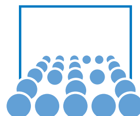
Further topics on SWE and CUDA

Oliver Meister

December 13th 2012



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
- CUDA: Synchronization, kernel calls, kernels

Assignment 1

computeNetUpdatesKernel main call:

```
dim3 dimBlock(TILE_SIZE, TILE_SIZE);
dim3 dimGrid(nx/TILE_SIZE, ny/TILE_SIZE);

computeNetUpdatesKernel<<<dimGrid,dimBlock>>>(
  hd, hud, hvd, bd,
  hNetUpdatesLeftD, hNetUpdatesRightD,
  huNetUpdatesLeftD, huNetUpdatesRightD,
  hNetUpdatesBelowD, hNetUpdatesAboveD,
  hvNetUpdatesBelowD, hvNetUpdatesAboveD,
  l_maximumWaveSpeedsD, nx,ny
);
```

Assignment 1

computeNetUpdatesKernel right edges call:

```
dim3 dimRightBlock(1, TILE_SIZE);
dim3 dimRightGrid(1, ny/TILE_SIZE);

computeNetUpdatesKernel<<<dimRightGrid, dimRightBlock>>>(
  hd, hud, hvd, bd,
  hNetUpdatesLeftD, hNetUpdatesRightD,
  huNetUpdatesLeftD, huNetUpdatesRightD,
  hNetUpdatesBelowD, hNetUpdatesAboveD,
  hvNetUpdatesBelowD, hvNetUpdatesAboveD,
  l_maximumWaveSpeedsD, nx, ny,
  nx, 0, dimGrid.x, 0
);
```

Assignment 1

computeNetUpdatesKernel top edges call:

```
dim3 dimTopBlock(TILE_SIZE, 1);
dim3 dimTopGrid(nx/TILE_SIZE, 1);

computeNetUpdatesKernel<<<dimTopGrid, dimTopBlock>>>(
  hd, hud, hvd, bd,
  hNetUpdatesLeftD, hNetUpdatesRightD,
  huNetUpdatesLeftD, huNetUpdatesRightD,
  hNetUpdatesBelowD, hNetUpdatesAboveD,
  hvNetUpdatesBelowD, hvNetUpdatesAboveD,
  l_maximumWaveSpeedsD, nx, ny,
  0, ny, 0, dimGrid.y
);
```

Assignment 2

computeNetUpdatesKernel:

```
i = i_offsetX + blockDim.x * blockIdx.x + threadIdx.x + 1;
j = i_offsetY + blockDim.y * blockIdx.y + threadIdx.y + 1;
```

Assignment 2

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 2

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 1

updateUnknownsKernel call:

```
dim3 dimBlock(TILE_SIZE,TILE_SIZE);
dim3 dimGrid(nx/TILE_SIZE,ny/TILE_SIZE);

// compute the update width.
float l_updateWidthX = i_deltaT / dx;
float l_updateWidthY = i_deltaT / dy;

// update the unknowns (global time step)
updateUnknownsKernel<<<dimGrid,dimBlock>>>(
  hNetUpdatesLeftD, hNetUpdatesRightD,
  huNetUpdatesLeftD, huNetUpdatesRightD,
  hNetUpdatesBelowD, hNetUpdatesAboveD,
  hvNetUpdatesBelowD, hvNetUpdatesAboveD,
  hd, hud, hvd, l_updateWidthX, l_updateWidthY,
  nx, ny);
```

Assignment 2

updateUnknownsKernel:

```
i = blockDim.x * blockIdx.x + threadIdx.x + 1;
j = blockDim.y * blockIdx.y + threadIdx.y + 1;

// compute the global cell position
cellPos = computeOneDPositionKernel(i, j, i_nY+2);
rPos = computeOneDPositionKernel(i-1, j, i_nY+1);
lPos = computeOneDPositionKernel(i, j, i_nY+1);
aPos = computeOneDPositionKernel(i, j-1, i_nY+1);
bPos = computeOneDPositionKernel(i, j, i_nY+1);

/* cont. */
```

Assignment 2

updateUnknownsKernel:

```
/* cont. */

//update the cell values: water height and momentum
io_h[cellPos] -= i_updateWidthX *
  (i_hNetUpdatesRightD[rPos] + i_hNetUpdatesLeftD[lPos])
   + i_updateWidthY *
  (i_hNetUpdatesAboveD[aPos] + i_hNetUpdatesBelowD[bPos]);

io_hu[cellPos] -=
  i_updateWidthX *
  (i_huNetUpdatesRightD[rPos] + i_huNetUpdatesLeftD[lPos]);

io_hv[cellPos] -=
  i_updateWidthY *
  (i_hvNetUpdatesAboveD[aPos] + i_hvNetUpdatesBelowD[bPos]);
```

