Advances in the Solution of NS Eqs. in GPU Hardware

by M.Storti[a], S.Costarelli[a], R.Paz[a], L.Dalcin[a], S. Idelsohn[abc]

[a]Centro de Investigación de Métodos Computacionales
CIMEC (CONICET-UNL), Santa Fe, Argentina
mario.storti@gmail.com, www.cimec.org.ar/mstorti
[b]Institució Catalana de Recerca i Estudis Avançats
(ICREA), Barcelona, Spain, www.icrea.cat
[c]International Center for Numerical Methods in Engineering (CIMNE)
Technical University of Catalonia (UPC), www.cimne.com

Scientific computing on GPU's

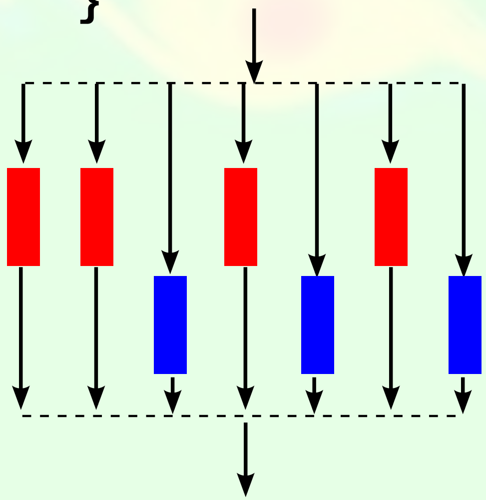
- ***Graphics Processing Units (GPU's)*** **are specialized hardware desgined to discharge computation from the CPU for *intensive graphics applications*.**
- **They have many cores (*thread processors*), currently the *Tesla K40 (Kepler GK110)* has *2880* cores at 745 Mhz (Builtin boost to 810, 875Mhz).**

- **The *raw computing power* is in the order of *Teraflops* (4.3 Tflops in SP and 1.43 Tflops in DP).**
- **Memory Bandwidth (GDDR5) 288 GB/sec. Memory size 12 GB/sec.**
- **Cost USD 5,000. Low end version Geforce GTX Titan: USD 1000.**

Scientific computing on GPU's (cont.)

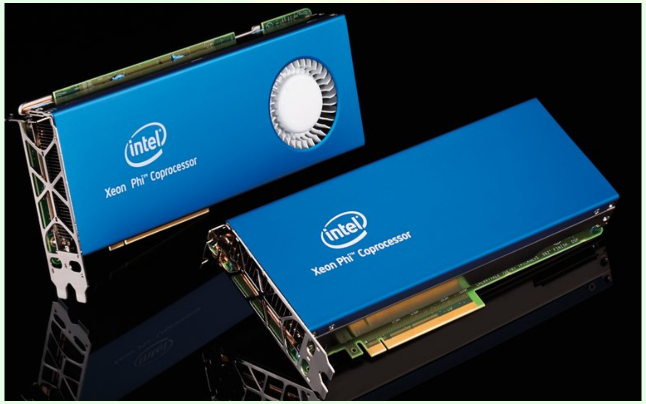
- **The difference between the GPU's architecture and standard multicore processors is that GPU's have much more computing units (*ALU's* (Arithmetic-Logic Unit) and *SFU's* (Special Function Unit), but few *control units*.**
- **The programming model is *SIMD (Single Instruction Multiple Data)*.**
- **GPU's compete with many-core processors (e.g. Intel's Xeon Phi) Knights-Corner, Xeon-Phi 60 cores).**

```
if (COND) {
    BODY-TRUE;
} else {
    BODY-FALSE;
}
```



Xeon Phi

- In December 2012 Intel launched the Xeon Phi coprocessor card: 3100 and 5110P. (2000 USD to 2600 USD). It has 60 cores with 22nm technology (clock speed 1GHz approx). “Supercomputer on a card” (SOC).
- Today limitation is that(with 22nm technology) is that 5e9 transistors can be put on a sinle chip. Today Xeon processors have typically 2.5e9 transistors.
- Xeon Phi has 60 cores equivalent to the original Pentium processor (40e6 transistors).



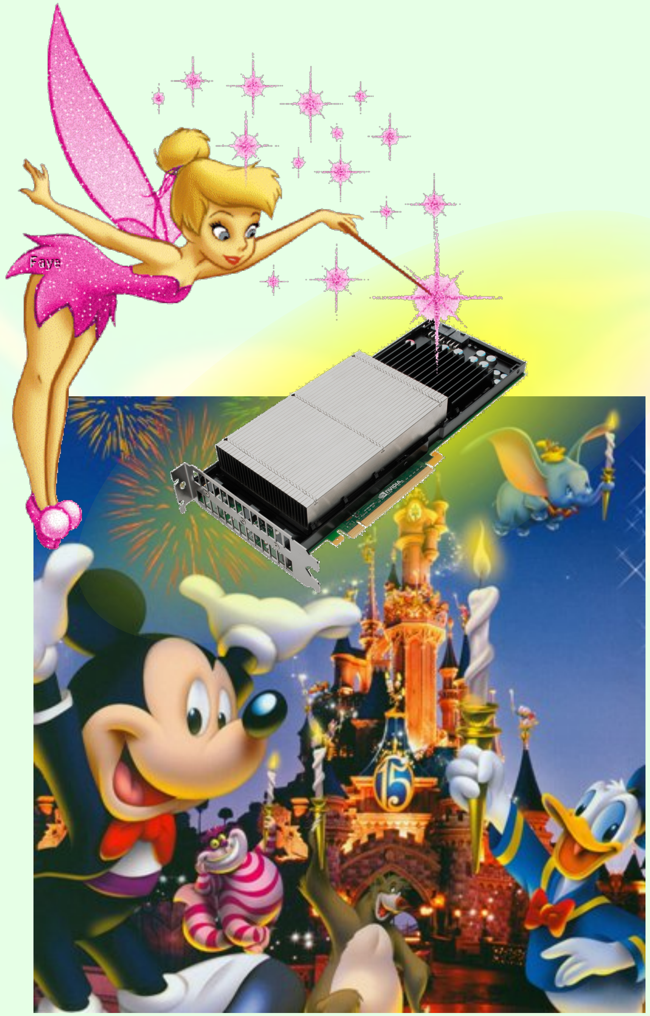
Xeon Phi (cont.)

- Xeon Phi is an *alien* computer. It fits in a PCI Express X 16 slot, and has its own basic Linux system. You can SSH to the card and run x86-64 code. Another workflow is to run the code in the host and send intensive computing tasks to the card (e.g. solving a linear system).
- On January 2013 Texas Advanced Computing Center (TACC) added Xeon Phi's to his Stampede supercomputer. Main CPUs are Xeon E5-2680. 128 nodes have Nvidia Kepler K20 GPUs. Estimated performance 7.6 Pflops. Tianhe-2 (China) the current fastes supercomputer (33.86 pflops) includes also Xeon Phi coprocessors.
- Part of Intel's Many Integrated Core (MIC) architecture. Previous codenames for the project: Larrabee, Knights Ferry, Knights-Corner.)

