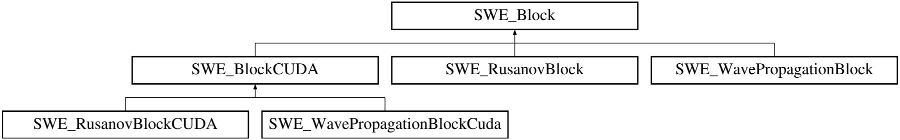
SWE and GPU Computing with CUDA

CSCS-FoMICS-USI Summer School on Computer Simulations in Science and Engineering

Michael Bader

July 8–19, 2013

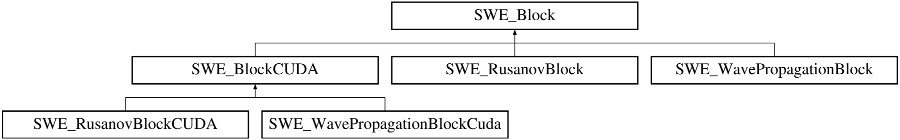
SWE Class Design



abstract class SWE_Block:

- base class to hold data structures (arrays h, hu, hv, b)
- manipulate ghost layers
- methods for initialization, writing output, etc.
- defines interface for main time-stepping loop: computeNumericalFluxes(), updateUnknowns(), . . .

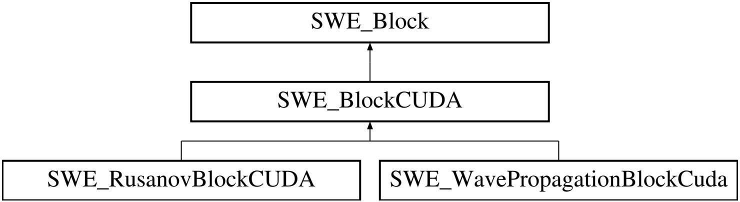
SWE Class Design (2)



derived classes:

- for different model variants: SWE_RusanovBlock, SWE_WavePropagationBlock, . . .
- for different programming models: SWE_BlockCUDA, SWE_BlockArBB, . . .
- override computeNumericalFluxes(), updateUnknowns(), . . .
  $\rightarrow$ methods relevant for parallelization

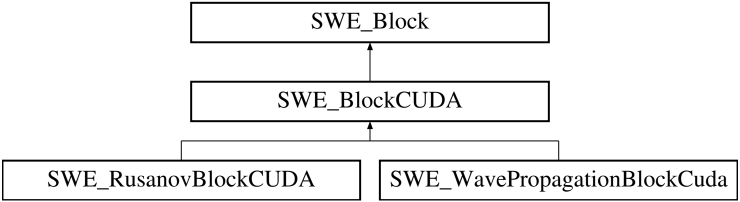
SWE Class Design – SWE_BlockCUDA



abstract class SWE_Block:

- base class to hold data structures (arrays h, hu, hv, b)
- manipulate ghost layers
- methods for initialization, writing output, etc.

SWE Class Design – SWE_BlockCUDA (2)



derived classes:

- for different model variants: SWE_RusanovBlock, SWE_WavePropagationBlock, . . .
- for different programming models: SWE_BlockCUDA, SWE_BlockArBB, . . .
- override computeNumericalFluxes(), updateUnknowns(), . . .
  $\rightarrow$ methods relevant for parallelization

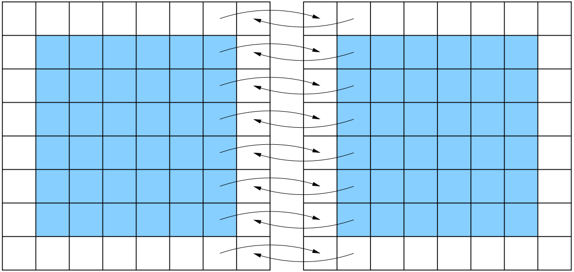
Example: SWE_WavePropagationBlockCUDA

```
class SWE_WavePropagationBlockCuda: public SWE_BlockCUDA {
  /*-- definition of member variables skipped --*/
  public:
    // compute a single time step (net-updates + update of the cells).
    void simulateTimestep( float i_dT );
    // simulate multiple time steps (start and end time provided as param
    float simulate(float, float);
    // compute the numerical fluxes (net-update formulation here).
    void computeNumericalFluxes();
    // compute the new cell values.
    void updateUnknowns(const float i_deltaT);
};
```

