Hyperbolic Conservation Laws on GPUs

Master Course, 2013,
University of Granada, Spain

2013-04-09

André R. Brodtkorb, Ph.D., Research Scientist
SINTEF ICT, Dept. of Appl. Math.

Outline

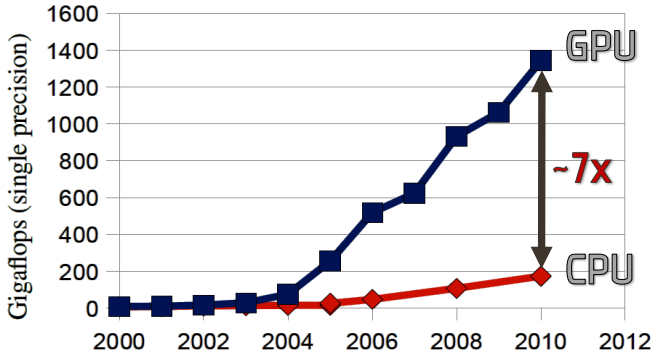
- Quick repetition on GPUs
- Mapping mathematics to computations
- The shallow water equations
- Validity of computed results
- Higher performance

Quick GPU Repetition

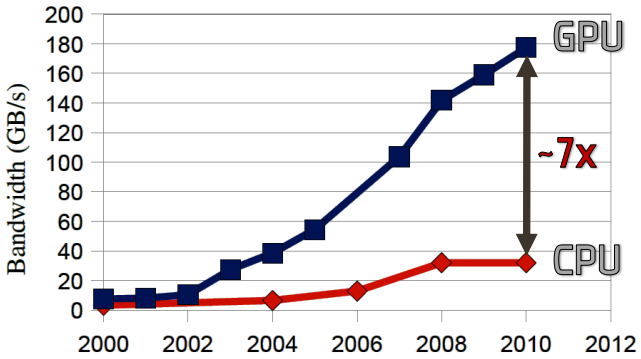
Massive Parallelism: The Graphics Processing Unit

	CPU	GPU
Cores	4	16
Float ops / clock	64	1024
Frequency (MHz)	3400	1544
GigaFLOPS	217	1580
Power consumption	~130 W	~250 W
Memory (GiB)	32+	3

Performance

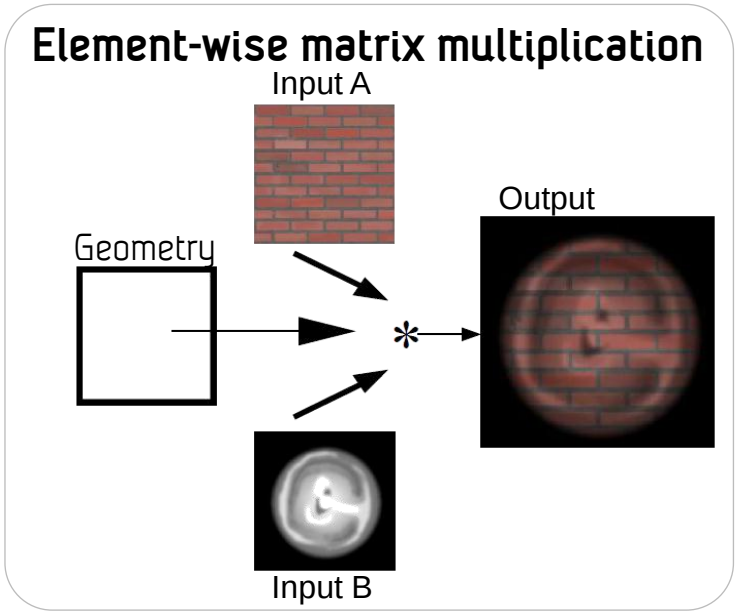


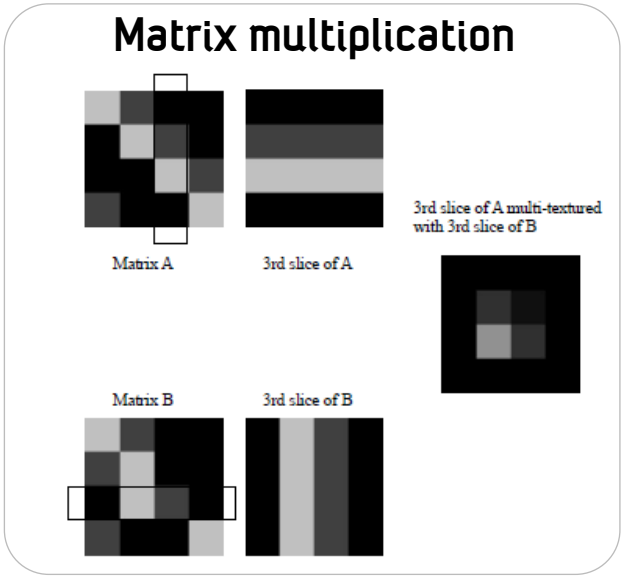
Memory Bandwidth



Early Programming of GPUs

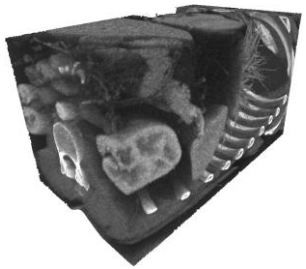
- GPUs were first programmed using OpenGL and other graphics languages
  - Mathematics were written as operations on graphical primitives
  - Extremely cumbersome and error prone



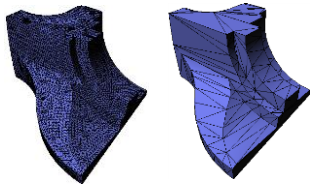


[1] Fast matrix multiplies using graphics hardware, Larsen and McAllister, 2001

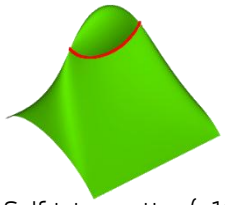
Examples of Early GPU Research at SINTEF



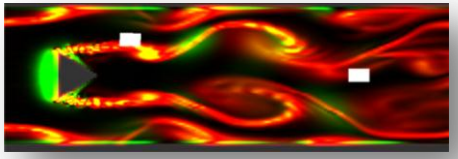
Registration of medical data (~20x)



Preparation for FEM (~5x)

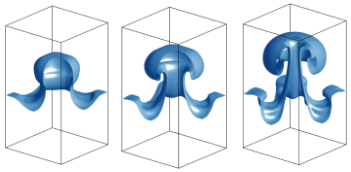


Self-intersection (~10x)

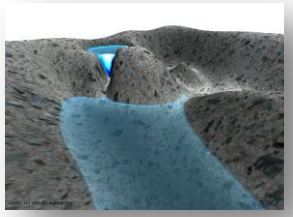


Fluid dynamics and FSI (Navier-Stokes)

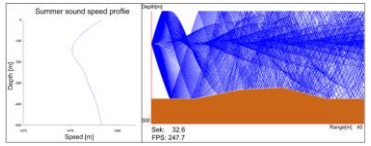
Inpainting (~400x matlab code)



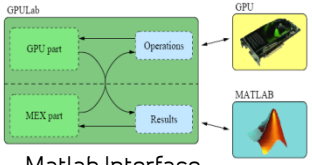
Euler Equations (~25x)



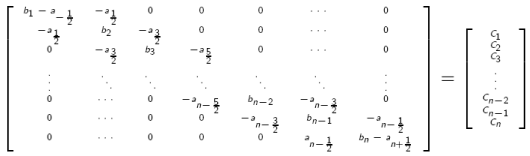
SW Equations (~25x)



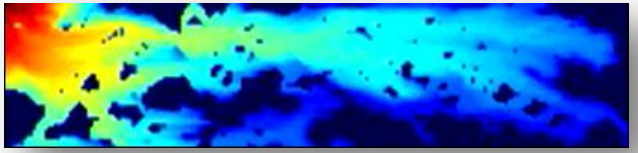
Marine aqoustics (~20x)



Matlab Interface

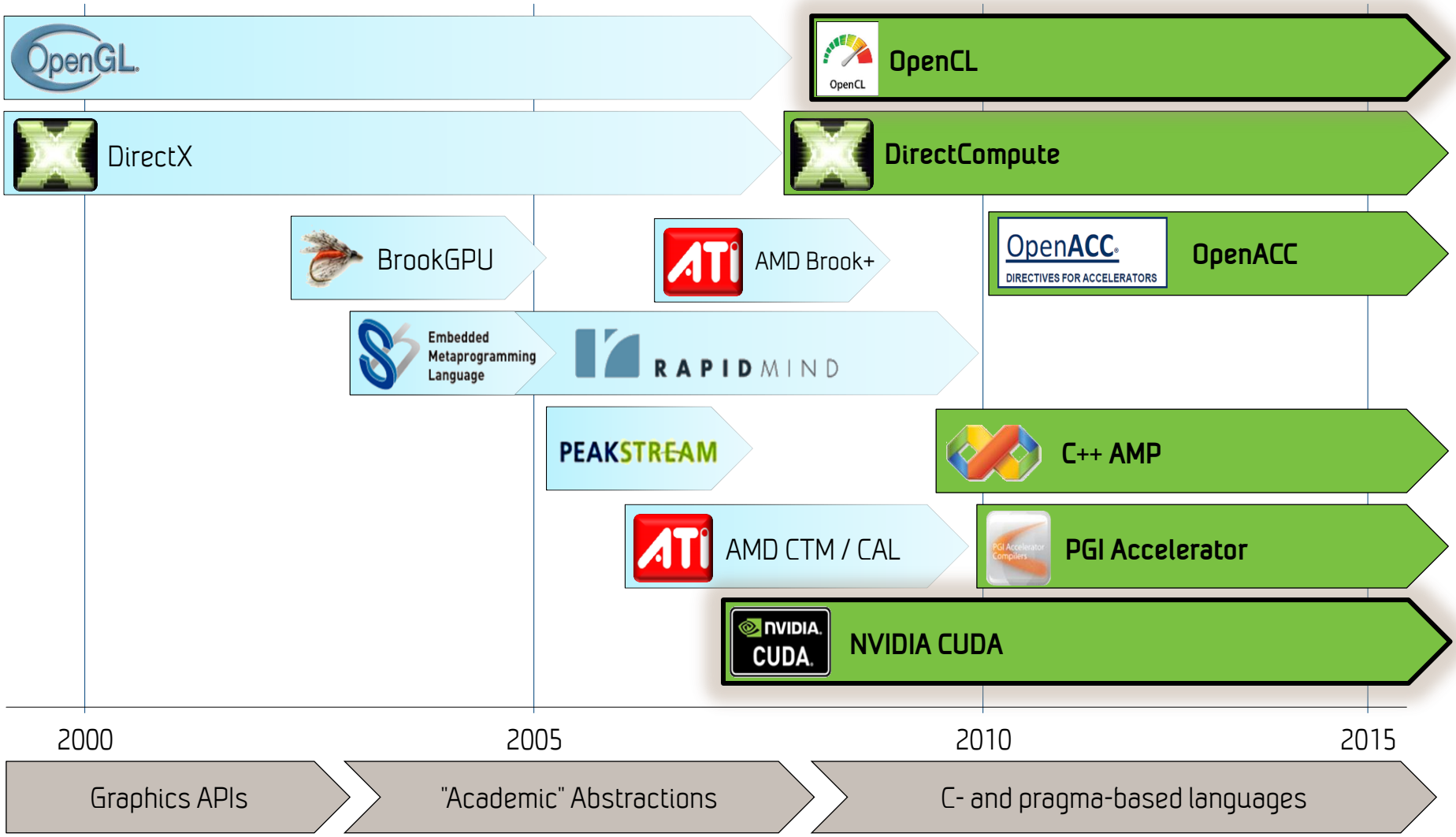


Linear algebra



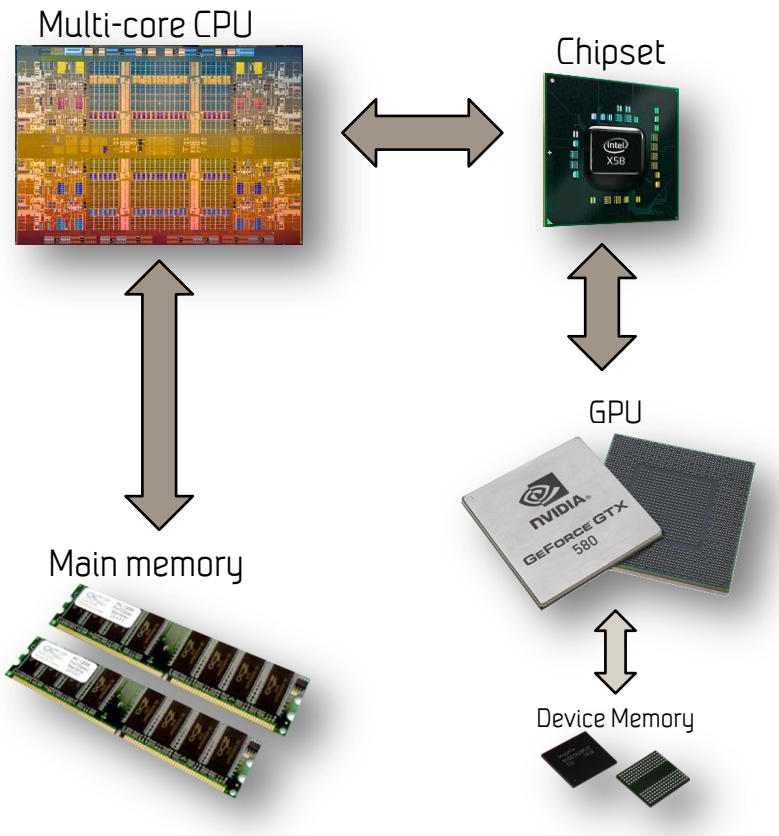
Water injection in a fluvial reservoir (20x)

Todays GPU Programming Languages

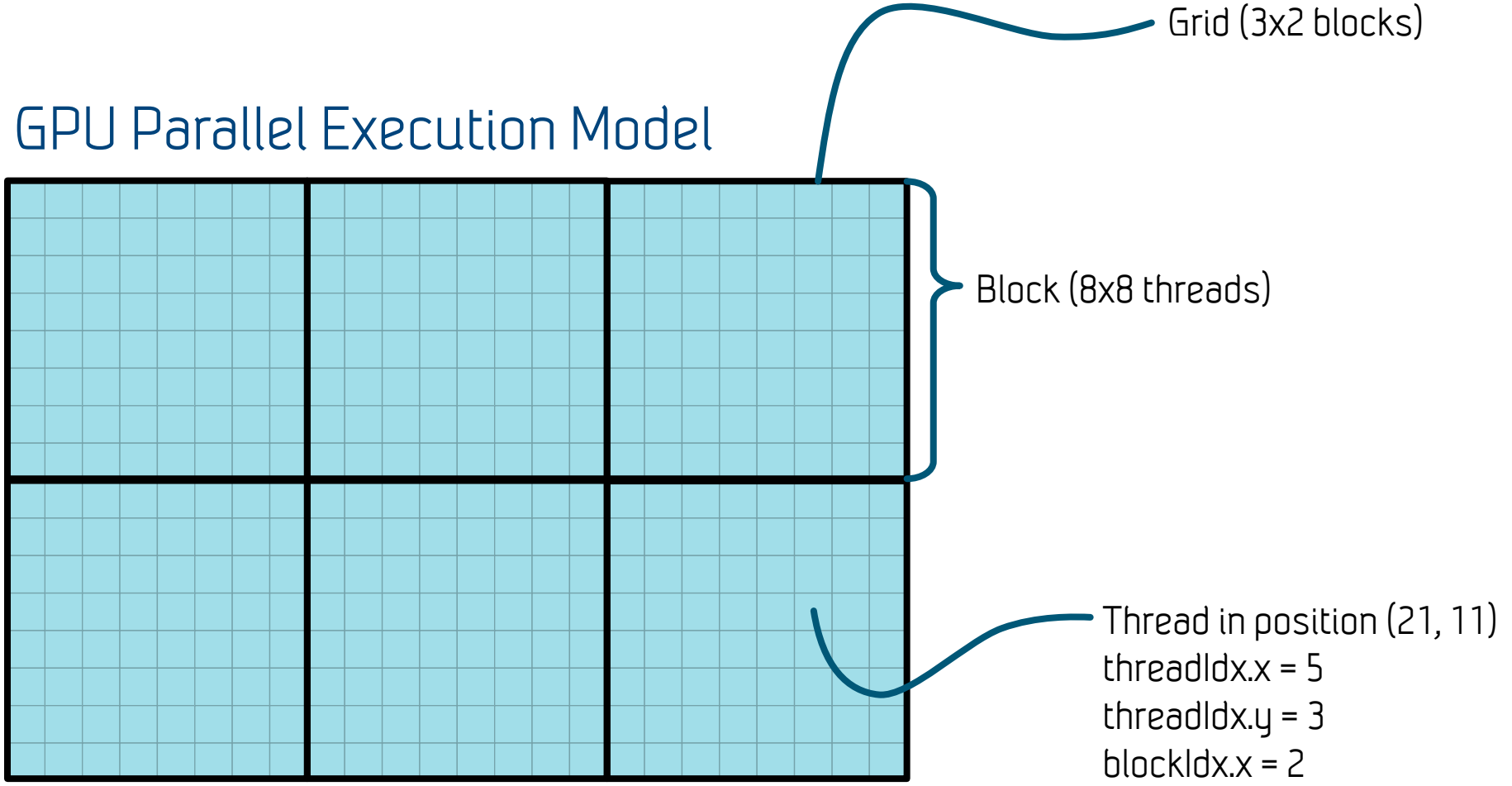


GPU Architecture

- The GPU is connected to the traditional CPU via the PCI-express bus, which is relatively slow
  - 15.75 GB/s each direction
- It is a massively parallel processor with high bandwidth
  - 7-10x as fast memory and computational speed as the CPU
- It has limited device memory
  - Typically up-to 6 GB

