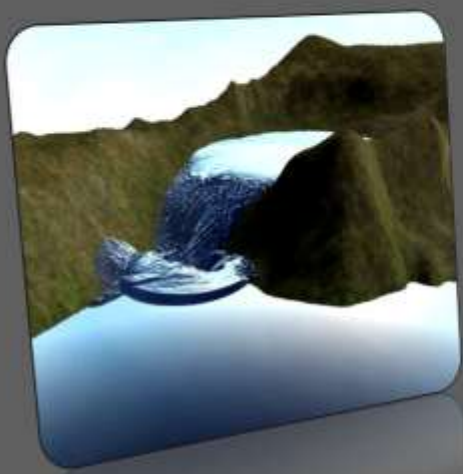
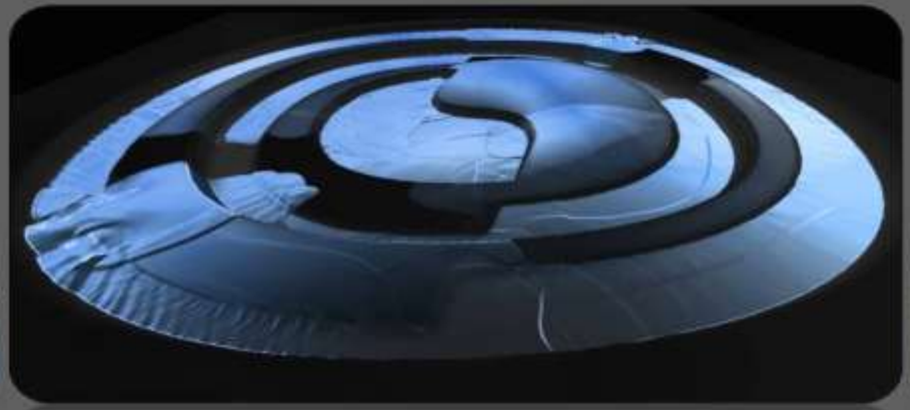
Compact Stencils for the Shallow Water Equations on Graphics Processing Units

Fenics '12
Simula
2012-06-07


André R. Brodtkorb <Andre.Brodtkorb@sintef.no>

Brief Outline

- **Introduction to Computing on GPUs**
- **The Shallow Water Equations**
- **Compact Stencils on the GPU**
- **Physical correctness**
- **Summary**

Introduction to GPU Computing

Long, long time ago, ...

1942: Digital Electric Computer
(Atanasoff and Berry)

1956

1947: Transistor
(Shockley, Bardeen, and Brattain)

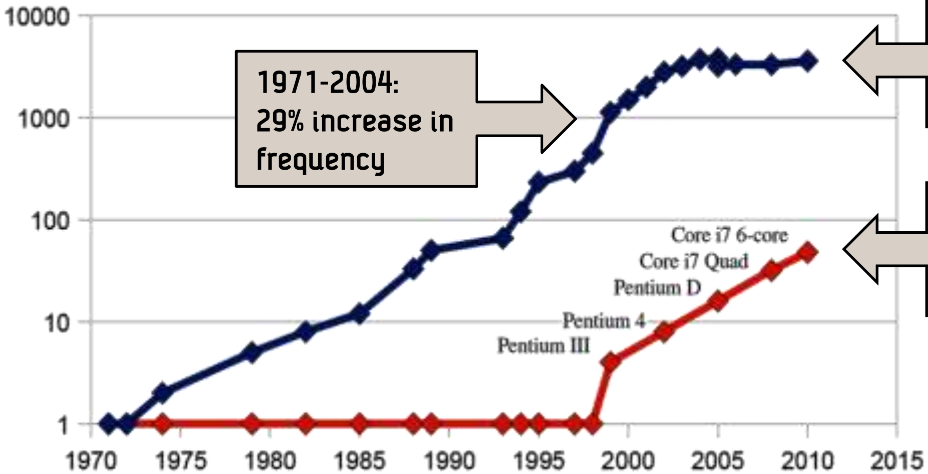
2000

1958: Integrated Circuit
(Kilby)

1971: Microprocessor
(Hoff, Faggin, Mazor)

1971- More transistors
(Moore, 1965)

The end of frequency scaling



2004-2011:
Frequency constant

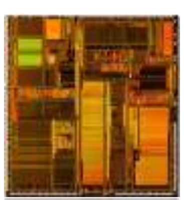
1999-2011:
25% increase in parallelism

A serial program uses 2% of available resources!

Parallelism technologies:
- Multi-core (8x)
- Hyper threading (2x)
- AVX/SSE/MMX/etc (8x)

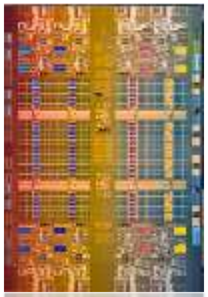
1971: Intel 4004,
2300 trans, 740 KHz

1982: Intel 80286,
134 thousand trans, 8 MHz



1993: Intel Pentium P5,
1.18 mill. trans, 66 MHz

2000: Intel Pentium 4,
42 mill. trans, 1.5 GHz

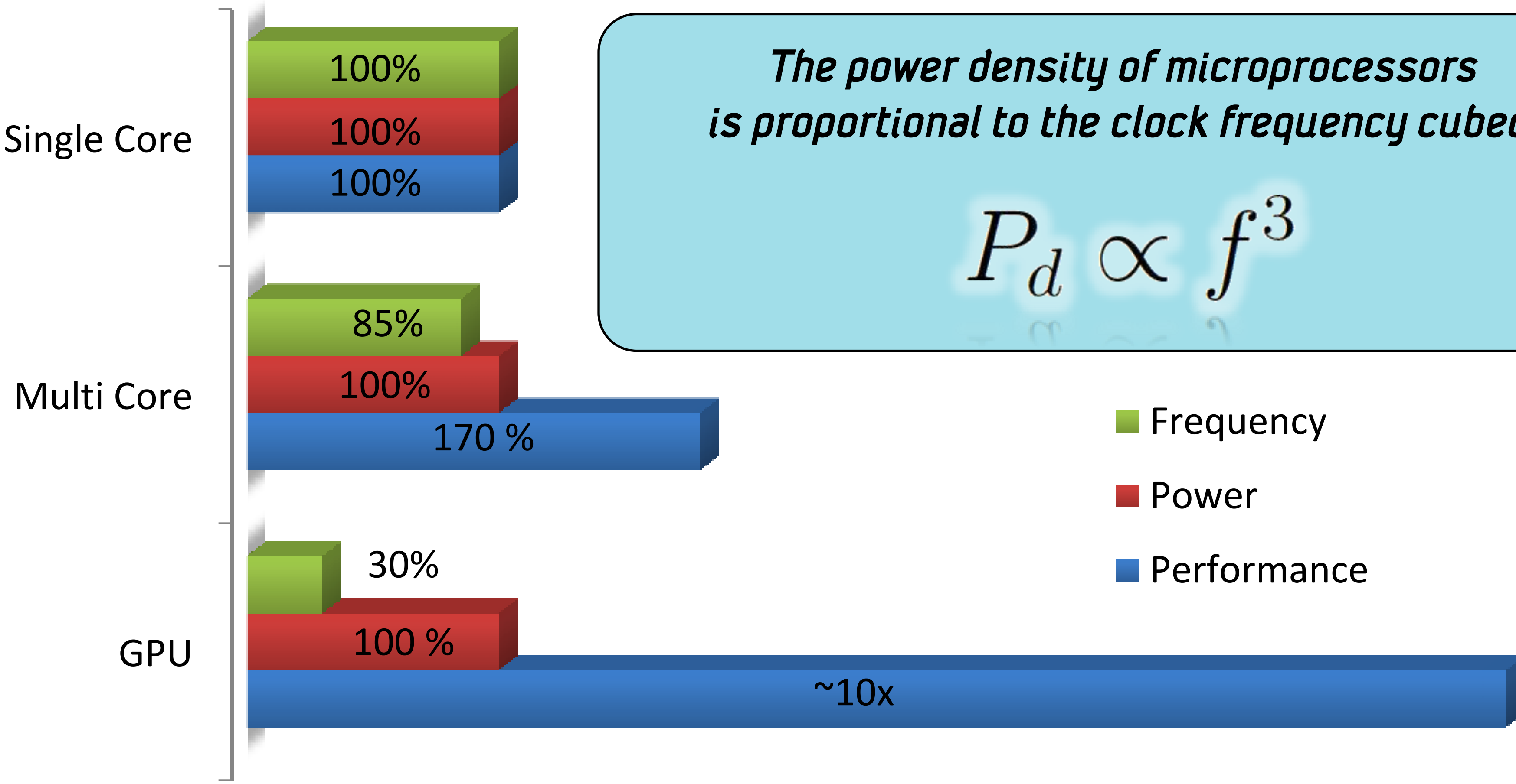


2010: Intel Nehalem,
2.3 bill. trans, 8 X 2.66 GHz

How does parallelism help?