(in file SWE_WavePropagationBlockCuda.hh)

Part I

Parallel Programming Patterns



Computing the Net Updates

Parallel Programming Patterns

- compute net updates on left/right edges:

```
for(int i=1; i < nx+2; i++) in parallel {
  for(int j=1; j < ny+1; j++) in parallel {
    float maxEdgeSpeed;
    fWaveComputeNetUpdates( 9.81,
        h[i-1][j], h[i][j], hu[i-1][j], hu[i][j], /* ... */ );
  }
}
```

- compute net updates on top/bottom edges:

```
for(int i=1; i < nx+1; i++) in parallel {
  for(int j=1; j < ny+2; j++) in parallel {
    fWaveComputeNetUpdates( 9.81,
        h[i][j-1], h[i][j], hv[i][j-1], hv[i][j], /* ... */);
  }
}
```

(function fWaveComputeNetUpdates() defined in file solver/FWaveCuda.h)

Computing the Net Updates

Options for Parallelism

Parallelization of computations:

- compute all vertical edges in parallel
- compute all horizontal edges in parallel
- compute vertical & horizontal edges in parallel (task parallelism)

Parallel access to memory:

- concurrent read to variables h, hu, hv
- exclusive write access to net-update variables on edges

Updating the Unknowns

Parallel Programming Patterns

- update unknowns from net updates on edges:

```
for(int i=1; i < nx+1; i++) in parallel {
   for(int j=1; j < ny+1; j++) in parallel {
      h[i][j] -= dt/dx * (hNetUpdatesRight[i-1][j-1]
                         + hNetUpdatesLeft[i][j-1] )
                + dt/dy * (hNetUpdatesAbove[i-1][j-1]
                          + hNetUpdatesBelow[i-1][j] );
      hu[i][j] -= dt/dx * (huNetUpdatesRight[i-1][j-1]
                          + huNetUpdatesLeft[i][j-1] );
      /* ... */
   }
}
```

Updating the Unknowns

Options for Parallelism

Parallelization of computations:

- compute all cells in parallel

Parallel access to memory:

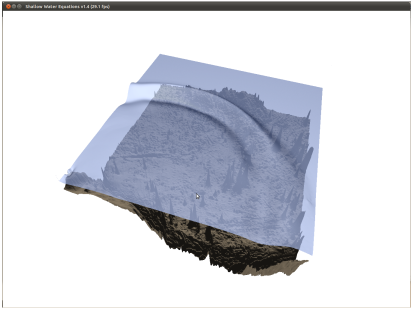
- concurrent read to net-updates on edges
- exclusive write access to variables h, hu, hv

“Vectorization property”:

- exactly the same code for all cell!

Part II

SWE and CUDA



SWE_BlockCUDA – GPU Memory

Additional Member Variables in class SWE_BlockCUDA:

- base class to hold data structures (arrays h, hu, hv, b)
- manipulate ghost layers
- methods for initialization, writing output, etc.

Allocate unknowns h, hu, hv, b in SWE_BlockCUDA:

```
int size = (nx+2)*(ny+2)*sizeof(float);
// allocate CUDA memory for unknows h,hu,hv and bathymetry b
cudaMalloc((void**)&hd, size);
cudaMalloc((void**)&hud, size);
cudaMalloc((void**)&hvd, size);
cudaMalloc((void**)&bd, size);
```

(see constructor SWE_BlockCUDA(...) in file SWE_BlockCUDA.cu)

SWE_BlockCUDA – GPU Memory (2)

