Smoothed Particle Hydrodynamics simulations on multi-GPU systems

E. Rustico, G. Bilotta, A. Hérault, C. Del Negro and G. Gallo
Università di Catania, INGV Osservatorio Etneo, CNAM (Paris)

February 16, 2012


RISTORANTE

Outline

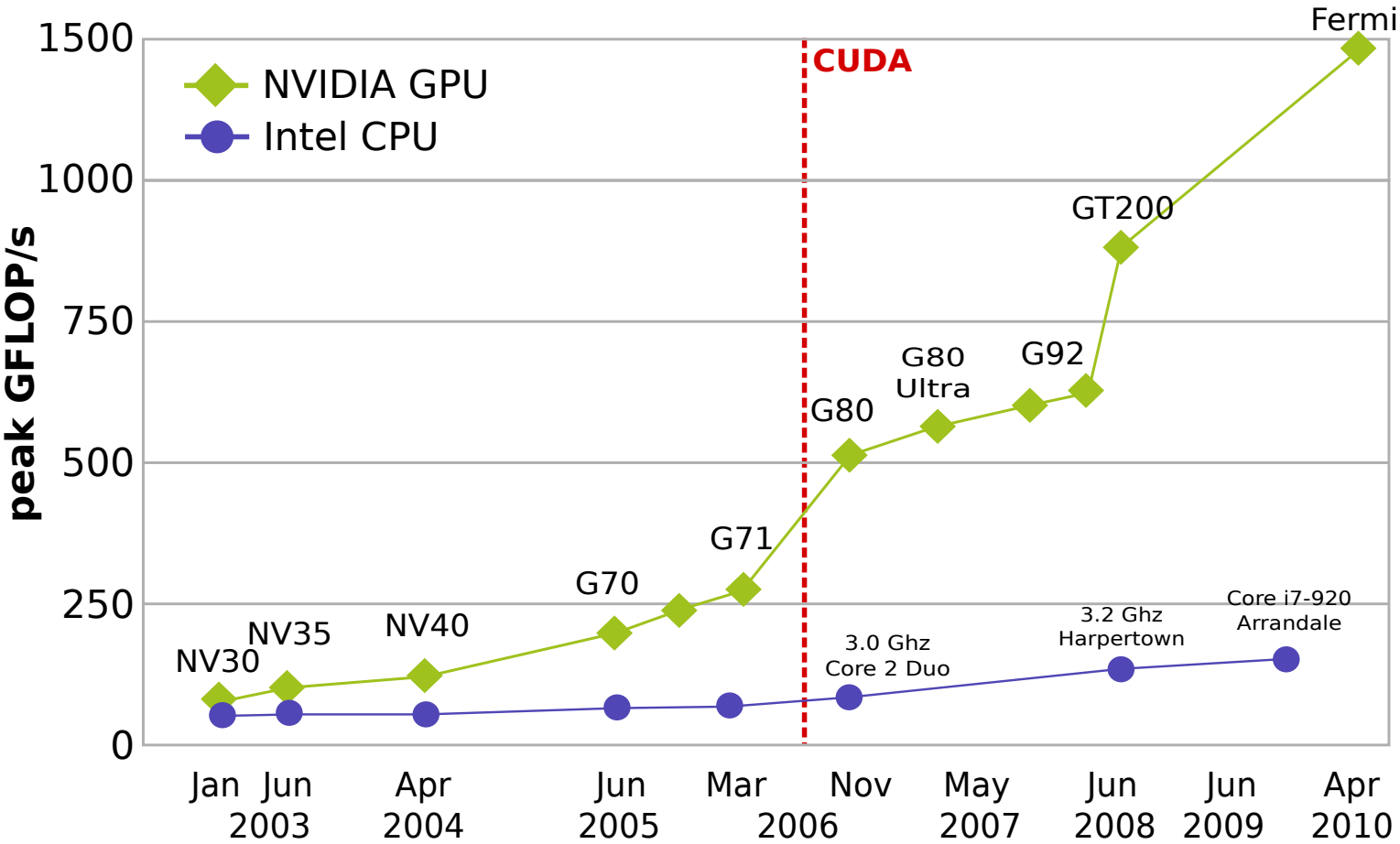
1981

1992

2001

2011

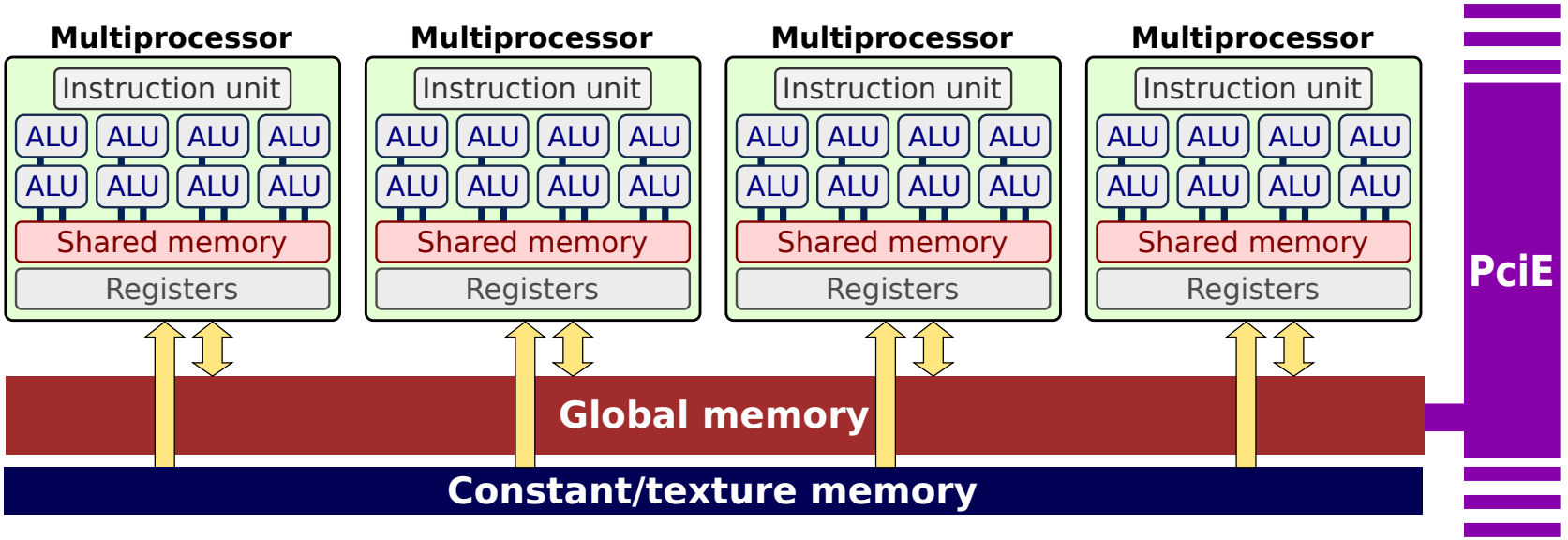
"Same" game, slightly different amount of computational requirements...



Gaming has been the main pulling factor for GPU computational power.
GPUs are nowadays faster than CPUs for *specific* kinds of computations

GPU programming Timeline

- **1980s**: first **Graphic Processing Units**
- **1990s**: almost every computer has its own **graphic accelerator**
- **2001**: release of the first **programmable shader**; researchers used to map a non-graphic problem to a graphic-one, in order to exploit the computational power of GPUs for general-purpose problems
- **2007**: release of CUDA, software and hardware platform with **explicit support** for general-purpose programmability



A multi-core GPU can be seen as a parallel computer with **shared memory**

Why CUDA?

- Simple programming **language**; use an extension of C++
- Simple programming **model**; it is easy to quickly gain a consistent speedup
- **Cheap** hardware (CUDA-capable hardware since 2007; 3 TFLOPS with less than $1,000 €$)
- Various optimized libraries available for common tasks (sorting, selection, linear algebra, FFT, image processing, computer vision...)
- ...aggressive marketing!

In november 2011, **three** CUDA-enabled clusters are among the top 5 in **TOP500**

Why **not** CUDA?