The power density of microprocessors is proportional to the clock frequency cubed:

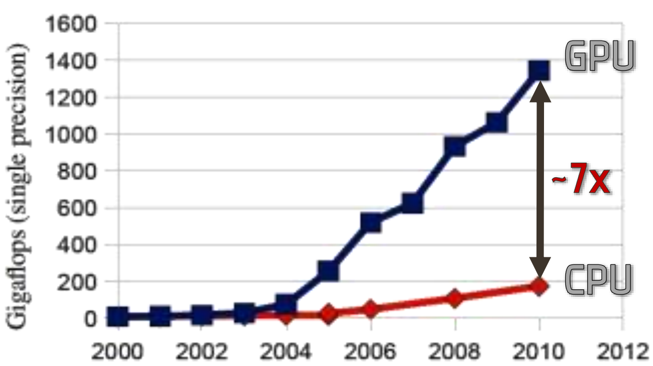
$$P_d \propto f^3$$



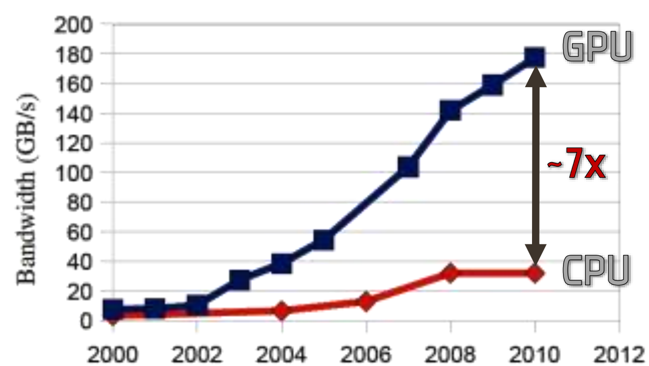
The GPU: Massive parallelism

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

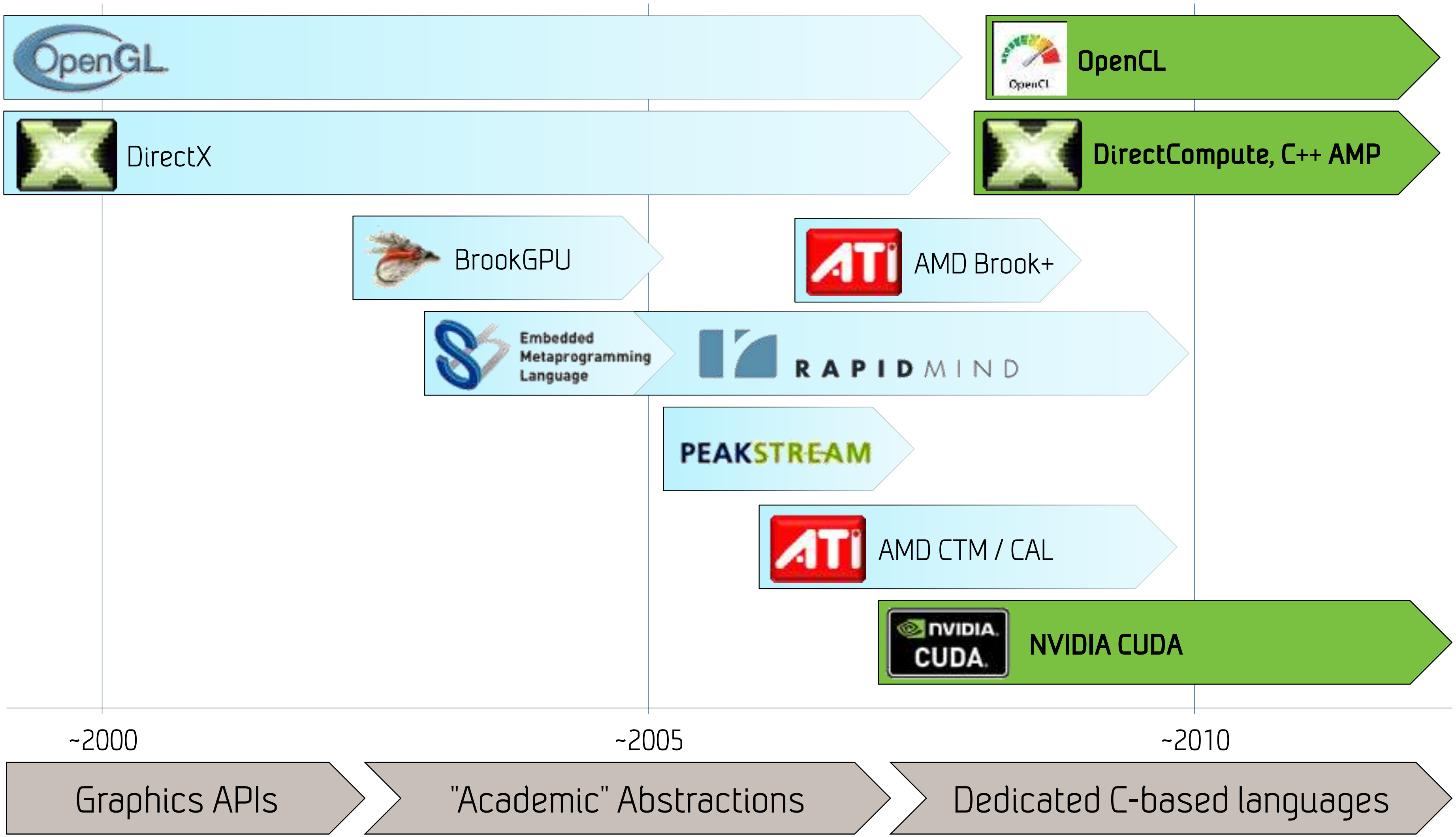
Performance



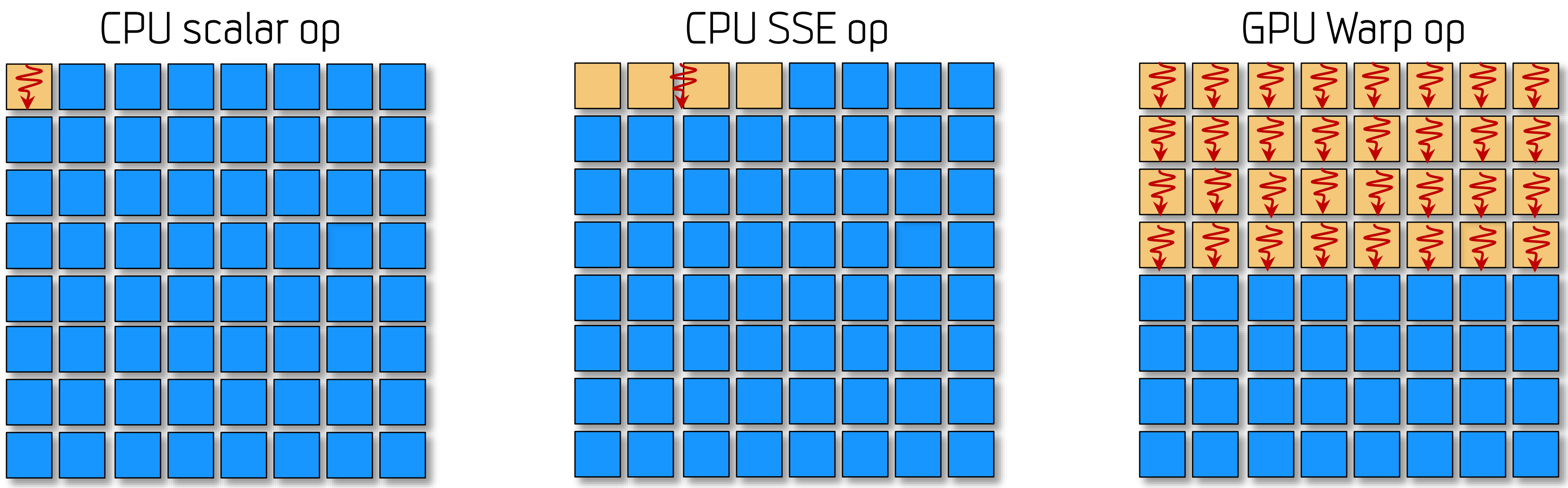
Memory Bandwidth



GPU Programming: From Academic Abuse to Industrial Use



GPU Execution mode

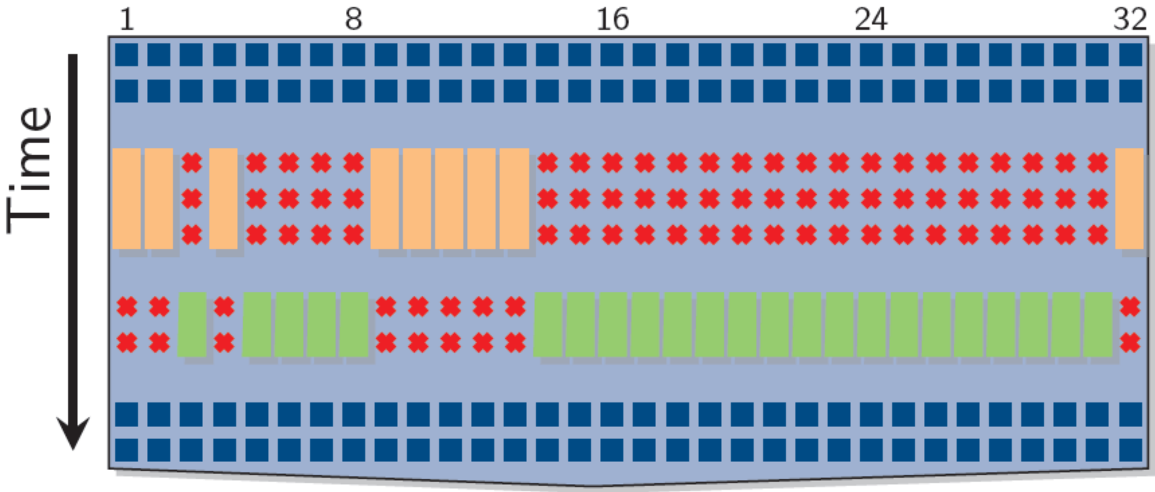


CPU scalar op
- 1 thread, 1 operand on 1 data element

CPU SSE op
- 1 thread, 1 operand on 2-4 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**

Warp Serialization and Masking




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

Hardware serializes and masks divergent code flow:

- Programmer is relieved of fiddling with element masks (which is necessary for SSE)
- But execution time is still the sum of branches taken
- Worst case:
    - All warp threads takes individual branches (1/32 perfomance)
- Thus, important to **minimize divergent code flow**!
    - Move conditionals into data, use min, max, conditional moves.