Define & Allocate Member Variables in SWE_BlockCUDA:

```
SWE_BlockCUDA::SWE_BlockCUDA(/*-- parameters--*/)
 : SWE_Block(_offsetX,_offsetY)
{ /*-- further  initializations  skipped  --*/
  int size = (nx+2)*(ny+2)*sizeof(float);
  // allocate CUDA memory for unknows h,hu,hv and bathymetry b
  cudaMalloc((void**)&hd, size);
     checkCUDAError("allocate device memory for h");
  cudaMalloc((void**)&hud, size);
     checkCUDAError("allocate device memory for hu");
  cudaMalloc((void**)&hvd, size);
     checkCUDAError("allocate device memory for hv");
  cudaMalloc((void**)&bd, size);
     checkCUDAError("allocate device memory for bd");
  /*-- allocation of ghost/copy layer to follow  --*/
}
```

(see file SWE_BlockCUDA.cu)

Excursion: Checking for CUDA Errors

- CUDA API functions typically return error code as value
- but no exceptions, (immediate) crashes, etc.
- error code should thus be checked after each function call

⇒ helper function defined in SWE_BlockCUDA:

```
void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if ( cudaSuccess != err)
    {
        fprintf (stderr, "\nCuda error (%s): %s.\n",
                 msg, cudaGetErrorString( err) );
        exit(-1);
    }
}
```

(see file SWE_BlockCUDA.cu)

SWE_BlockCUDA – Synchronize Memory

Methods to copy CPU memory to GPU memory:

- called after each external write to arrays h, hu, hv, b (read data from file, set initial conditions, etc.)
- allows to implement individual methods on GPU
- SWE allows data in main memory to be not up-to-date (goal: perform simulation entirely on GPU)

Interface defined in class SWE_Block:

```
void SWE_Block::synchAfterWrite() {
    synchWaterHeightAfterWrite();
    synchDischargeAfterWrite();
    synchBathymetryAfterWrite();
}
```

(see file SWE_Block.cpp)

CUDA Example: Synchronize Water Height

Method synchWaterHeightAfterWrite():

- synchronize array h on CPU and GPU memory
- **after an external update of the water height h**
  (i.e., after an update of CPU main memory)
- copies entire array h (incl. ghost layers) into array hd

```
void SWE_BlockCUDA::synchWaterHeightAfterWrite() {
  /*-- --*/
  int size = (nx+2)*(ny+2)*sizeof(float);
  cudaMemcpy(hd,h.elemVector(), size, cudaMemcpyHostToDevice);
     checkCUDAError("memory of h not transferred");
}
```

(see file SWE_BlockCUDA.cu)

SWE_BlockCUDA – Synchronize Memory (2)

Methods to copy GPU memory to CPU memory:

- called before each external output of arrays h, hu, hv, b (write output to file, etc.)
- allows to implement individual methods on GPU
- helpful for debugging

Interface defined in class SWE_Block:

```
void SWE_Block::synchBeforeRead() {
    synchWaterHeightBeforeRead();
    synchDischargeBeforeRead();
    synchBathymetryBeforeRead();
}
```

(see file SWE_Block.cpp)

CUDA Example: Synchronize Water Height

Method synchWaterHeightBeforeRead():

- synchronize array h on GPU and CPU memory
- **after an update of the water height hd on the GPU**
  (e.g., after computation of one or more time steps on the GPU)
- copies entire array hd (incl. ghost layers) into array h

```
void SWE_BlockCUDA::synchWaterHeightBeforeRead() {
  /*-- --*/
  int size = (nx+2)*(ny+2)*sizeof(float);
  cudaMemcpy(h.elemVector(),hd, size, cudaMemcpyDeviceToHost);
     checkCUDAError("memory of h not transferred");
  /*-- --*/
}
```

(see file SWE_BlockCUDA.cu)

CUDA: Set Ghost Layer

Implementation in SWE_Block::setGhostLayer():

1. call setBoundaryConditions()
   → set simple, block-local boundary conditions ("real boundaries")
2. transfer data between ghost and copy layers
   → to be discussed in more detail (later)

```
void SWE_BlockCUDA::setBoundaryConditions() {
  /*-- some code skipped --*/
  if (boundary[BND_LEFT] == PASSIVE || /*-- --*/) {
     /*-- --*/
  }
  else {
     dim3 dimBlock(1,TILE_SIZE);
     dim3 dimGrid(1,ny/TILE_SIZE);
     kernelLeftBoundary<<<dimGrid,dimBlock>>>(
        hd,hud,hvd,nx,ny,boundary[BND_LEFT]);
  };
```