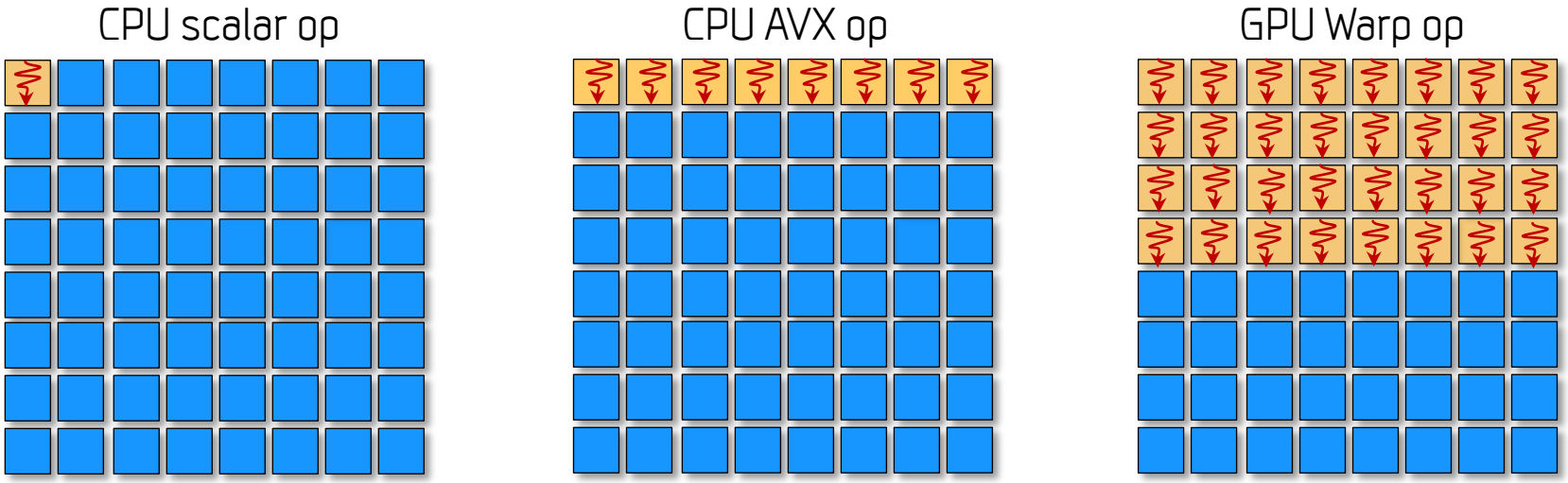


GPU Parallel Execution Model



- The same program is launched for all threads "in parallel"
- The thread identifiers are used to calculate its global position
- The thread position is used to load and store data, and execute code
- The parallel execution means that synchronization can be very expensive

GPU Vector Execution Model



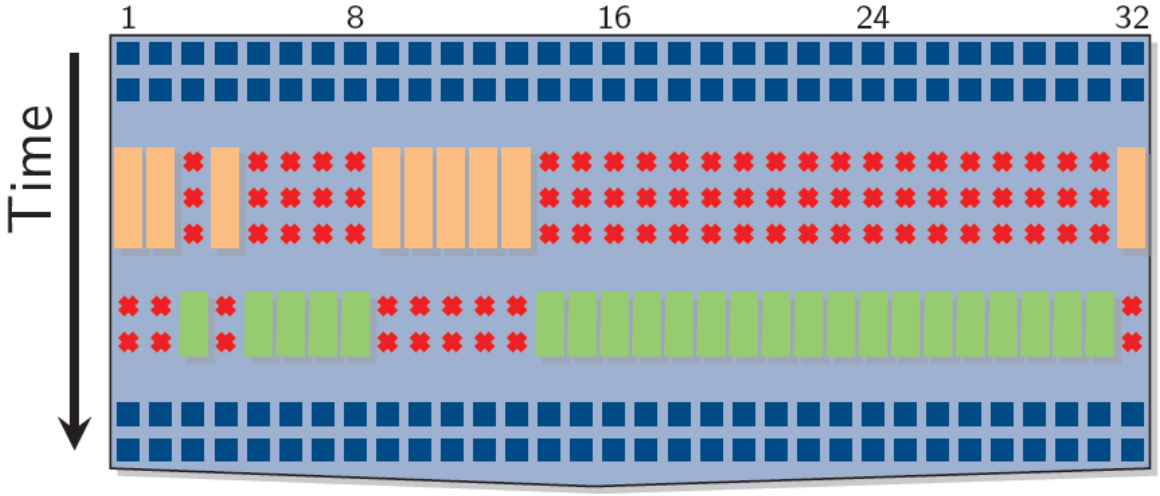
CPU scalar op: 1 thread, 1 operand on 1 data element

CPU SSE/AVX op: 1 thread, 1 operand on 2-8 data elements

GPU Warp op: 1 **warp** = 32 threads, 32 operands on 32 data elements

- Exposed as **individual threads**
- Actually runs the **same instruction**
- Divergence implies **serialization and masking**

Branching can be expensive




```
// Non-divergent code
if( x > 0 ) {
    y = pow(x, exp);
    y *= Ks;
    z = y + Ka;
} else {
    x = 0;
    z = Ka;
}
// Non-divergent code
```

Hardware automatically serializes and masks divergent code flow:

- Execution time is the sum of all branches taken
- Programmer is relieved of fiddling with element masks (which is necessary for SSE)
- Worst case 1/32 performance
- Important to **minimize divergent code flow**!
  - Move conditionals into data, use min, max, conditional moves.

Mathematics to computations

Hyperbolic Conservation Laws

- A conservation law describes that a quantity is conserved
  - Example: conservation of mass (amount of water) in shallow water
- A hyperbolic conservation law means that a disturbance propagates with a finite speed
  - Example: a wave will not travel infinitely fast: it has a maximum propagation speed
- Many natural phenomena can (partly) be described mathematically as such hyperbolic conservation laws
  - Gas dynamics
  - Magneto-hydrodynamics
  - Shallow water
  - …

Partial differential equations (PDEs)

- Partial differential equations are used to describe a multitude of physical phenomena
  - Tsunamis, sound waves, heat propagation, pressure waves, …
- The heat equation is a prototypical PDE

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}$$

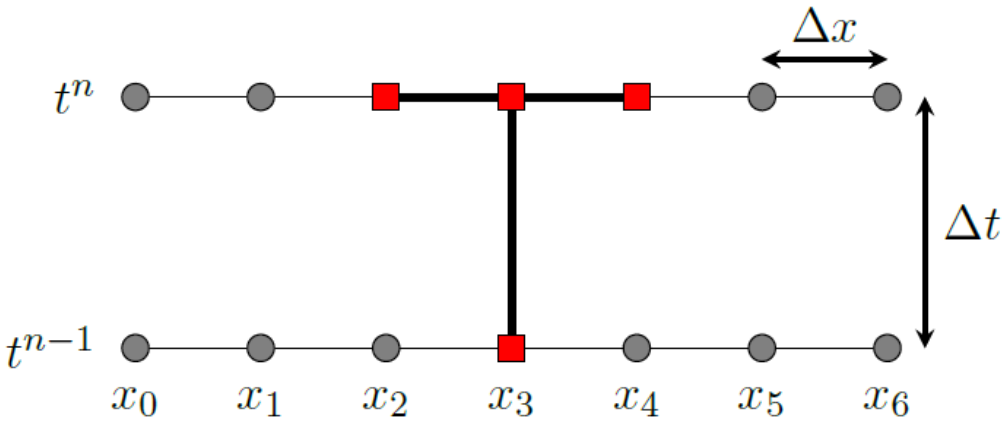
- u is the temperature, kappa is the diffusion coefficient, t is time, and x is space.
- Describes diffusive heat transport in a medium

Solving the heat equation

- We can discretize this PDE by replacing the continuous derivatives with discrete approximations

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \quad \Rightarrow \quad \frac{1}{\Delta t}(u_i^n - u_i^{n-1}) = \frac{\kappa}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n)$$

- The discrete approximations use a set of grid points in space and time



- The choice of discrete derivatives and grid points gives rise to different discretizations with different properties

Solving the heat equation

- From the discretized PDE, we can create a numerical scheme by reordering the terms

$$\frac{1}{\Delta t}(u_i^n - u_i^{n-1}) = \frac{\kappa}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n)$$

$$-ru_{i-1}^n + (1+2r)u_i^n - ru_{i+1}^n = u_i^{n-1}, \qquad r = \frac{\kappa \Delta t}{\Delta x^2}$$

- This gives us one equation per grid point which we must solve

Solving a PDE

- We organize all the equations we have into a matrix equation Ax=b
  - We gather the *coefficients* in A
  - We gather the unknowns (u^n) in the vector x
  - We gather the known state (u^{n-1})in the vector b

$$-ru_{i-1}^n + (1+2r)u_i^n - ru_{i+1}^n = u_i^{n-1}$$

$$\begin{bmatrix} ??? & 0 & 0 & 0 & 0 & 0 \\ -r & 1+2r & -r & 0 & 0 & 0 & 0 \\ 0 & -r & 1+2r & -r & 0 & 0 & 0 \\ 0 & 0 & -r & 1+2r & -r & 0 & 0 \\ 0 & 0 & 0 & -r & 1+2r & -r & 0 \\ 0 & 0 & 0 & 0 & -r & 1+2r & -r \\ 0 & 0 & 0 & 0 & 0 & ??? \end{bmatrix} \begin{bmatrix} u_0^n \\ u_1^n \\ u_2^n \\ u_3^n \\ u_4^n \\ u_5^n \\ u_6^n \end{bmatrix} = \begin{bmatrix} u_0^{n-1} \\ u_1^{n-1} \\ u_2^{n-1} \\ u_3^{n-1} \\ u_4^{n-1} \\ u_5^{n-1} \\ u_6^{n-1} \end{bmatrix}$$

- For the first and last equations, we need boundary conditions!

Boundary conditions

- Boundary conditions describe how the solution should behave at the boundary of our domain
- Different boundary conditions give very different solutions!
- A simple boundary condition to implement is fixed "boundaries"
  - This simply sets the temperature at the end points to a fixed value

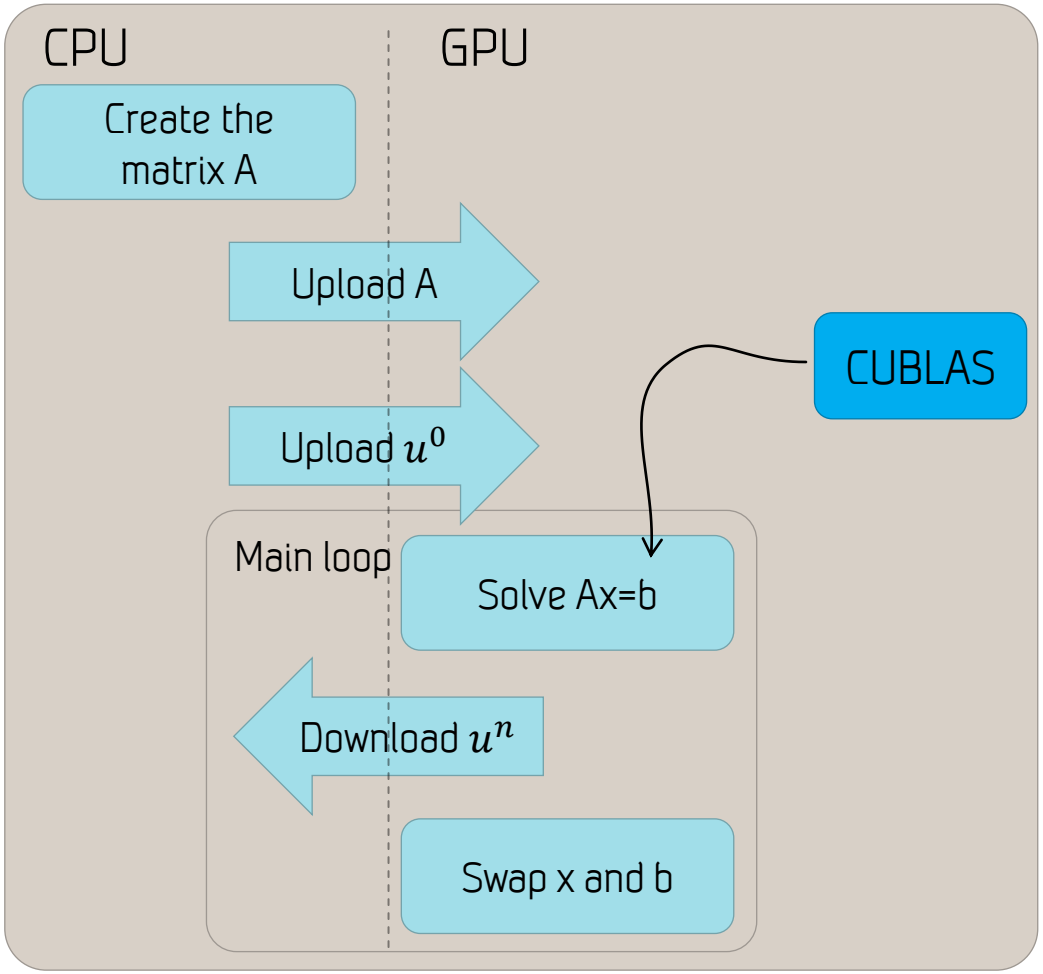
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -r & 1+2r & -r & 0 & 0 & 0 & 0 \\ 0 & -r & 1+2r & -r & 0 & 0 & 0 \\ 0 & 0 & -r & 1+2r & -r & 0 & 0 \\ 0 & 0 & 0 & -r & 1+2r & -r & 0 \\ 0 & 0 & 0 & 0 & -r & 1+2r & -r \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0^n \\ u_1^n \\ u_2^n \\ u_3^n \\ u_4^n \\ u_5^n \\ u_6^n \end{bmatrix} = \begin{bmatrix} u_0^{n-1} \\ u_1^{n-1} \\ u_2^{n-1} \\ u_3^{n-1} \\ u_4^{n-1} \\ u_5^{n-1} \\ u_6^{n-1} \end{bmatrix}$$