((version texstuff-1.2.1-144-g90c3aa5 Wed Mar 12 15:08:41 2014 +0100) (date Wed Mar 12 17:13:43 2014 +0100))

GPU's in HPC

- **Some HPC people are skeptical about the *efficient computing power* of GPU's for scientific applications.**
- **In many works *speedup* is referred to available CPU processors, which is not consistent.**
- **Delivered speedup w.r.t. mainstream x86 processors is often much lower than expected.**
- **Strict *data parallelism* is difficult to achieve on CFD applications.**
- **Unfortunately, this idea is reinforced by the fact that GPU's come from the videogame *special effects* industry, not with scientific computing.**



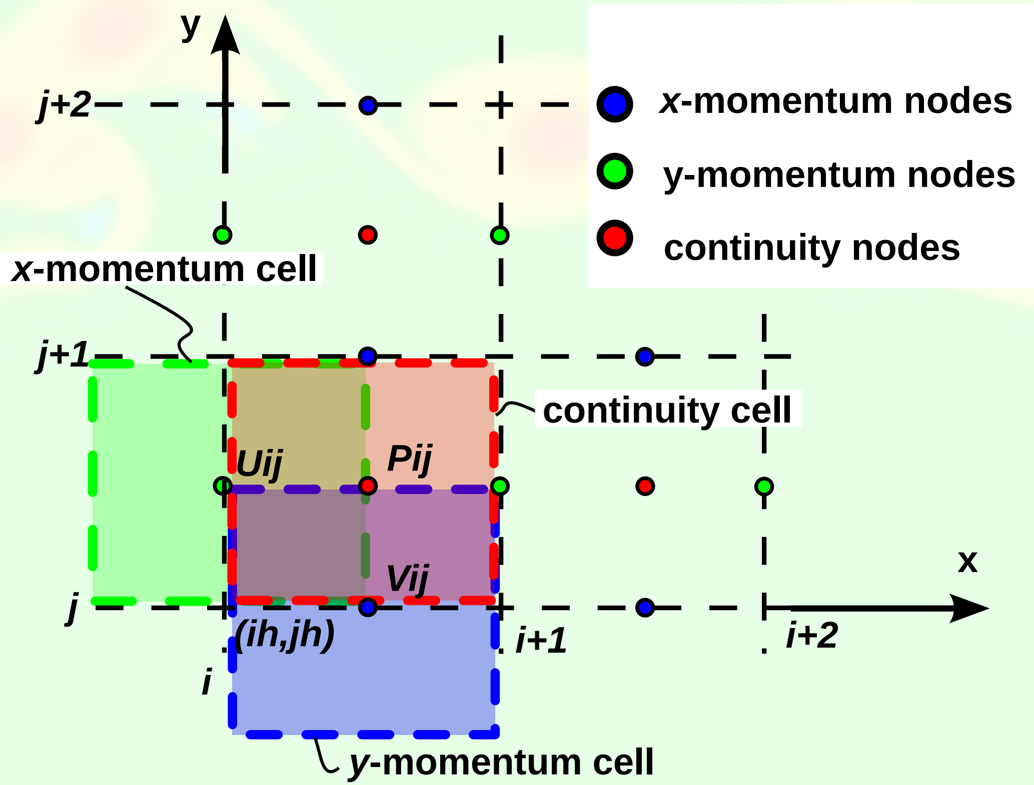
Solution of incompressible Navier-Stokes flows on GPU

- GPU's are less efficient for algorithms that require access to the *card's (device) global memory*. Shared memory is much faster but usually *scarce* (16K per thread block in the Tesla C1060) ☹.
- The best algorithms are those that make computations for one cell requiring only information on that cell and their neighbors. These algorithms are classified as *cellular automata (CA)*.
- *Lattice-Boltzmann* and *explicit F⋆M (FDM/FVM/FEM)* fall in this category.
- *Structured meshes* require less data to exchange between cells (e.g. neighbor indices are computed, no stored), and so, they require less shared memory. Also, very fast solvers like *FFT-based* (*Fast Fourier Transform*) or *Geometric Multigrid* are available ☺.

Fractional Step Method on structured grids with QUICK

Proposed by *Molemaker et.al. SCA'08: 2008 ACM SIGGRAPH,* Low viscosity flow simulations for animation.

- Fractional Step Method (a.k.a. pressure segregation)
- u, v, w and continuity cells are *staggered* (MAC=Marker And Cell).
- *QUICK* advection scheme is used in the predictor stage.
- Poisson system is solved with *IOP (Iterated Orthogonal Projection)* (to be described later), on top of *Geometric MultiGrid*



Solution of the Poisson with FFT

- Solution of the *Poisson equation* is, for large meshes, the more CPU consuming time stage in Fractional-Step like Navier-Stokes solvers.
- We have to solve a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$
- The Discrete Fourier Transform (DFT) is an orthogonal transformation $\hat{\mathbf{x}} = \mathbf{O}\mathbf{x} = \mathrm{fft}(\mathbf{x})$.
- The inverse transformation $\mathbf{O}^{-1} = \mathbf{O}^{T}$ is the inverse Fourier Transform $\mathbf{x} = \mathbf{O}^{T}\hat{\mathbf{x}} = \mathrm{ifft}(\hat{\mathbf{x}})$.
- If the operator matrix $\mathbf{A}$ is *spatially invariant* (i.e. the stencil is the same at all grid points) and the b.c.'s are periodic, then it can be shown that $\mathbf{O}$ diagonalizes $\mathbf{A}$, i.e. $\mathbf{O}\mathbf{A}\mathbf{O}^{-1} = \mathbf{D}$.
- So in the transformed basis the system of equations is diagonal

$$
\begin{aligned}
(\mathbf{O}\mathbf{A}\mathbf{O}^{-1})\,(\mathbf{O}\mathbf{x}) &= (\mathbf{O}\mathbf{b}),\\
\mathbf{D}\hat{\mathbf{x}} &= \hat{\mathbf{b}},
\end{aligned}
\tag{1}
$$

- For $N = 2^p$ the Fast Fourier Transform (FFT) is an algorithm that computes the DFT (and its inverse) in $O(N\log(N))$ operations.

CONICET

UNL

C I M E C

Solution of the Poisson with FFT (cont.)

- **So the following algorithm computes the solution of the system in $O(N \log(N))$ ops.**
  - $\hat{\mathbf{b}} = \mathrm{fft}(\mathbf{b})$, **(transform r.h.s)**
  - $\hat{\mathbf{x}} = \mathbf{D}^{-1}\hat{\mathbf{b}}$, **(solve diagonal system $O(N)$)**
  - $\mathbf{x} = \mathrm{ifft}(\hat{\mathbf{x}})$, **(anti-transform to get the sol. vector)**
- **Total cost: 2 FFT's, plus one element-by-element vector multiply (the reciprocals of the values of the diagonal of $\mathbf{D}$ are precomputed)**
- **In order to precompute the diagonal values of $\mathbf{D}$,**
  - **We take any vector $\mathbf{z}$ and compute $\mathbf{y} = \mathbf{A}\mathbf{z}$,**
  - **then transform $\hat{\mathbf{z}} = \mathrm{fft}(\mathbf{z})$, $\hat{\mathbf{y}} = \mathrm{fft}(\mathbf{y})$,**
  - $D_{jj} = \hat{y}_j / \hat{z}_j$.

Solution of the Poisson equation on embedded geometries

- **FFT solver and GMG are very fast but have several restrictions: invariance of translation, periodic boundary conditions. They are not well suited for embedded geometries.**
- **One approach for the solution is the *IOP (Iterated Orthogonal Projection)* algorithm.**
- **It is based on solving iteratively the Poisson eq. on the *whole domain (fluid+solid)*. Solving in the whole domain is fast, because algorithms like Geometric Multigrid or FFT can be used. Also, they are very efficient running on GPU's .**
- **However, if we solve in the whole domain, then we can't enforce the boundary condition $(\partial p/\partial n) = 0$ at the solid boundary which, then means the violation of the *condition of impenetrability at the solid boundary* .**

