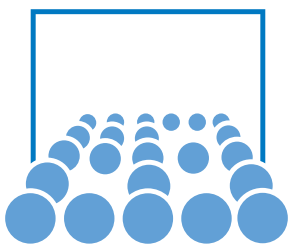
Further topics on SWE and CUDA

Oliver Meister

January 22nd 2014



Last Tutorial

The Shallow Water Equations

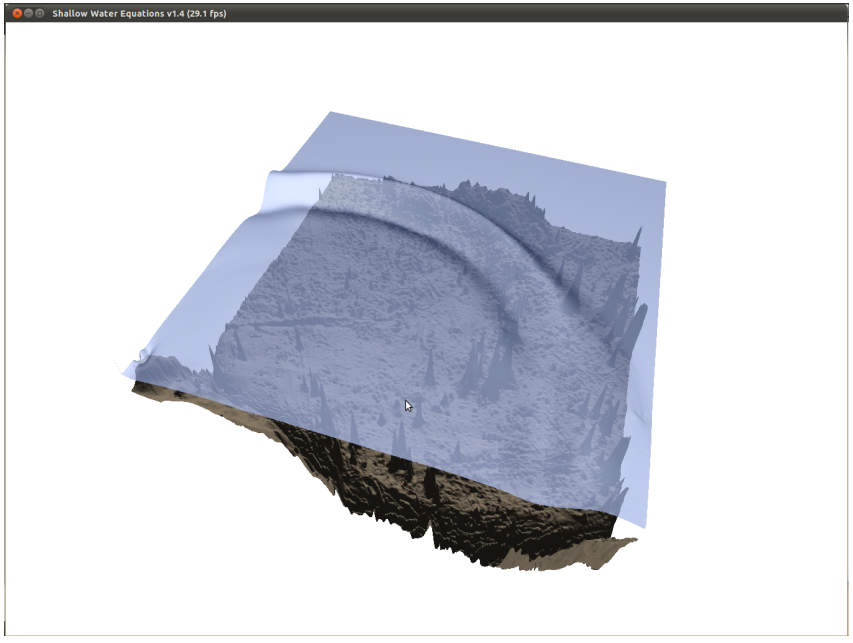
- hyperbolic equation
- Finite Volume discretization
- Cartesian grid partitioned into blocks with ghost layers

SWE Code

- Euler time step:
  - set boundary conditions
  - compute net updates
  - set time step size
  - update cell unknowns
- parallelization concepts

Teil I

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

$\Rightarrow$ 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 Parallelization

Goal: "run everything on the GPU" $\rightarrow$ remember main loop:

```
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)

```
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: Cell update kernel

Task: Implement the net updates kernel in `src/SWE_WavePropagationBlockCuda_kernels.cu`

a) For the vertical edge on the left of each cell and the horizontal edge on the bottom of each cell:

- **i)** Call `fWaveComputeNetUpdates` and read the correct arguments from the data arrays.
- **ii)** Write the net updates back to the correct positions in the global net update arrays.
- **iii)** Update the maximum wave speed.

You can refer to the C++ version of the kernel in `src/SWE_WavePropagationBlock.cpp`.

Assignment 1 - Case Study

i) What is a flux solver? Which data does it operate on?
A flux solver computes *net updates* on edges from neighboring cell data. It reads state vectors $\mathbf{q}_i$ and $\mathbf{q}_{i+1}$ from both cells and produces update vectors $\delta\mathbf{q}_{i,i+1}$ and $\delta\mathbf{q}_{i+1,i}$ for both cells.

ii) What are the source terms in the shallow water equations?
The bathymetry term and boundary conditions

iii) What components does a block consist of and how do they differ?

- Inner layer (computes on local data only)
- Copy layer (requires data from ghost layer, copied to neighbor blocks)
- Ghost layer (required by copy layer, copied from neighbor blocks or set by boundary condition)

iv) What data dependency exists between `computeFluxes()` and `eulerTimestep()`? - The net updates and time step size

Assignment 3a - net updates kernel

computeNetUpdatesKernel vertical edges:

```
lpos = computeOneDPositionKernel(i-1, j, i_nY+2);
rpos = computeOneDPositionKernel(i, j, i_nY+2);

// compute the net-updates
fWaveComputeNetUpdates( 9.81, i_h[lpos], i_h[rpos],
  i_hu[lpos], i_hu[rpos], i_b[lpos], i_b[rpos], l_netUpdates);

// compute the location of the net-updates
updPos = computeOneDPositionKernel(i-1, j, i_nY+1);

// store the horizontal net-updates
o_hNetUpdatesLeftD[updPos] = l_netUpdates[0];
o_hNetUpdatesRightD[updPos] = l_netUpdates[1];
o_huNetUpdatesLeftD[updPos] = l_netUpdates[2];
o_huNetUpdatesRightD[updPos] = l_netUpdates[3];
```

