Gene Golub SIAM Summer School 2012 – Simulation and Supercomputing in the Geosciences

**SWE – An Education-Oriented Code to Solve the Shallow Water Equations**

Michael Bader, Alexander Breuer

Technische Universität München
Starting Point: Lecture on Parallel Programming

- classical approaches for shared & distributed memory: OpenMP and MPI
- “something more fancy” → GPU computing (CUDA, e.g.)
- motivating example to teach different models and compare their properties

“Motivating Example”:

- not just Jacobi or Gauß-Seidel
- not the heat equation again . . .
- inspired by a CFD code: “Nast” by Griebel et al.
- turned out to become shallow water equations
- and then there was: G2S3!
Towards Tsunami Simulation with SWE

Shallow Water Code – Summary

- Finite Volume discretization on regular Cartesian grids → simple numerics (but can be extended to state-of-the-art)
- patch-based approach with ghost cells for communication → wide-spread design pattern for parallelization
Shallow Water Code – Bells & Whistles

- included augmented Riemann solvers
  → allows to simulate inundation
  (George, 2008; Bale, LeVeque, et al., 2002)
- developed towards hybrid parallel architectures
  → now runs on GPU cluster
Part I

Model and Discretization
Model: The Shallow Water Equations

Simplified setting (no friction, no viscosity, no coriolis forces, etc.):

\[
\begin{bmatrix}
  h \\
  hu \\
  hv
\end{bmatrix}_t + \begin{bmatrix}
  hu \\
  hu^2 + \frac{1}{2}gh^2 \\
  huv
\end{bmatrix}_x + \begin{bmatrix}
  hv \\
  huv \\
  hv^2 + \frac{1}{2}gh^2
\end{bmatrix}_y = 0.
\]

Finite Volume Discretization:

- generalized 2D hyperbolic PDE: \( q = (h, hu, hv)^T \)
  
  \[
  \frac{\partial}{\partial t} q + \frac{\partial}{\partial x} F(q) + \frac{\partial}{\partial y} G(q) = 0
  \]

- wave propagation form:
  
  \[
  Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right) \\
  - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^n + B^- \Delta Q_{i,j+1/2}^n \right).
  \]
Model: The Shallow Water Equations

Simplified setting (no friction, no viscosity, no coriolis forces, etc.):

\[
\begin{bmatrix}
    h \\
    hu \\
    hv
\end{bmatrix}_t + \begin{bmatrix}
    hu \\
    hu^2 + \frac{1}{2}gh^2 \\
    huv
\end{bmatrix}_x + \begin{bmatrix}
    hv \\
    huv \\
    hv^2 + \frac{1}{2}gh^2
\end{bmatrix}_y = 0.
\]

Flux Computation on Edges:

- wave propagation form:
  \[
  Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right) \\
  - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^n + B^- \Delta Q_{i,j+1/2}^n \right).
  \]

- simple fluxes: Rusanov/(local) Lax-Friedrich
- more advanced: f-Wave or (augmented) Riemann solvers (George, 2008; LeVeque, 2011), no limiters
Finite Volume Discretization

Unknowns and Numerical Fluxes:

- unknowns $h$, $hu$, $hv$, and $b$ located in cell centers
- two sets of “net updates”/numerical fluxes per edge: $A^+ \Delta Q_{i-1/2,j}$, $B^- \Delta Q_{i,j+1/2}$, etc.
Patches of Cartesian Grid Blocks

Spatial Discretization:

- regular Cartesian meshes; allow multiple patches
- ghost and copy layers to implement boundary conditions, for more complicated domains, and for parallelization
Part II

Implementation
Main Loop – Euler Time-stepping

```java
while (t < ... ) {
    // set boundary conditions
    splash.setGhostLayer();

    // compute fluxes on each edge
    splash.computeNumericalFluxes();

    // set largest allowed time step:
    dt = splash.getMaxTimestep();
    t += dt;

    // update unknowns in each cell
    splash.updateUnknowns(dt);
}
```