(see file SWE_BlockCUDA.cu)

CUDA: Set (Simple) Boundary Conditions

```
__global__
void kernelLeftBoundary(float* hd, float* hud, float* hvd,
                        int nx, int ny, BoundaryType bound)
{
  // determine j coordinate of current ghost cell:
  int j = 1 + TILE_SIZE*blockIdx.y + threadIdx.y;
  // determine position of ghost and copy cell in array:
  int ghost = getCellCoord(0,j,ny);
  int inner = getCellCoord(1,j,ny);

  // consider only WALL & OUTFLOW boundary conditions:
  hd[ghost] = hd[inner];
  hud[ghost] = (bound==WALL) ? -hud[inner] : hud[inner];
  hvd[ghost] = hvd[inner];
}
```

(in file SWE_BlockCUDA_kernels.cu)

CUDA Parallelization – Roadmap

Goal: "run everything on the GPU"

⇒ **functions and kernels to implement:**

- compute fluxes on each edge:
  → **splash.computeNumericalFluxes();**

```
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(nx/TILE_SIZE,ny/TILE_SIZE);
computeNetUpdatesKernel<<<dimGrid,dimBlock>>>(
   hd, hud, hvd, bd, /* ... */, nx,ny);
```

- update unknowns in each cell:
  → **splash.updateUnknowns(dt);**

```
dim3 dimBlock(TILE_SIZE, TILE_SIZE);
dim3 dimGrid(nx/TILE_SIZE, ny/TILE_SIZE);
updateUnknownsKernel<<<dimGrid,dimBlock>>>(
   hd, hud, hvd, /* ... */, nx, ny, dt, 1.0f/dx, 1.0f/dy);
```

Part III

Optimization of the SWE-CUDA Kernels

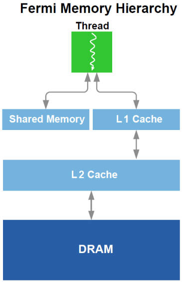


image: NVIDIA

SWE-CUDA – Memory-Bound Performance

A performance estimate for SWE:

- assumption: performance is **memory-bound**
- presentation laptop has a bandwidth (GPU main memory) of 11.2 GB/s
- what is the possible performance of the SWE code?

Memory transfer in SWE:

- consider mesh of size 1024×1024, thus 1 M<cells>
- variables h, hu, hv, b: 4×4 bytes, thus 16 MB
- net updates: 4×4 bytes per edge, thus 32 MB
- how many read & write accesses in each kernel?

SWE-CUDA – Memory-Bound Performance (2)

Memory accesses in computeNetUpdates:

- read variables h, hu, hv, b: 16 MB
- write netUpdates: 32 MB

Memory accesses in updateUnknowns:

- read netUpdates: 32 MB
- write variables h, hu, hv: 12 MB

Total memory transfer:

- neglect computation of maximum wave speed
- read 48 MB, write 44 MB per time step
- 11.2 GB/s $\approx$ 120 time steps per sec.?

SWE-CUDA – Memory-Bound Performance (3)

Road blocks for memory-bound performance:

- assumed that each kernels reads/writes any piece of data only once
- currently not the case for read accesses

Read accesses in computeNetUpdates:

- each cell reads h, hu, hv, b from left/bottom and right/top cell
  $\rightarrow$ doubles number of read accesses
- kernel is called twice (left/right and bottom/top updates)
  $\rightarrow$ doubles number of read accesses
- new value: read 192 MB, write 44 MB per time step
  $\rightarrow$ 11.2 GB/s $\approx$ 60 time steps per sec.?