- Bound to a specific manufacturer
- No complete abstraction from hardware (like OpenCL)
- Theoretical peak power often smaller than the main competitor (ATI)

Example: array initialization, serial CPU code

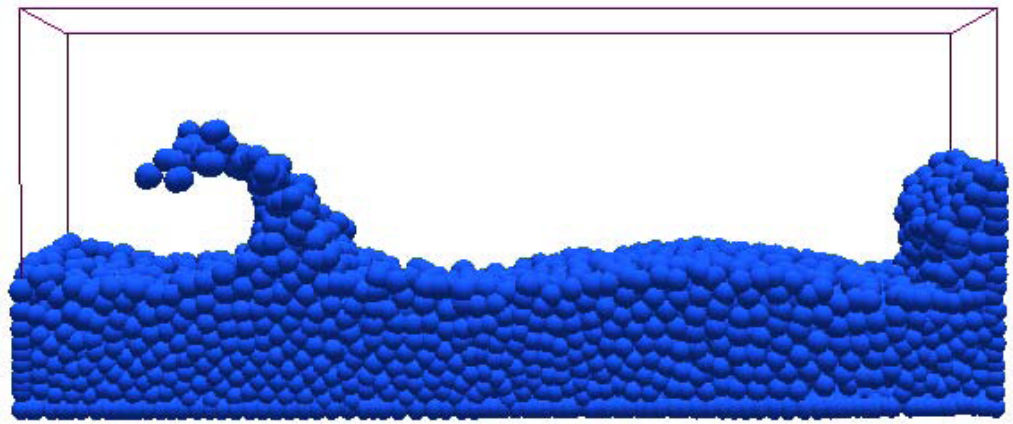
```
1 for (i=0; i<4096; i++)
2     cpu_array[i] = i;
```

Example: array initialization, parallel GPU code

```
1 #define BLKSIZE 64
2 ...
3 __global__ void init_kernel(int* gpuPointer) {
4     index = blockDim.x*blockIdx.x + threadIdx.x;
5     gpuPointer[index] = index;
6 }
7 ...
8 init_kernel<<< 4096/BLKSIZE, BLKSIZE >>>(gpu_array);
```

SPH

Smoothed Particle Hydrodynamics is a mesh-free, Lagrangian numerical method for fluid modeling, originally developed by Gingold, Monaghan and Lucy for astrophysical simulations.



The fluid is discretized in a set of arbitrarily distributed particles each carrying scalar and vector properties (mass, density, position, velocity, etc.).

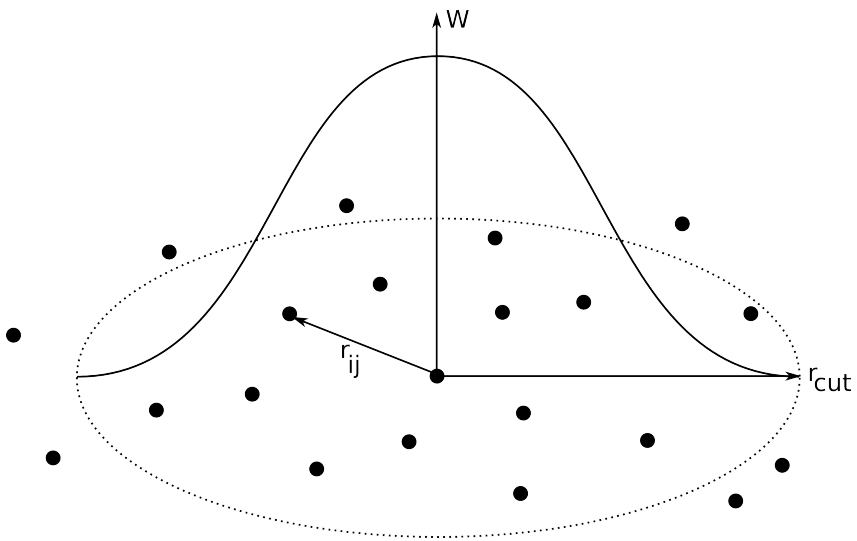
It is possible to interpolate the value of a field A at any point r of the domain by convolving the field values with a **smoothing kernel** W:

$$A(\mathbf{r}) \simeq \sum_j m_j \frac{A_j}{\rho_j} W(|\mathbf{r} - \mathbf{r}_j|, h),$$

where

- m_j is the mass of particle j
- A_j is the value of the field A for particle j
- ρ_j is the density associated with particle j
- $\mathbf{r}$ denotes position
- W is the *smoothing kernel function*, which is chosen to have compact support with radius proportional to a given *smoothing length* h

As the smoothing kernel W has compact support, each particle only interacts with a small set of **neighbors**:



Because each particle accesses its neighbors multiple times during an iteration, it is convenient to compute the neighbors list **once** and store it

Each iteration of the simulation consists of the following steps:

1. For each particle, compute and store the **list of neighbors**
2. For each particle, compute the interaction with all neighbors and compute the resulting **force** and the maximum allowable δt
3. Find the **minimum** allowable δt (to be used in next iteration)
4. For each particle, **integrate** the force over the δt and compute the resulting position and velocity

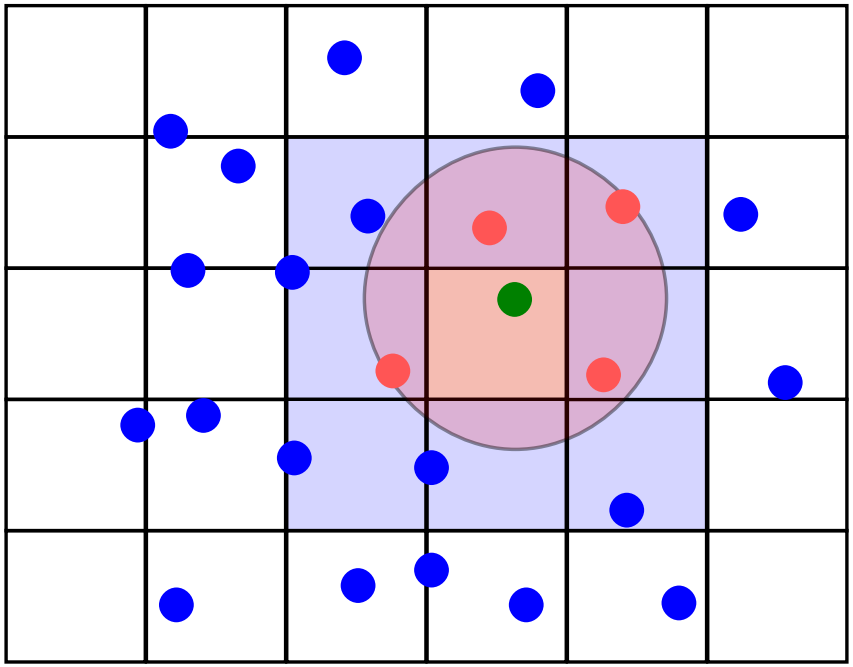
The current implementation actually relies on a **predictor-corrector** integration scheme with **adaptive** δt, where steps 2-4 are performed twice each iteration

Single-GPU SPH

The SPH model exposes a high degree of parallelism and it steps can be straightforwardly implemented as GPU functions (i.e. CUDA kernels):

1. **BuildNeibs** - Computed every k^{th} iteration
2. **Forces** - The most expensive step
3. **MinimumScan** - Computed with CUDPP minimum scan
4. **Euler** - The fastest step

The naïf algorithm to gather the neighbor list for each particle is $O(n^2)$ (quadratic with the number of particles).
While the SPH model is **gridless**, organizing the particles in a *virtual grid* with cells as wide as the influence radius boosts the search:



Cells are completely **transparent to the SPH model** and have side equal to the influence radius

Differences with respect to the CPU implementation

- Data are stored in a *structure of arrays* instead of *array of structures* fashion (to improve **coalescence**; cfr. CUDA Best Practices)
- Each interaction is computed **twice**, as on the GPU it is fastest not to deal with concurrent writes
- Negligible **numerical differences**: GPU uses by default single-precision float and order of additions may differ

The GPU-based implementation reached a 100× speedup over the reference CPU code (serial, single thread). Details and results in A. Herault, G. Bilotta, and R. A. Dalrymple, *SPH on GPU with CUDA*, Journal of Hydraulic Research, 48(Extra Issue):74–79, 2010