Solving the heat equation

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -r & 1+2r & -r & 0 & 0 & 0 & 0 \\ 0 & -r & 1+2r & -r & 0 & 0 & 0 \\ 0 & 0 & -r & 1+2r & -r & 0 & 0 \\ 0 & 0 & 0 & -r & 1+2r & -r & 0 \\ 0 & 0 & 0 & 0 & -r & 1+2r & -r \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} u_0^n \\ u_1^n \\ u_2^n \\ u_3^n \\ u_4^n \\ u_5^n \\ u_6^n \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} u_0^{n-1} \\ u_1^{n-1} \\ u_2^{n-1} \\ u_3^{n-1} \\ u_4^{n-1} \\ u_5^{n-1} \\ u_6^{n-1} \end{bmatrix}}_{b}$$

- We now have a well-formed problem, if we give some initial heat distribution, u^0
- We can solve the matrix equation $Ax = b$ using linear algebra solvers (Gaussian elimination, conjugate gradients, tri-diagonal solvers, etc.)
- Choosing the right solver is often key to performance: CUBLAS, CUSPARSE, CUSP, …

Solving the heat equation on the GPU



$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -r & 1+2r & -r & 0 & 0 & 0 & 0 \\ 0 & -r & 1+2r & -r & 0 & 0 & 0 \\ 0 & 0 & -r & 1+2r & -r & 0 & 0 \\ 0 & 0 & 0 & -r & 1+2r & -r & 0 \\ 0 & 0 & 0 & 0 & -r & 1+2r & -r \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The Heat Equation on the GPU

- The example so far is quite inefficient and boring…
  - It solves only in 1D
    - Many real-world problems require 2D or 3D simulations
  - It does not utilize any knowledge about the matrix A or the solution
    - A is tridiagonal: we are storing and computing n^2 elements, whilst we only need to store the $3n$ non-zero elements
  - It uses a regular grid
    - Non-regular grids give us local refinement where we need it

- Adding more features gives a more complex picture
  - The matrix A quickly gets more complex with more features (2D/3D/non-regular grids/etc.)
  - More complex problems have more equations, and the A matrix must often be re-calculated for each simulation step (non-constant coefficients)

The Heat Equation on the GPU

- The presented numerical scheme is called an *implicit* scheme

- Implicit schemes are often sought after
  - They allow for large time steps,
  - They can be solved using standard tools
  - Allow complex geometries
  - They can be very accurate
  - …

- However…
  - Solution time is often a function of how long it takes to solve Ax=b and linear algebra solvers can be **slow and memory hungry**, especially on the GPU
  - for many time-varying phenomena, we are also interested in the temporal dynamics of the problem

Explicit scheme for the heat equation

- For problems in which disturbances travel at a finite speed, we can change the time derivative from a backward to a forward difference.

$$\frac{1}{\Delta t}(u_i^{n} - u_i^{n-1}) = \frac{\kappa}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n)$$

$$\frac{1}{\Delta t}(u_i^{n+1} - u_i^{n}) = \frac{\kappa}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n)$$

- This gives us an *explicit* numerical scheme (compared to the *implicit* scheme already shown)

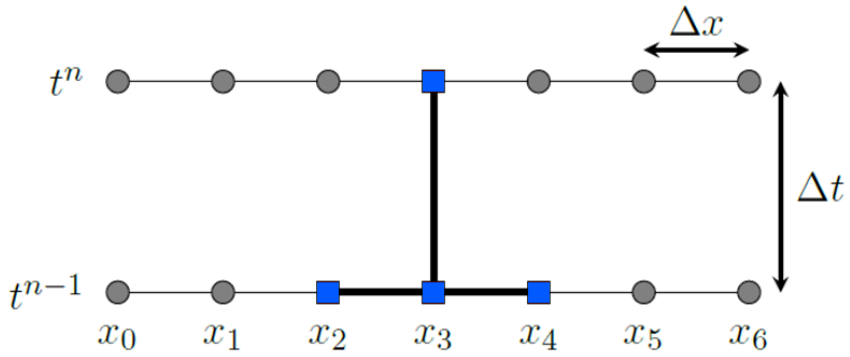
$$-ru_{i-1}^n + (1+2r)u_i^n - ru_{i+1}^n = u_i^{n-1}$$

$$u_i^{n+1} = ru_{i-1}^n + (1-2r)u_i^n + ru_{i+1}^n$$

Explicit scheme for the heat equation

- An explicit scheme for the heat equation gives us an explicit formula for the solution at the next timestep for each cell!
  - It is simply a weighted average of the two nearest neighbors and the cell itself

$$u_i^{n+1} = ru_{i-1}^n + (1-2r)u_i^n + ru_{i+1}^n$$



- This is perfectly suited for the GPU: each grid cell at the next time step can be computed independently of all other grid cells!

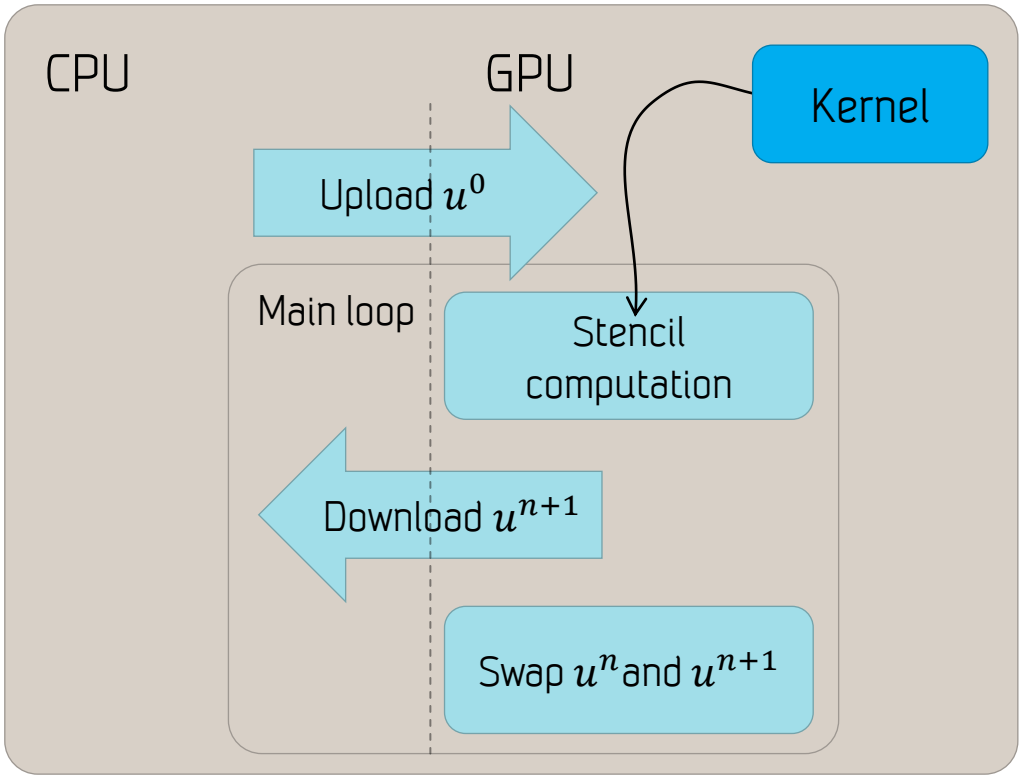
- However, we must have much smaller time steps than in the implicit scheme

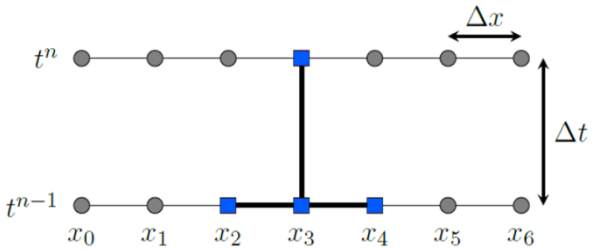
Timestep restriction

- Consider what would happen if you used a timestep of e.g., 10 hours for a stencil computation.
  - It is impossible, numerically, for a disturbance to travel more than one grid cell
  - Physically, however, the disturbance might have travelled half the domain
  - Using too large timesteps leads to unstable simulation results

- The restriction on how large the timestep can be is called the Courant-Friedrichs-Levy condition
  - Find the fastest propagation speed within the domain, and the timestep is inversely proportional to this speed.
  - For the heat equation $\frac{1}{2} > \frac{\kappa \Delta t}{\Delta x^2}$

Solving the heat equation on the GPU





GPU Kernel

CPU Code

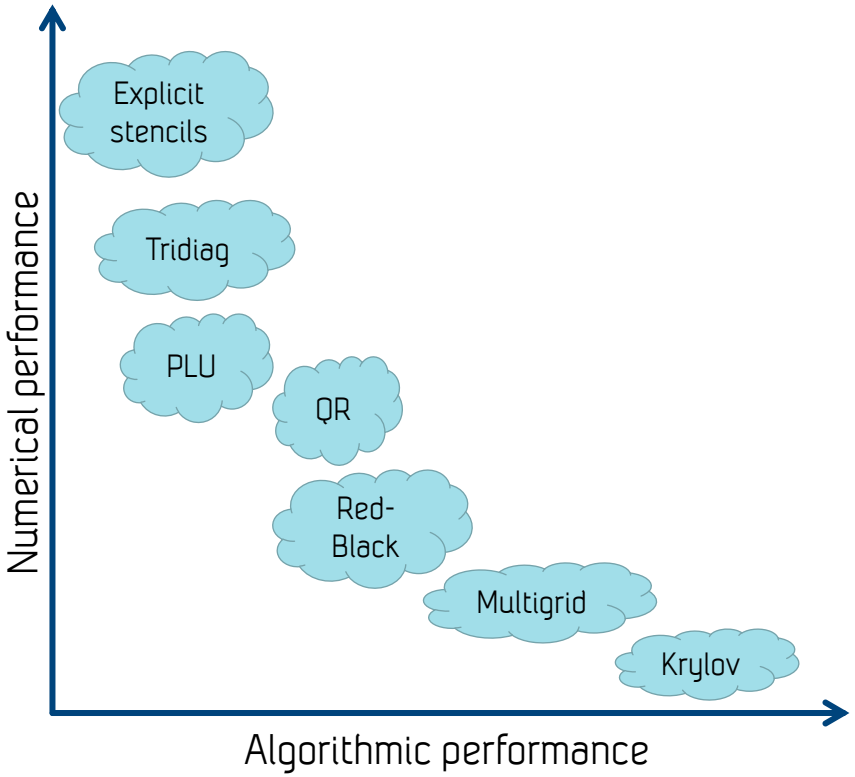
```
//Allocate memory
cudaMalloc(u0_ptr, …);
cudaMalloc(u1_ptr, …);

//Upload data
cudaMemcpy(u0_tr, …);
cudaMemcpy(u1_tr, …);

while (simulate == true) {
  //Launch kernel with n threads
  heatEquation<<<1, n>>>(u0_ptr, u1_ptr, r);

  //Download results
  cudaMemcpy(u1_ptr, …);

  //Swap pointers
  std::swap(u0_ptr, u1_ptr);
}
```

GPU Kernel

```
__global__ heatEquation(float* u0_ptr, float* u1_ptr,
                                              float r) {
 //Find location of this element in the global grid
 int x = blockIdx.x * blockDim.x + threadIdx.x;

 //Read the existing state from global memory (u0)
 float u0_l = u0_ptr[x-1];
 float u0_c = u0_ptr[x];
 float u0_r = u0_ptr[x+1];

 //Compute the heat equation
 float u1 = r*u0_l + (1.0-2.0*r)*u0_c + r*u0_r;

 //Write the new solution to global memory
 u1_ptr[x] = u1;
}
```

Choosing a solution method for GPUs

- For all problems, the total performance is the product of the algorithmic **and** the numerical performance
  - Your mileage may vary: algorithmic performance is highly problem dependent
- On GPUs, it can sometimes be a good idea to do a "stupid thing many times"
- Sparse linear algebra solvers often have high algorithmic performance
  - But only able to utilize a fraction of the capabilities of CPUs, and **worse on GPUs**
- **Explicit stencil** schemes (compact stencils) often able to **efficiently exploit the GPU execution model**
  - But can have low algorithmic performance



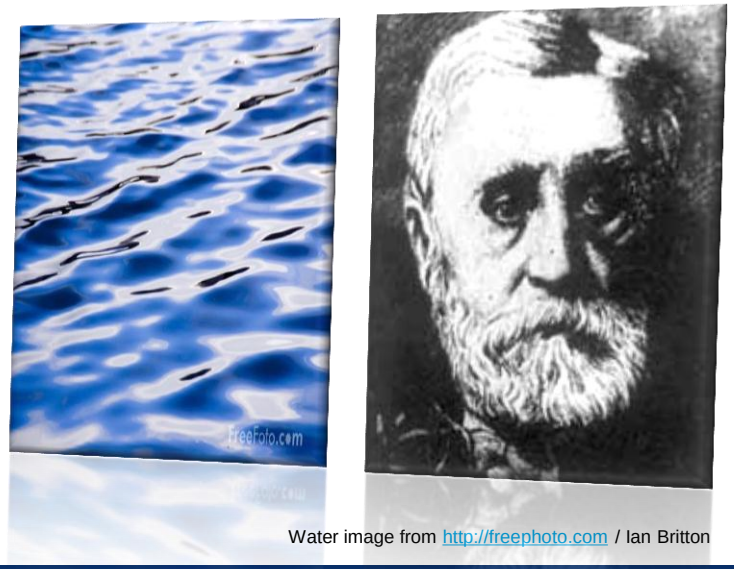
Shallow Water

The Shallow Water Equations

- A hyperbolic partial differential equation
  - First described by de Saint-Venant (1797-1886)
  - Conservation of mass and momentum
  - Gravity waves in 2D free surface

- Gravity-induced fluid motion
  - Governing flow is horizontal

- Not only used to describe physics of water:
  - Simplification of atmospheric flow
  - Avalanches
  - …



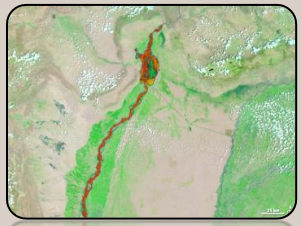
Water image from http://freephoto.com / Ian Britton

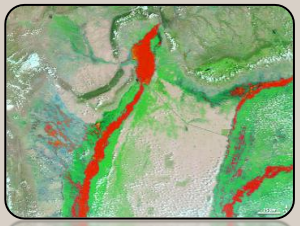
Target Application Areas

Tsunamis

2011: Japan (5321+)
2004: Indian Ocean (230 000)

Floods



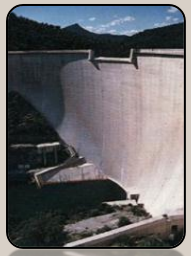


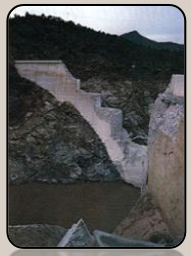
2010: Pakistan (2000+)
1931: China floods (2 500 000+)

Storm Surges

2005: Hurricane Katrina (1836)
1530: Netherlands (100 000+)

Dam breaks



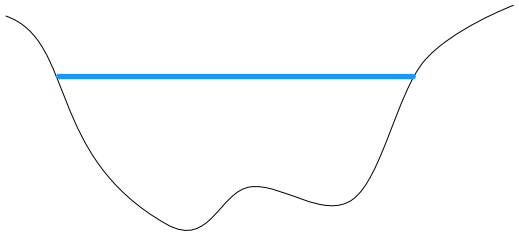


1975: Banqiao Dam (230 000+)
1959: Malpasset (423)

Images from wikipedia.org, www.ecolo.org

Do we need more speed?