Read accesses in updateUnknowns:

- actually no extra read or write accesses

CUDA Parallelization – Level 2

Optimization of kernels:

- coalesced access to GPU memory
- use of shared memory and registers

```
__shared__ float Fds[TILE_SIZE+1][TILE_SIZE+1];
__shared__ float Gds[TILE_SIZE+1][TILE_SIZE+1];
/* ... */
int iEdge = getEdgeCoord(i,j,ny);   // index of right/top Edge
Fds[tx+1][ty] = Fhd[iEdge];
Gds[tx][ty+1] = Ghd[iEdge];
/* ... */
 h = hd[iElem] - dt *( (Fds[tx+1][ty]-Fds[tx][ty])*dxi
                      +(Gds[tx][ty+1]-Gds[tx][ty])*dyi );
```

(in file SWE_RusanovBlockCUDA_kernels.cu)

Maximum Wave Speeds

Parallel Reduction Revisited

Computation of “Net Updates”:

- kernel computes wave speeds for every edge/cell
- also required to compute the CFL condition
  $\rightarrow$ parallel maximum computation required

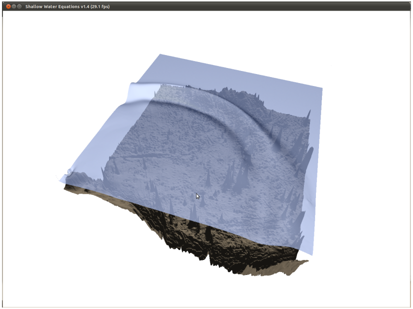
Optimization approach:

- keep wave speeds in shared memory
- compute maximum wave speed of a tile in shared memory
- subsequent parallel reduction only on tile-maxima

Some Aspects of CUDA Parallelization

Level 3: more advanced optimizations

- “kernel fusion”: merge computation of fluxes with updates of unknowns
- merge maximum-reduction on wave speeds (for CFL condition) with flux computation (or update of velocities)
- allows interactive/“real-time” simulation (800$\times$800 cells)



Net Updates and Updating Unkowns

Parallel Programming Patterns Revisited

Anticipate new parallel program:

For each cell in parallel(!) compute:

1. net updates for all edges (vertical & horizontal)
2. update cell unknowns from net updates
   write to next-timestep copies of h, hu, hv!

Parallel access to memory:

1. concurrent read to h, hu, hv; exclusive write to net updates
2. concurrent read to net updates; exclusive read to h, hu, hv

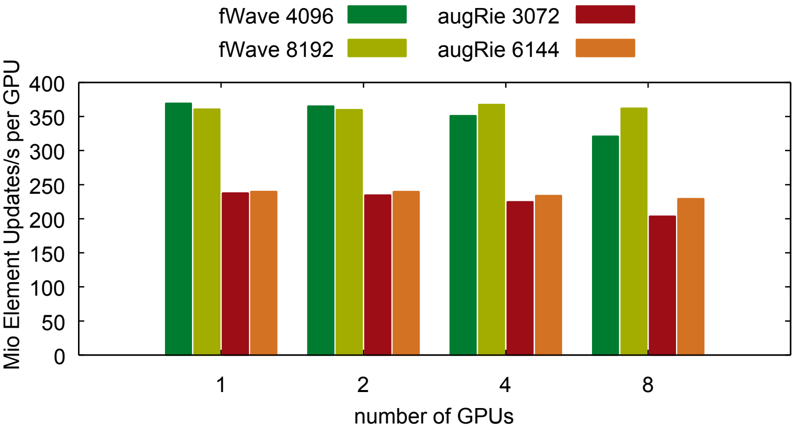
$\Rightarrow$ 2 after 1 for all cells, so everything is fine?

$\Rightarrow$ **unfortunately not!** (consider CUDA blocks, warps, etc.)

$\Rightarrow$ **may be cured:** old/new copy for h, hu, hv

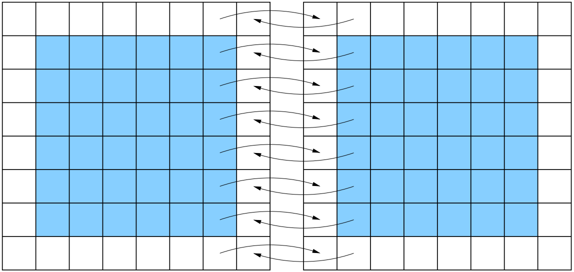
Performance on a Small GPU Cluster

(8 NVIDIA Tesla M2090 GPUs using MPI+CUDA)