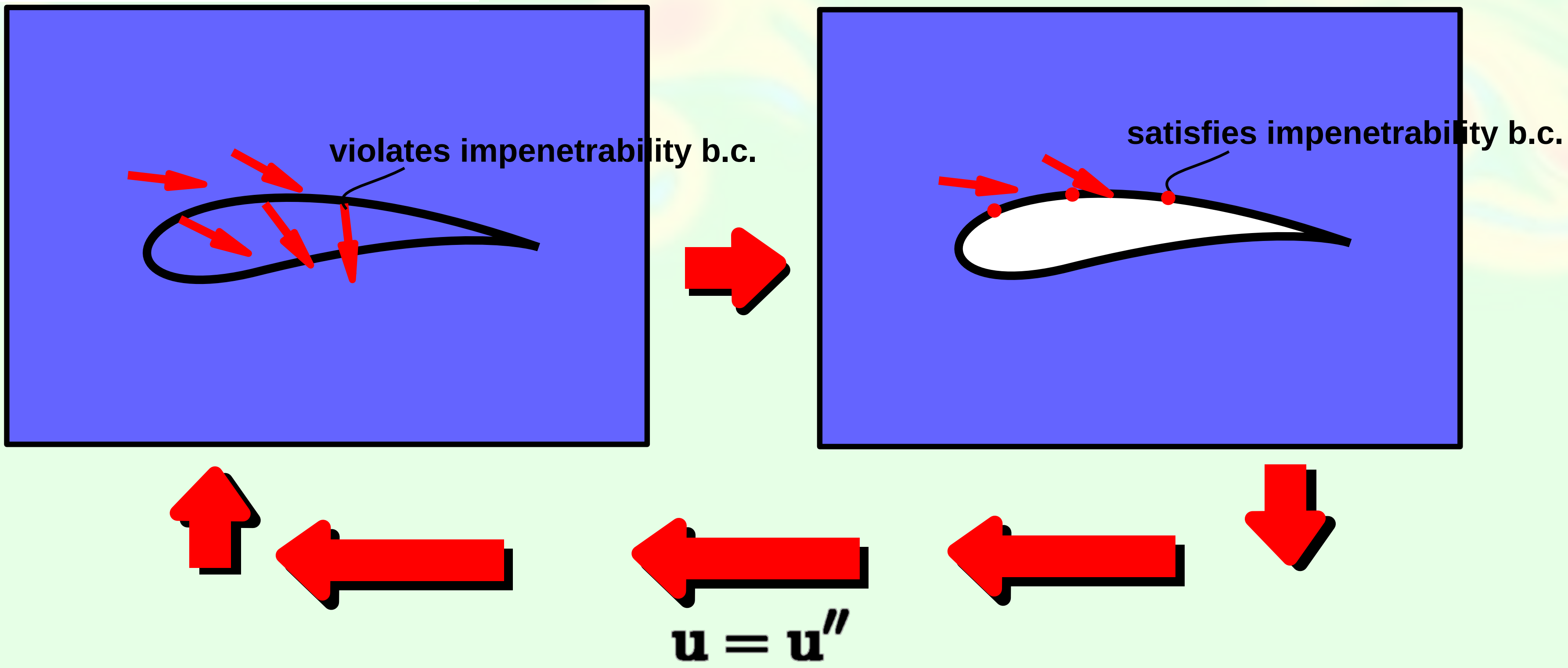
The IOP (Iterated Orthogonal Projection) method

The method is based on succesively solve for the incompressibility condition (on the whole domain: solid+fluid), and impose the boundary condition.

$$\mathbf{u}' = \mathbf{\Pi}_{\mathrm{div}}(\mathbf{u}) \begin{cases} \mathbf{u}' = \mathbf{u} - \nabla P, \\ \Delta P = \nabla \cdot \mathbf{u}, \end{cases}$$

on the whole domain (fluid+solid)

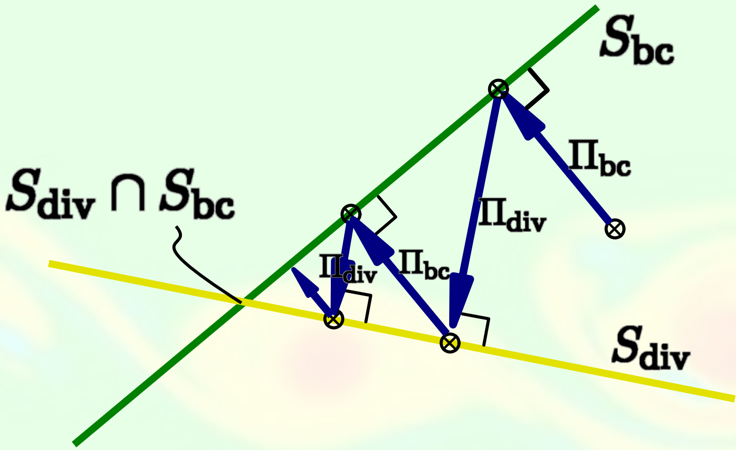
$$\mathbf{u}'' = \mathbf{\Pi}_{\mathrm{bdy}}(\mathbf{u}') \begin{cases} \mathbf{u}'' = \mathbf{u}_{\mathrm{bdy}}, \ \text{in } \Omega_{\mathrm{bdy}}, \\ \mathbf{u}'' = \mathbf{u}', \ \text{in } \Omega_{\mathrm{fluid}}. \end{cases}$$



The IOP (Iterated Orthogonal Projection) method (cont.)

- **Fixed point iteration**

$$\mathbf{w}^{k+1} = \mathbf{\Pi}_{\rm bdy}\mathbf{\Pi}_{\rm div}\mathbf{w}^k.$$

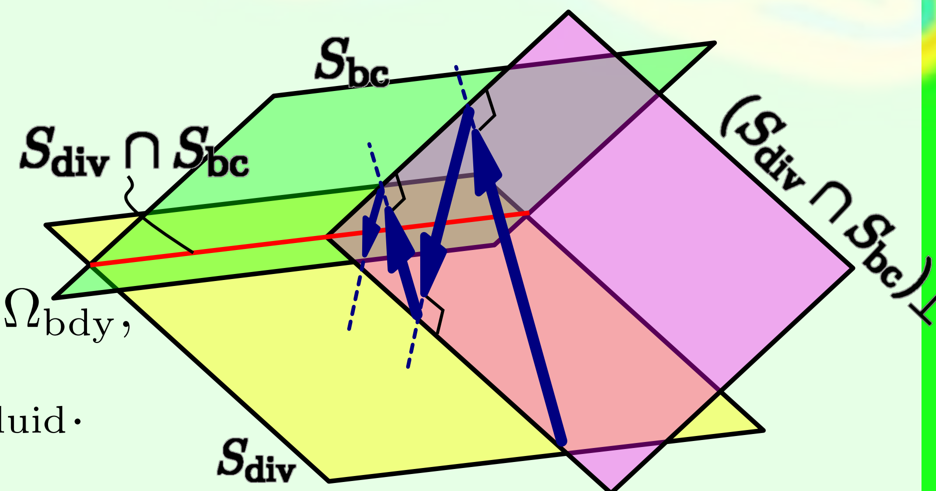


- **Projection on the space of *divergence-free* velocity fields:**

$$\mathbf{u}' = \mathbf{\Pi}_{\rm div}(\mathbf{u}) \begin{cases} \mathbf{u}' = \mathbf{u} - \nabla P, \\ \Delta P = \nabla \cdot \mathbf{u}, \end{cases}$$

- **Projection on the space of velocity fields that satisfy the *impenetrability boundary condition***