→ defines interface for abstract class SWE_Block
Set Ghost Layers – Boundary Conditions

Split into two methods:

- `setGhostLayer()`: interface function in SWE Block, needs to be called by main loop
- `setBoundaryConditions()`: called by `setGhostLayer()`; sets "real" boundary conditions (WALL, OUTFLOW, etc.)

```c
switch(boundary[BND_LEFT]) {
  case WALL:
    {
      for (int j=1; j<=ny; j++) {
        h[0][j] = h[1][j];
        b[0][j] = b[1][j];
        hu[0][j] = -hu[1][j];
        hv[0][j] = hv[1][j];
      };
      break;
    }
  case OUTFLOW:
    { /* ... */
```

(cmp. file SWE_Block.cpp)
Compute Numerical Fluxes

main loop to compute net updates on left/right edges:

```cpp
for (int i=1; i < nx+2; i++) {
    for (int j=1; j < ny+1; j++) {
        float maxEdgeSpeed;
        wavePropagationSolver.computeNetUpdates(
            h[i-1][j], h[i][j],
            hu[i-1][j], hu[i][j],
            b[i-1][j], b[i][j],
            hNetUpdatesLeft[i-1][j-1], hNetUpdatesRight[i-1][j-1],
            huNetUpdatesLeft[i-1][j-1], huNetUpdatesRight[i-1][j-1],
            maxEdgeSpeed
        );
        maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
    }
}
```

(cmp. file SWE_WavePropagationBlock.cpp)
main loop to compute net updates on top/bottom edges:

```cpp
for(int i=1; i < nx+1; i++) {
    for(int j=1; j < ny+2; j++) {
        float maxEdgeSpeed;
        wavePropagationSolver.computeNetUpdates(
            h[i][j-1], h[i][j],
            hv[i][j-1], hv[i][j],
            b[i][j-1], b[i][j],
            hNetUpdatesBelow[i-1][j-1], hNetUpdatesAbove[i-1][j-1],
            hvNetUpdatesBelow[i-1][j-1], hvNetUpdatesAbove[i-1][j-1],
            maxEdgeSpeed
        );
        maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
    }
}
```

(cmp. file SWE_WavePropagationBlock.cpp)
Determine Maximum Time Step

- variable `maxWaveSpeed` holds maximum wave speed
- updated during computation of numerical fluxes in method `computeNumericalFluxes()`:
  
  ```cpp
  maxTimestep = std::min( dx/maxWaveSpeed, dy/maxWaveSpeed );
  ```

- simple “getter” method defined in class `SWE::Block`:
  ```cpp
  float getMaxTimestep() { return maxTimestep; }
  ```

- hence: `getMaxTimestep()` for current time step should be called after `computeNumericalFluxes()`
Update Unknowns – Euler Time Stepping

```c
for (int i=1; i < nx+1; i++) {
    for (int j=1; j < ny+1; j++) {
        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]);
        hv[i][j] -= dt/dy * (hvNetUpdatesAbove[i-1][j-1]
                            + hvNetUpdatesBelow[i-1][j]);
        /* ... */
    }
}
```

(cmp. file SWE_WavePropagationBlock.cpp)
Goals for (Parallel) Implementation

Spatial Discretization:
- allow different parallel programming models
- and also to switch between different numerical models
⇒ class hierarchy of numerical vs. programming models

Hybrid Parallelization:
- support two levels of parallelization
- coarse-grain parallelism across Cartesian grid patches
- fine-grain parallelism on patch-local loops
⇒ separate fine-grain and coarse-grain parallelism (plug&play principle)
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()`, ...
  → methods relevant for parallelization
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(), updateUnkowns(), ...
  → 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 parameters).
    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 III

Parallel Programming Patterns
Computing the Net Updates

Parallel Programming Patterns

• compute net updates on left/right edges:

```c
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:

```c
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)

M. Bader, A. Breuer: SWE – An Education-Oriented Code to Solve the Shallow Water Equations
Gene Golub SIAM Summer School 2012 – Simulation and Supercomputing in the Geosciences,
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:

```c
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 IV

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:

```c
int size = (nx+2)*(ny+2)*sizeof(float);
// allocate CUDA memory for unknowns 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:

```cpp
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:

```c
void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if ( cudaSuccess != err )
    {
        fprintf ( stderr , "Cuda 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:

```cpp
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

```c
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:

```cpp
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

```c
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 Parallelization – Afternoon Session 1

Goal: “run everything on the GPU” → remember main loop:

```c
while( t < ... ) {
    // set boundary conditions
    splash.setGhostLayer();

    // compute fluxes on each edge
    splash.computeNumericalFluxes();

    // set largest allowed time step:
    dt = splash.getMaxTimestep();
    t += dt;

    // update unknowns in each cell
    splash.updateUnknowns(dt);
}
```
**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)

```cpp
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

```c
__global__
void kernelLeftBoundary(float* hd, float* hud, float* hvd,
                        int nx, int ny, BoundaryType bound)
{
    int j = 1 + TILE_SIZE*blockIdx.y + threadIdx.y;
    // determine j coordinate of current ghost cell:
    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)
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);
    ```
CUDA Parallelization – Afternoon Session 1

Roadmap:

1. replace example solution by code template:
   → in SWE “home” directory:

   ```
   cp src_skeleton/* src/
   ```

2. two CUDA kernels to be implemented
   in the file SWE_WavePropagationBlockCuda_kernels.cu:

   ```
   void computeNetUpdatesKernel([...])
   void updateUnkownsKernel([...])
   ```

3. hint: first use the following pattern:
   → transfer variables h, hu, hv to GPU memory
   → call to CUDA kernel
   → transfer updated variables back to CPU memory
CUDA Parallelization – Afternoon Session 1

Goal: “run really(?) everything on the GPU”

- focus on computation of net updates and Euler time step, first
- missing: set largest allowed time step
  → `splash.getMaxTimestep();`
- requires computation of a maximum/minimum (CFL condition: maximum wave speed required)
  → best done in kernel for net updates
- will be left for session 2 (or even 3)
  → use fixed time step until then . . .

```c
// update unknowns in each cell
splash.updateUnknowns(dt);

→ set dt to some good value
→ or trust method computeMaxTimestep() in class SWE_Block
```
Part V

Optimization of the SWE-CUDA Kernels

Fermi Memory Hierarchy

image: NVIDIA
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 $1024 \times 1024$, thus $1 \text{M}\text{<cells>}
- variables $h$, $hu$, $hv$, $b$: $4 \times 4$ bytes, thus $16 \text{MB}$
- net updates: $4 \times 4$ bytes per edge, thus $32 \text{MB}$
- how many read & write accesses in each kernel?
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
- $11.2 \text{ GB/s} \approx 120$ 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
  \( \rightarrow 11.2 \text{ GB/s} \approx 60 \text{ time steps per sec.} \)

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

```c
__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
  → 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 (800×800 cells)
Net Updates and Updating Unknowns
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
   \textit{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 \textbf{unfortunately not!} (consider CUDA blocks, warps, etc.)
   \Rightarrow \textbf{may be cured:} old/new copy for h, hu, hv
Performance Contest

SWE on a Tesla C2070 (mathgpu)

- 448 stream processing units
- memory bandwidth: 97.6 GB/s (acc. to bandwidth test)
- theoretical peak performance: $\approx 1$ TFlop/s

How much do you get?

- in terms of memory throughput?
- in terms of Flop/s?
- in terms of processed cells per second?
Part VI

Parallelization on Hybrid Architectures
Exchange of Values in Ghost/Copy Layers

Straightforward Approach:

- boundary conditions OUTFLOW, WALL vs. CONNECT or PARALLEL
- disadvantage: method setGhostLayer() needs to be implemented for each derived class
Exchange of Values in Ghost/Copy Layers (2)

Implemented via 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
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 memory
        cudaMemcpy(hd, h[0], (ny+2)*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(hud, hud[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

```cpp
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

```c
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
References/Literature