Open source implementation: GPU-SPH

`www.ce.jhu.edu/dalrymple/GPUSPH`

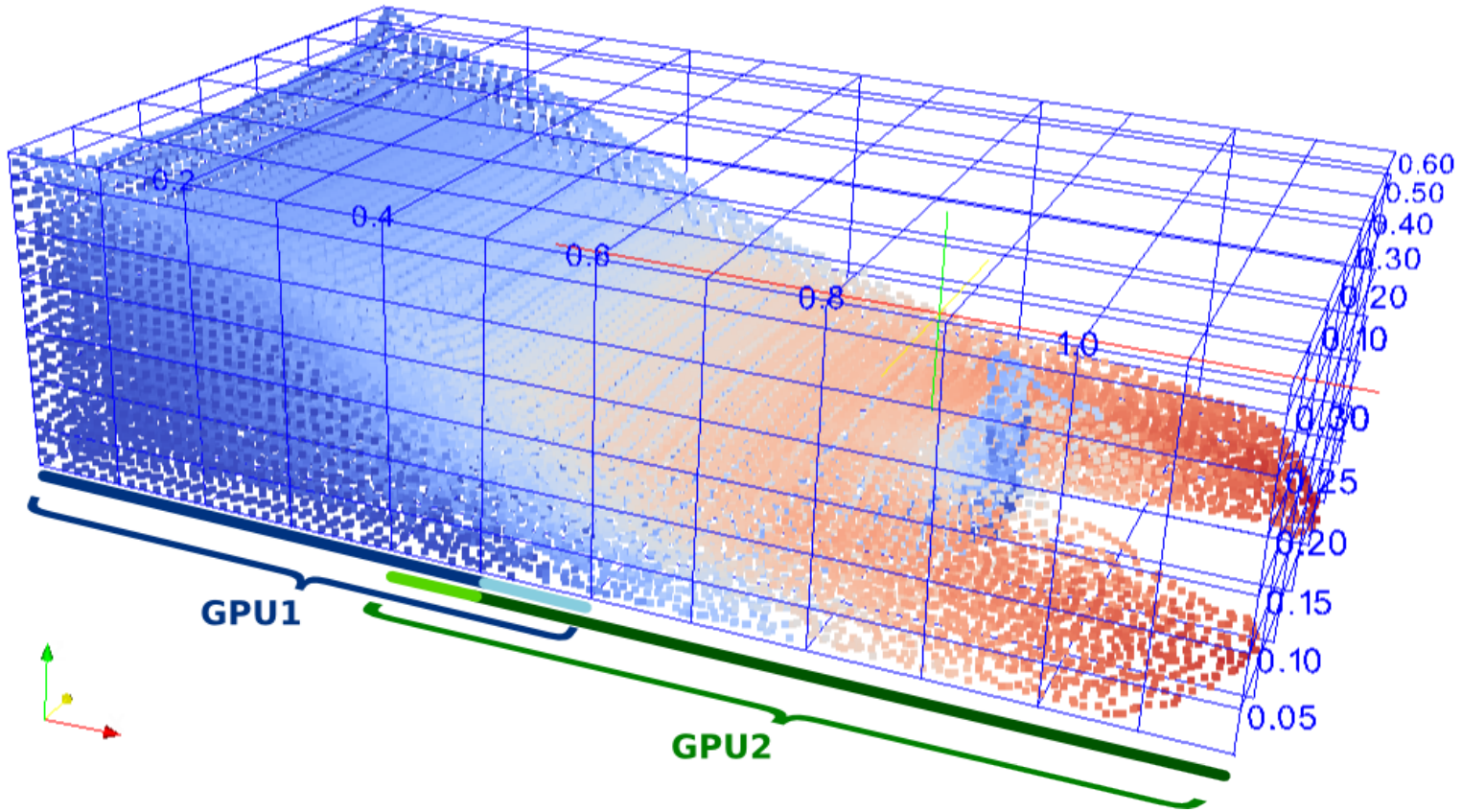
Multi-GPU SPH

Why exploiting multiple devices simultaneously?

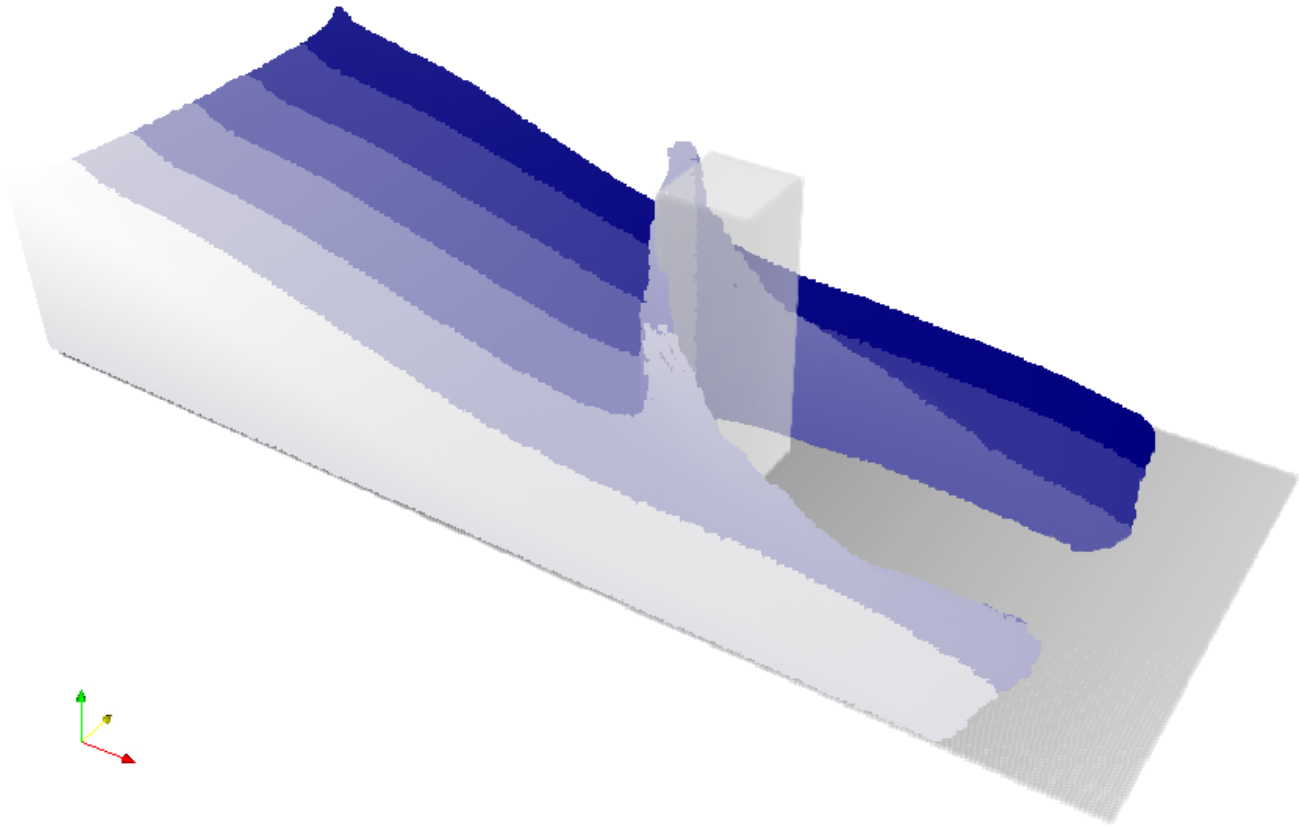
- **Performance** - faster simulations
- **Problem size** - bigger simulations

How to exploit multiple devices simultaneously?