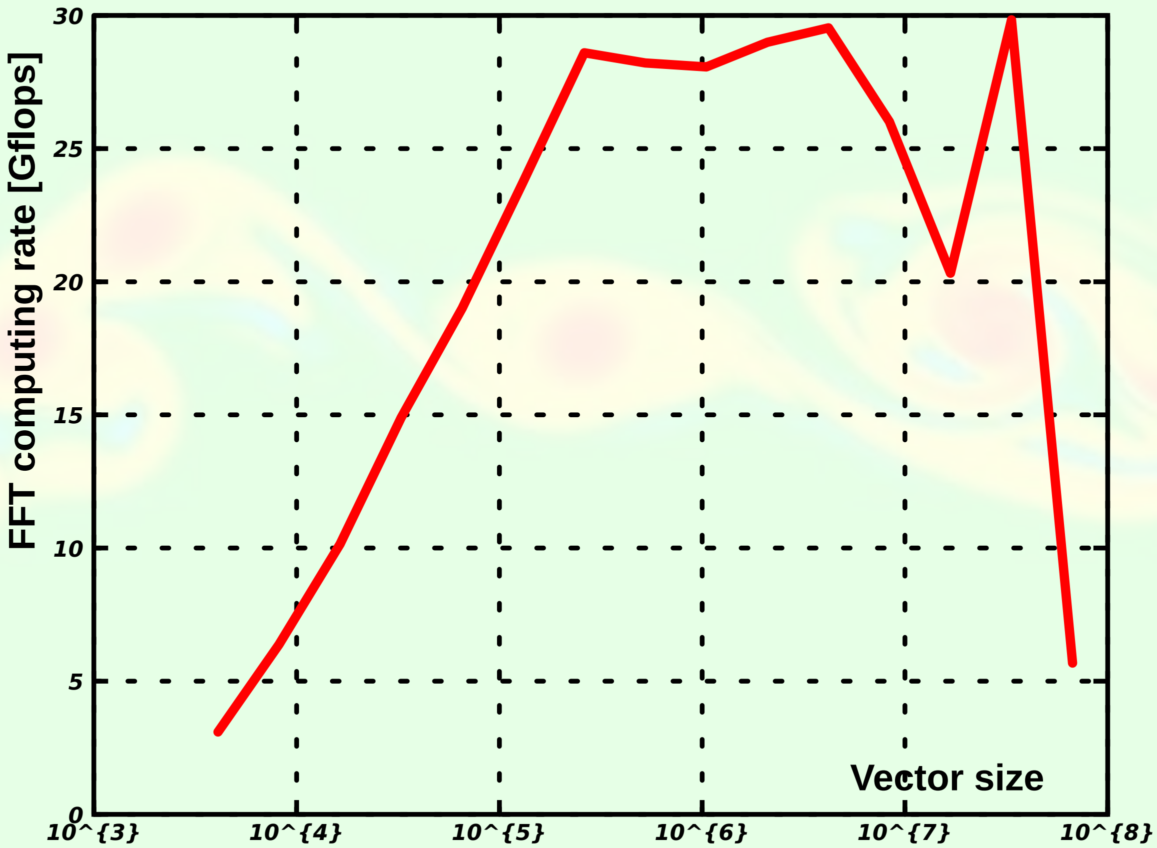
$$\mathbf{u}'' = \mathbf{\Pi}_{\rm bdy}(\mathbf{u}') \begin{cases} \mathbf{u}'' = \mathbf{u}_{\rm bdy}, & \text{in } \Omega_{\rm bdy}, \\ \mathbf{u}'' = \mathbf{u}', & \text{in } \Omega_{\rm fluid}. \end{cases}$$



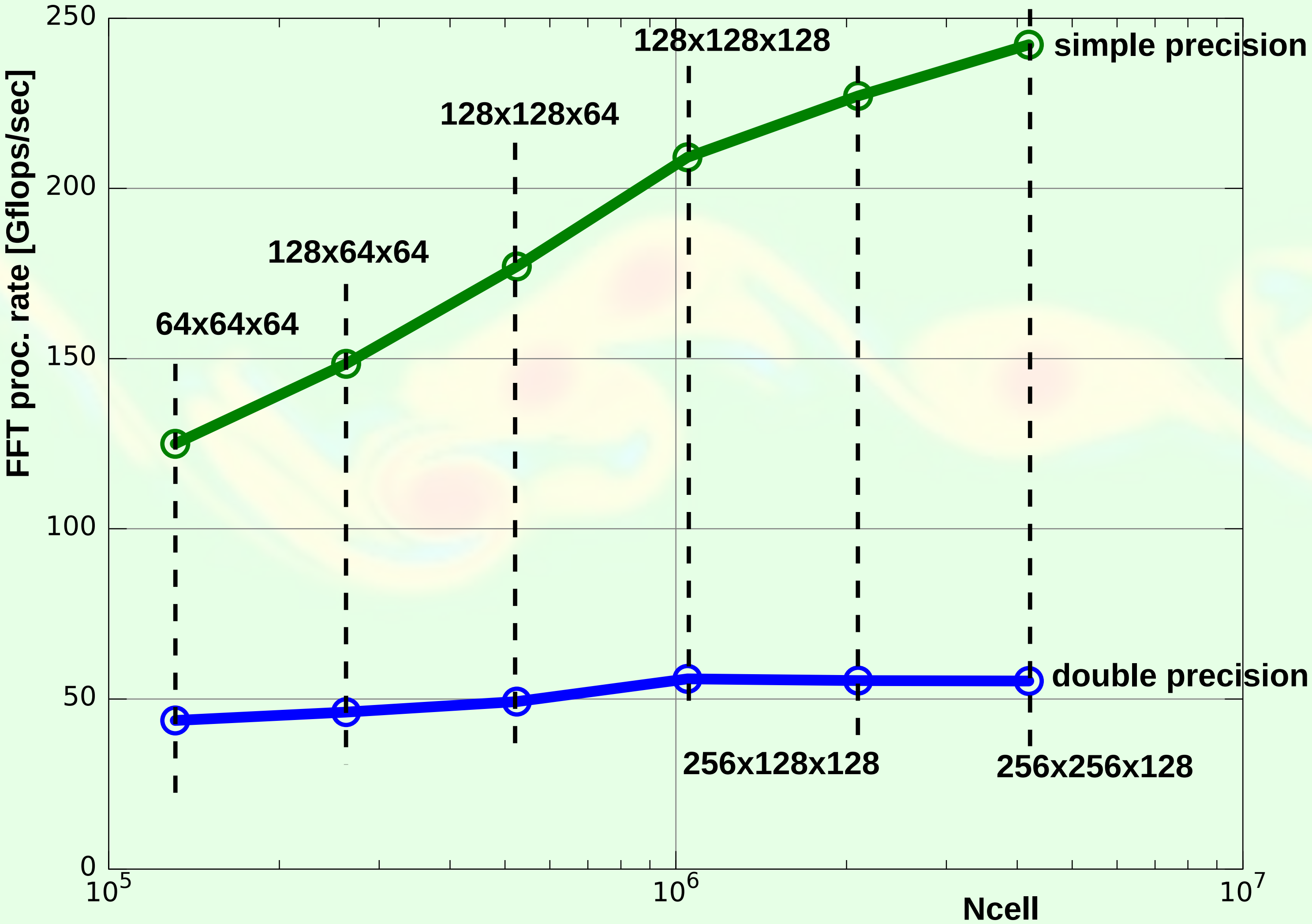
((version texstuff-1.2.1-144-g90c3aa5 Wed Mar 12 15:08:41 2014 +0100) (date Wed Mar 12 17:13:43 2014 +0100))