- Many existing dam break inundation maps are based on 1D simulations
  - Approximate valleys using 1D cross sections
  - Much bias to individual engineer skills
  - Assumptions only hold for valleys



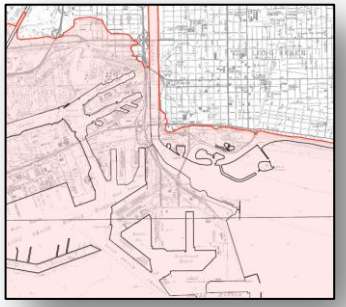
- Many dams and levees even lack emergency action plans!
  - In US: **42% of high hazard**, **44% of significant hazard** dams without plans
  - **114.000 miles** of levee systems

- **Simulation using GPUs enables high quality 2D simulations**

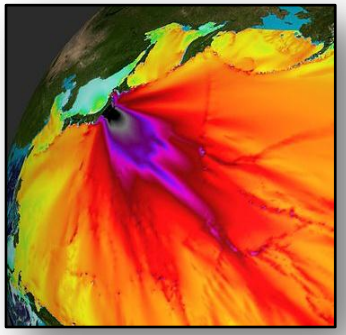
See also M. Altinakar, P. Rhodes, Faster-than-Real-Time Operational Flood Simulation using GPGPU Programming

Using GPUs for Shallow Water Simulations

- In preparation for events: Evaluate possible scenarios
  - Simulation of many ensemble members
  - Creation of inundation maps and emergency action plans



- In response to ongoing events
  - Simulate possible scenarios *in real-time*
  - Simulate strategies for action (deployment of barriers, evacuation of affected areas, etc.)



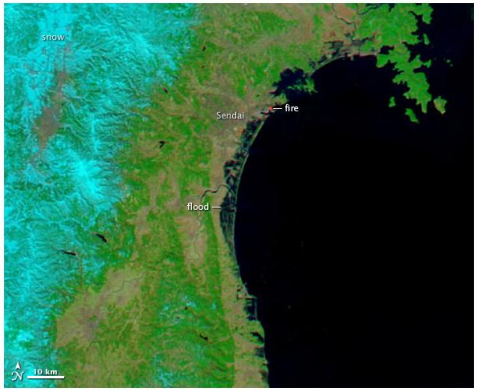
- **High requirements to performance => Use the GPU**

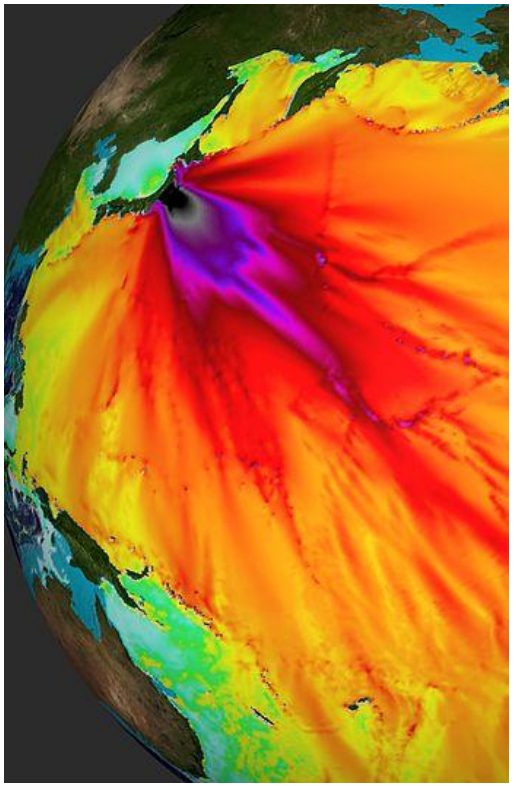
Simulation result from NOAA

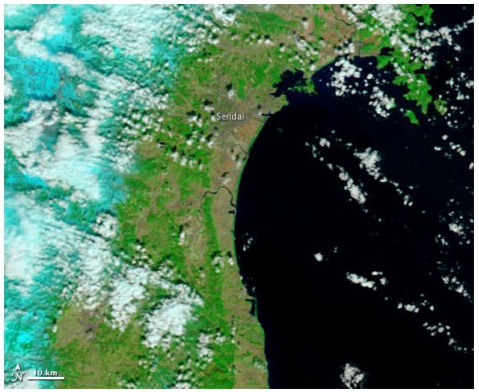
Inundation map from "Los Angeles County Tsunami Inundation Maps", http://www.conservation.ca.gov/cgs/geologic_hazards/Tsunami/Inundation_Maps/LosAngeles/Pages/LosAngeles.aspx

2011 Japan Tsunami

- Tsunami warnings must be issued in real-time
  - Huge computational domains
  - Rapid wave propagation
  - Uncertainties wrt. Tsunami cause
- Warnings must be accurate
  - Wrongful warning are dangerous!
- GPUs can be used to increase quality of warnings

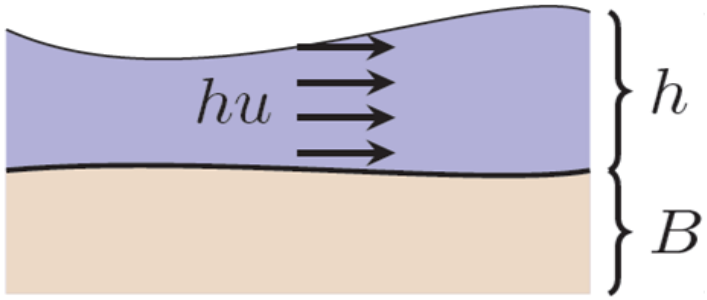






Images from US Navy (top), NASA (left), NOAA (right)

The Shallow Water Equations



$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}$$

Vector of Conserved variables

Flux Functions

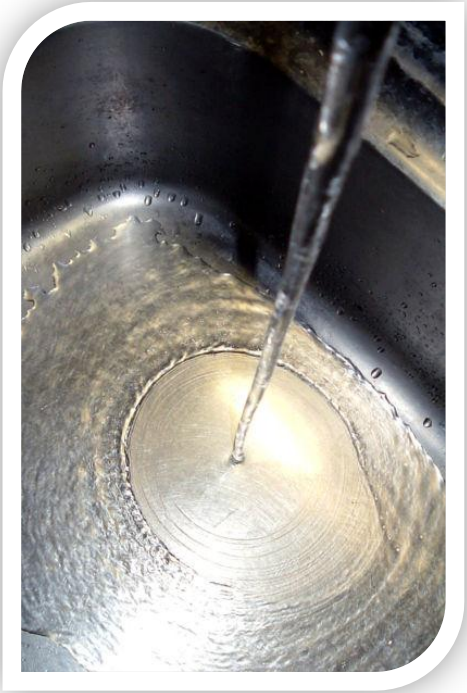
Bed slope source term

Bed friction source term

The Shallow Water Equations

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}$$

- A Hyperbolic partial differential equation
  - Enables explicit schemes
- Solutions form discontinuities / shocks
  - Require high accuracy in smooth parts without oscillations near discontinuities
- Solutions include dry areas
  - Negative water depths ruin simulations
- Often high requirements to accuracy
  - Order of spatial/temporal discretization
  - Floating point rounding errors
- Can be difficult to capture "lake at rest"



A standing wave or *shock*

Finding the "perfect" numerical scheme

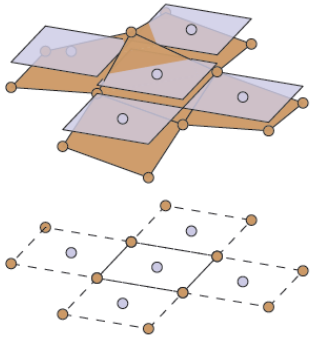
- We want to find a numerical scheme that
  - Works well for our target scenarios
    - Handles dry zones (land)
    - Handles shocks gracefully (without smearing or causing oscillations)
    - Preserves "lake at rest"
    - Has the accuracy for capturing the required physics
    - Preserves the physical quantities
  - Fits GPUs well
    - Works well with single precision
    - Is embarrassingly parallel
    - Has a compact stencil
    - …
  - …

Chosen numerical scheme: A. Kurganov and G. Petrova, A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System *Communications in Mathematical Sciences*, 5 (2007), 133-160

- Second order accurate fluxes
- Total Variation Diminishing
- Well-balanced (captures lake-at-rest)
- Compact stencil (Good ,but not perfect, match with the GPU)

Discretization

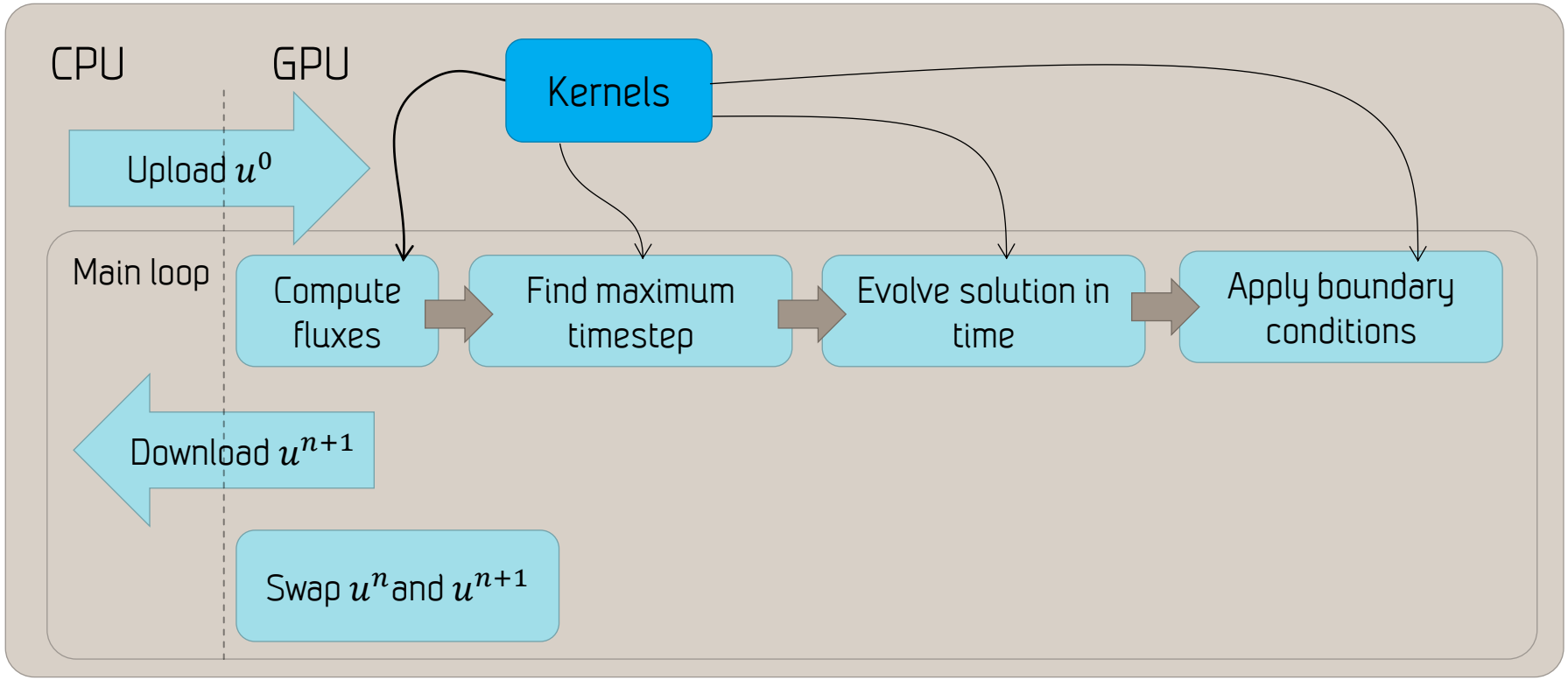
- Our grid consists of a set of *cells* or *volumes*
  - The bathymetry is a piecewise bilinear function
  - The physical variables (h, hu, hv), are piecewise constants per volume



- Physical quantities are transported across the cell interfaces

- Source terms (bathymetry and friction) are per cell

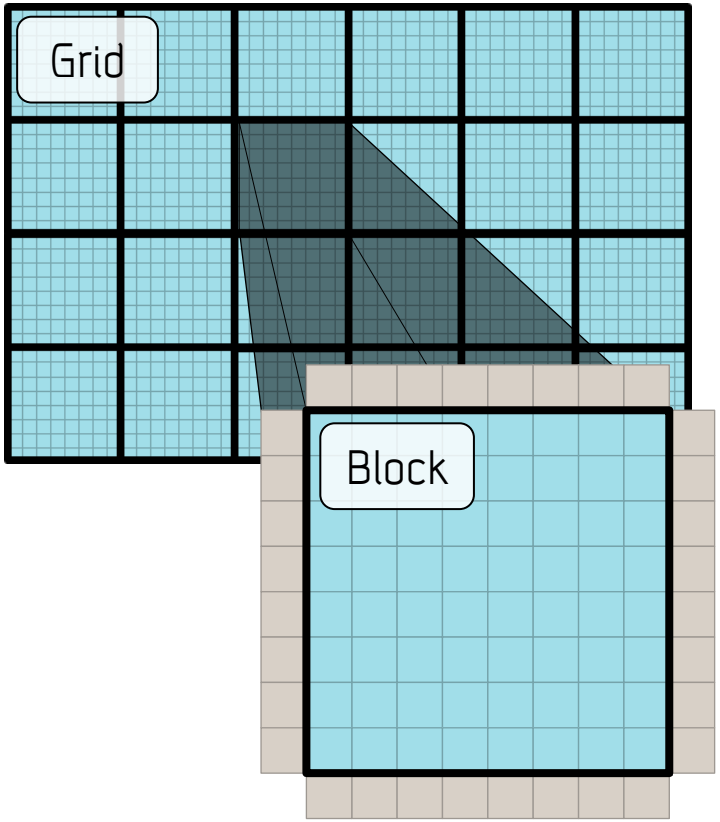
Simulation setup



Flux kernel domain decomposition: grids and blocks

- Observations:
  - Our shallow water problem is 2D
  - The GPU requires a parallel algorithm
  - The GPU has native support for 2D grids and blocks

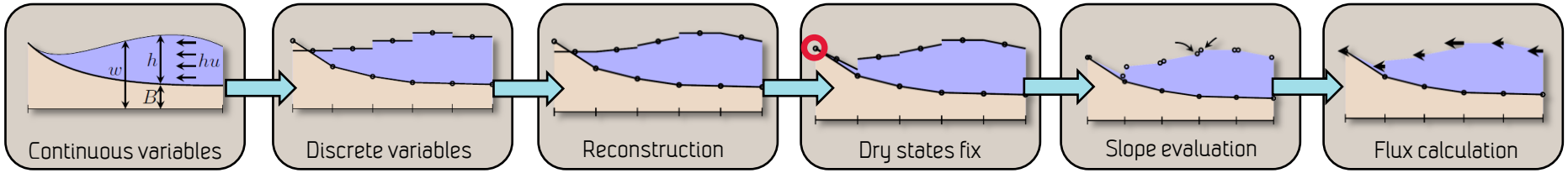
- Main idea:
  - Split up the computation into independent 2D blocks
  - Each block is similar to a node in an MPI cluster
  - Execute all blocks in parallel



Computing fluxes (spatial discretization)

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}$$

$$\frac{dQ_{ij}}{dt} = H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) - \left[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\right] - \left[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\right]$$

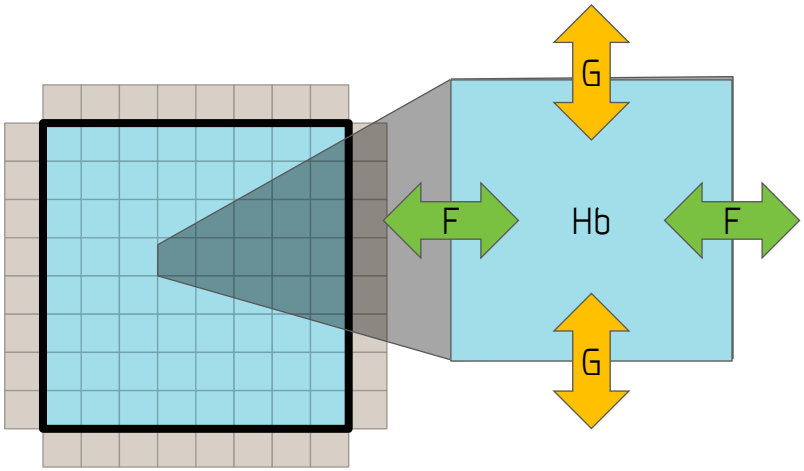


The flux calculation is a set of stencil operations:

1. slope reconstruction, **2.** dry states fix, **3.** slope evaluation and **4.** flux calculation.