Assignment 3a - net updates kernel

computeNetUpdatesKernel horizontal edges:

```
bpos = computeOneDPositionKernel(i, j-1, i_nY+2);
apos = computeOneDPositionKernel(i, j, i_nY+2);

// compute the net-updates
fWaveComputeNetUpdates( 9.81, i_h[bpos], i_h[apos],
  i_hv[bpos], i_hv[apos], i_b[bpos], i_b[apos], l_netUpdates);

// compute the location of the net-updates
updPos = computeOneDPositionKernel(i, j-1, i_nY+1);

// store the vertical net-updates
o_hNetUpdatesBelowD[updPos] = l_netUpdates[0];
o_hNetUpdatesAboveD[updPos] = l_netUpdates[1];
o_hvNetUpdatesBelowD[updPos] = l_netUpdates[2];
o_hvNetUpdatesAboveD[updPos] = l_netUpdates[3];
```

Assignment 3b - visual output

Let's see what the output looks like! :)

Teil II

Optimization of the SWE-CUDA Kernels

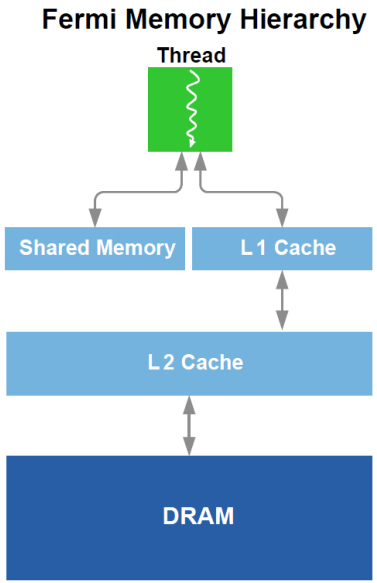


image: NVIDIA

SWE-CUDA – Memory-Bound Performance

A performance estimate for SWE:

- assumption: performance is **memory-bound**
- presentation laptop has a bandwidth (GPU main memory) of 14.4 GB/s
- what is the best possible performance of the SWE code?

Memory transfer in SWE:

- consider mesh of size 256×256, thus 65.6 k cells
- variables h, hu, hv, b: 4×4 bytes per cell, thus 1 MiB
- net updates: $2 \times 2 \times 4$ bytes per edge, thus 2 MiB
- how many read & write accesses in each kernel?

SWE-CUDA – Memory-Bound Performance (2)

Memory accesses in computeNetUpdates:

- read variables h, hu, hv, b: 1 MiB
- write netUpdates: 2 MiB

Memory accesses in updateUnknowns:

- read netUpdates: 2 MiB
- write variables h, hu, hv: 786 kiB

Total memory transfer:

- neglect computation of maximum wave speed
- read 3 MiB, write 2.75 MiB per time step
- Estimated timesteps per second: $14.4\,\text{GB/s} \div 3\,\text{MiB} \approx 4800\frac{1}{s}$
- Measured timesteps per second: $250\frac{1}{s}$

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 kernel reads h, hu, hv, b from 3 cells
  $\rightarrow$ triples number of read accesses
- new value: read 5 MiB, write 2.75 MiB per time step
  $\rightarrow$ 14.4 GB/s $\div$ 5 MiB $\approx$ 2900 time steps per sec.?

Read accesses in updateUnknowns:

- actually no extra read or write accesses

Optimizations

Task: Think of possible optimizations for the CUDA kernels

1. How could memory access in the kernels be improved?

2. Is there a way to make the computation of the maximum wave speed faster?

3. What else could be done?

Optimizations

Task: Think of possible optimizations for the CUDA kernels

1. How could memory access in the kernels be improved?
   Store cell data in shared memory, reassign threads for coalesced access
2. Is there a way to make the computation of the maximum wave speed faster?

3. What else could be done?

Optimizations

Task: Think of possible optimizations for the CUDA kernels

1. How could memory access in the kernels be improved?
   Store cell data in shared memory, reassign threads for coalesced access
2. Is there a way to make the computation of the maximum wave speed faster?
   Use a binary fan-in for the reduction
3. What else could be done?