Implementation details on the GPU

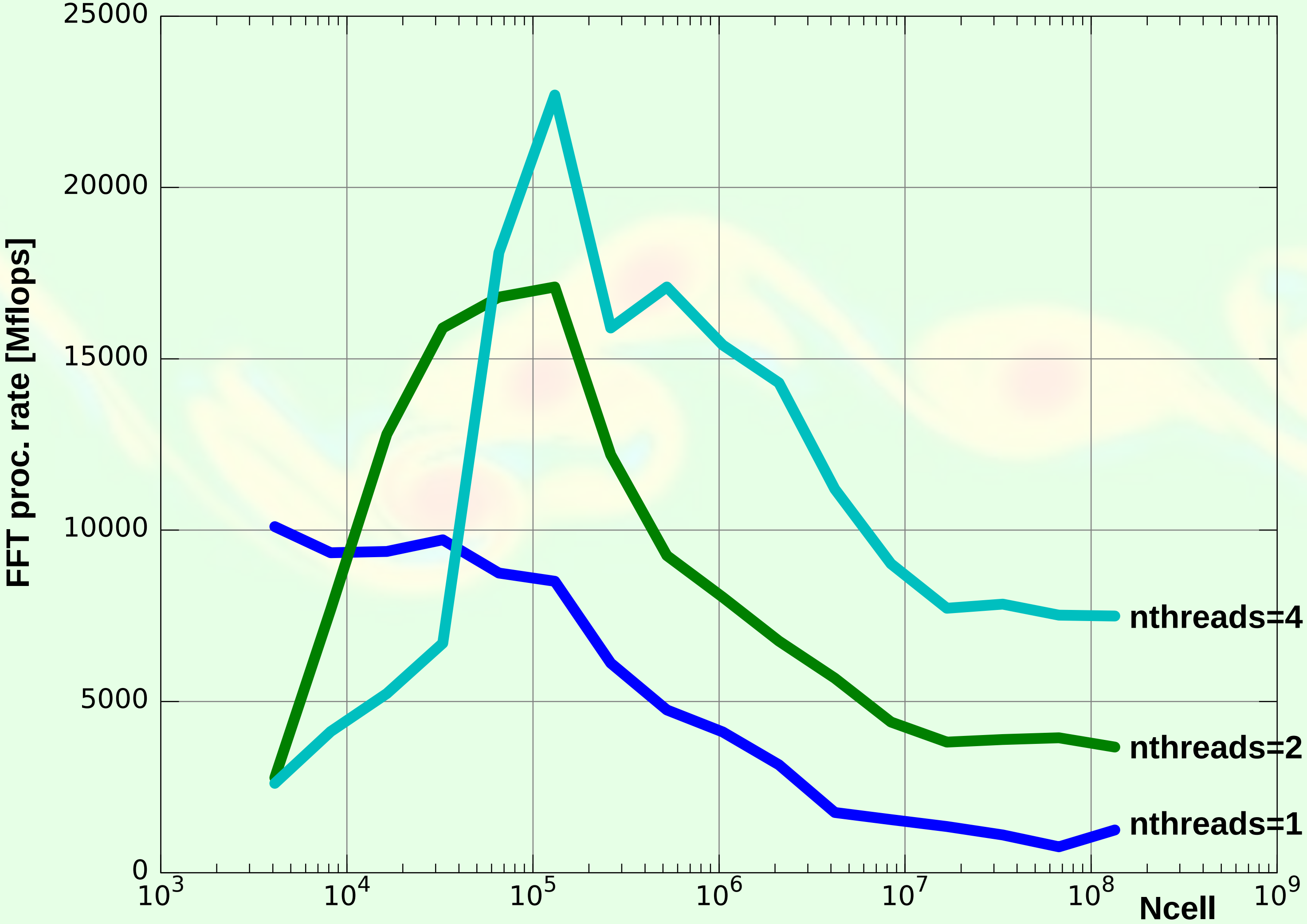
- We use the ***CUFFT library***.
- Per iteration: 2 FFT's and Poisson residual evaluation. The FFT on the ***GPU Tesla C1060*** performs at ***27 Gflops***, (in double precision) where the operations are counted as $5N\log_2(N)$.



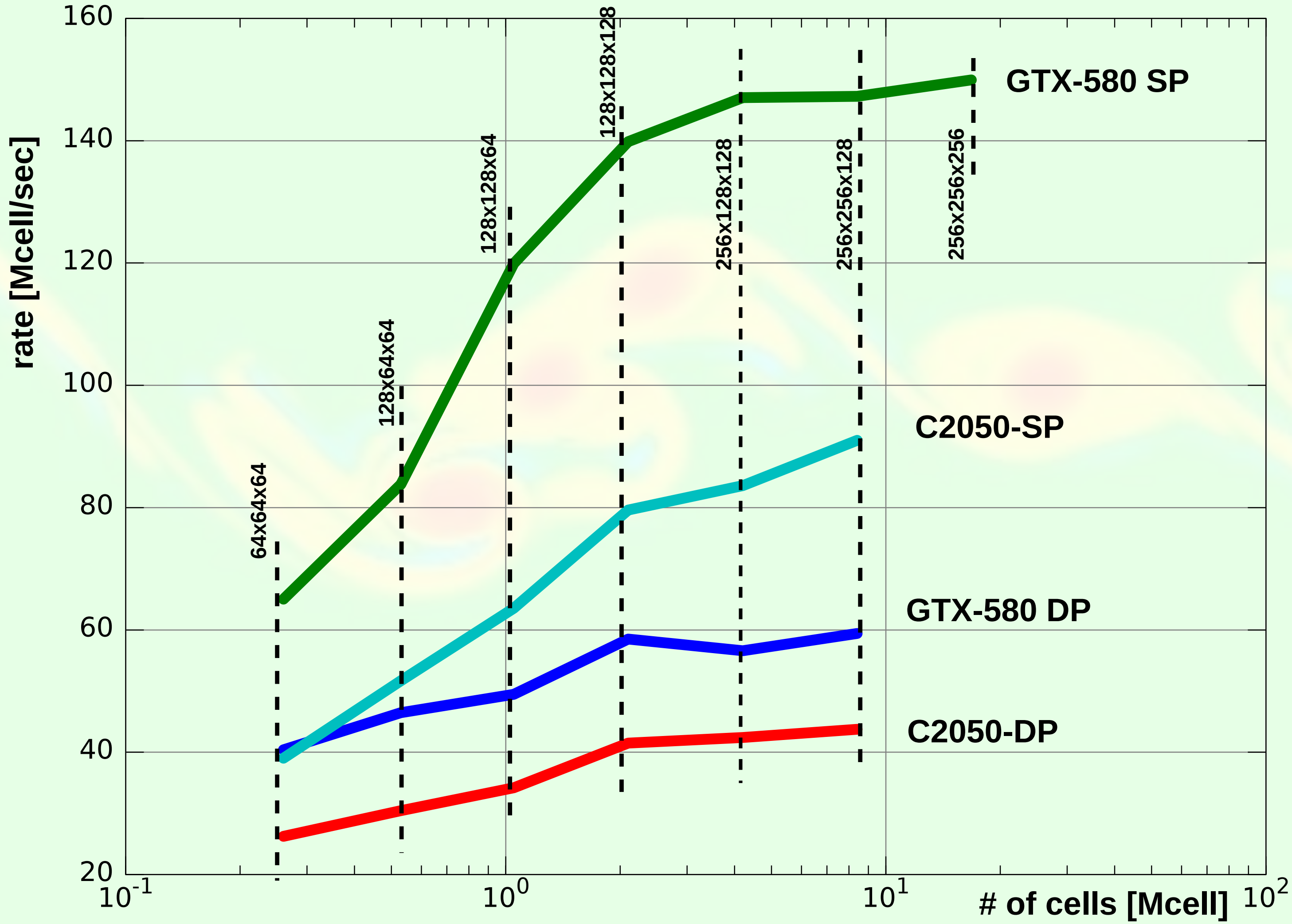
FFT computing rates in GPGPU. GTX-580



FFTW on i7-3820@3.60Ghz (Sandy Bridge)



NSFVM Computing rates in GPGPU. Scaling



NSFVM and “Real Time” computing

- For a 128x128x128 mesh ($\approx$ 2Mcell), we have a computing time of 2 Mcell/(140 Mcell/sec) = 0.014 secs/time step.
- That means 70 steps/sec.
- A von Neumann stability analysis shows that the QUICK stabilization scheme is inconditionally stable if advanced in time with Forward Euler.
- With a second order Adams-Bashfort scheme the critical CFL is 0.588.
- For NS eqs. the critical CFL has been found to be somewhat lower ($\approx$ 0.5).
- If $L = 1$, $u = 1$, $h = 1/128$, $\Delta t = 0.5h/u = 0.004$ [sec], so that we can compute in 1 sec, 0.28 secs of simulation time. We say ST/RT=0.28.