Example: Warp Serialization in Newton's Method

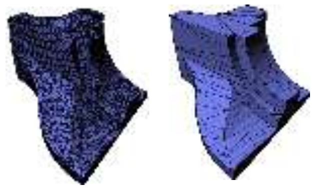
- First if-statement
  - Masks out superfluous threads
  - Not significant
- Iteration loop
  - Identical for all threads
- Early exit
  - Possible divergence
  - Only beneficial when all threads in warp can exit
- **Removing** early exit increases performance from 0.84ms to 0.69ms (kernel only)

```
__global__
void
newton(float* x,const float* a,const float* b,const float* c,int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if( i < N ) {
    const float la = a[i];
    const float lb = b[i];
    const float lc = c[i];
    float lx = 0.f;
    for(int it=0; it<MAXIT; it++) {
      float f = la*lx*lx + lb*lx + lc;
      if( fabsf(f) < 1e-7f) {
        break;
      }
      float df = 2.f*la*lx + lb;
      lx = lx - f/df;
    }
    x[i] = lx;
  }
}
```

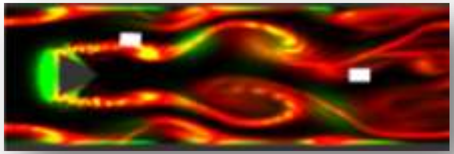
(But fails 7 of 1 000 000 times since multiple zeros isn't handled properly, but that is a different story ☺)

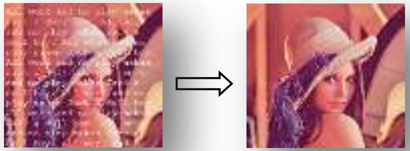
Examples of early GPU research

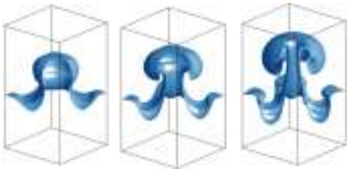

Registration of medical data (~20x)

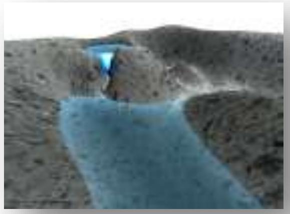

Preparation for FEM (~5x)

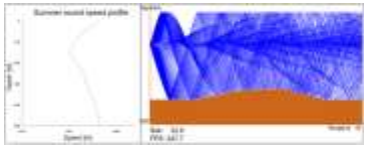

Self-intersection (~10x)

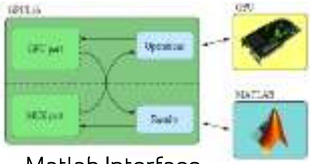

Fluid dynamics and FSI (Navier-Stokes)

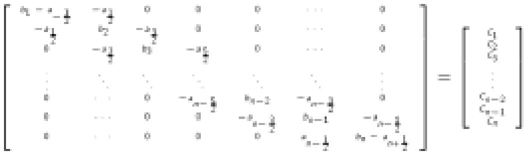

Inpainting (~400x matlab code)

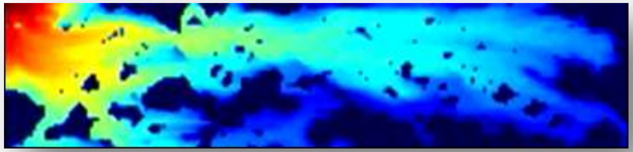

Euler Equations (~25x)


SW Equations (~25x)


Marine aqoustics (~20x)

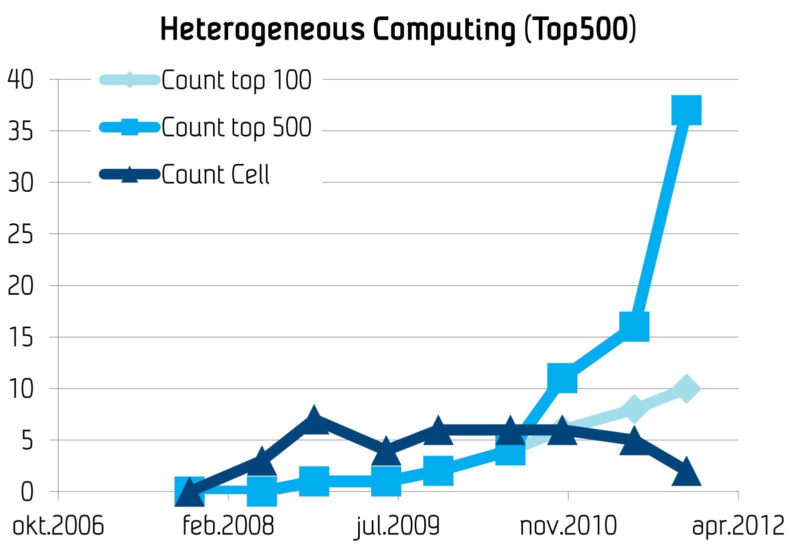

Matlab Interface

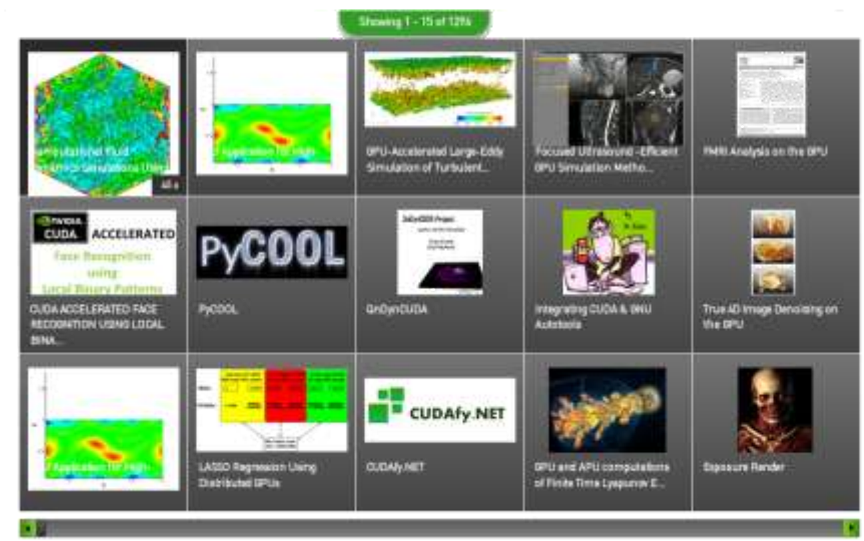

Linear algebra


Water injection in a fluvial reservoir (20x)

Examples from SINTEF

Examples of GPU use today





Screenshot from NVIDIA website

Compact stencils on the GPU: Efficient Flood Simulations

The Shallow Water Equations

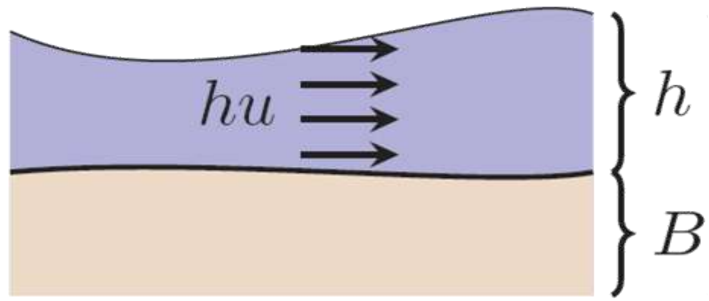
- A hyperbolic partial differential equation
  - First described by de Saint-Venant (1797-1886)
  - Conservation of mass and momentum
  - Gravity waves in 2D free surface

- Gravity-induced fluid motion
  - Governing flow is horizontal

- Not only for water:
  - Simplification of atmospheric flow
  - Avalanches
  - ...

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

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

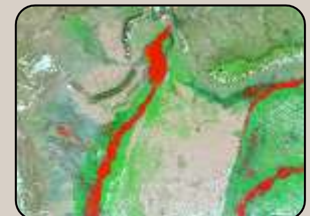
Bed friction source term

Target Application Areas

Tsunamis

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

Floods

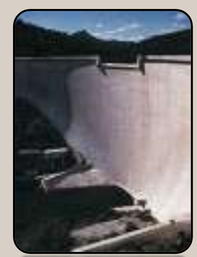


2010: Pakistan (2000+)
1931: China floods (2 500 000+)

Storm Surges

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

Dam breaks



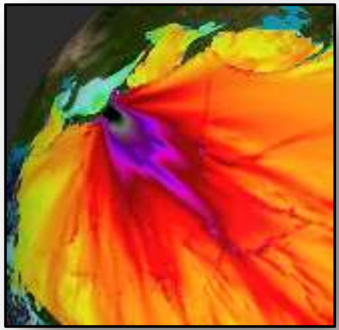
1975: Banqiao Dam (230 000+)
1959: Malpasset (423)

Images from wikipedia.org, www.ecolo.org

Two important uses of shallow water simulations

- In preparation for events: Evaluate possible scenarios
  - Simulation of many ensemble members
  - Creation of inundation maps
  - Creation of Emergency Action Plans

- In response to ongoing events
  - Simulate possible scenarios *in real-time*
  - Simulate strategies for flood protection (sand bags, etc.)
  - Determine who to evacuate based on simulation, not guesswork



- **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

Solving a partial differential equation on the GPU

- Before we start with the shallow water equations, let us examine something slightly less complex:
  the heat equation
- Describes diffusive heat conduction
- Prototypical partial differential equation

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

- u is the temperature, kappa is the diffusion coefficient, t is time, and x is space.

Finding a solution to the heat equation

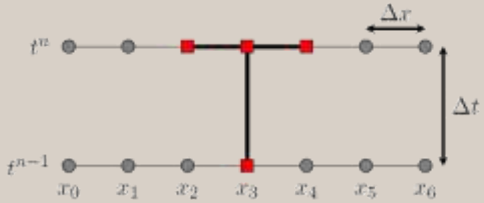
$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}$$

- Solving such partial differential equations analytically is nontrivial in all but a few very special cases

- Solution strategy: replace the continuous derivatives with approximations at a set of grid points

