2. Comparison between GPU and “traditional” multicore CPUs.
3. Multiple GPUs

For the first set of tests we measured the performance of four different GPU kernels using the CUDA Visual Profiler.

These tests have been executed on a **old** Nvidia Tesla 870 (capability = 1.0) with the following features:

- ✓ 128 Processor cores
- ✓ Clock rate: 1.3 Ghz
- ✓ Global memory: 1.5 Gbytes
- ✓ Memory bandwidth: 76.8 Gbytes/sec.

Results (2)

Small test case: 1000 iterations on 55800 fluid nodes (62x30x30)

| <i>GPU kernel</i> | GPU time (in sec.) | Global load coherent | Global store coherent | Global store uncoherent |
|------------------------|-----------------------|-------------------------|--------------------------|----------------------------|
| Load/local store | 0.15 | 828308 | 33132352 | 0 |
| Load/scattered store | 0.9 | 16131112 | 1743904 | 251093664 |
| Load/F.P./local store | 0.33 | 8285634 | 33142536 | 0 |
| Load/F.P./scatt. store | 1.0 | 16132222 | 1744024 | 251110368 |

- ✓ The first GPU kernel is a simple memory copy
- ✓ The second kernel is also a memory copy but the store operations make use of the connectivity matrix
- ✓ The third kernel carries out the LB update but stores the results without using the info in the connectivity matrix
- ✓ The fourth kernel corresponds to the real LB update.

Results (3)

- ✓ Sustained bandwidth for the first kernel is ~ 57 Gbyte/sec.
- ✓ The LB update carries out 56 Million Lattice UPdates/Sec. (MLUPS)
- ✓ The penalty due to *uncoalesced* memory accesses is very large
(there is a factor 6 of difference between the first and the second kernel)
but it is somehow alleviated by the floating point operations of the LB
(the difference between the third and the fourth kernel is only a factor 3).

Results (4)

A slightly different test case:

1000 iterations on 65536 fluid nodes (64x32x32)

| <i>GPU kernel</i> | GPU time (in sec.) | Global load coherent | Global store coherent | Global store uncoherent |
|------------------------|--------------------------|-------------------------|--------------------------|----------------------------|
| Load/F.P./scatt. store | .82 | 18944000 | 18432000 | 163840000 |

- ✓ The number of *uncoalesced* memory accesses decreases of ~ 37%
- ✓ The LB update carries out ~ 80 MLUPS (an improvement of ~ 40%)
- ✓ For any other lattice size the performance is $56 \leq \text{MLUPS} \leq 80$
- ✓ Note that regardless of the number of *uncoalesced* memory accesses, the indirect addressing scheme requires a number of load operations that is almost double with respect to implementations using the *full matrix* (*no indirect addressing*) representation.

Results (5)

In the second set of tests we compared the performance of two GPUs:

GeForce 8600M GT:

- ✓ clock rate equal to 750 MHz
- ✓ total global memory 256 MBytes
- ✓ total number of registers *per block* 8192

with the Intel Core 2 Duo at 2.4 GHz of a Mac Book Pro that contains a GeForce 8600 as video card

Two test cases:

1. a small regular domain (64x32x32)
2. an irregular geometry representing (at low resolution) a human coronary artery having a bounding box 89x234x136.

With the *full matrix* representation this test would not fit in the memory of a GPU!

Results (6)

For both test cases the GeForce 8600M GT outperforms the Intel Core 2 Duo by a factor 4 and executes more than 10 MLUPS. An excellent result for a simple video card of a laptop!

Then, we compared the performance of the Tesla S870

- ✓ clock rate equal to 1.3 MHz
- ✓ total global memory 1.5 GBytes
- ✓ total number of registers *per* block 8192
- ✓ 128 processor cores.

with an Intel Xeon E5405 2.4 GHz 12MB L2 Cache
Quad-Core Processor.

The LB update on the Tesla S870 outperforms a multithreaded version for the Intel Quad-core by a factor 10 for the small domain test case and by a factor ~8 for the irregular geometry test case.

Results (7)

In the third set of tests we used a *cluster* of GPUs:

- ✓ 4 Quad core Xeon 2.8Ghz connected by Infiniband
- ✓ each node equipped with two pre-production GT200 GPUs (for a total of 8 GPUs)
- ✓ total global memory of the GPUs is 32 Gbytes.

Fluid populations need to be exchanged among GPUs

- ✓ GPUs do not communicate each other directly.
 1. data travel from GPU to the *source* CPU
 2. then from the *source* CPU to the CPU of the *target* GPU
 3. finally from the *target* CPU to the *target* GPU.
- ✓ A buffering mechanism has been implemented to minimize the communication between a GPU and the host CPU.

Speedup of a small test case on 28 GPUs