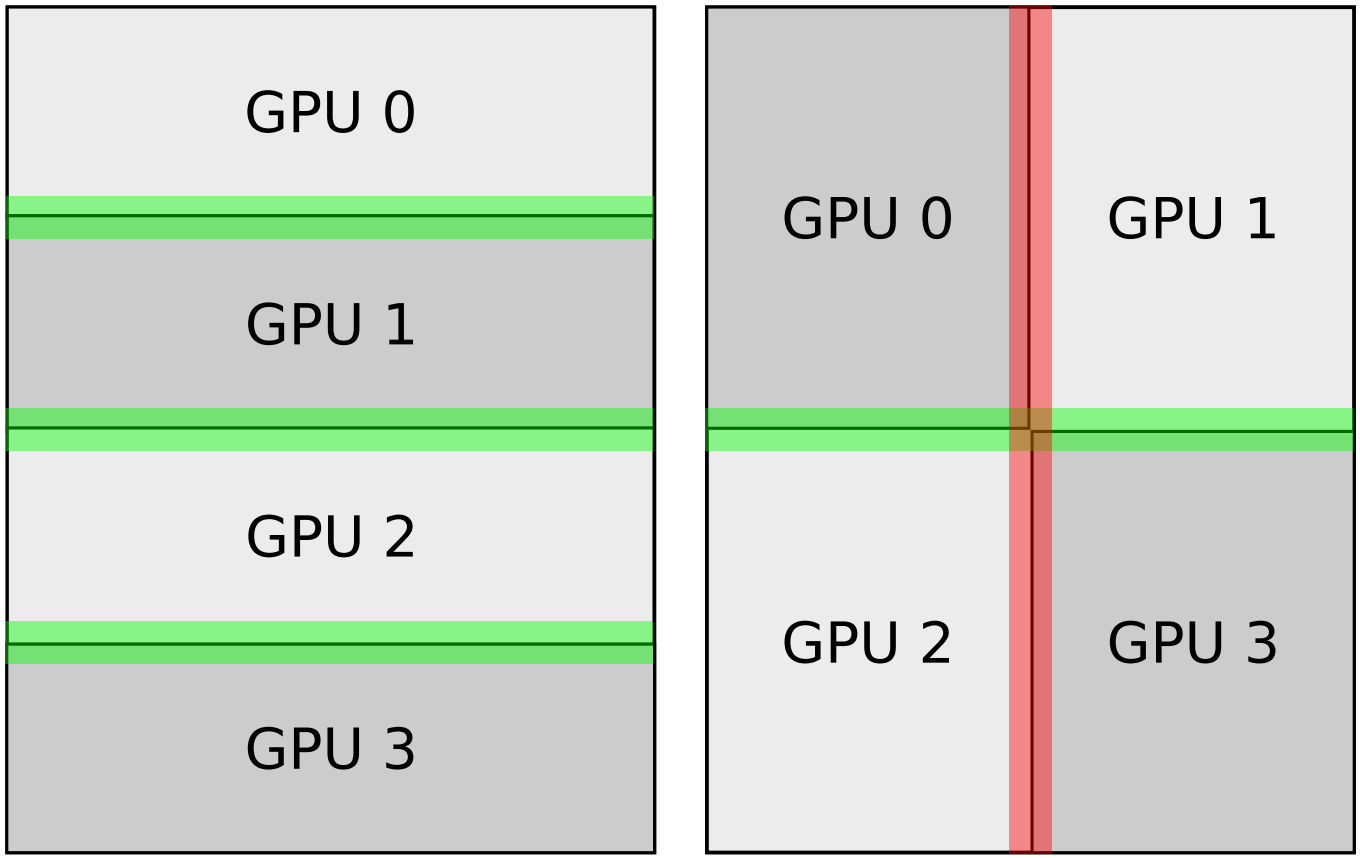
- Distribute the **computational phases** (*pipeline*; not scalable)
- Distribute the **problem domain** (scalable; boundary conditions)
  - Greedy list-split (unfeasible: no guarantees for neighbors)
  - Cells-based split (list-split + ordering)



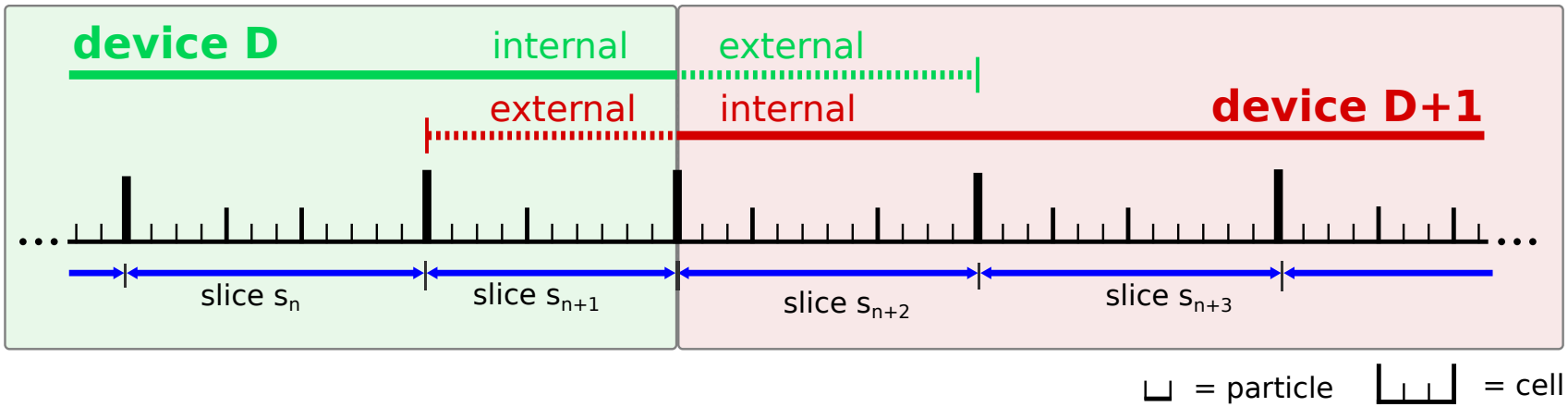
To split the domain, we exploit the virtual grid used for fast-neighbor search



Particles are enumerated in linear memory according to the cartesian axis chosen to split



Avoid multiple split planes in the same simulation



External particles: read-only copy of neighboring devices, to compute the forces for edging particles

Internal particles: particles needed by neighboring devices

Total overlap: **two cells**, **to be updated after every force computations**.

Up to CUDA 3.2, no direct device-device communication: we need a CPU buffer and multiple transfers

1. **BuildNeibs** - Read all particles, write for internal
2. **Forces** - Ditto
3. **MinimumScan** - Read only internal
4. **Euler** - Write both internal and external

Plus: after each execution of **Forces**, must exchange overlapping borders with neighbor devices.

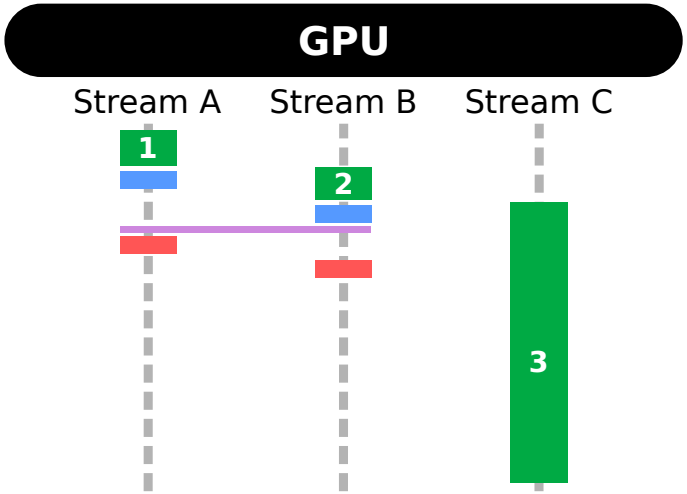
We need a technique to hide those transfers, or the overhead will "kill" the simulations

kernel_forces_async

- Launch kernel_forces on edging internal slices (**1**,**2**)
- Launch kernel_forces on the rest of slices (**3**)
- Launch download of internal slices (forces)