- Solve for each grid point numerically on a computer

- Use many grid points, and high order of approximation to get good results



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

The Heat Equation with an implicit scheme

1. We can construct an *implicit* scheme by carefully choosing the "correct" approximation of derivatives

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

2. This ends up in a system of linear equations

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

3. Solve Ax=b using standard GPU methods to evolve the solution in time

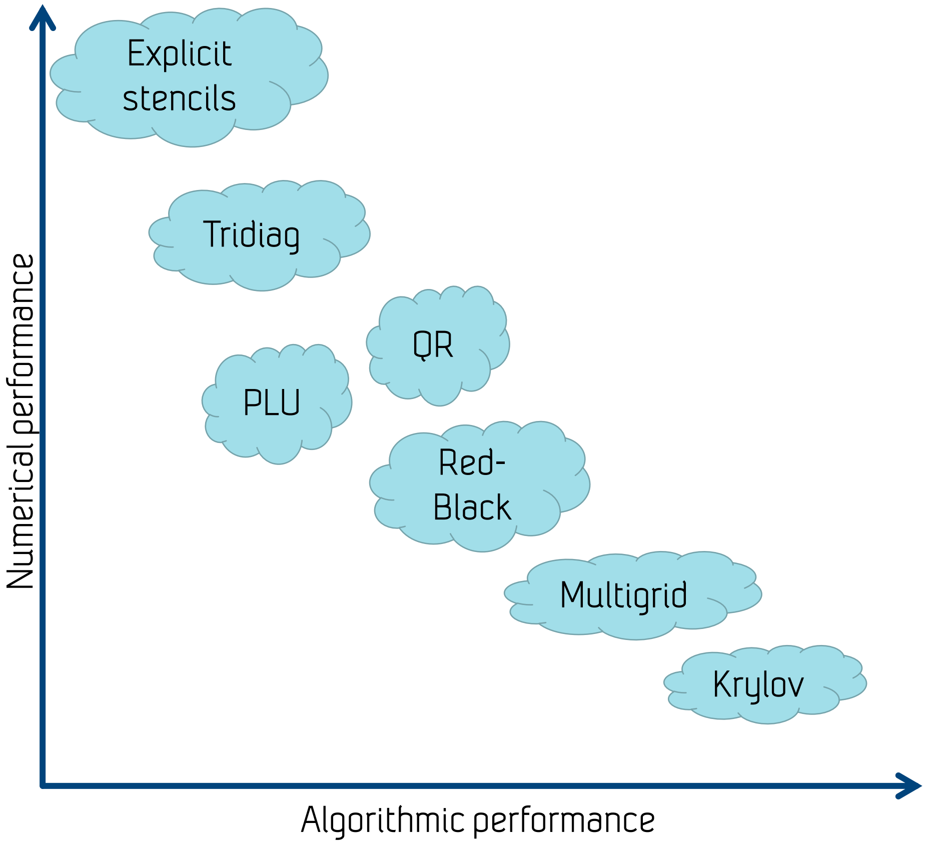
The Heat Equation with an implicit scheme

- Such 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…
  - for many time-varying phenomena, we are also interested in the temporal dynamics of the problem
  - Linear algebra solvers can be **slow and memory hungry**, especially on the GPU

Algorithmic and numerical performance

- 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
- Sparse linear algebra solvers have low numerical performance
  - Only able to utilize a fraction of the capabilities of CPUs, and **worse on GPUs**
- For suitable problems, explicit schemes with compact stencils can give the best performance
  - Able to reach near-peak performance

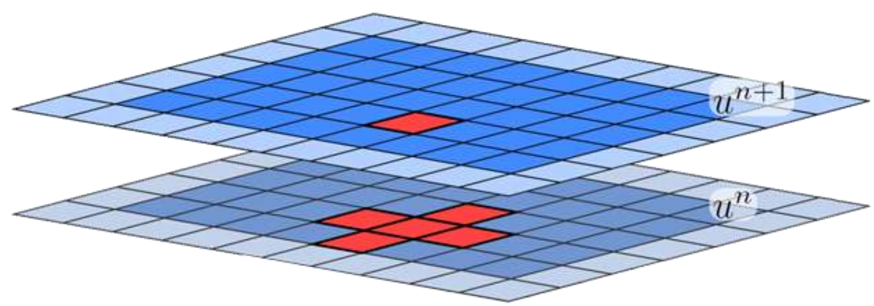


Explicit schemes with compact stencils

- Explicit schemes can give rise to compact stencils
  - Embarrassingly parallel
  - Perfect for the GPU!

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

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



Back to 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"
    - Have the accuracy required for capturing the physics
    - Preserves the physical quantities
  - Fits GPUs well
    - Works well with single precision
    - Is embarrassingly parallel
    - Has a compact stencil
    - …
  - …

The Finite Volume Scheme of Choice*

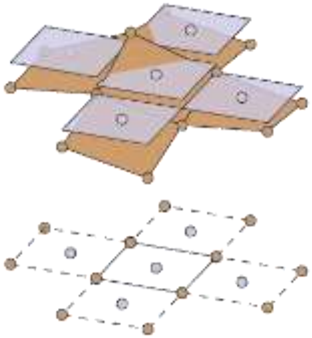
Scheme of choice: A. Kurganov and G. Petrova, A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System *Communications in Mathematical Sciences*, 5 (2007), 133-160

- Second order accurate fluxes
- Total Variation Diminishing
- Well-balanced (captures lake-at-rest)
- Good (but not perfect) match with GPU execution model

* With all possible disclaimers

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



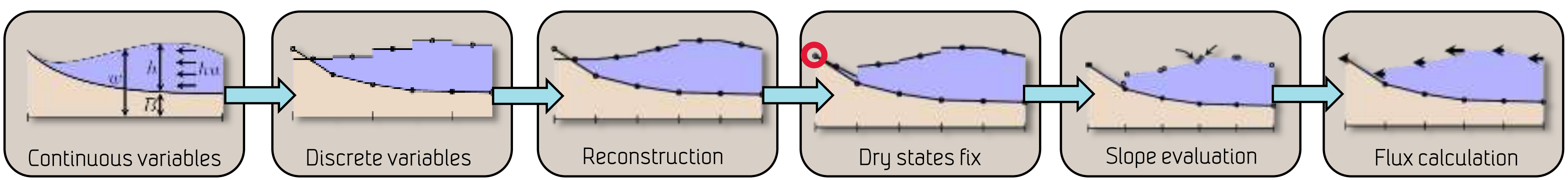
- Physical quantities are transported across the cell interfaces

- Algorithm:
  1. Reconstruct physical variables
  2. Evolve the solution
  3. Average over grid cells

Kurganov-Petrova Spatial Discretization (Computing fluxes)

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

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



Temporal Discretization (Evolving in time)

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

Gather all known terms

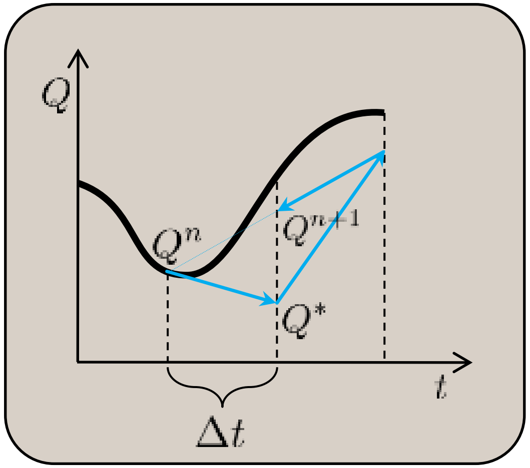
$$\boxed{\frac{dQ_{ij}}{dt}} = \boxed{H_f(Q_{ij})} + R(Q)_{ij}$$

Use second order Runge-Kutta to solve the ODE

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

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

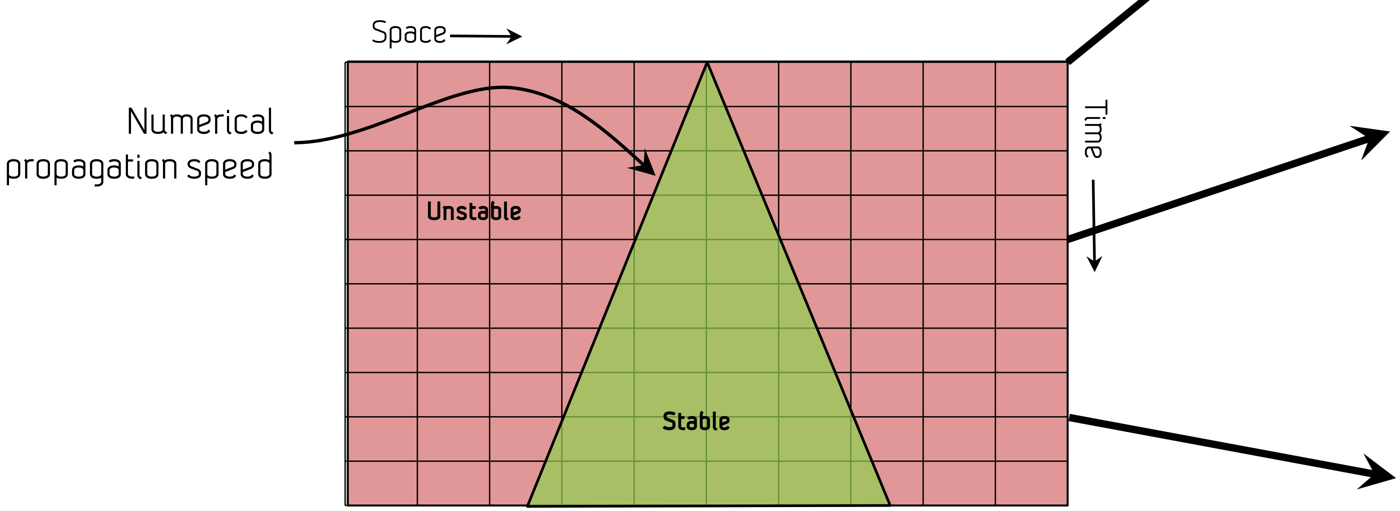
$$\Delta t \leq \tfrac{1}{4}\min\left\{\Delta x / \max_\Omega \left|u \pm \sqrt{gh}\right|, \Delta y / \max_\Omega \left|v \pm \sqrt{gh}\right|\right\}$$



