Further topics on SWE and CUDA

Alexander Pöppl
January 13th 2016
Last Tutorial

The Shallow Water Equations

- set of hyperbolic equations
- describe gravity-induced water waves in a shallow domain (wavelength $\gg$ depth)
- Finite Volume discretization with cell-averaged states and numerical fluxes

SWE Code

- Cartesian grid partitioned into blocks with ghost layers
- Euler time step:
  - set boundary conditions
  - compute net updates
  - set time step size
  - update cell unknowns
- parallelization concepts
a) Model and discretization:

- What do the shallow water equations describe and what unknowns appear in them?

- What are fluxes and how can they be approximated numerically?
Assignment T5.1 - Case Study (Recap)

a) Model and discretization:

- What do the shallow water equations describe and what unknowns appear in them?  
  The shallow water equations describe the large scale evolution of water (or other liquid) waves, affected by gravity (and bathymetry) in a vertically integrated domain. Vertical flows are neglected. The unknowns in the equations are water height and momentum.

- What are fluxes and how can they be approximated numerically?
a) Model and discretization:

- What do the shallow water equations describe and what unknowns appear in them?
  The shallow water equations describe the large scale evolution of water (or other liquid) waves, affected by gravity (and bathymetry) in a vertically integrated domain. Vertical flows are neglected. The unknowns in the equations are water height and momentum.

- What are fluxes and how can they be approximated numerically?
  Fluxes describe spatial displacement of a conserved quantity, such as mass or momentum transport. They are approximated with numerical fluxes generated by flux solvers (Upwind, Lax-Friedrichs, ...).
Assignment T5.1 - Case Study

b) Implementation of SWE:
   - What components does a block consist of?

   - What data dependency exists between `setBoundaryLayer()`, `computeFluxes()` and `eulerTimestep()`?
b) Implementation of SWE:

- What components does a block consist of?
  - Inner layer (local data access only)
  - Copy layer (access to ghost layer)
  - Ghost layer (access to neighbor block/boundary condition)

- What data dependency exists between `setBoundaryLayer()`, `computeFluxes()` and `eulerTimestep()`?

  - `computeFluxes()` requires boundary states from `setBoundaryLayer()` and inner states from `eulerTimestep()`.
  - `eulerTimestep()` requires net updates and time step size from `computeFluxes()`.
b) Implementation of SWE:

- What components does a block consist of?
  - Inner layer (local data access only)
  - Copy layer (access to ghost layer)
  - Ghost layer (access to neighbor block/boundary condition)

- What data dependency exists between `setBoundaryLayer()`, `computeFluxes()` and `eulerTimestep()`?
  `computeFluxes()` requires boundary states from `setBoundaryLayer()` and inner states from `eulerTimestep()`. `eulerTimestep()` requires net updates and time step size from `computeFluxes()`.
Assignment H5.2b - Coalesced access

SWE_WavePropagationBlockCuda_kernels.cu:
Change access pattern in `computeNetUpdatesKernel` and `updateUnknownsKernel`:

```
l_cellIndexI += blockDim.x * blockIdx.x + threadIdx.x + 1;
l_cellIndexJ += blockDim.y * blockIdx.y + threadIdx.y + 1;
/* ... */
l_cellIndexI = blockDim.x * blockIdx.x + threadIdx.x + 1;
l_cellIndexJ = blockDim.y * blockIdx.y + threadIdx.y + 1;
```

SWE_WavePropagationBlockCuda.cu:
Update block sizes in `computeNumericalFluxes` for the right and top block (the inner block is fine):

```
dim3 dimRightBlock(1, TILE_SIZE);
/* ... */
dim3 dimTopBlock(TILE_SIZE, 1);
```
Assignment H5.2b - Coalesced access

SWE_WavePropagationBlockCuda_kernels.cu:
Change access pattern in computeNetUpdatesKernel and updateUnknownsKernel:

```c
l_cellIndexI += blockDim.y * blockIdx.x + threadIdx.y + 1;
l_cellIndexJ += blockDim.x * blockIdx.y + threadIdx.x + 1;
/* ... */
l_cellIndexI = blockDim.y * blockIdx.x + threadIdx.y + 1;
l_cellIndexJ = blockDim.x * blockIdx.y + threadIdx.x + 1;
```

SWE_WavePropagationBlockCuda.cu:
Update block sizes in computeNumericalFluxes for the right and top block (the inner block is fine):

```c
dim3 dimRightBlock(TILE_SIZE, 1);
/* ... */
dim3 dimTopBlock(1, TILE_SIZE);
```
Part 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:

```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 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, "\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`

```cpp
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`)

Alexander Pöppl: Further topics on SWE and CUDA
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)

```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

__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)
Optimization of the SWE-CUDA Kernels

Fermi Memory Hierarchy

- Thread
- Shared Memory
- L1 Cache
- L2 Cache
- DRAM

image: NVIDIA
A performance estimate for SWE:

- assumption: performance is **memory-bound**
- NVidia NVS 5200M 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 $256 \times 256$, thus $65.6 \text{k cells}$
- variables $h$, $hu$, $hv$, $b$: $4 \times 4$ bytes per cell, thus $1 \text{ MiB}$
- net updates: $2 \times 2 \times 4$ bytes per edge, thus $2 \text{ MiB}$
- how many read & write accesses in each kernel?
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: \( \frac{14.4 \text{ GB/s}}{3 \text{ MiB}} \approx 4.8 \text{kHz} \)
- Measured timesteps per second: 1 kHz
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
  → triples number of read accesses
- new value: read 5 MiB, write 2.75 MiB per time step
  → 14.4 GB/s ÷ 5 MiB ≈ 2900 time steps per sec.?

Read accesses in updateUnKnowns:

- actually no extra read or write accesses
Task: Think of possible optimizations for the CUDA kernels

1. Can we use shared memory to improve memory access in the kernels?

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

3. Would it be feasible to fuse the kernels computeNetUpdatesKernel and updateUnknownsKernel?
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
Fs[tx+1][ty] = Fhd[iEdge];
Gds[tx][ty+1] = Ghd[iEdge];
/* ... */
```

```c
h = hd[iElem] - dt *( (Fs[tx+1][ty]-Fs[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

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

⇒ execute 1, synchronize, and then execute 2 – should work, right?
Net Updates and Updating Unknowns
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
   ⇒ execute 1, synchronize, and then execute 2 – should work, right?
   ⇒ unfortunately not! (no synchronization between blocks)
Net Updates and Updating Unknowns
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
   ⇒ execute 1, synchronize, and then execute 2 – should work, right?
   ⇒ unfortunately not! (no synchronization between blocks)
   ⇒ may be cured: old/new copy for h, hu, hv
References/Literature