Computing fluxes

$$\frac{dQ_{ij}}{dt} = H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) - \left[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\right] - \left[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\right]$$



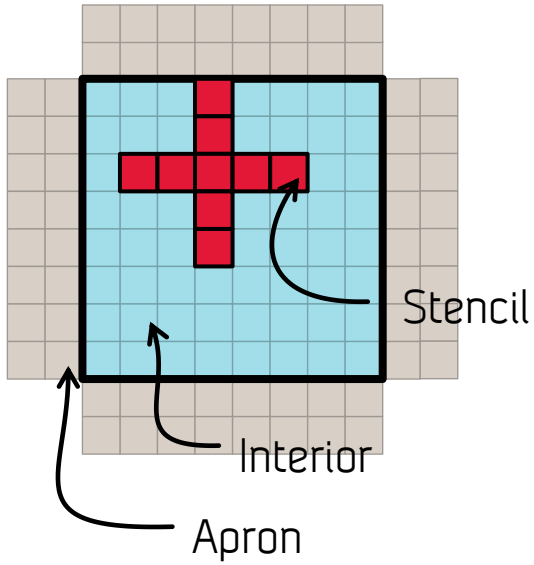
- The fluxes, F and G, are computed for each cell interface
- The source term, Hb, is computed for each cell
- Shared memory is used to limit data traffic and reuse data

Reusing data and results

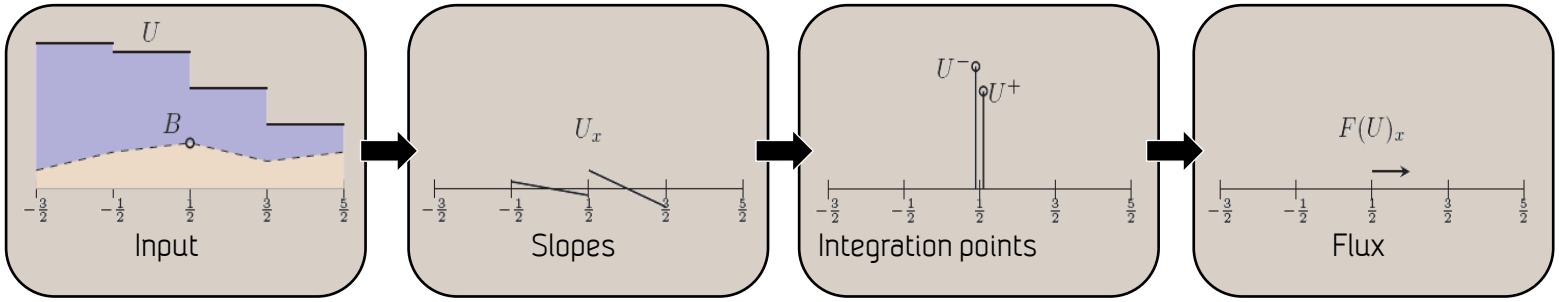
- Shared memory
  - Shared memory is a programmer controlled cache on the GPU
  - It is small, fast, and very useful for collaboration between threads within a block

- We can read in the physical variables into shared memory to save memory bandwidth
- We can let each thread compute the flux across the south and west interface, and store the flux in shared memory to save computations

```
//Declare a shared variable
__shared__ F[block_width][block_width];
...
//Compute the flux and store in shared memory
float f_west  = computeFluxWest(...);
F[ty][tx] = f_west;
__syncthreads();

//Use the results computed by other threads
float r = (F[ty][tx]  - F[ty][tx+1]) / dx
```



Per interface fluxes



- The flux for a single interface is computed from four input cells
  1. Reconstruct a slope within each cell (flux limited derivative)
  2. Evaluate the physical quantity at the cell interface from the right and left cell
  3. If there is a difference, compute the flux using the central upwind flux function

- n threads, and n+1 interfaces
  - one warp performs extra calculations!
  - Alternative is one thread per stencil operation, bu t that gives many idle threads, and extra register pressure

Slope reconstruction

- The slope is reconstructed using a slope limiter (generalized minmod in our case)
  - Compute the forward, backward, and central difference approximation to the derivative
  - Choose the least steep slope, or zero if signs differ
- Branching gives divergent code paths
  - Use branchless implementation (2007)
  - Much faster than naïve approach

$$\mathrm{MM}(a,b,c) = \begin{cases} \min(a,b,c), & \{a,b,c\} > 0 \\ \max(a,b,c), & \{a,b,c\} < 0 \\ 0, & \end{cases}$$

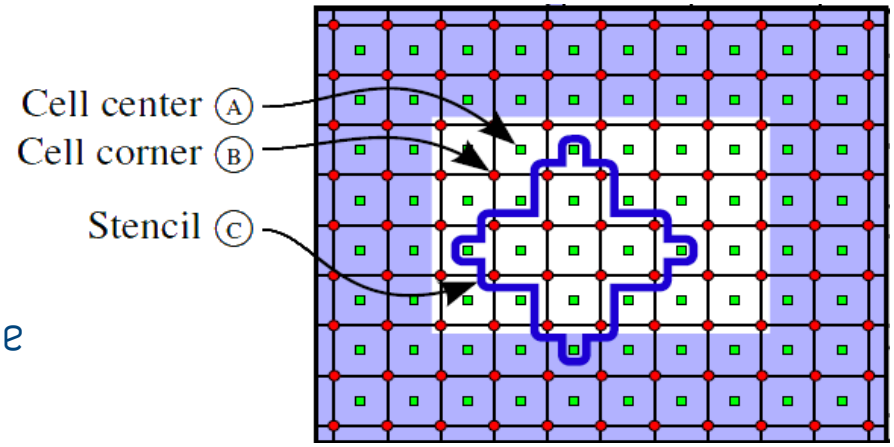
```
float minmod(float a, float b, float c) {
  return 0.25f*sign(a)
    *(sign(a) + sign(b))
    *(sign(b) + sign(c))
    *min( min(abs(a), abs(b)), abs(c) );
}
```

(2007) T. Hagen, M. Henriksen, J. Hjelmervik, and K.-A. Lie.
How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine.
Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF, (211–264). Springer Verlag, 2007.

Flux kernel – Block size

- Choosing the "correct" block size is difficult and affects performance dramatically

- Our block size is 16x14:
  - Warp size: multiple of 32
  - Shared memory use: close to 16 KB
  - Occupancy
    - Use 48 KB shared mem, 16 KB cache
    - Three resident blocks
    - Trades cache for occupancy
  - Fermi cache
  - Global memory access



Maximum timestep

- The maximum allowable timestep is related to the largest wave speed in the domain

$$\Delta t \leq \tfrac{1}{4} \min\left\{ \Delta x / \max_\Omega |u \pm \sqrt{gh}|, \Delta y / \max_\Omega |v \pm \sqrt{gh}| \right\}$$

- The wave speeds are computed per cell interface, and we need to find the maximum
  - This is called a reduction

- We follow the CUDA SDK reduction example with some modifications
  1. First, perform in-block shared memory reduction in the flux kernel to reduce 17x15 wave speeds to 1.
  2. Run a separate reduction pass "identical" to the SDK example

Reduction

- Reduction is a memory bound operation: We need to approach peak memory bandwidth

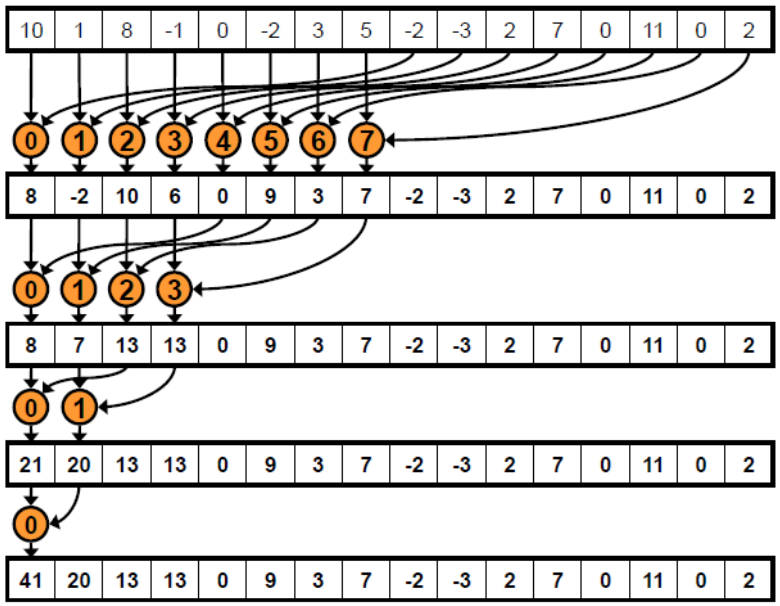
- Launch one block that covers all the elements we want to reduce
  1. Reduce from n elements to n_threads elements by striding through global memory
     - Use many threads to saturate memory bus (Littles law)
     - Striding pattern gives coalesced reads

```
__global__ void DtKernel(float* L_ptr, unsigned int n) {
 __shared__ float data[threads];
 volatile float* sdata = &data[0];
 unsigned int tid = threadIdx.x;

 //Reduce to "threads" elements
 sdata[tid] = FLT_MAX;
 for (unsigned int i=tid; i<n; i += threads) {
  sdata[tid] = min(sdata[tid], L_ptr[i]);
 }
 __syncthreads();
```

Reduction

2. Reduce from n_threads elements to 1 element using shared memory

```
if (threads >= 512) {
  if (tid < 256) sdata[tid] = min(sdata[tid], sdata[tid + 256]);
  __syncthreads();
}
...
if (threads >= 64) {
  if (tid < 32) sdata[tid] = min(sdata[tid], sdata[tid + 32]);
  __syncthreads();
}
if (tid < 16) {
  sdata[tid] = min(sdata[tid], sdata[tid + 16]);
  sdata[tid] = min(sdata[tid], sdata[tid + 8]);
  sdata[tid] = min(sdata[tid], sdata[tid + 4]);
  sdata[tid] = min(sdata[tid], sdata[tid + 2]);
  sdata[tid] = min(sdata[tid], sdata[tid + 1]);
}
```



Temporal Discretization (Evolving in time)

$$\frac{dQ_{ij}}{dt} = H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) - \left[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\right] - \left[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\right]$$

Gather all known terms into R

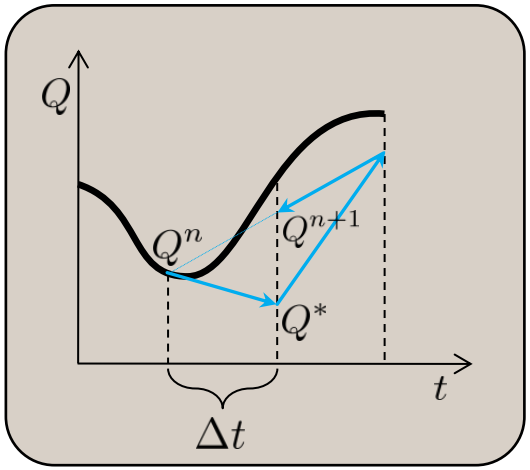
$$\frac{dQ_{ij}}{dt} = H_f(Q_{ij}) + R(Q)_{ij}$$

Use second order Runge-Kutta to solve the ODE

$$Q^*_{ij} = \left[Q^n_{ij} + \Delta t R(Q^n)_{ij}\right] / \left[1 + \Delta t \tilde{H}_f(Q^n_{ij})\right]$$

$$Q^{n+1}_{ij} = \left[\tfrac{1}{2}Q^n_{ij} + \tfrac{1}{2}\left[Q^*_{ij} + \Delta t R(Q^*)_{ij}\right]\right] / \left[1 + \tfrac{1}{2}\Delta t \tilde{H}_f(Q^*_{ij})\right]$$

$$\Delta t \leq \tfrac{1}{4}\min\left\{\Delta x / \max_\Omega \left|u \pm \sqrt{gh}\right|, \Delta y / \max_\Omega \left|v \pm \sqrt{gh}\right|\right\}$$



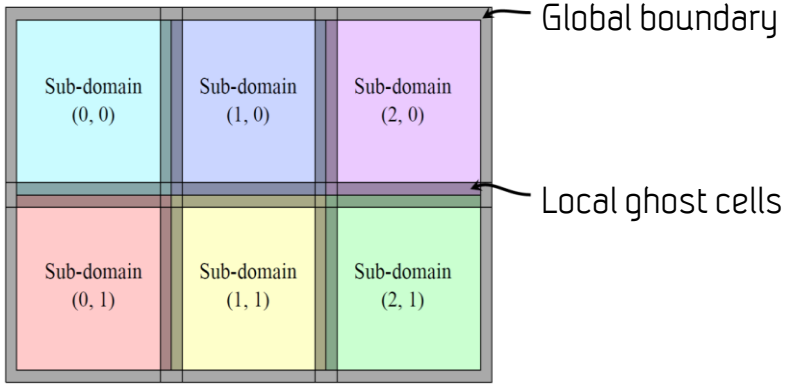
Runge-Kutta kernel

- The Runge-Kutta kernel essentially performs a simple forward Euler integration step, and also computes the friction source term
- Embarrassingly parallel computation: no dependencies on other threads
- Memory bound, but reaches good memory throughput: coalesced reads etc.