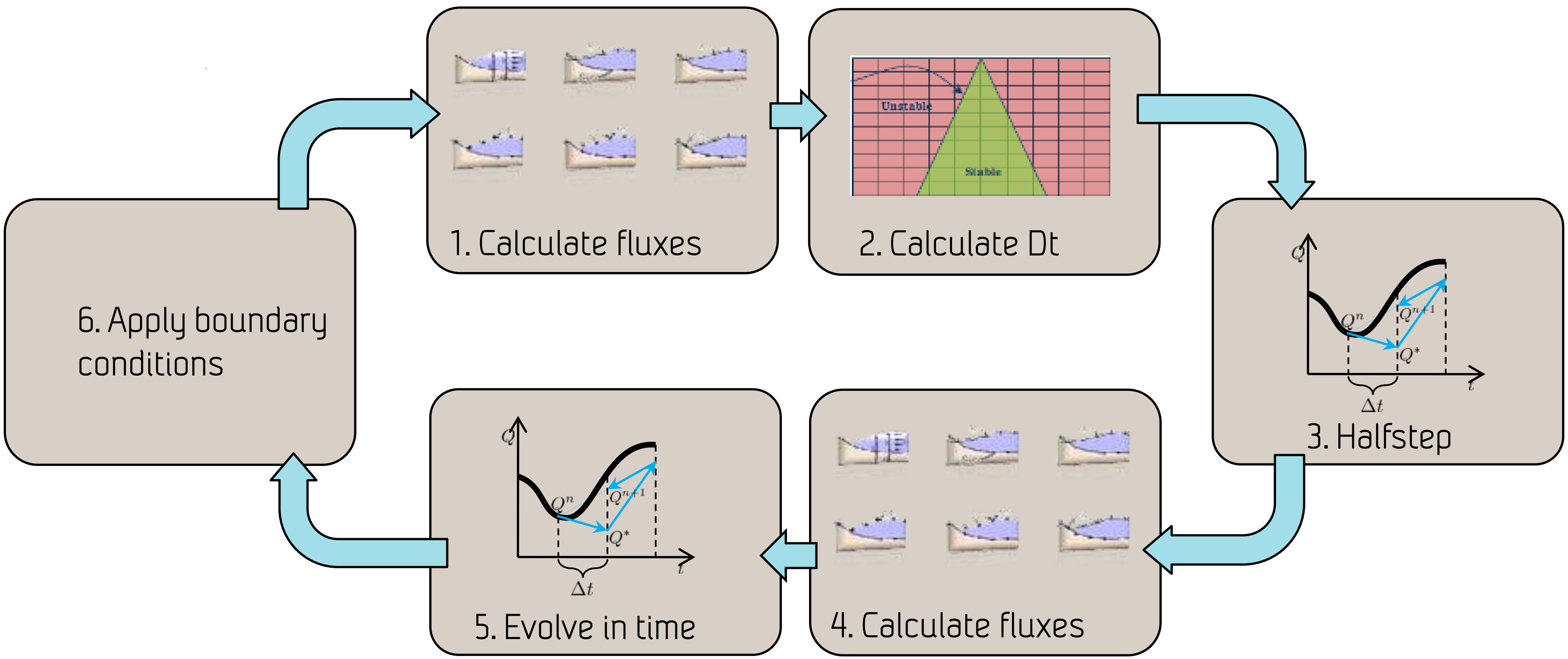
Courant-Friedrichs-Lewy condition

- Explicit scheme, time step restriction:
  - Time step size restricted by a Courant-Friedrichs-Lewy condition
  - Each wave is allowed to travel at most one quarter grid cell per time step:

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

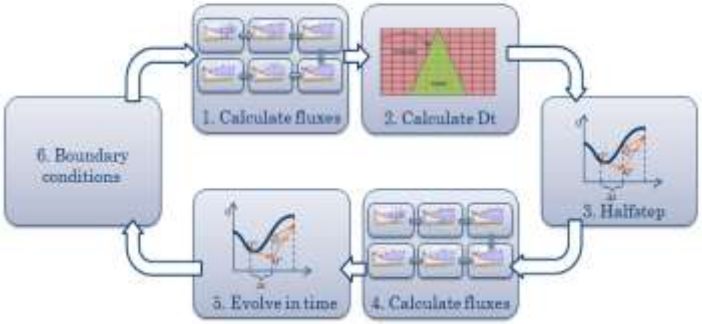


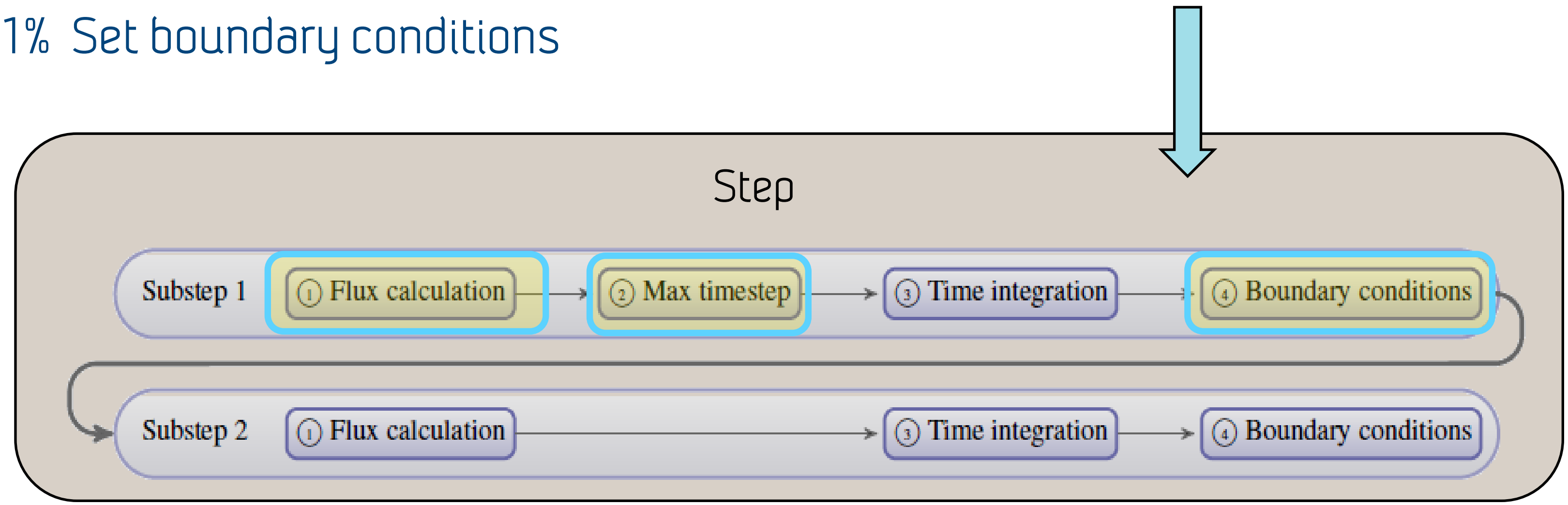
A Simulation Cycle



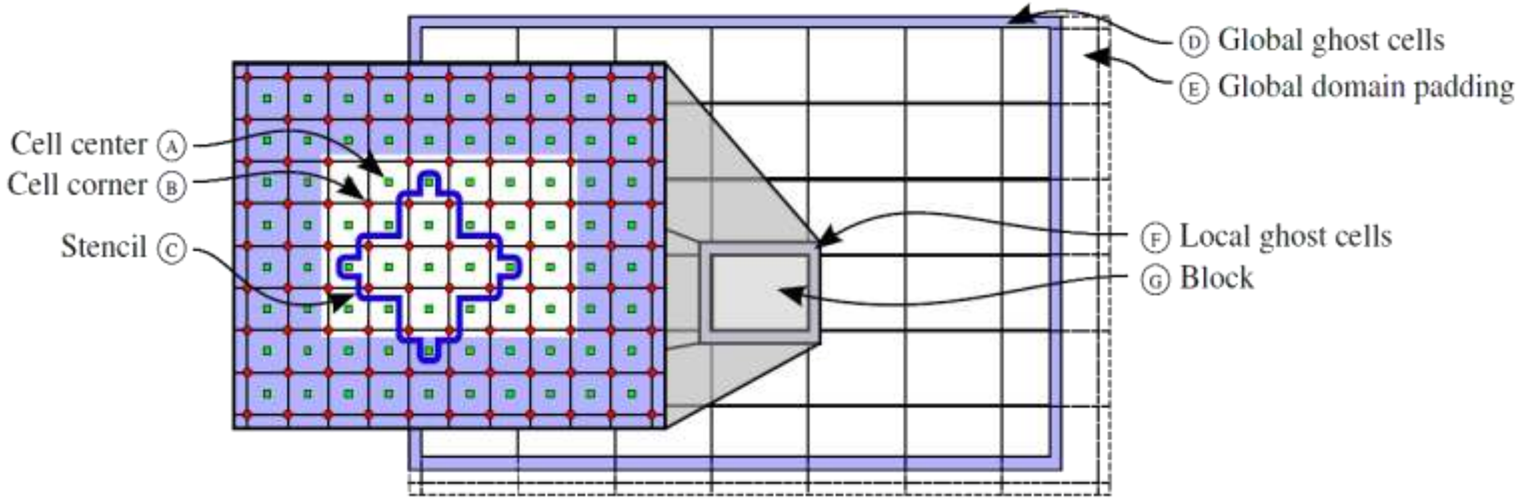
Implementation – GPU code

- Four CUDA kernels:
  - 87% Flux
  - <1% Timestep size (CFL condition)
  - 12% Forward Euler step
  - <1% Set boundary conditions





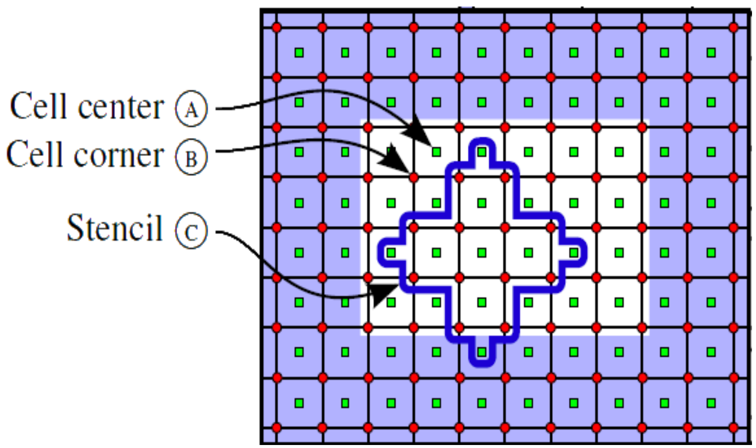
Flux kernel – Domain decomposition



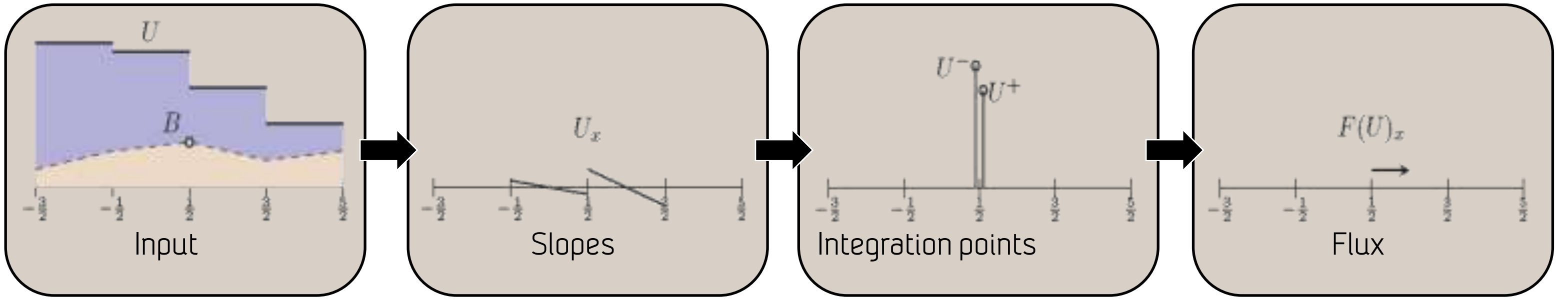
- A nine-point nonlinear stencil
  - Comprised of simpler stencils
  - Heavy use of shared mem
  - Computationally demanding
- Traditional Block Decomposition
  - Overlaping ghost cells (aka. apron)
  - Global ghost cells for boundary conditions
  - Domain padding