Teil I

Optimization of the SWE-CUDA Kernels

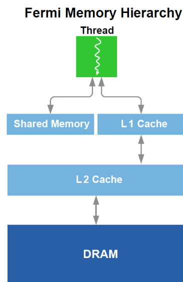


image: NVIDIA

SWE-CUDA – Memory-Bound Performance

A performance estimate for SWE:

- assumption: performance is **memory-bound**
- presentation laptop has a bandwidth (GPU main memory) of 9.6 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, 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: $9.6\,\text{GB/s} \div 3\,\text{MiB} \approx 3000 \frac{1}{s}$
- Measured timesteps per second: $386 \frac{1}{s}$

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
  $\rightarrow$ 9.6 GB/s $\div$ 5 MiB $\approx$ 1800 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 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

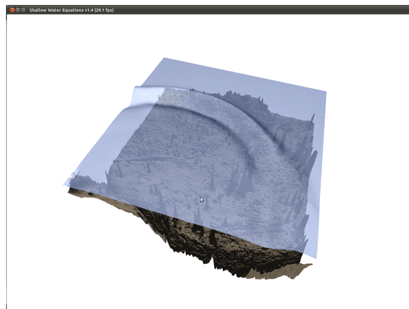
Optimization approach:

- 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

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

Parallel access to memory:

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

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

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

Parallel access to memory:

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

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

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

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 write 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

Assignment

1. Improve the `computeNetUpdates` kernel by using shared memory access to read the cell data.
2. Optimize the kernel for coalesced memory access, use the Nvidia profiler to check for warp serialization
3. (Optional) Implement a binary fan-in into the kernel, in order to reduce the maximum wave speeds for each block to a single value (reduce over all blocks is already integrated via `thrust`)

Performance Contest

SWE on a Tesla C2070 (mathgpu)

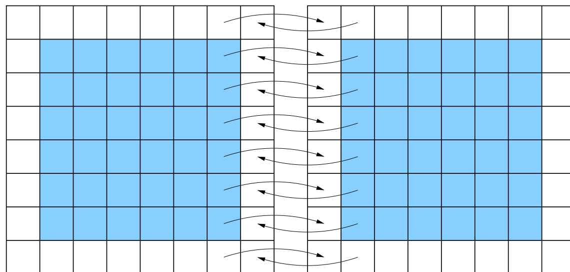
- 448 stream processing units
- memory bandwidth: 97.6 GB/s (acc. to bandwidth test)
- theoretical peak performance: $\approx$ 1 TFlop/s

How much do you get?

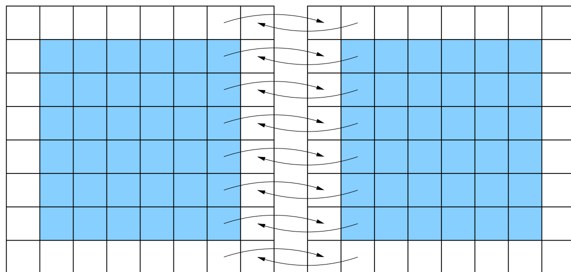
- in terms of memory throughput?
- in terms of Flop/s?
- in terms of processed cells per second?

Teil II

Parallelization on Hybrid Architectures



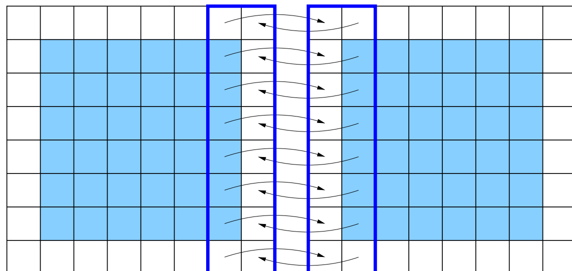
Exchange of Values in Ghost/Copy Layers



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

SWE_BlockCUDA – Update of Ghost Layers

Memory-Synchronization Revisited

- ghost layers might be updated in each time step
  $\rightarrow$ 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. dev
     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,
             rightInflow->h.elemVector(), 1, MPI_COL, rightRank,
              MPI_COMM_WORLD,&status);

MPI_Sendrecv(rightOutflow->h.elemVector(), 1, MPI_COL, rightRank
             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

- 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.