Optimizations

Task: Think of possible optimizations for the CUDA kernels

1. How could memory access in the kernels be improved?
   Store cell data in shared memory, reassign threads for coalesced access
2. Is there a way to make the computation of the maximum wave speed faster?
   Use a binary fan-in for the reduction
3. What else could be done?
   kernel fusion, loop unrolling, ...

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 Edg
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

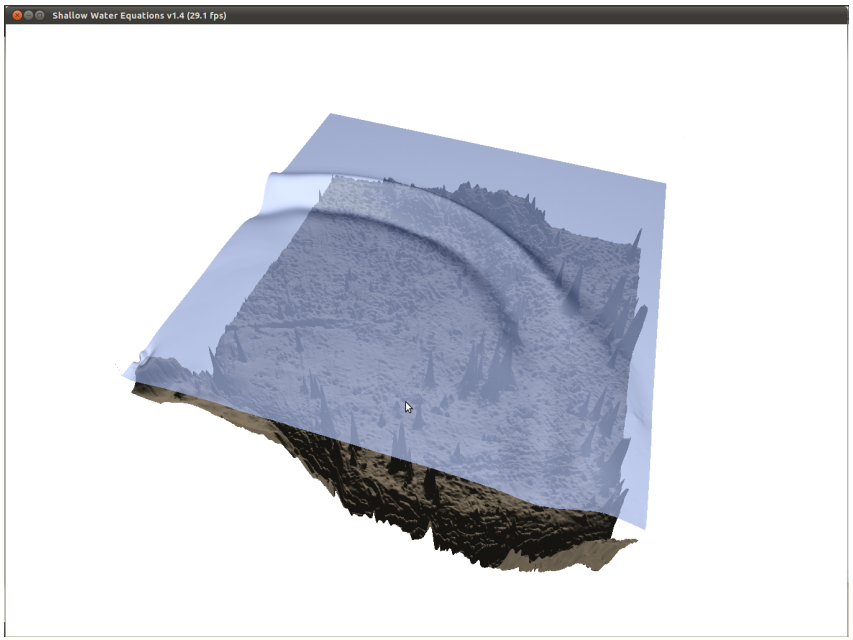
Optimization approach:

- 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 (800×800 cells)



Net Updates and Updating Unkowns

Parallel Programming Patterns Revisited

Idea of kernel fusion:

Compute for each cell in parallel:

1. net updates for all edges (vertical & horizontal)
2. update cell unknowns from net updates

Parallel access to memory:

1. concurrent read to h, hu, hv; exclusive write to net updates (now located only in shared memory!)
2. concurrent read to net updates; exclusive write to h, hu, hv

$\Rightarrow$ execute 1, synchronize, and then execute 2 – should work, right?

Net Updates and Updating Unkowns

Parallel Programming Patterns Revisited

Idea of kernel fusion:

Compute for each cell in parallel:

1. net updates for all edges (vertical & horizontal)
2. update cell unknowns from net updates

Parallel access to memory:

1. concurrent read to h, hu, hv; exclusive write to net updates (now located only in shared memory!)
2. concurrent read to net updates; exclusive write to h, hu, hv

$\Rightarrow$ execute 1, synchronize, and then execute 2 – should work, right?

$\Rightarrow$ **unfortunately not!** (no synchronization between blocks)

Net Updates and Updating Unkowns

Parallel Programming Patterns Revisited

Idea of kernel fusion:

Compute for each cell in parallel:

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
   (now located only in shared memory!)
2. concurrent read to net updates; exclusive write to h, hu, hv

$\Rightarrow$ execute 1, synchronize, and then execute 2 – should work, right?

$\Rightarrow$ **unfortunately not!** (no synchronization between blocks)

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

References/Literature

- George, D. L. (2008), *Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation*. J. Comput. Phys. 227 (6), p. 3089–3113
- Bale, D. S. (2002), R. J. LeVeque, S. Mitran, and J. A. Rossmanith, *A wave-propagation method for conservation laws with spatially varying flux functions*. SIAM J. Sci. Comput. 24, p. 955–978.
- M. Bader (2012) and A. Breuer: *Teaching Parallel Programming Models on a Shallow-Water Code*. In: 11th Int. Symp. on Parall. and Dist. Computing (ISPDC 2012). IEEE Computer Society.