Flux kernel – Block size

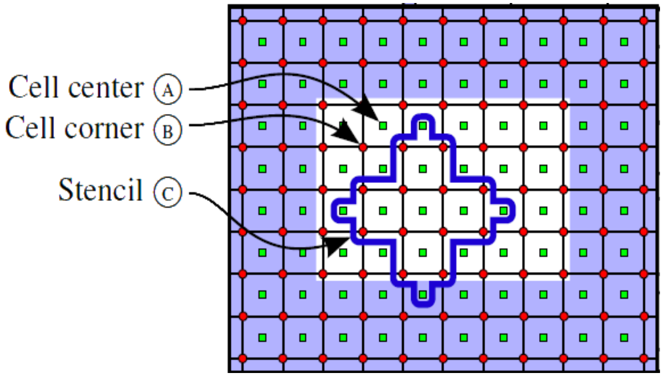
- Block size is 16x14
  - Warp size: multiple of 32
  - Shared memory use: 16 shmem buffers use ~16 KB
  - Occupancy
    - Use 48 KB shared mem, 16 KB cache
    - Three resident blocks
    - Trades cache for occupancy
  - Fermi cache
  - Global memory access



Flux kernel - computations



- Calculations
  - Flux across north and east interface
  - Bed slope source term for the cell
  - Collective stencil operations
- n threads, and n+1 interfaces
  - one warp performs extra calculations!
  - Alternative is one thread per stencil operation (Many idle threads, and extra register pressure)



Flux kernel – flux limiter

- Limits the fluxes to obtain non-oscillatory solution
  - Generalized minmod limiter
    - Least steep slope, or
    - Zero if signs differ
  - Creates divergent code paths

- Use branchless implementation (2007)
  - Requires special sign function
  - Much faster than naïve approach

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

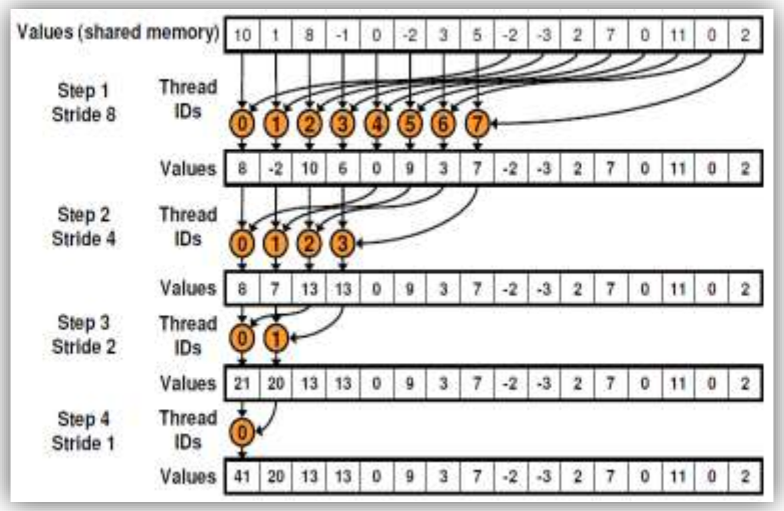
```
float minmod(float a, float b, float c) {
 return 0.25f
  *sign(a)
  *(sign(a) + sign(b))
  *(sign(b) + sign(c))
  *min( min(abs(a), abs(b)), abs(c) );
}
```

(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.

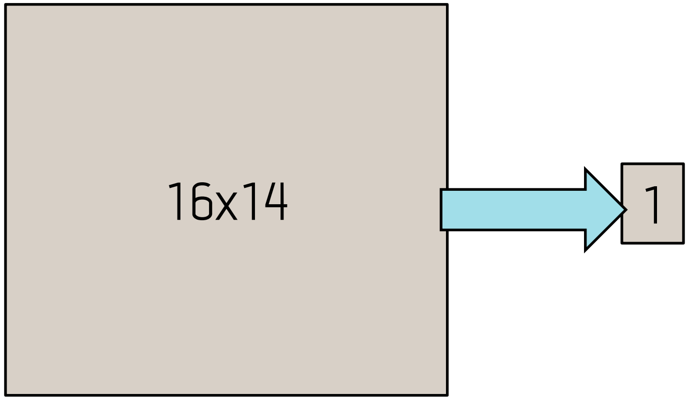
Timestep size kernel

- Flux kernel calculates wave speed per cell
  - Find global maximum
  - Calculate timestep using the CFL condition
  - Parallel reduction:
    - Models CUDA SDK sample
    - Template code
    - Fully coalesced reads
    - Without bank conflicts

Image from "Optimizing Parallel Reduction in CUDA", Mark Harris

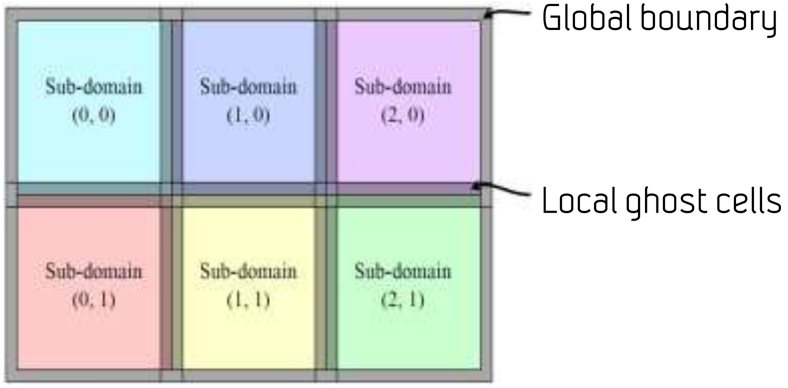


- Optimization
  - Perform partial reduction in flux kernel
  - Reduces memory and bandwidth by a factor 192

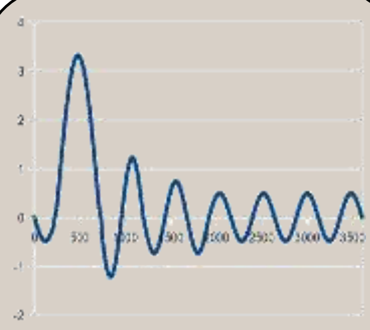


Boundary conditions kernel

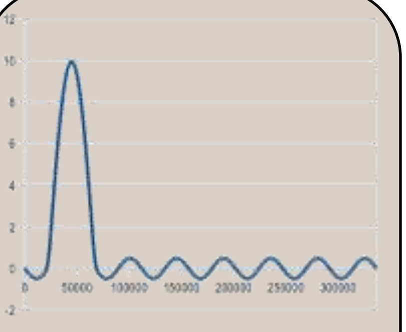
- Global boundary uses ghost cells
  - Fixed inlet / outlet discharge
  - Fixed depth
  - Reflecting
  - Absorbing



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



3.5m Tsunami, 1h



10m Storm Surge, 4d

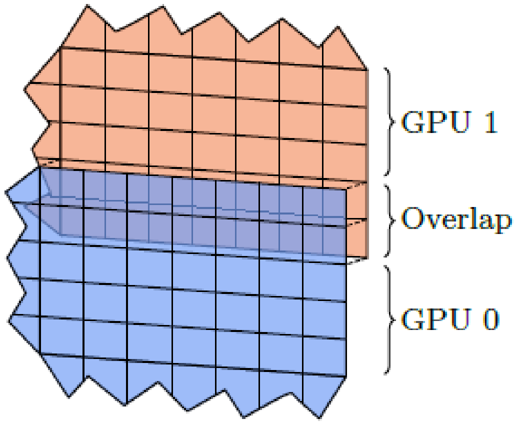
Boundary conditions kernel

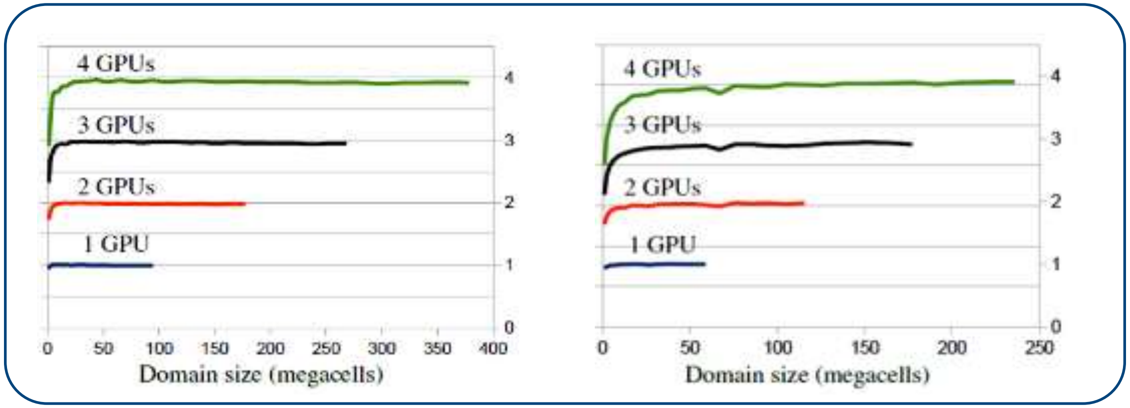
- Use CPU-side if-statement instead of GPU-side
  - Similar to CUDA SDK reduction sample, using templates:
  - One block sets all four boundaries
  - Boundary length (>64, >128, >256, >512)
  - Boundary type ("none", reflecting, fixed depth, fixed discharge, absorbing outlet)
  - In total: 4*5*5*5*5 = 2500 realizations