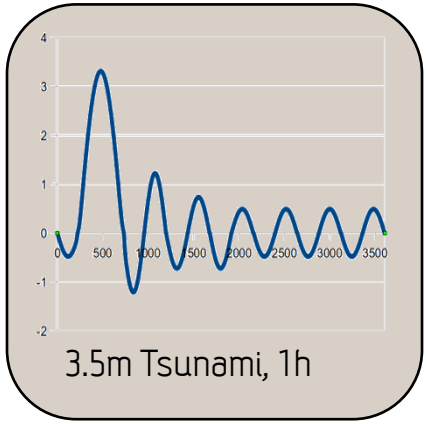
```
__global__ RKKernel(…) {
  //Read in U1, R1, dt, etc.
  …

  //Compute the bed friction term alpha
  alpha = dt*g*sqrt(u*u+v*v)/hc^2;

  //Evolve the solution
  Q1 =  U1 + dt*R1;
  Q2 = (U2 + dt*R2)/(1.0f+alpha);
  Q3 = (U3 + dt*R3)/(1.0f+alpha);

  //Write out to global memory
  …
}
```

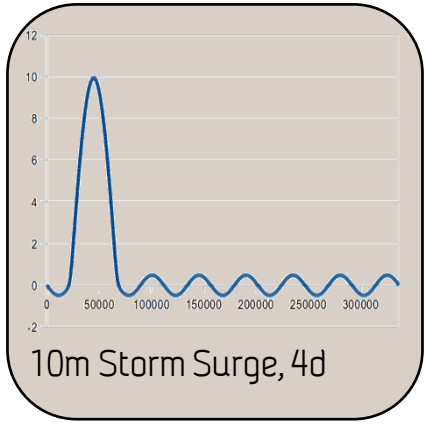
Boundary conditions kernel

- Ghost cells used for boundary
  - Fixed inlet / outlet discharge
  - Fixed depth
  - Reflecting
  - Outflow/Absorbing



- Can also supply hydrograph
  - Tsunamies
  - Storm surges
  - Tidal waves



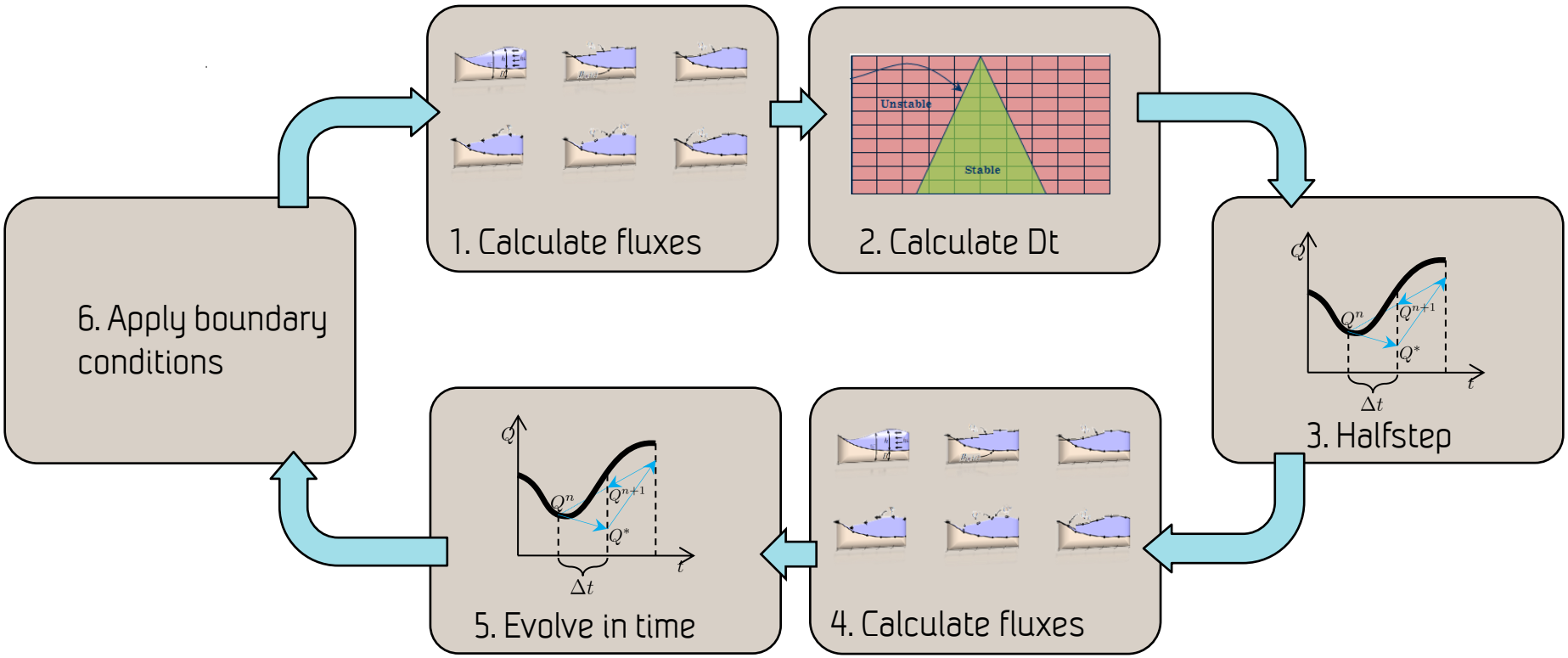


Boundary conditions kernel

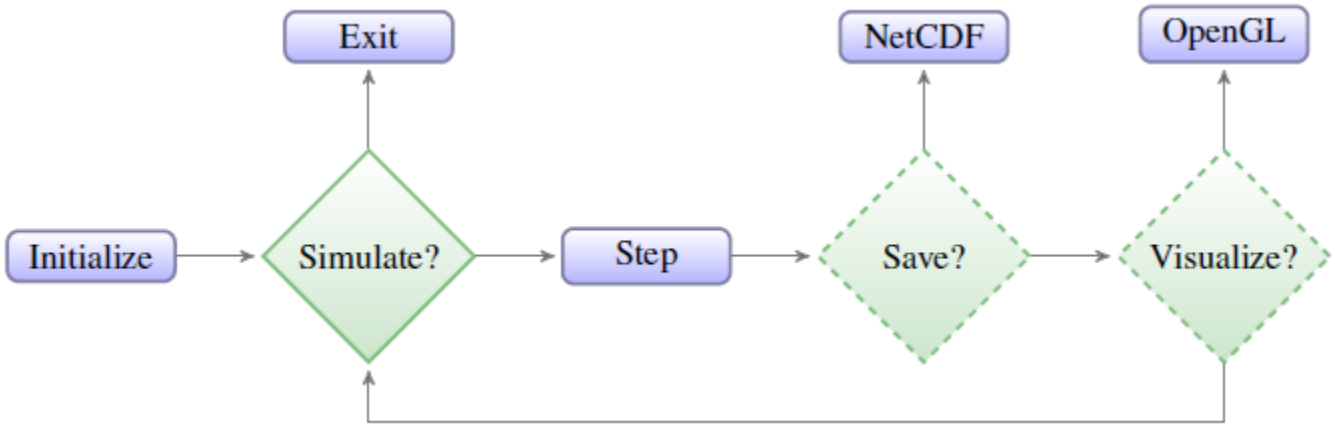
- Similar to CUDA SDK reduction sample, using templates:
  - One block sets all four boundaries
  - Boundary length (›64, ›128, ›256, ›512)
  - Boundary type ("none", reflecting, fixed depth, fixed discharge, absorbing outlet)
  - In total: 4*5*5*5*5 = 2500 kernel realizations!

```
switch(block.x) {
  case 512: BCKernelLauncher‹512, N, S, E, W›(grid, block, stream); break;
  case 256: BCKernelLauncher‹256, N, S, E, W›(grid, block, stream); break;
  case 128: BCKernelLauncher‹128, N, S, E, W›(grid, block, stream); break;
  case  64:  BCKernelLauncher‹  64, N, S, E, W›(grid, block, stream); break;
}
```

Summary: A Simulation Cycle



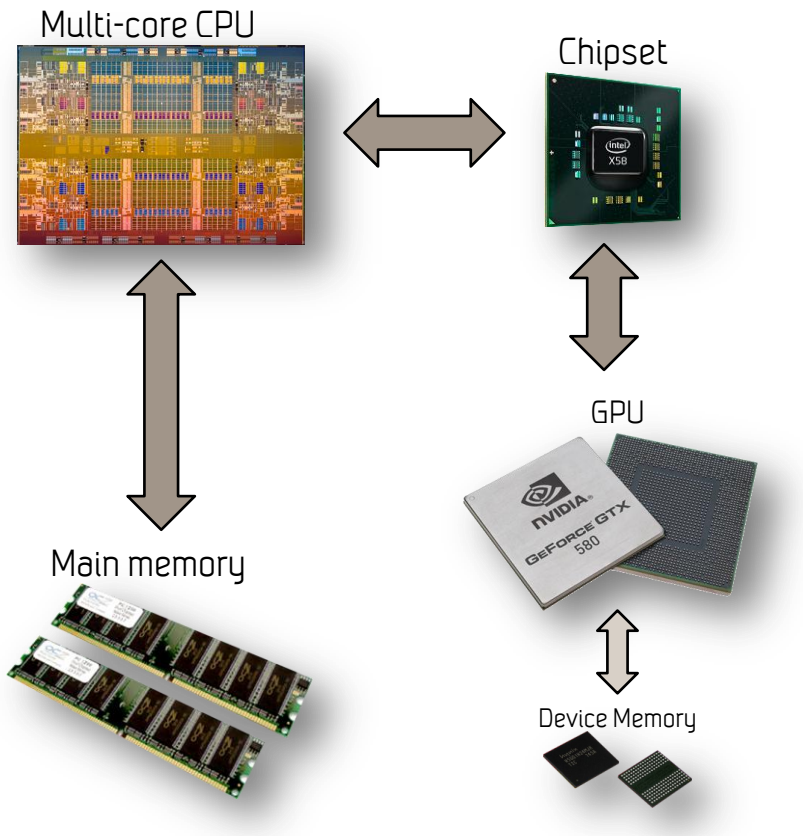
Visualization



- Real-time visualization of results is important
  - Get an initial feel for the solution
  - Check parameters and validity visually
  - …
- We have implemented direct OpenGL visualization

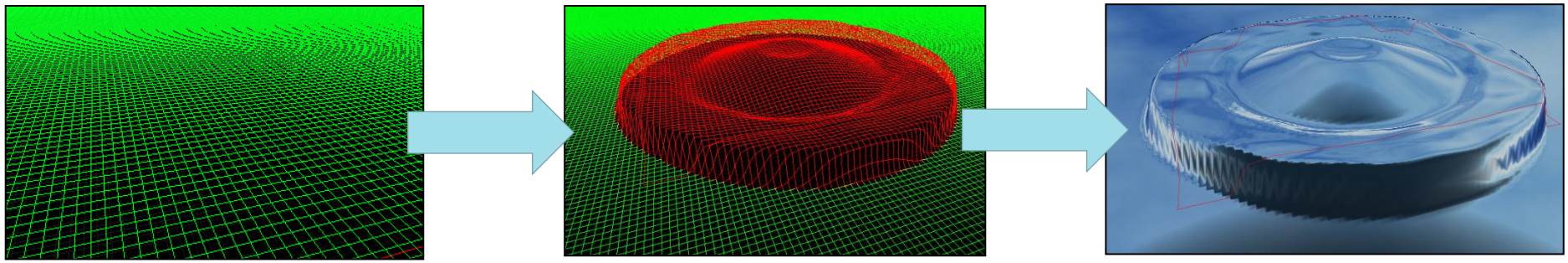
Direct OpenGL Visualization

- OpenGL is a graphics API for hardware accelerated rendering
- Cuda runs on the GPU, and so does OpenGL
- We can visualize results directly, by doing a fast GPU (CUDA) to GPU (OpenGL) copy



Rendering

- With the data in OpenGL, it is "simple" to visualize it
  - Visualization in OpenGL is at least a course by itself… I teach two of them☺
- Start with a 2D mesh
  - Displace vertices in the height dimension and generate a normal
  - Render each pixel using Fresnel shading

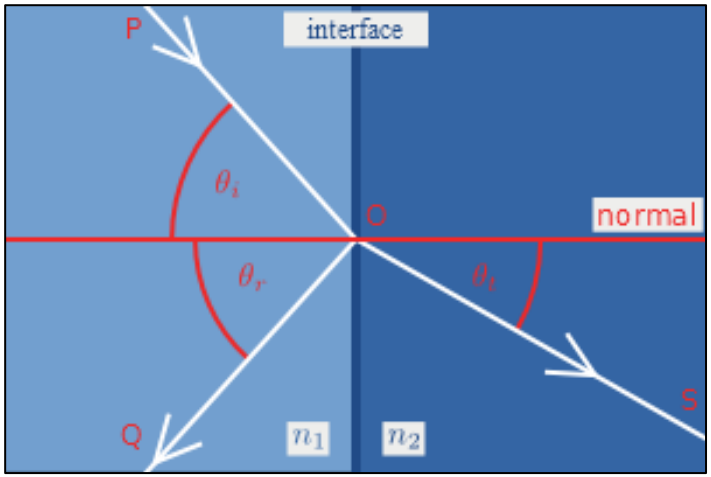


Vertex displacement

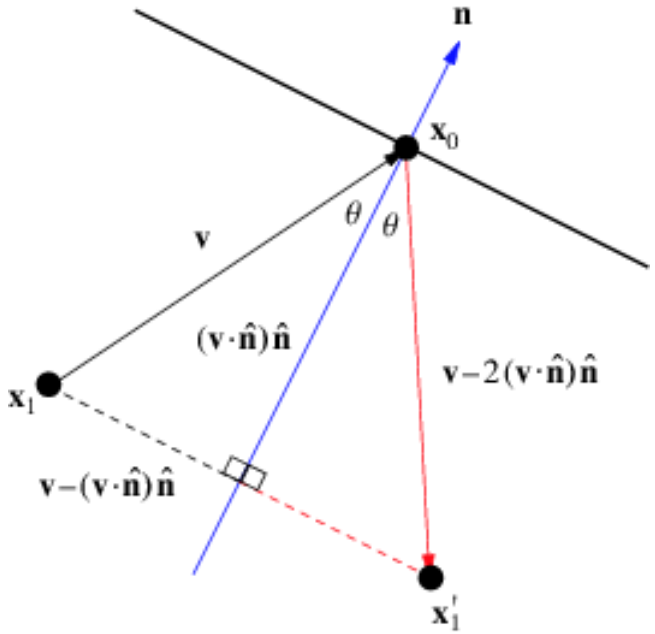
GLSL is used to programmatically move the vertex along the y-axis (up)

```
void main(void) {
        //Look up height from texture
        float c = texture2D(height_map,  gl_MultiTexCoord0.xy).x;

        //Displace the vertex according to height map
        vec4 position = gl_Vertex;
        position.y = c*dz;

        //Set output vertex position
        gl_Position = gl_ModelViewProjectionMatrix*position;
}
```

Fresnel Equations



- We have an incoming ray, P, going from a medium with refraction index n1 (e.g., air) to a medium with refraction index n2 (e.g., water)
- We can find the angle of refraction by using Snell's law $n_1 \sin\theta_i = n_2 \sin\theta_t$
- The reflected ray is easily found using the dot product



- Then calculate level of refraction using Schlick's approximation

Fresnel shading

```
void main() {
        const float eta_air = 1.000293;
        const float eta_water = 1.3330;

        vec3 texcoord_reflect = reflect(-v, n);
        vec3 texcoord_refract = refract(-v, n, eta);

        float fresnel = R0 + (1.0-R0)*pow((1.0-dot(v, n)), 5.0);

        vec4 reflect = texture(skybox, texcoord_reflect);
        vec4 refract = texture(skybox, texcoord_refract);

        out_color = mix(refract, reflect, fresnel);
}
```

Copying data from CUDA to OpenGL

- CUDA works with pitched linear memory, OpenGL works with textures
- We "map" the OpenGL texture to a CUDA pointer, and copy simulation results from CUDA to the mapped pointer

```
//Initialization
cudaGraphicsGLRegisterImage(&resource,
   texture, GL_TEXTURE_2D,
   cudaGraphicsMapFlagsWriteDiscard);

//In render loop
cudaGraphicsMapResources(1, resource, 0);
cudaGraphicsSubResourceGetMappedArray(&array, resource, 0, 0);
cudaMemcpy2DToArray(array, 0, 0, cuda_ptr, cuda_pitch,
   width_in_bytes, height,
   cudaMemcpyDeviceToDevice);
cudaGraphicsUnmapResources(1, resource, 0);
```

Verifying results

Single Versus Double Precision

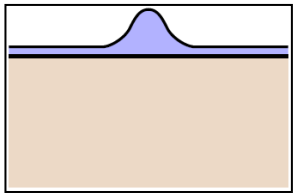
Given erroneous data, double precision
calculates a more accurate (but still wrong) answer

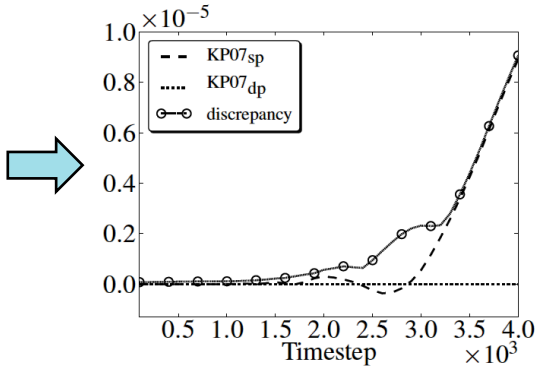
Single precision benefits:

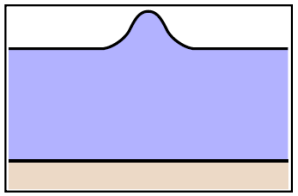
- Uses *half* the storage space
- Uses *half* the bandwidth
- Executes (at least) *twice* as fast

