The Shallow Water Equations and CUDA

Oliver Meister

November 28th 2012
Last Tutorial

Optimizations

- coalesced memory access
- loop unrolling
- thread granularity

Performance Measurements

- runtime
- occupancy
- warp serialize

What can still be done?

- Binary Fan-In for reduction
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
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
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
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
Towards Tsunami Simulation with SWE (2)

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
Towards Tsunami Simulation with SWE (2)

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
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
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
Model and Discretization
Model: The Shallow Water Equations

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

\[
\begin{bmatrix}
    h \\
    hu \\
    hv \\
\end{bmatrix}
\begin{bmatrix}
    hu \\
    hu^2 + \frac{1}{2}gh^2 \\
    huv \\
\end{bmatrix}
\begin{bmatrix}
    hv \\
    huv \\
    hv^2 + \frac{1}{2}gh^2 \\
\end{bmatrix}
\begin{bmatrix}
    0 \\
    -ghb_x \\
    -ghb_y \\
\end{bmatrix}
\]

Finite Volume Discretization:

- generalized 2D hyperbolic PDE: \( q = (h, hu, hv)^T \)

\[
\frac{\partial}{\partial t} q + \frac{\partial}{\partial x} F(q) + \frac{\partial}{\partial y} G(q) = s(q)
\]

- wave propagation form:

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

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

\[
\begin{bmatrix}
h \\
h u \\
h v \\
\end{bmatrix}_t + \begin{bmatrix}
h u \\
h u^2 + \frac{1}{2}gh^2 \\
h u v \\
\end{bmatrix}_x + \begin{bmatrix}
h v \\
h u v \\
h v^2 + \frac{1}{2}gh^2 \\
\end{bmatrix}_y = \begin{bmatrix}
0 \\
-ghb_x \\
-ghb_y \\
\end{bmatrix}
\]

Flux Computation on Edges:

- wave propagation form:

\[
Q^{n+1}_{i,j} = Q^n_{i,j} - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q^1_{i-1/2,j} + A^- \Delta Q^n_{i+1/2,j} \right) - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q^1_{i,j-1/2} + B^- \Delta Q^n_{i,j+1/2} \right).
\]

- 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
Teil II

Implementation
Main Loop – Euler Time-stepping

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

    // compute fluxes on each edge
    computeNumericalFluxes();

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

    // update unknowns in each cell
    updateUnknowns(dt);
}
```
Main Loop – Euler Time-stepping

\[
\text{while( } t < \ldots \text{ ) \{} \\
\quad // set boundary conditions \\
\quad \text{setGhostLayer();} \\
\quad \\
\quad // compute fluxes on each edge \\
\quad \text{computeNumericalFluxes();} \\
\quad \\
\quad // set largest allowed time step: \\
\quad dt = \text{getMaxTimestep();} \\
\quad t += dt; \\
\quad \\
\quad // update unknowns in each cell \\
\quad \text{updateUnknowns(dt);} \\
\text{\}};
\]

\[\rightarrow \text{defines interface for abstract class SWE\_Block}\]
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()`:
  \[
  \text{maxTimestep} = \text{std::min}( \text{dx}/\text{maxWaveSpeed}, \text{dy}/\text{maxWaveSpeed} );
  \]
- 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
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(), updateUnkowns(),...
  ```
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.
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
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).
    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)
Teil 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)
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
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!
Teil 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)
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 unknowns 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, "\nCuda error (%s): %s.\n", msg, cudaGetErrorString( err) );
        exit(-1);
    }
}
```

(see file SWE_BlockCUDA.cu)
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`)
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

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

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

    // compute fluxes on each edge
    computeNumericalFluxes();

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

    // update unknowns in each cell
    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)

```c
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)
{
    // 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)
Assignment

Functions and kernels to be implemented:

In the file SWE_WavePropagationBlockCuda.cu

- compute fluxes on each edge:
  \[ \text{computeNumericalFluxes();} \]
  
  \[ \text{dim3 dimBlock(TILE\_SIZE, TILE\_SIZE);} \]
  \[ \text{dim3 dimGrid(nx/TILE\_SIZE, ny/TILE\_SIZE);} \]
  \[ \text{computeNetUpdatesKernel<<<dimGrid, dimBlock>>>(} \]
  \[ \text{hd, hud, hvd, bd, /* ... */}, nx, ny); \]

- update unknowns in each cell:
  \[ \text{updateUnknowns(dt);} \]
  
  \[ \text{dim3 dimBlock(TILE\_SIZE, TILE\_SIZE);} \]
  \[ \text{dim3 dimGrid(nx/TILE\_SIZE, ny/TILE\_SIZE);} \]
  \[ \text{updateUnknownsKernel<<<dimGrid, dimBlock>>>(} \]
  \[ \text{hd, hud, hvd, /*...*/}, nx, ny, dt, 1.0f/dx, 1.0f/dy); \]
Assignment

In the file `SWE_WavePropagationBlockCuda_kernels.cu`

- CUDA kernels to be implemented:

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

- **important**: Refer to the F-Wave solver CPU implementation `SWE_WavePropagationBlock.cpp` for the kernels, the task is not to implement a completely new solver, but to migrate a C++ version to CUDA instead.

- **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
Assignment

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

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

    // update unknowns in each cell
    updateUnknowns(dt);

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

1. Write the missing initialization and kernel calls in SWE_WavePropagationBlockCuda.cu.

2. Port the F-Wave solver kernels computeNetUpdatesKernel and updateUnknownsKernel from the C++ version in SWE_WavePropagationBlock.cpp to a CUDA version in SWE_WavePropagationBlockCuda_kernels.cu.
Assignment

Installation instructions:

- Install required libraries (if necessary).
  Ubuntu: `sudo apt-get install libxi-dev libxmu-dev scons`
- Download and extract exercise files
- To compile, change to the SWE directory and type `scons`
- If CUDA is not found, you can set the CUDA folder manually in `./SWE_gnu_cuda.py` and call `scons buildVariablesFile=./SWE_gnu_cuda.py` instead.
- You can use build variable files to modify other parameters as well, for example to enable OpenGL output. Look in `./build/options` for a few examples.
- The executable is generated in `./build`.