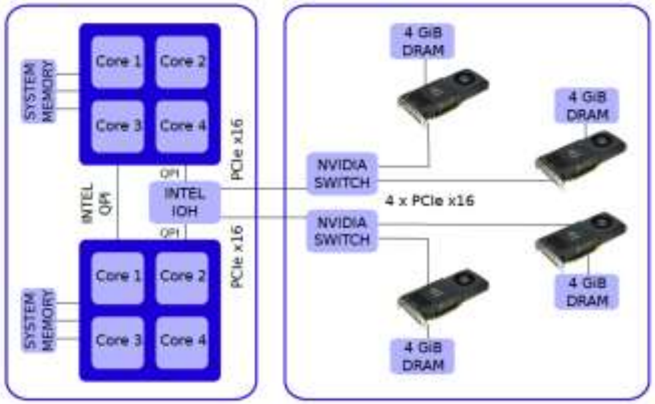
```
switch(block.x) {
 case 512: BCKernelLauncher<512, N, S, E, W>(grid, block, stream); break;
 case 256: BCKernelLauncher<256, N, S, E, W>(grid, block, stream); break;
 case 128: BCKernelLauncher<128, N, S, E, W>(grid, block, stream); break;
 case  64:  BCKernelLauncher<  64, N, S, E, W>(grid, block,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

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



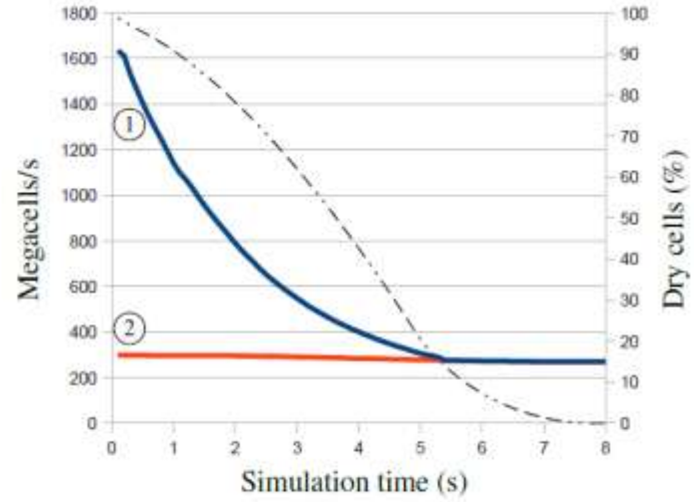


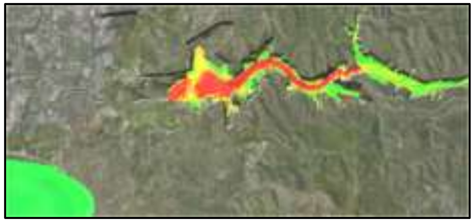


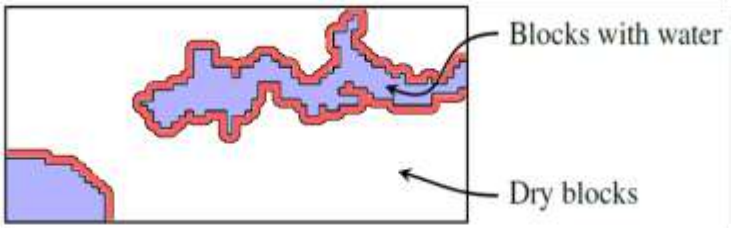
Collaboration with Martin L. Sætra

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

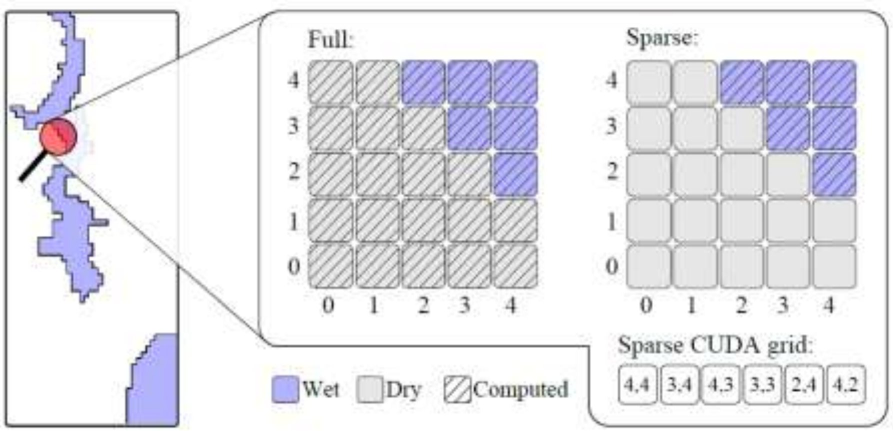






Sparse domain optimization

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



Sparse Compute:

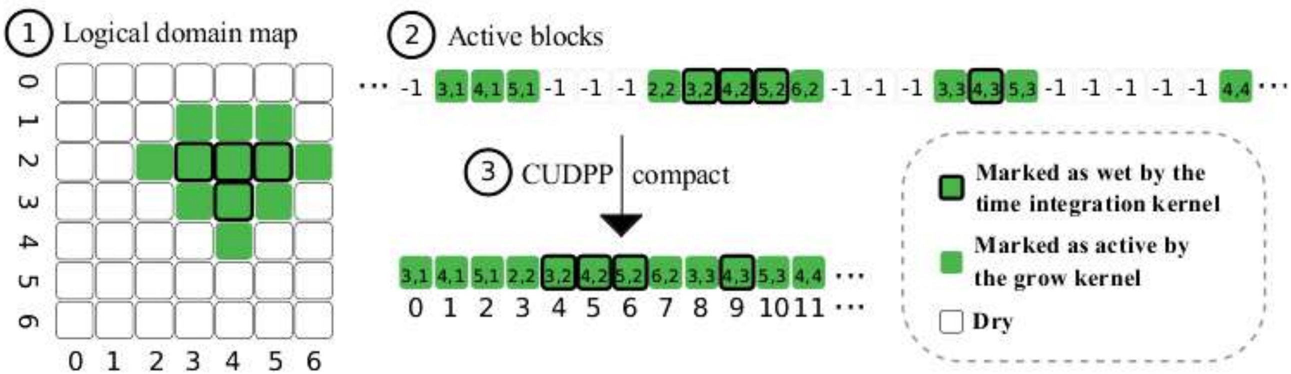
Do not perform any computations on dry parts of the domain

Sparse Memory:

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

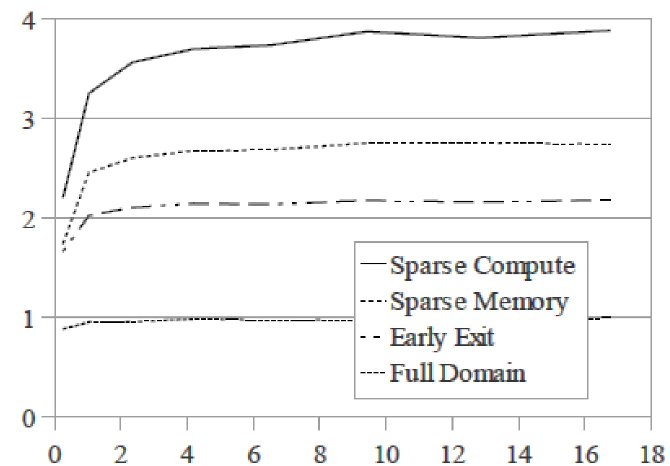
Ph.D. work of Martin L. Sætra

Sparse domain optimization



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

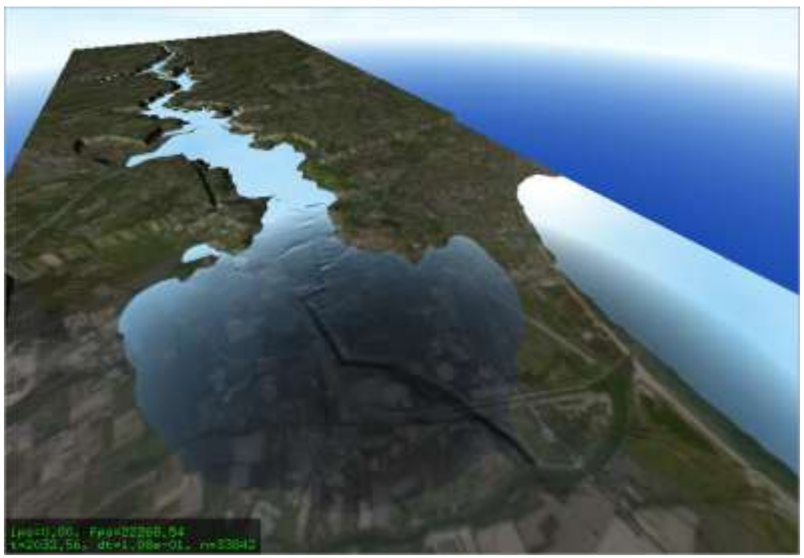
- Similarly for memory, *but it gets quite complicated…*
- 2x improvement over early exit (mileage may vary)!

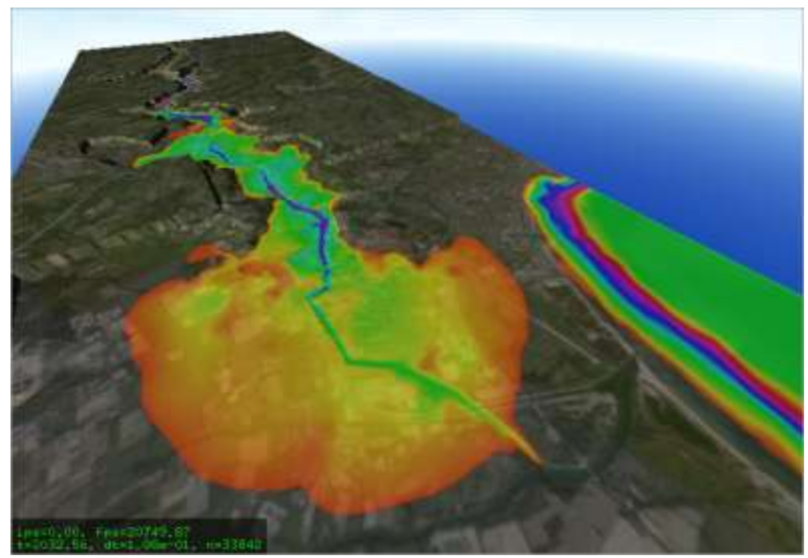


Comparison using an average of 26% wet cells

Real-time visualization

- When the data is on the GPU, visualize it directly
  - Has about 10% performance impact
  - http://www.youtube.com/watch?v=FbZBR-FjRwY





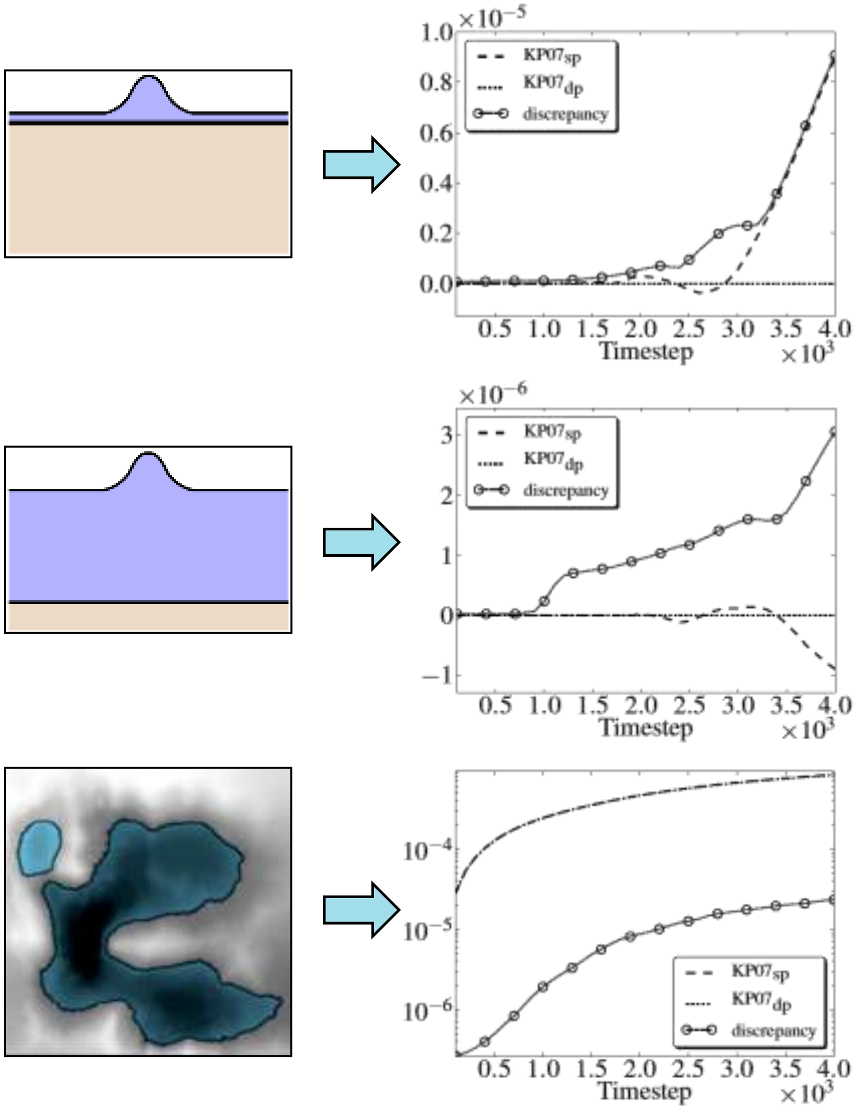
Accuracy and Physical correctness

Accuracy: Single Versus Double Precision

- What is the relative error in mass conservation for single and double precision?
- What is the discrepancy between the two?

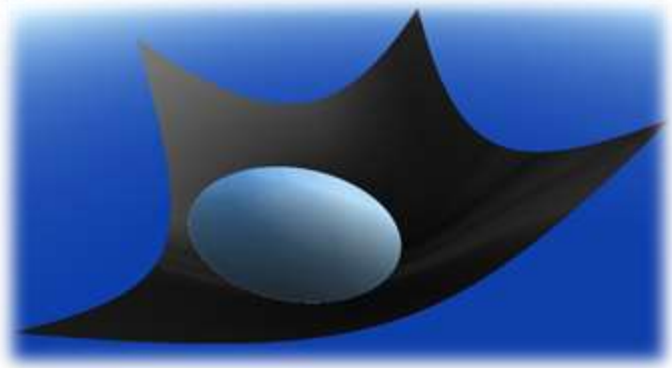
- Three different test cases
  - Low water depth (wet only)