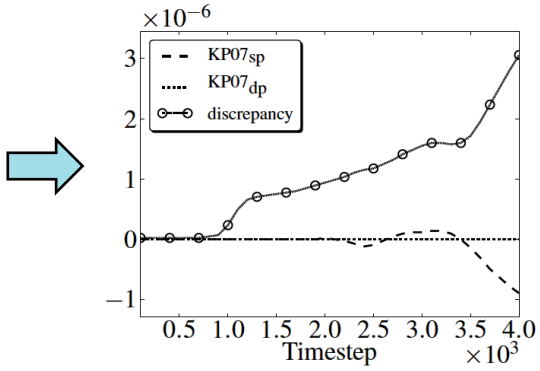
Single Versus Double Precision Example

- Three different test cases
  - Low water depth (wet-wet)
  - High water depth (wet-wet)
  - Synthetic terrain with dam break (wet-dry)

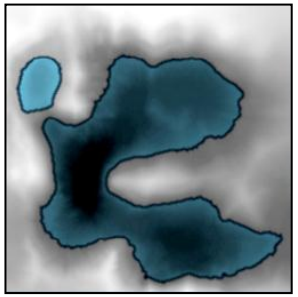


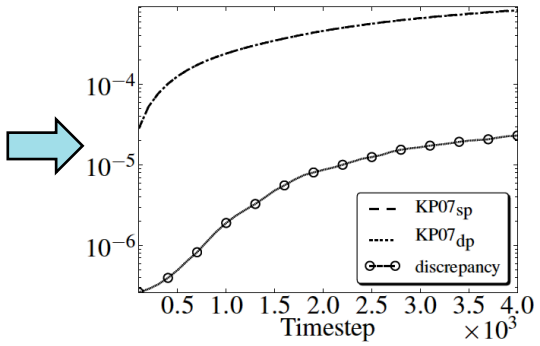






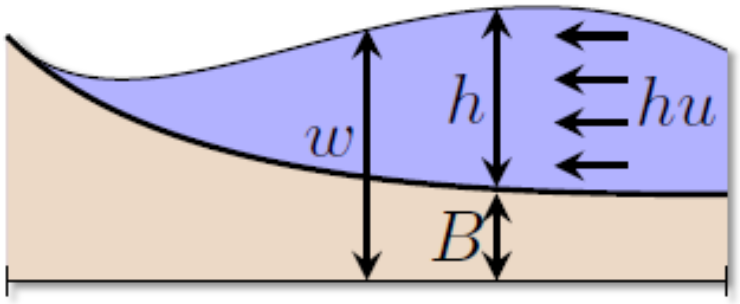
- Conclusions:
  - Loss in conservation on the order of machine epsilon
  - Single precision gives larger error
  - Errors related to the wet-dry front is more than an order of magnitude larger (model error)
  - **Single precision is sufficiently accurate for this scheme**





More on Accuracy

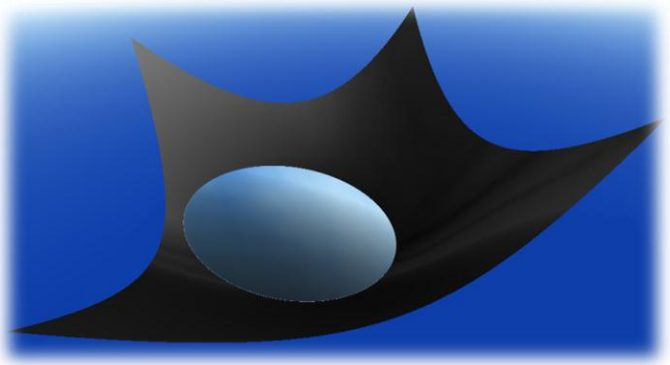
- We were experiencing large errors in conservation of mass for special cases
- The equations is written in terms of w = B+h to preserve "lake at rest"



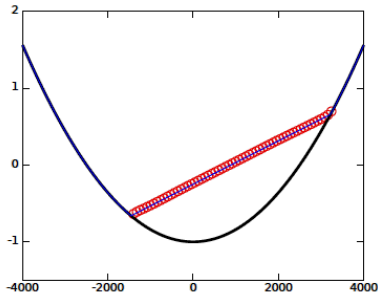
- Large B, and small h
  - The scale difference gives major floating point errors (h flushed to zero)
  - Even double precision is insufficient
- Solve by storing only h, and reconstruct w when required!
  - Single precision sufficient for most real-world cases
  - Always store the quantity of interest!

Verification: Parabolic basin

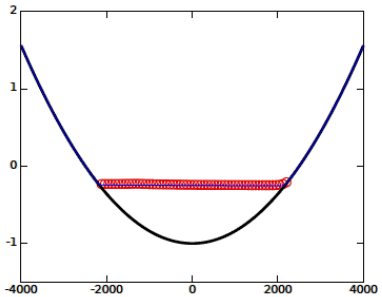
- Single precision is sufficient, but do we solve the equations?
- Test against analytical 2D parabolic basin case (Thacker)
  - Planar water surface oscillates
  - 100 x 100 cells
  - Horizontal scale: 8 km
  - Vertical scale: 3.3 m



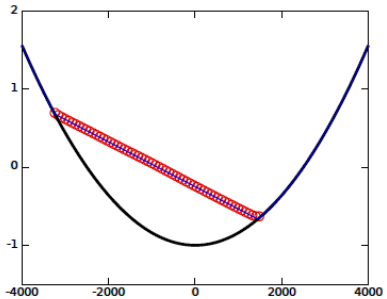
- Simulation and analytical match well
  - But, as most schemes, growing errors along wet-dry interface



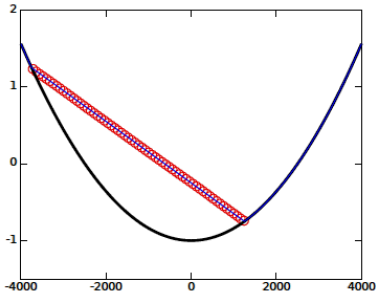
(a) $t \approx 1\pi/8\omega$ (1392.85 s)



(b) $t \approx 2\pi/8\omega$ (2787.95 s)

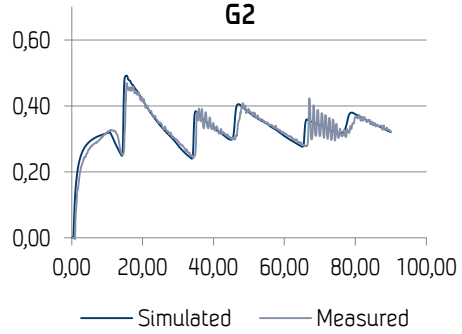


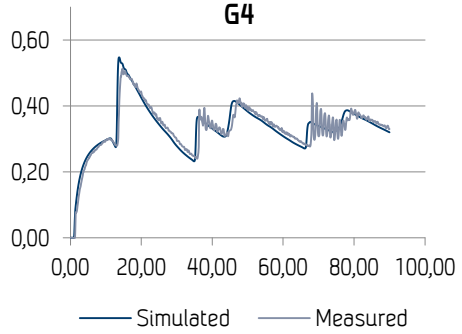
(c) $t \approx 3\pi/8\omega$ (4198.64 s)

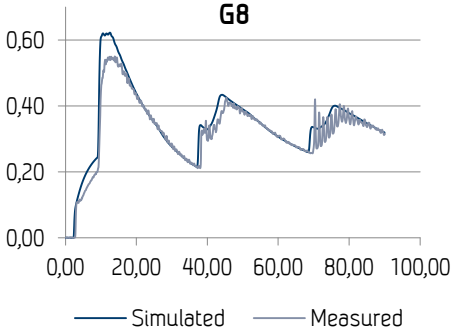


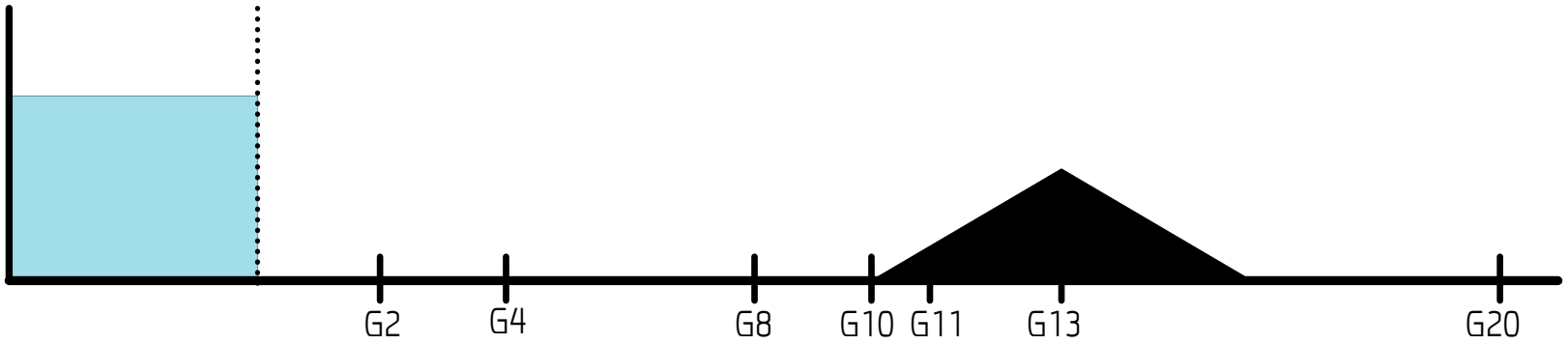
(d) $t \approx 4\pi/8\omega$ (5559.27 s)

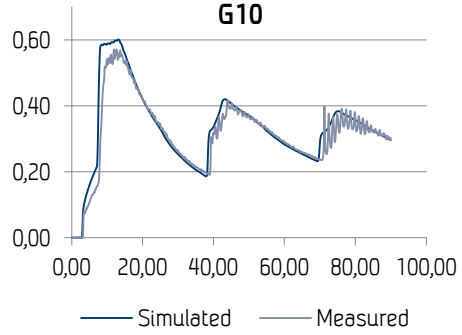
1D Validation: Flow over Triangular bump (90s)

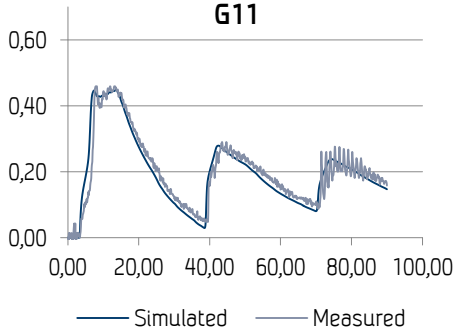


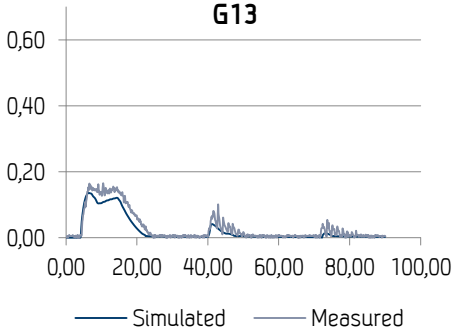


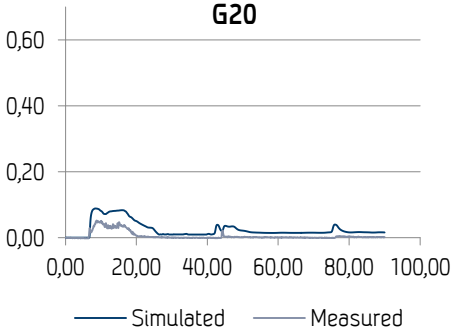






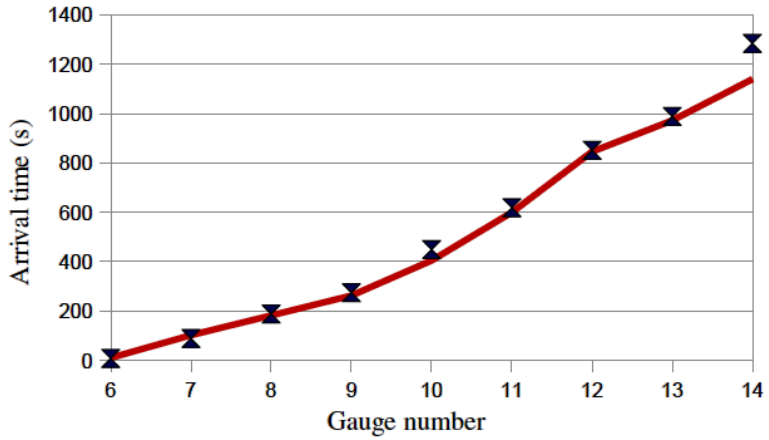






Validation: Barrage du Malpasset

- We model the equations correctly, but can we model real events?
- South-east France near Fréjus: Barrage du Malpasset
  - Double curvature dam, 66.5 m high, 220 m crest length, 55 million m^3
  - Bursts at 21:13 December 2nd 1959
    - Reaches Mediterranean in 30 minutes (speeds up-to 70 km/h)
    - 423 casualties, $68 million in damages
  - Validate against experimental data from 1:400 model
    - 482 000 cells (1099 x 439 cells)
    - 15 meter resolution
- **Our results match experimental data very well**
  - Discrepancies at gauges 14 and 9 present in most (all?) published results



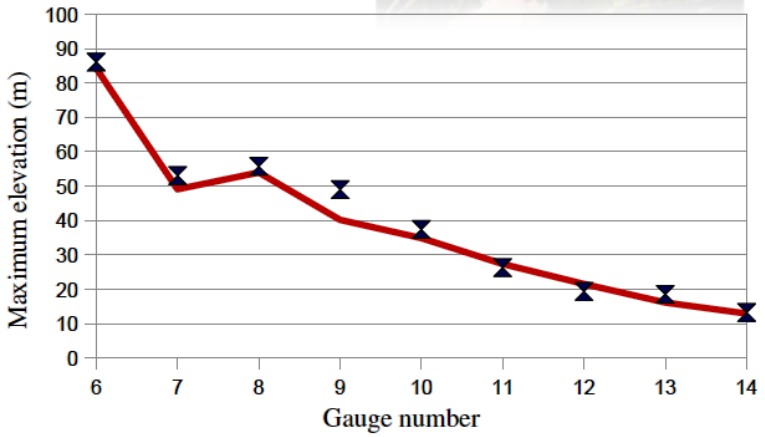


Image from google earth, mes-ballades.com

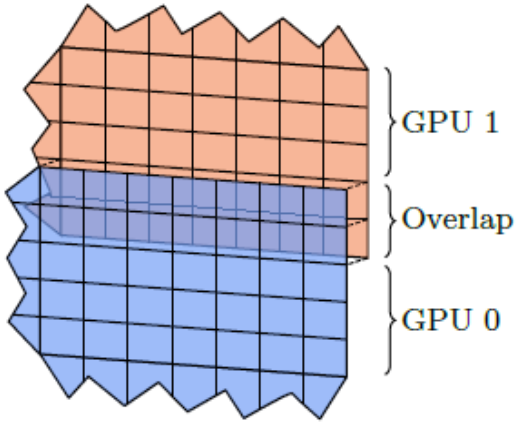
Higher performance

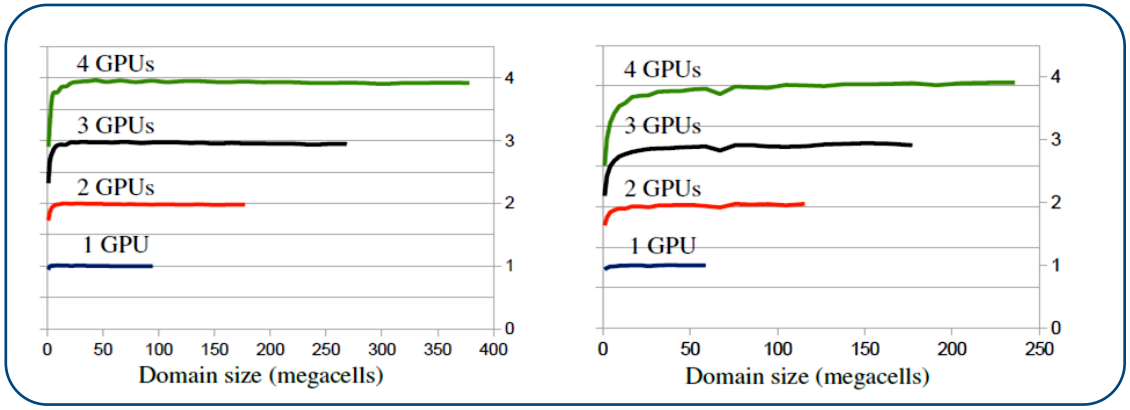
Higher performance

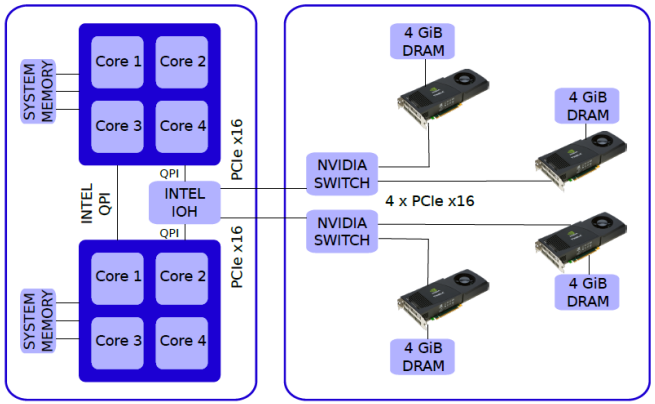
- Tuning GPU code for optimal performance gets you only so far
- The real big performance improvements come from algorithmic improvements
- We have looked at different ways of optimizing performance
  - Multi-GPU
  - Early exit and sparse computations
  - Adaptive local refinement
  - Mixed-order schemes