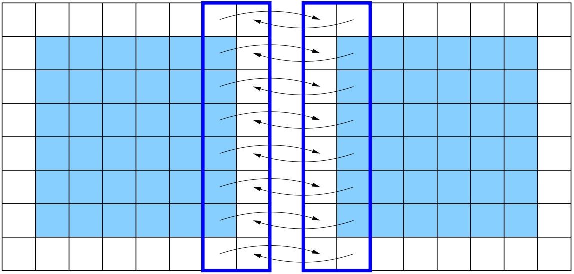


Part IV

Parallelization on Hybrid Architectures



Exchange of Values in Ghost/Copy Layers



Remember Proxy Objects:

- grabGhostLayer() to write into ghost layer
- registerCopyLayer() to read from copy layer
- return proxy object (class SWE_Block1D) that references one row/column of the grid

SWE_BlockCUDA – Update of Ghost Layers

Memory-Synchronization Revisited

- ghost layers might be updated in each time step
  → conditions PASSIVE, CONNECT
- updated ghost layers in CPU memory need to be copied to GPU

```
void SWE_BlockCUDA::synchGhostLayerAfterWrite() {
  if  (boundary[BND_LEFT] == PASSIVE ||
       boundary[BND_LEFT] == CONNECT) {
     // transfer h, hu, hv from left ghost layer to resp. device memo
     cudaMemcpy(hd, h[0], (ny+2)*sizeof(float),
                          cudaMemcpyHostToDevice);
     /*-- same for hud/hu and hvd/hv --*/
  };
```

(in file SWE_BlockCUDA.cu)

SWE_BlockCUDA – Update of Copy Layers

Memory-Synchronization Revisited

- copy layers need to be updated in each time step
  → conditions PASSIVE, CONNECT
- requires transfer from GPU to CPU memory

```
void SWE_BlockCUDA::synchCopyLayerBeforeRead() {
  /*-- left and right copy layer skipped --*/
  int size = 3*(nx+2);
  // bottom copy layer
  if (... || boundary[BND_BOTTOM] == CONNECT) {
     dim3 dimBlock(TILE_SIZE,1);
     dim3 dimGrid(nx/TILE_SIZE,1);
     kernelBottomCopyLayer<<<dimGrid,dimBlock>>>(
         hd,hud,hvd,bottomLayerDevice+size,nx,ny);
     cudaMemcpy(bottomLayer+size, bottomLayerDevice+size,
                    size*sizeof(float), cudaMemcpyDeviceToHost);
  };
  /*-- ... --*/
```

(in file SWE_BlockCUDA.cu)

MPI Parallelization – Exchange of Ghost/Copy Layers

```
SWE_Block1D* leftInflow = splash.grabGhostLayer(BND_LEFT);
SWE_Block1D* leftOutflow = splash.registerCopyLayer(BND_LEFT);

SWE_Block1D* rightInflow = splash.grabGhostLayer(BND_RIGHT);
SWE_Block1D* rightOutflow = splash.registerCopyLayer(BND_RIGHT);

MPI_Sendrecv(leftOutflow->h.elemVector(), 1, MPI_COL, leftRank, 1,
             rightInflow ->h.elemVector(), 1, MPI_COL, rightRank, 1,
             MPI_COMM_WORLD,&status);

MPI_Sendrecv(rightOutflow->h.elemVector(), 1, MPI_COL, rightRank,4,
             leftInflow ->h.elemVector(), 1, MPI_COL, leftRank, 4,
             MPI_COMM_WORLD,&status);
```

(cmp. file examples/swe_mpi.cpp)

Teaching Parallel Programming with SWE

SWE in Lectures, Tutorials, Lab Courses:

- non-trivial example, but model & implementation easy to grasp
- allows different parallel programming models
  (MPI, OpenMP, CUDA, Intel TBB/ArBB, OpenCL, ...)
- prepared for hybrid parallelisation

Some Extensions:

- ASAGI - parallel server for geoinformation
  (S. Rettenberger, Master's thesis)
- OpenGL real-time visualisation of results
  (T. Schnabel, student project)

→ **http://www5.in.tum.de/SWE/** → **https://github.com/TUM-I5**