(launch video nsfvm-bodies-all),

CONICET

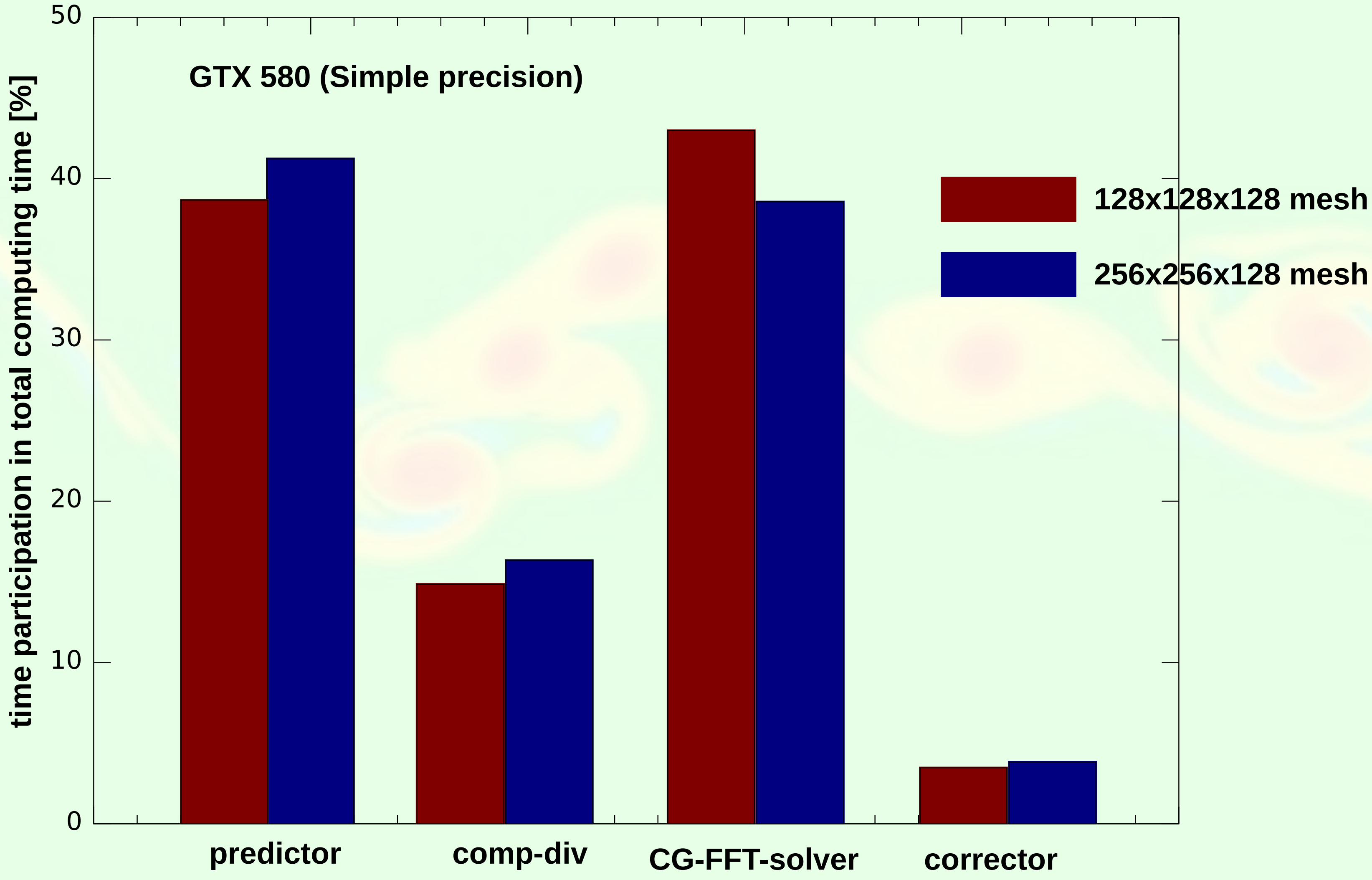
C I M E C

NSFVM and "Real Time" computing (cont.)

Descripcion	video	Malla	Ncell	2D/3D?	Umax	CFL	Rate [Mcell/sec]	Tcomp/Tsim	Tvideo/Tsim	Tcomp/Tvideo
Cylinder moving randomly in a square cavity	vrtx3d-cylinder.avi	128x128	16K	2D	3	0.5	90	**0.14**	**1.27**	**0.11**
2-D Flow around a moving square body	vrtx3d-moving-square.avi	128x128	16K	2D	0.66	0.5	90	**0.031**	**1.6**	**0.019**
3-D Falling block off centered	falling-block-offcentered.avi	128x128x128	2M	3D	3	0.5	140	**11.5**	**10**	**1.15**
3-D Cube moving randomly in a 3-D cavity	moving-cube-random.avi	128x128x128	2M	3D	3.8	0.5	140	**14.5**	**5**	**3**
2-D Flow around a cylinder at Re=1000	cylinder-nsfvm-re1000.avi	256x1024	262K	2D	2	0.5	90	**3**	**3.52**	**0.85**

((version texstuff-1.2.1-144-g90c3aa5 Wed Mar 12 15:08:41 2014 +0100) (date Wed Mar 12 17:13:43 2014 +0100))

Computing times in GPGPU. Fractional Step components



Current work

Current work is done in the following directions

- **Improving performance by replacing the *QUICK* advection scheme by *MOC+BFECC* (which could be more GPU-friendly).**
- **Implementing a CPU-based *renormalization* algorithm for free surface (level-set) flows.**
- **Another important issue is improving the representation (accuracy) of the solid body surface by using an *immersed boundary* technique.**

((version texstuff-1.2.1-144-g90c3aa5 Wed Mar 12 15:08:41 2014 +0100) (date Wed Mar 12 17:13:43 2014 +0100))

Why leave QUICK?

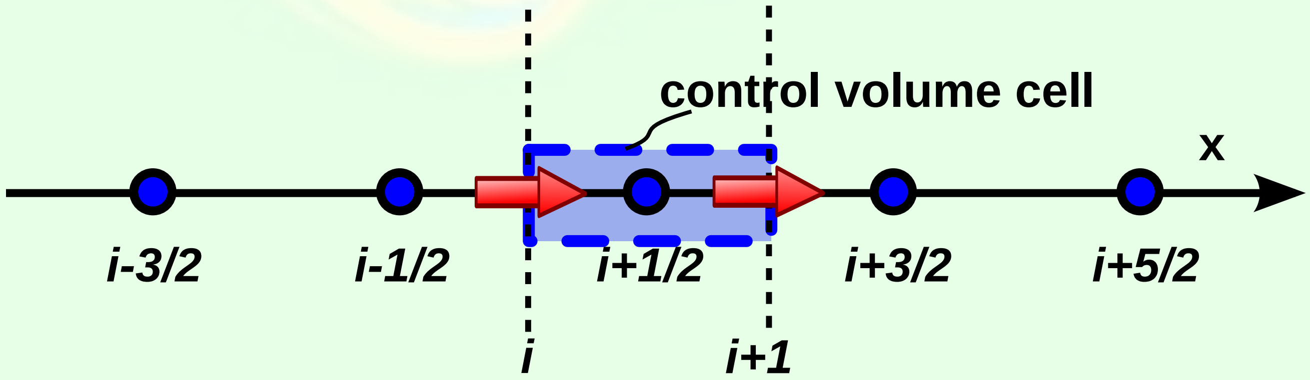
- **One of steps of the Fractional Steps Method is the advection step. We have to advect the velocity field and we desire a method as less diffusive as possible, and that allows as large the CFL number as possible.**
- **Also, of course, we want a GPU friendly algorithm.**
- **Previously we used QUICK, but it has a stencil that extends more than one cell in the upwind direction. This increases *shared memory* usage and data transfer. We seek for another low dissipation scheme with a more compact stencil.**

Quick advection scheme

1D Scalar advection diffusion: a= **advection velocity,** ϕ **advected scalar.**

$$\frac{\partial}{\partial x}(a\phi)\Big|_{i+1/2} \approx \frac{(a\phi^Q)_{i+1} - (a\phi^Q)_i}{\Delta x},$$

$$\phi_i^Q = \begin{cases} 3/8\,\phi_{i+1/2} + 6/8\,\phi_{i-1/2} - 1/8\,\phi_{i-3/2}, & \textbf{if } a > 0, \\ 3/8\,u_{i-1/2} + 6/8\,u_{i+1/2} - 1/8\,u_{i+3/2}, & \textbf{if } a < 0, \end{cases}$$