Multi-GPU simulations

- Because we have a finite domain of dependence, we can create independent partitions of the domain and distribute to multiple GPUs
- Modern PCs have up-to four GPUs
- Near-perfect weak and strong scaling



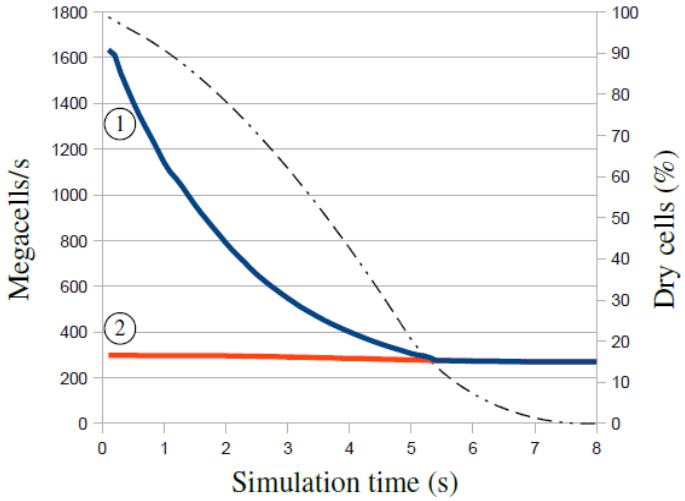




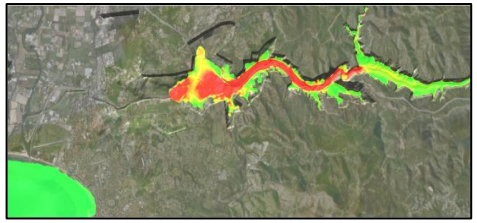
Collaboration with Martin L. Sætra

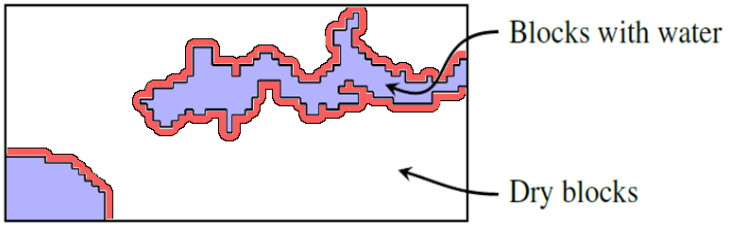
Early exit optimization

- Observation: Many dry areas do not require computation
  - Use a small buffer to store wet blocks
  - Exit flux kernel if nearest neighbors are dry



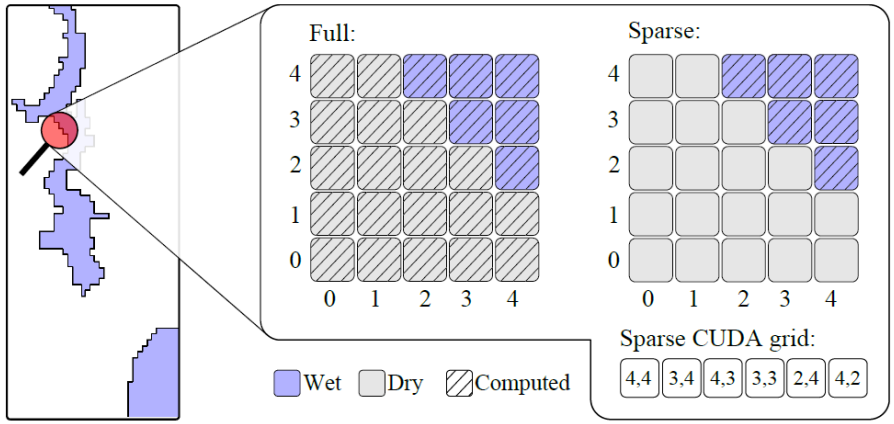
- Up-to 6x speedup (mileage may vary)
  - Blocks still have to be scheduled
  - Blocks read the auxiliary buffer
  - One wet cell marks the whole block as wet





Sparse domain optimization

- The early exit strategy launches too many blocks
- Dry blocks should not need to check that they are dry!



Sparse Compute:

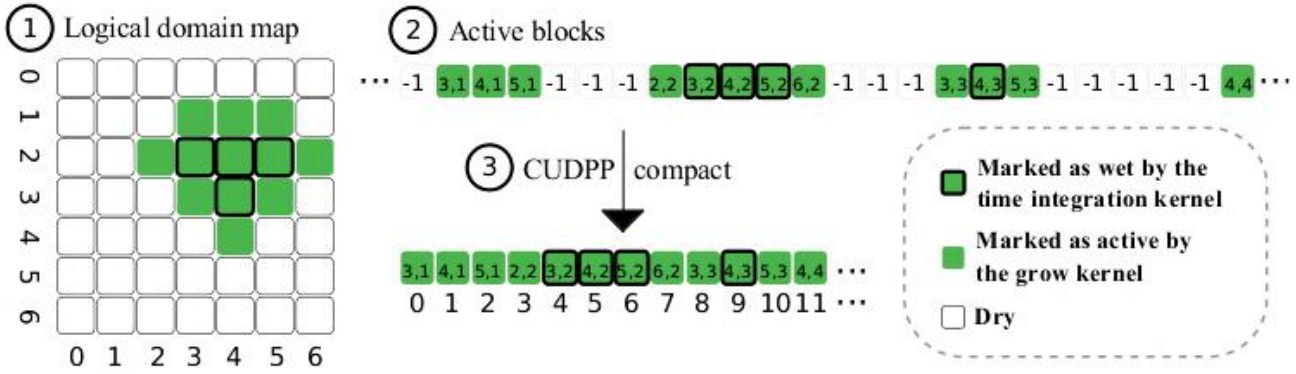
Do not perform any computations on dry parts of the domain

Sparse Memory:

Do not save any values in the dry parts of the domain

Ph.D. work of Martin L. Sætra

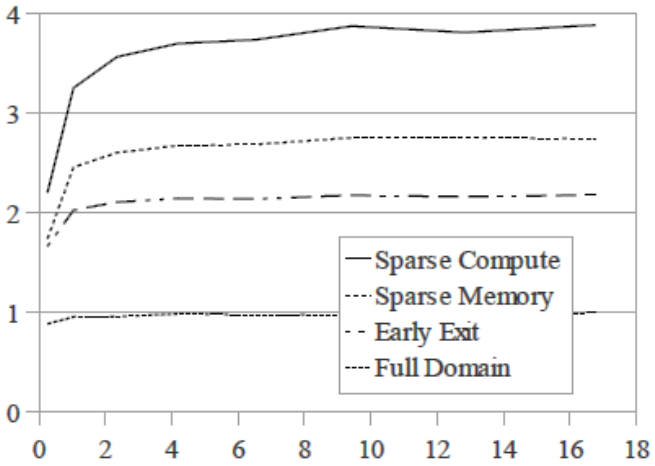
Sparse domain optimization



1. Find all wet blocks
2. Grow to include dependencies
3. Sort block indices and launch the required number of blocks

- Similarly for memory, *but it gets quite complicated*…

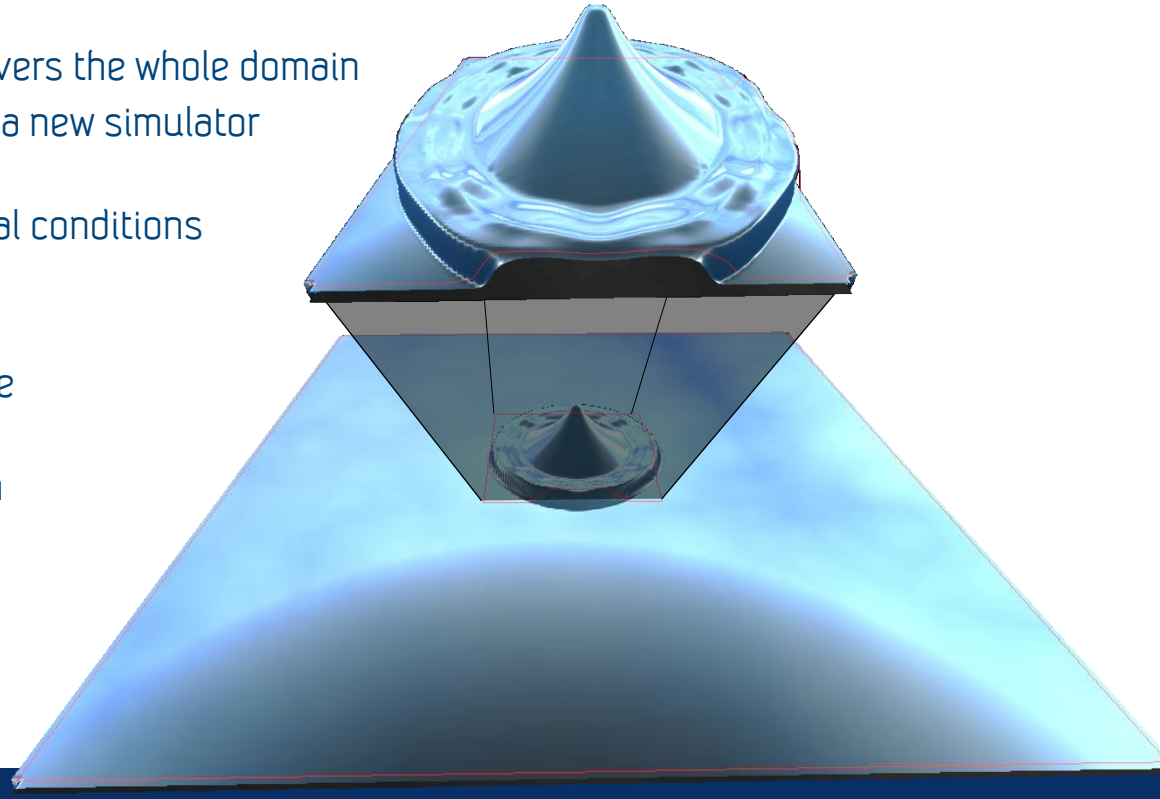
- 2x improvement over early exit (mileage may vary)!



Comparison using an average of 26% wet cells

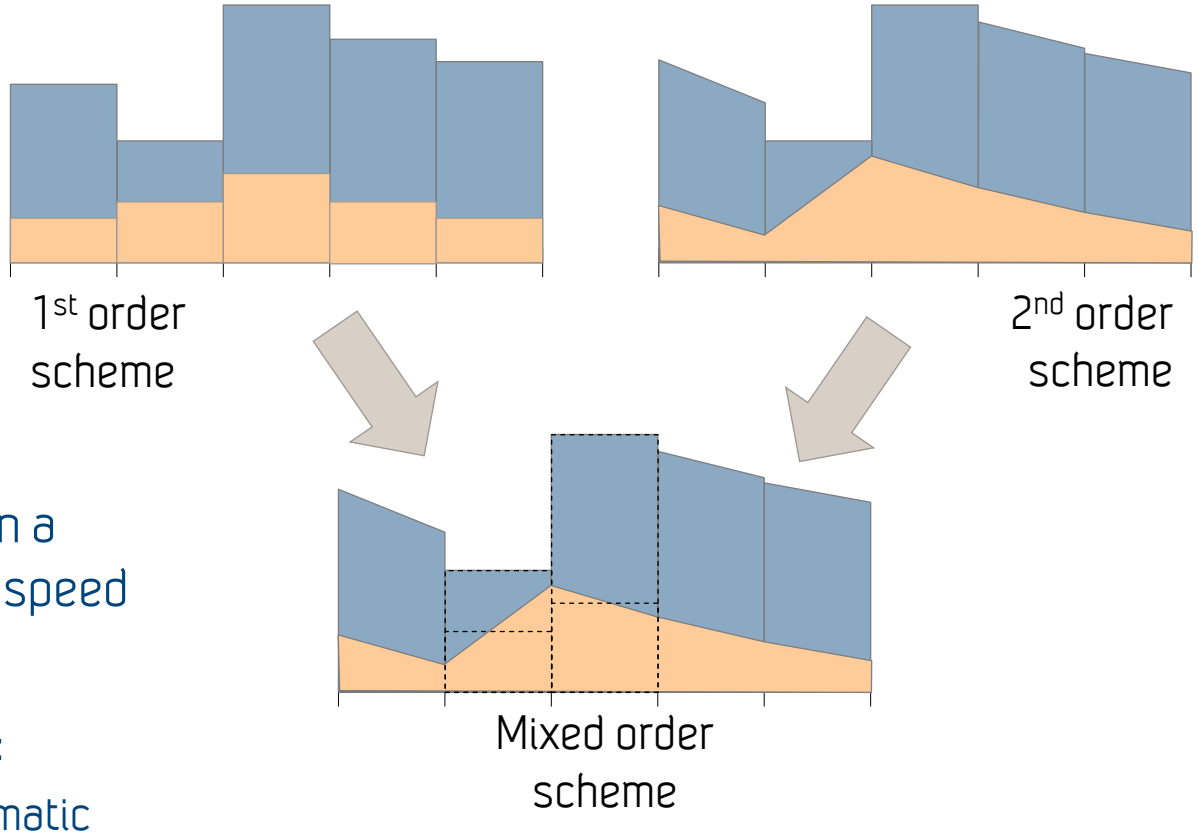
Adaptive local refinement

- It is often most interesting to have high resolution in only certain areas
- Adaptive local refinement performs refinement only where it is needed!
  - This saves both memory and computations
- Simple idea:
  - Have a coarse simulator that covers the whole domain
  - For each area of interest, create a new simulator with higher resolution
    - Use the coarse grid as initial conditions
    - Use coarse grid fluxes as boundary conditions
    - Average the values from the fine grid to the coarse
    - Correct boundary condition fluxes for conservation
  - Use multiple refinement levels where needed



Collaboration with Martin L. Sætra

Mixed order schemes



- The small size of Dt is often a problem for the simulation speed
- Use a mixed order scheme:
  - Use first order for problematic interfaces
  - Use second order everywhere else

Summary

- We have looked at
  - Quick repetition on GPUs
  - Mapping mathematics to computations
  - The shallow water equations
  - Validity of computed results
  - Higher performance

- Most important take-home lessons:
  - It is extremely difficult to get everything right with GPUs:
    Always get it working before optimizing it
  - Micro-optimization of a single code will only get you so far:
    Doing algorithmic tricks often gives much higher performance gains