The Shallow Water Equations and CUDA

Oliver Meister

January 8th 2014
Last Tutorial

Discretized Heat Equation

- System matrix with fixed number of non-zeros per row
- Matrix-vector product and vector arithmetics

ELLPACK kernel

- Coalesced access to indices and data array
- Can degenerate to dense matrix

CUDA libraries

- cuBLAS: dense linear algebra
- cuSPARSE: sparse linear algebra
Assignment 1a: Kernel call

Task: Kernel call

\[
dim3 \ grid((N - 1)/\text{TILE\_SIZE} + 1, 1, 1); \\
\text{dim3 block(}\text{TILE\_SIZE}, 1, 1)\]; \\
k_{\text{ell\_mat\_vec\_mm}} \llll \text{grid, block} \rrrr (...) \;
\]  
→ Same as for scalar CSR kernel
Assignment 1b: ELLPACK kernel

Issues: None!
Assignment 1c,d: cuBLAS calls

cublasCreate(&handle);
ELL_create(N, num_cols_per_row, indices, data, x, y, ...);

//choose something bigger than epsilon initially
err = 2.0f * epsilon;
for (i = 0; err > epsilon; ++i) {
    ELL_kernel(N, num_cols_per_row, indices_d, data_d, x_d, y_d);

    cublasSnrm2(handle, N, y_d, 1, &err);
    alpha = dt/(dx * dx) * c;
    cublasSaxpy(handle, N, &alpha, y_d, 1, x_d, 1);

    //** Output: see next slide **/
}

ELL_destroy(indices_d, data_d, x_d, y_d);
cublasDestroy(handle);
Assignment 1c,d: Vector output

Output: Two possible solutions

```c
if ((i & 511) == 0) {
    cudaMemcpy(x, x_d, N * sizeof(float), cudaMemcpyDeviceToHost)
    // OR (better)
    cublasGetVector(N, sizeof(float), x_d, 1, x, 1)

    // (...)
}
```
// choose something bigger than epsilon initially
err = 2.0f * epsilon;

for (i = 0; err > epsilon; ++i) {
    alpha = 1.0f; beta = 0.0f;
cusparseShybmv(cusparseHandle,  
    CUSPARSE_OPERATION_NON_TRANSPOSE,  
    &alpha, descr, hyb_d, x_d, &beta, y_d);

cublasSnrm2(cublasHandle, N, y_d, 1, &err);

    alpha = dt/(dx * dx) * c;
cublasSaxpy(cublasHandle, N, &alpha, y_d, 1, x_d, 1);
}

Assignment 2c: ELLPACK vs. cuSPARSE

Performance by Heat Equation kernel

Matrix size n * n

time [s]

ellpack  cusparse  band

Oliver Meister: The Shallow Water Equations and CUDA
Tutorial Parallel Programming and High Performance Computing, January 8th 2014
Assignment 2c: ELLPACK vs. cuSPARSE

Performance by Heat Equation kernel

Matrix size $n \times n$

time difference [s]

ellpack  cusparse  band
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
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
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
Teil I

Model and Discretization
Model: The Shallow Water Equations

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

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

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.):

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

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

Unknnowns 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.
Spatial Discretization:

- regular Cartesian meshes; allow multiple patches
- multiple layers for boundary conditions, complicated domains and parallelization:
  - inner layer (local data), copy layer (copied to other patches) and ghost layer (boundary or copied from other patches)
Case Study: Shallow Water Equations

Task: Read part 1 of the case study on the worksheet. Answer the following questions:

a) What is a flux solver? Which data does it operate on?

b) What are the source terms in the shallow water equations?

*Note:* Keep in mind that it is not necessary for you to understand every detail, only the general concept.
Teil II

Implementation

- SWE_Block
- SWE_BlockCUDA
- SWE_RusanovBlock
- SWE_WavePropagationBlock
- SWE_RusanovBlockCUDA
- SWE_WavePropagationBlockCuda
Main Loop – Euler Time-stepping

```java
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

```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);
}
```

→ 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.)

```cpp
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:
    { /* ... */
```

(Oliver Meister: The Shallow Water Equations and CUDA Tutorial Parallel Programming and High Performance Computing, January 8th 2014)
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)
Compute Numerical Fluxes (2)

main loop to compute net updates on top/bottom edges:

```c
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

```cpp
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()`, ...
  \( \rightarrow \) 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`)

Oliver Meister: The Shallow Water Equations and CUDA  
Tutorial Parallel Programming and High Performance Computing, January 8th 2014
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 cells!
Case Study: Shallow Water Equations

Task: Read part 2 of the case study on the worksheet. Answer the following questions:

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

b) What data dependency exists between `computeFluxes()` and `eulerTimestep()`?

→ Homework!