Method Of Characteristics (MOC)

- The *Method Of Characteristics (MOC)* consists in tracking the position of the node following the characteristics to the position it had at time t^n and taking its value there,

$$\Phi(\mathbf{x}^{n+1}, t^{n+1}) = \Phi(\mathbf{x}^n, t^n)$$

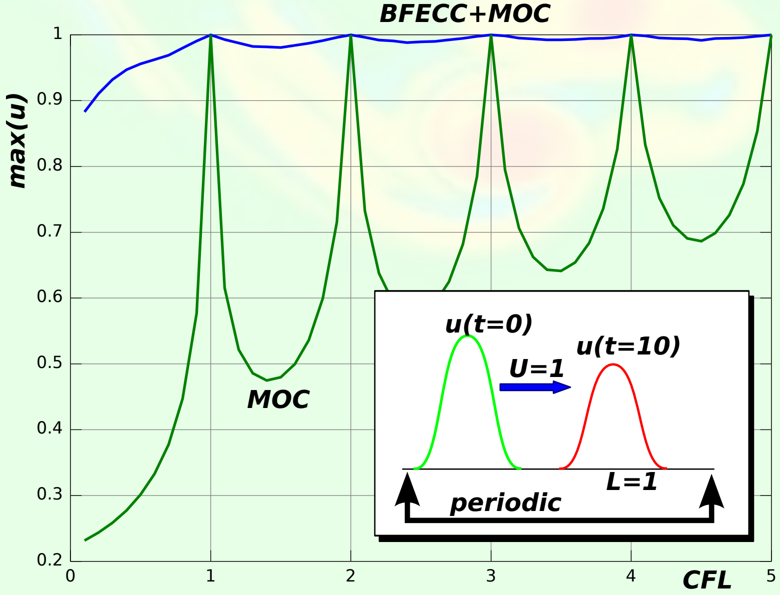
  If x^n doesn't happen to be a mesh node it involves a *projection*.
- It's the basis of the *Lagrangian* methods for dealing with advection terms.

((version texstuff-1.2.1-144-g90c3aa5 Wed Mar 12 15:08:41 2014 +0100) (date Wed Mar 12 17:13:43 2014 +0100))

Method Of Characteristics (MOC) (cont.)

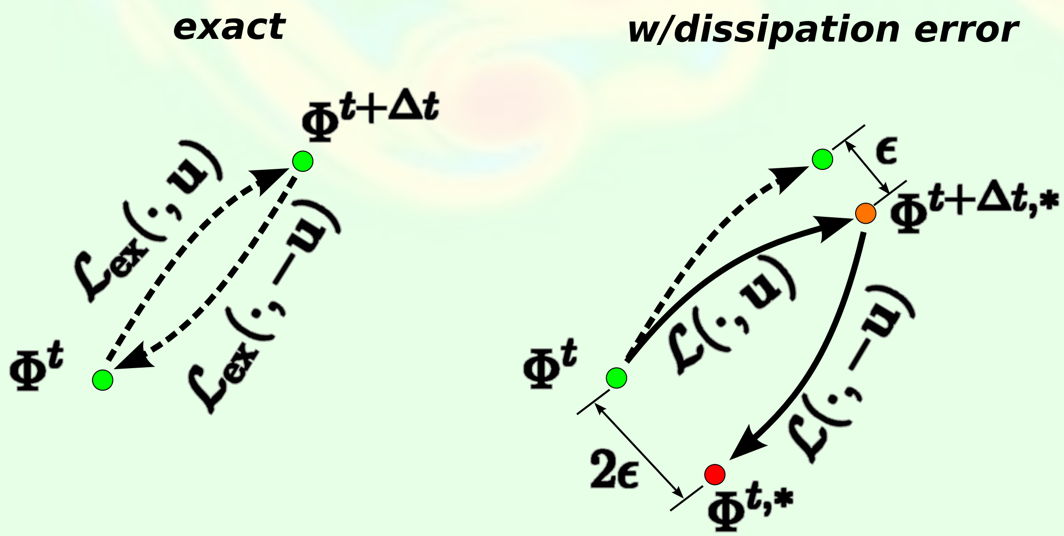
- **So typically MOC has very low diffusion if CFL is an integer number ☺, and too diffusive if it is an semi-integer number ☹.**
- **Of course, in the general case (non uniform meshes, non uniform velocity field) we can't manage to have an integer CFL number for all the nodes.**

(launch video video-moc-cfl1), *(launch video video-moc-cfl05)*.

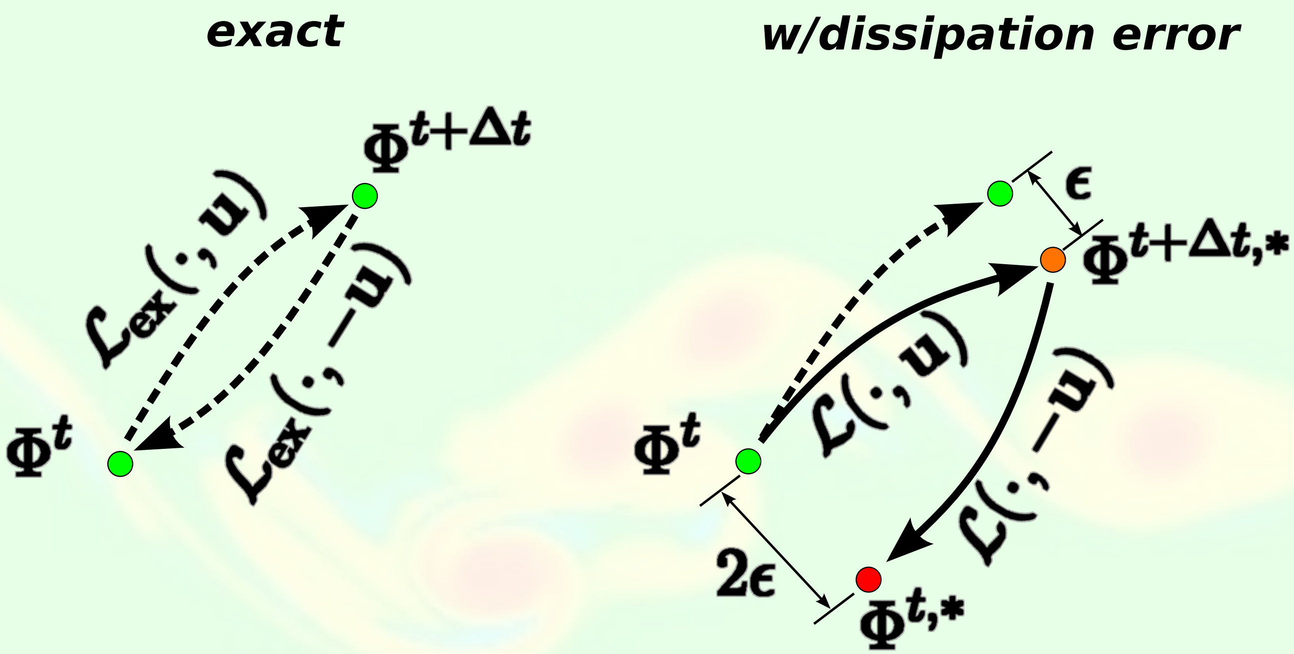


MOC+BFECC

- Assume we have a *low order (dissipative)* operator (may be SUPG, MOC, or any other) $\Phi^{t+\Delta t} = \mathcal{L}(\Phi^t, \mathbf{u})$.
- The *Back and Forth Error Compensation and Correction (BFECC)* allows to eliminate the dissipation error.
  - ▷ Advance *forward* the state $\Phi^{t+\Delta t,*} = \mathcal{L}(\Phi^t, \mathbf{u})$.
  - ▷ Advance *backwards* the state $\Phi^{t,*} = \mathcal{L}(\Phi^{t+\Delta t,*}, -\mathbf{u})$.