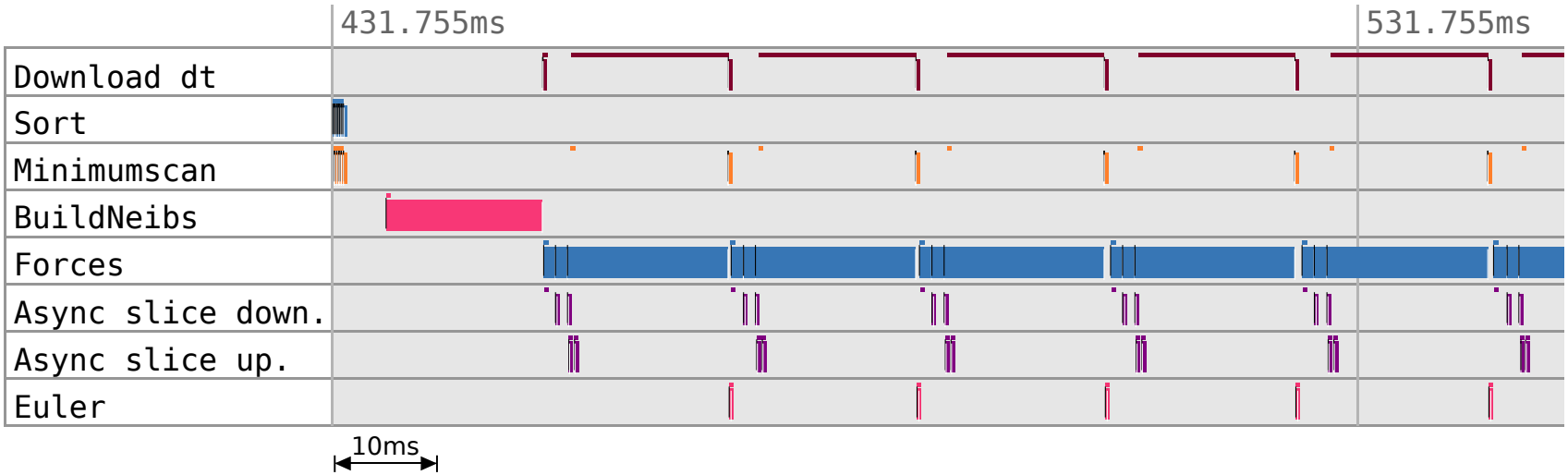
GPUThread BARRIER

- Launch upload of external slices (forces)



Key idea: exchange the slices as soon as they are ready, while still computing the forces of the internal particles, by using the **asynchronous API**

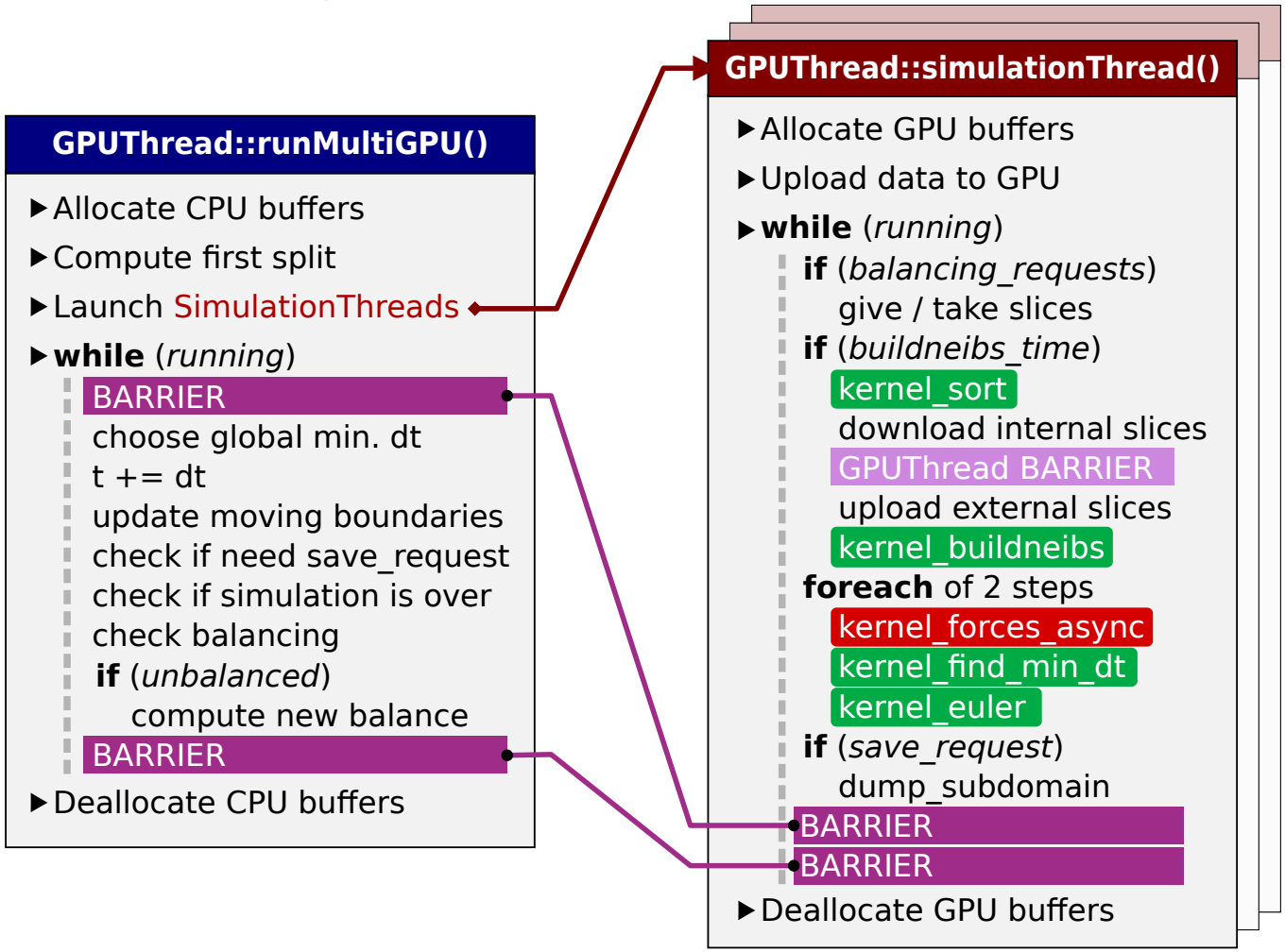
Know-how acquired with a multi-GPU Cellular Automaton (MAGFLOW)



Actual timeline, with kernel lengths in scale. Only one GPU with 2 neighboring devices is shown. The little dots mark the moment operations were issued on the CPU.

The timeline has been produced with a **custom visualization tool**

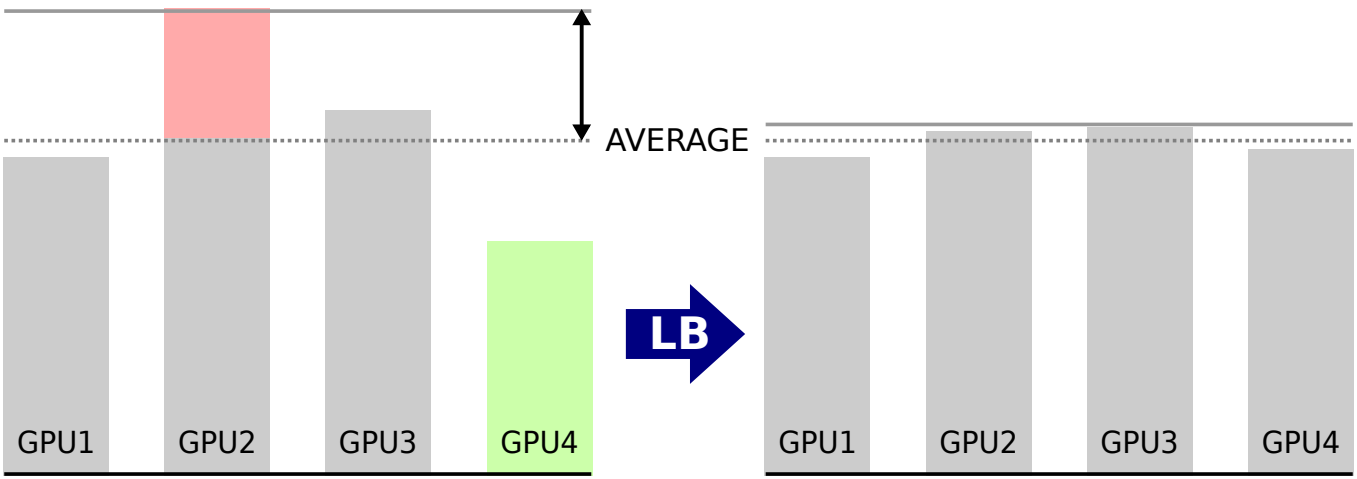
Overall simulator design:



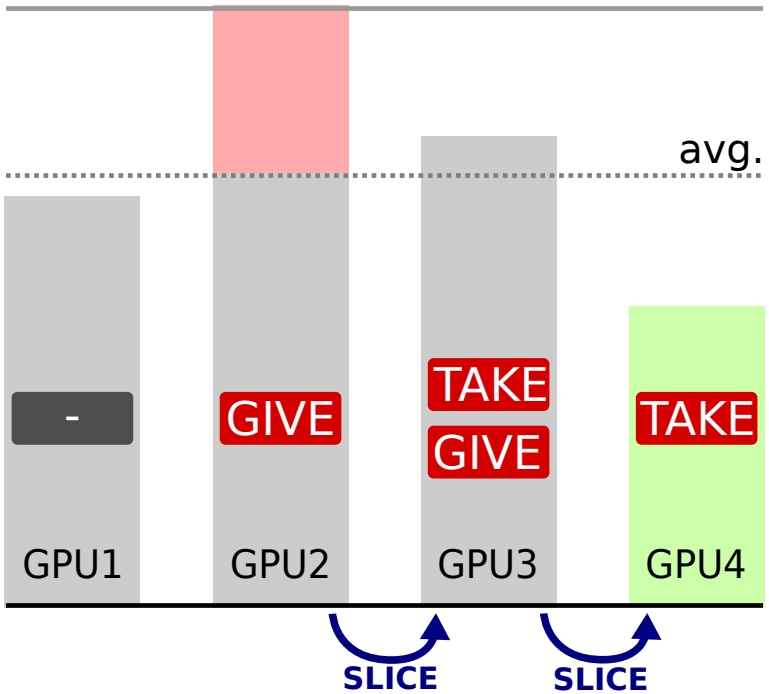
One `simulationThread` (CPU thread) per device

Load balancing

It is important to balance the workload, as the fluid topology can greatly influence the overall performance.



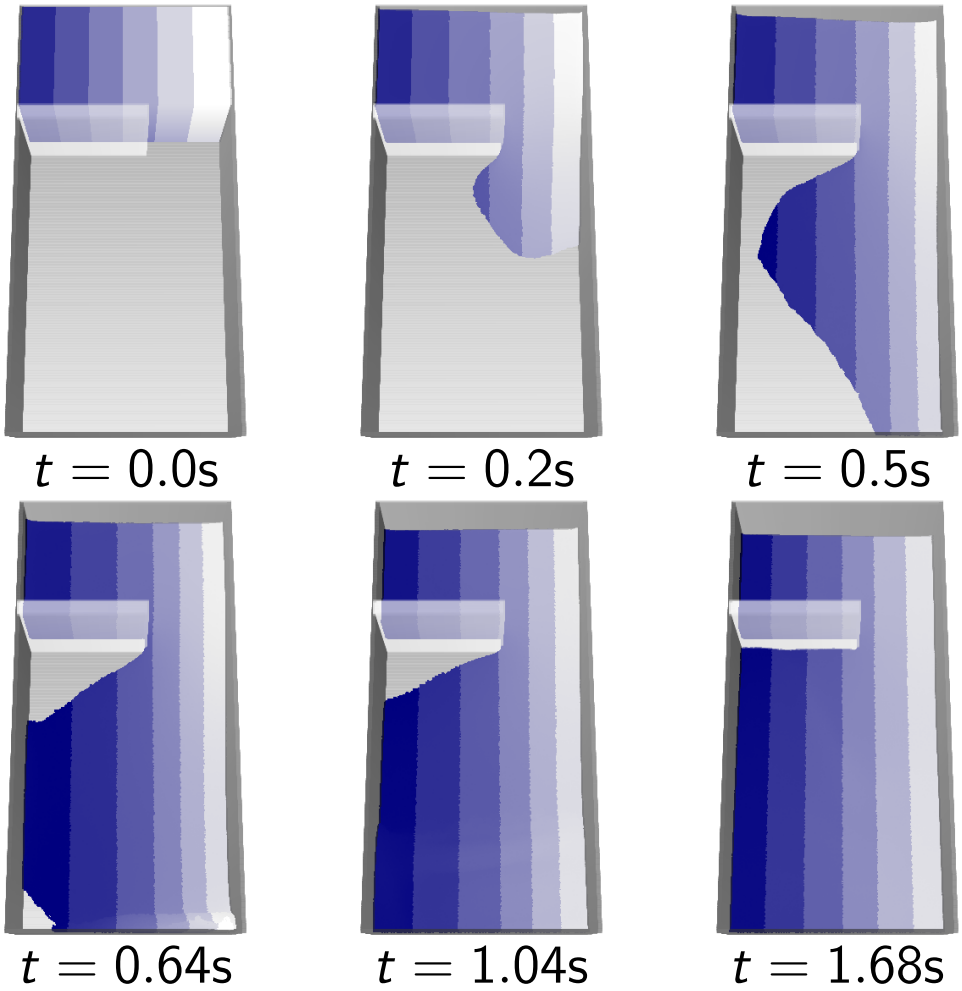
```
1  AG ← global avg. duration of kernel Forces
2  Ts ← avg. duration of one slice = AG / num. slices
3  foreach GPU
4      Ag ← self avg. duration of kernel Forces
5      delta ← Ag - AG
6      if (abs(delta) > Ts * HLB_THRESHOLD)
7          if (delta > 0)
8              mark as GIVING_CANDIDATE
9          else
10             mark as TAKING_CANDIDATE
11 foreach GPU
12     if a couple GIVING/TAKING is found
13         mark GIVING_CANDIDATE as GIVING
14         mark TAKING_CANDIDATE as TAKING
15         foreach intermediate GPU
16             mark as GIVING and TAKING
17         break
```



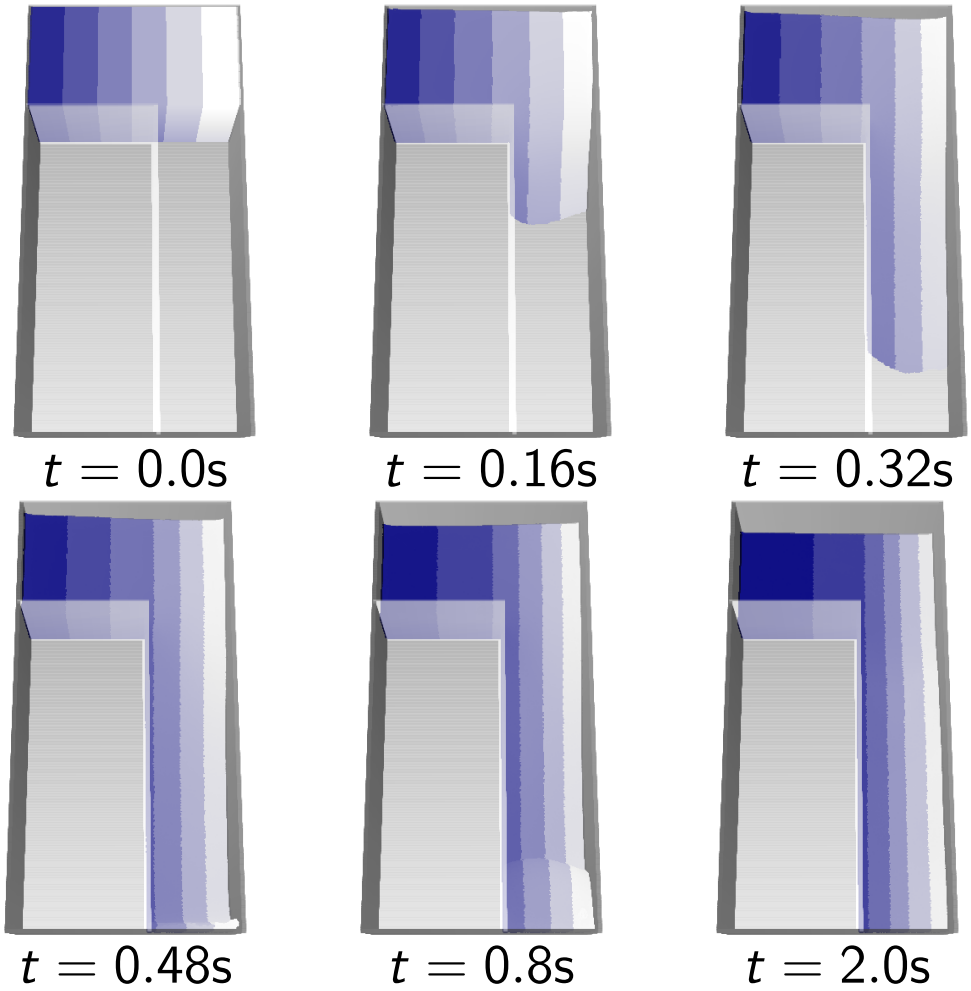
Simple policy:

- One transfer at a time
- Time for one slice computed from average
- Transfer between the first detected giving/taking pair
- Static threshold (no check for local minima nor ping-pong transfers)

BoreInABox



BoreInABox corridor variant

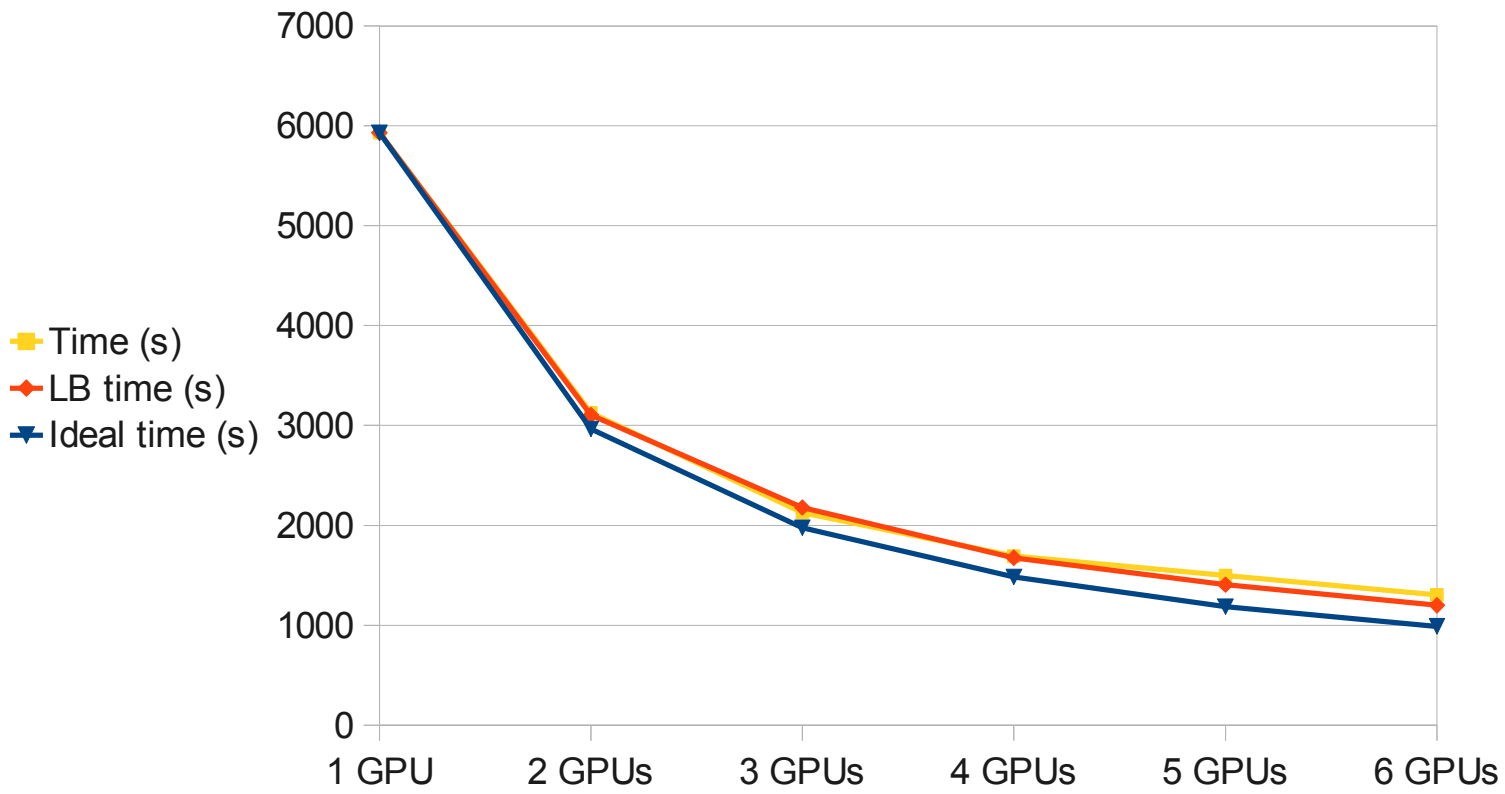


Videos available for download at:
`http://www.dmi.unict.it/~rustico/sphvideos/`

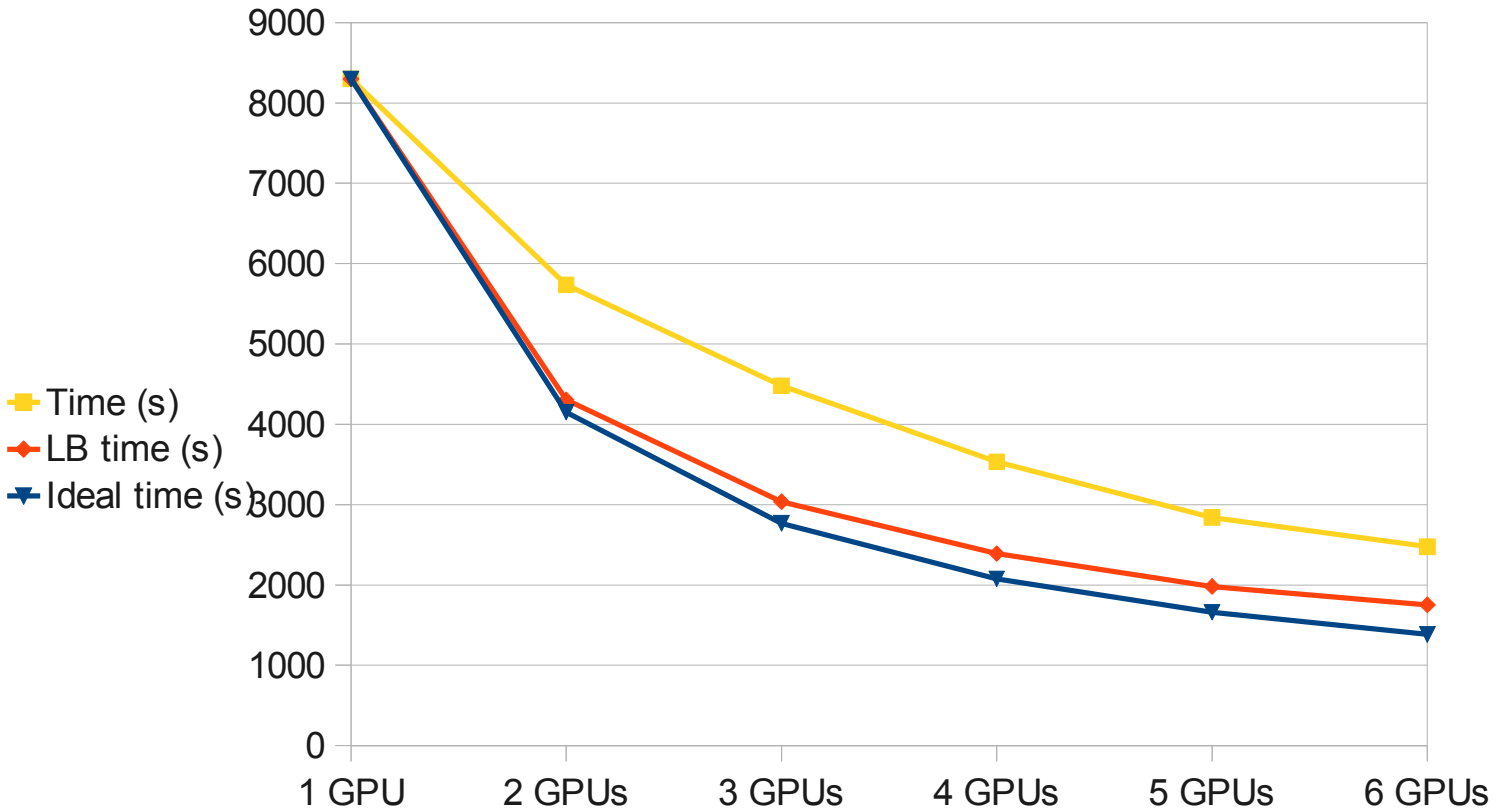
Hardware platform

- TYAN-FT72 rack
- Dual-Xeon 16 cores
- 16Gb RAM
- 6×GTX480 cards on PCIe2

Results - **DamBreak3D** (symmetric problem)



Results - **BoreInABox** (asymmetric problem)



A possible performance metric abstracting from the number of particles and the simulation settings is the number of **iterations** computed, times the number of **particles**, divided by the execution time in **seconds**.