  - High water depth (wet only)
  - Synthetic terrain with dam break (wet-dry)

- Conclusions:
  - We have loss in conservation on the order of machine epsilon
  - Single precision gives larger error than double
  - Errors related to the wet-dry front is more than an order of magnitude larger
  - **For our application areas, single precision is sufficient**

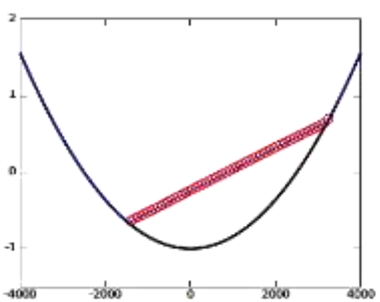


Verification: Parabolic basin

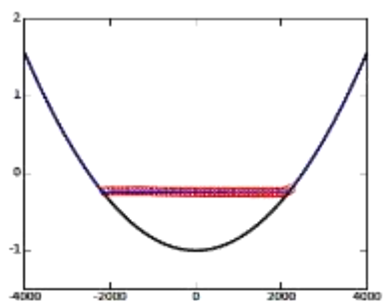
- Single precision is sufficient, but do we solve the equations?
- Test against analytical 2D parabolic basin case (Thacker)
  - Planar water surface oscillates
  - 100 x 100 cells
  - Horizontal scale: 8 km
  - Vertical scale: 3.3 m



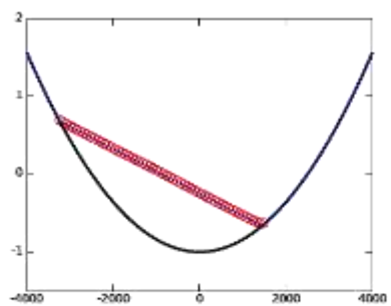
- Simulation and analytical match well
  - But, as most schemes, growing errors along wet-dry interface



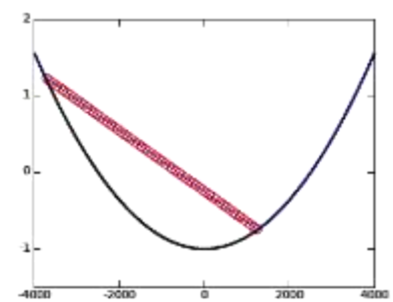
(a) $t \approx 1\pi/8\omega$ (1392.85 s)



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



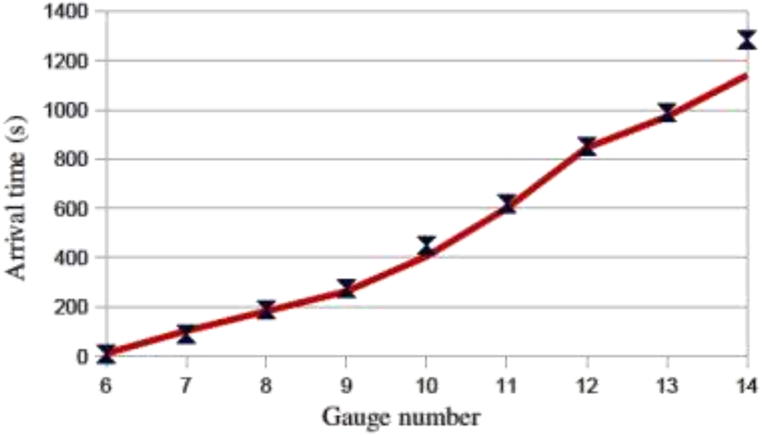
(c) $t \approx 3\pi/8\omega$ (4198.64 s)



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

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



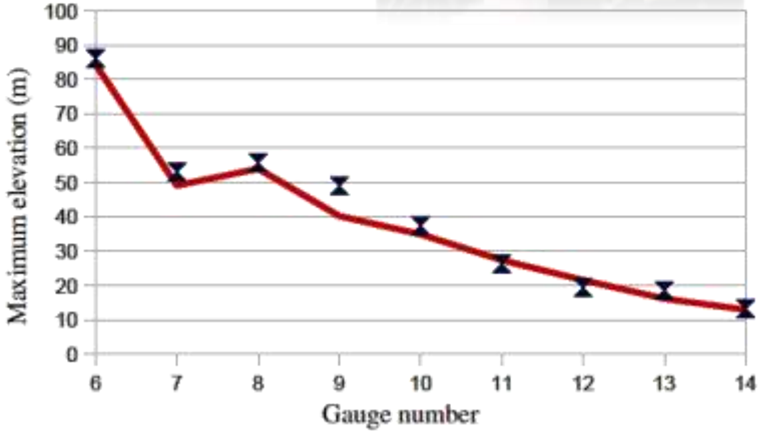


Image from google earth, mes-ballades.com

Summary

Summary

- Shallow water simulations on the GPU vastly outperform CPU implementations
  - Able to run faster-than-real-time!

- Physical correctness can be ensured
  - Even single precision is sufficiently accurate

- Multi-GPU and sparse domain optimizations
  - Two GPUs give twice the performance
  - Computation on land avoided

Thank you for your attention

Questions?

Contact:
André R. Brodtkorb
Email: Andre.Brodtkorb@sintef.no
Homepage: http://babrodtk.at.ifi.uio.no/
Youtube: http://youtube.com/babrodtk
SINTEF: http://www.sintef.no/heterocomp

"This slide is intentionally left blank"