In thousands, we can therefore measure *kip/s* (kilo-iterations times particles per second)

Kip/s, with and without load balancing, with a symmetric problem (DamBreak3D):

DamBreak3D	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
Kip/s	9,977	19,149	27,802	35,213	39,784	45,599
LB kip/s	-	19,380	27,191	35,336	42,578	49,491
Ideal kip/s	-	19,955	29,932	39,910	49,887	59,865

With an asymmetric problem (BoreInABox):

BoreInABox	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
Kip/s	8,770	12,713	16,275	20,649	25,657	29,418
LB kip/s	-	16,940	24,115	30,548	36,800	41,745
Ideal kip/s	-	17,541	26,311	35,082	43,852	52,623

A common starting point to analyze a parallel implementation is to model the problem as made by a **parallelizable** part and a **serial**, non parallelizable one.

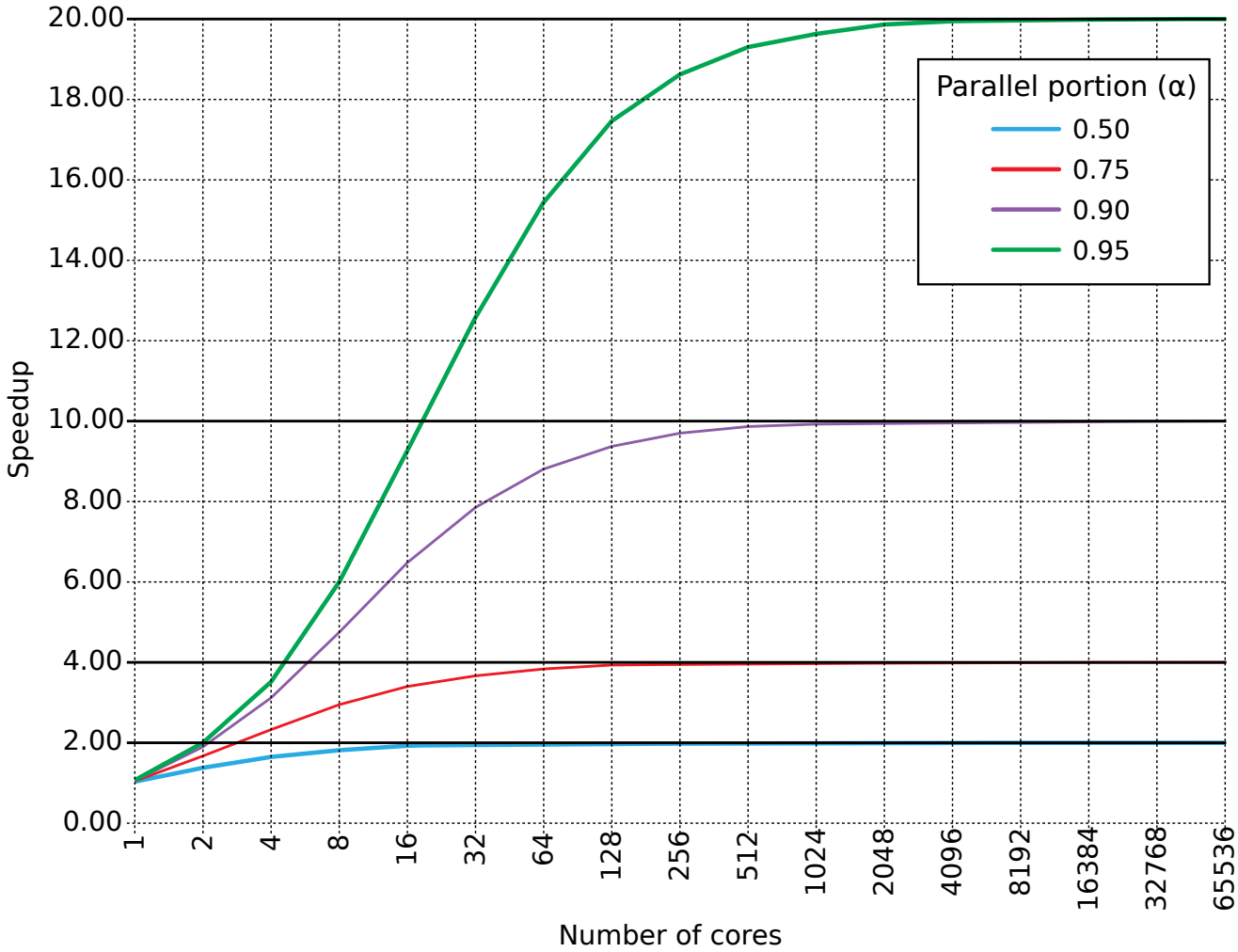
Let α be the parallelizable fraction of the problem and β the non-parallelizable one; it is $\alpha + \beta = 1$, with α, β non-negative real numbers.

Amdahl's law measures the speedup in terms of execution time and gives an upper-bound inversely proportional to β.

With a slight change in the notation, Amdahl's law can be written as

$$S_A(N) = \frac{1}{\beta + \frac{\alpha}{N}}$$

where N is the number of used cores and S_A the expected speed-up


Parallel portion (α)
0.50
0.75
0.90
0.95
Speedup
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
16.00
18.00
20.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of cores

The **Karp–Flatt** metric was proposed in 1990 as a measure of the **efficiency** of the parallel implementation of a problem. The new efficiency metric is the experimental measure of the serial fraction β of a problem:

$$\beta_{KF} = \frac{\frac{1}{S_e} - \frac{1}{N}}{1 - \frac{1}{N}}$$

S_e is the empirically measured speedup and N is the number of cores (GPUs).

An efficient parallelization presents a **constant** β

Measured β, with and without load balancing, with a symmetric problem (DamBreak3D):

DamBreak3D	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
β_{KF}, no LB	0.053	0.036	0.048	0.063	0.061
β_{KF}, LB	0.053	0.056	0.037	0.041	0.040

With an asymmetric problem (BoreInABox):

BoreInABox	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
β_{KF}, no LB	$0,429$	$0,289$	$0,222$	$0,181$	$0,153$
β_{KF}, LB	$0,053$	$0,056$	$0,048$	$0,048$	$0,050$

Future work

- Extend to GPU **clusters**, achieving a third level of parallelism (what is the best split?)
- Smarter **load balancing** (adaptive threshold and time analysis)

References

- E. Rustico, G. Bilotta, A. Hérault, C. Del Negro, G. Gallo, *Smoothed Particle Hydrodynamics simulations on multi-GPU systems*, 20th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Special Session on GPU Computing and Hybrid Computing, 2012, Garching/Munich (DE)
- G. Bilotta, E. Rustico, A. Hérault, A. Vicari, C. Del Negro and G. Gallo, *Porting and optimizing MAGFLOW on CUDA*, Annals of Geophysics, Vol. 54, n. 5, doi: 10.4401/ag-5341
- A. Hérault, G. Bilotta, E. Rustico, A. Vicari and C. Del Negro, *Numerical Simulation of lava flow using a GPU SPH model*, Annals of Geophysics, Vol. 54, n. 5, doi: 10.4401/ag-5343
- A. Hérault, G. Bilotta, E. Rustico, A. Vicari and C. Del Negro, *Numerical Simulation of lava flow using a GPU SPH model*, Annals of Geophysics, Vol. 54, n. 5, doi: 10.4401/ag-5343
- A. Herault, G. Bilotta, and R. A. Dalrymple, *SPH on GPU with CUDA*, Journal of Hydraulic Research, 48(Extra Issue):74–79, 2010
- J. J. Monaghan, *Smoothed particle hydrodynamics*, Annual Review of Astronomy and Astrophysics 30, pages 543–574, 1977

Thank you

...that's all, folks!

Eugenio Rustico
http://www.dmi.unict.it/~rustico
rustico@dmi.unict.it