MOC+BFECC (cont.)

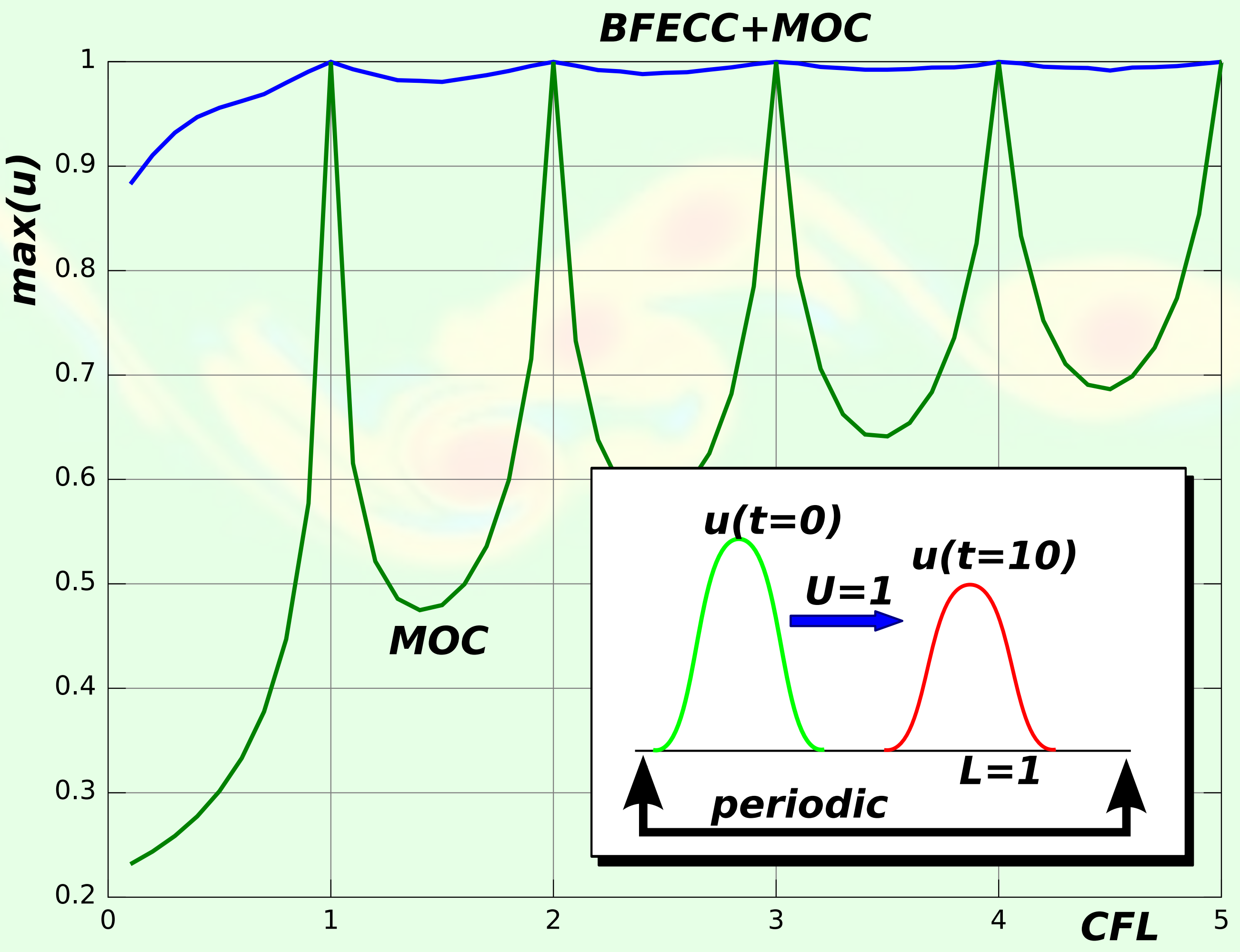


- If $\mathcal{L}$ introduces some *dissipative* error ϵ, then $\Phi^{t,*} \neq \Phi^t$, in fact $\Phi^{t,*} = \Phi^t + 2\epsilon$.
- So that we can *compensate* for the error:

$$\begin{aligned} \Phi^{t+\Delta t} &= \mathcal{L}(\Phi^t, \Delta t) - \epsilon, \\ &= \Phi^{t+\Delta t,*} - 1/2(\Phi^{t,*} - \Phi^t) \end{aligned} \tag{2}$$

MOC+BFECC (cont.)

(launch video video-moc-bfecc-cfl05).



MOC+BFECC (cont.)

Nbr of Cells	QUICK-SP	BFECC-SP	QUICK-DP	BFECC-DP
$64 \times 64 \times 64$	**29.09**	**12.38**	**15.9**	**5.23**
$128 \times 128 \times 128$	**75.74**	**18.00**	**28.6**	**7.29**
$192 \times 192 \times 192$	**78.32**	**17.81**	**30.3**	**7.52**

Cubic cavity. Computing rates for the whole NS solver (one step) in [Mcell/sec] obtained with the BFECC and QUICK algorithms on a NVIDIA GTX 580. 3 Poisson iterations were used.

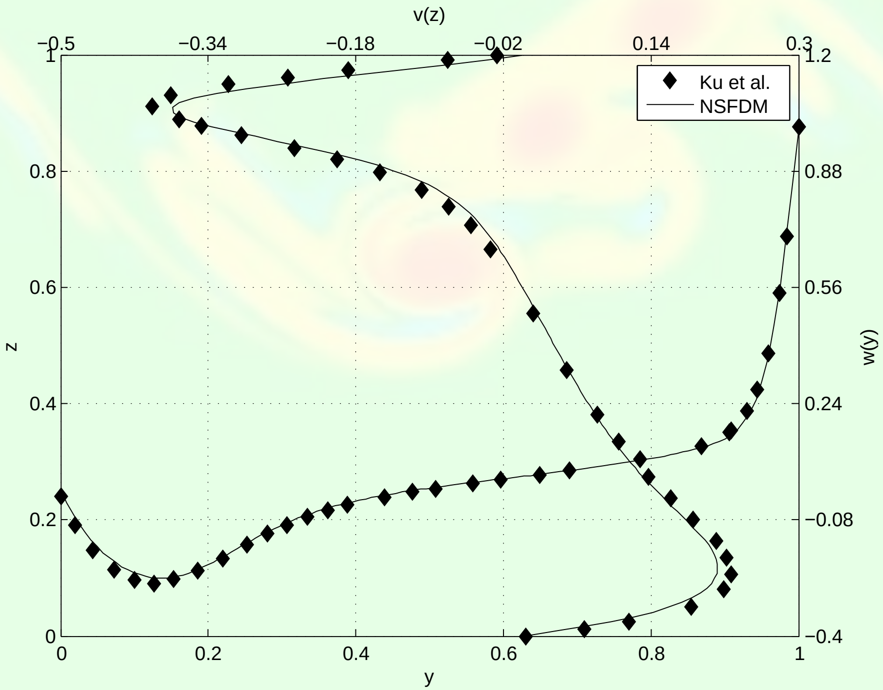
[jump to Conclusions]

Analysis of performance

- Regarding the performance results shown in above, it can be seen that the computing rate of QUICK is at most *4x faster* than that of BFECC. So BFECC is more efficient than QUICK whenever used with *CFL > 2*, being the *critical CFL for QUICK 0.5*. The CFL used in our simulations is typically CFL ≈ 5 and, thus, at this CFL the *BFECC version runs 2.5 times faster than the QUICK* version.
- The speedup of MOC+BFECC versus QUICK *increases with the number of Poisson iterations*. In the limit of very large number of iters (very low tolerance in the tolerance for Poisson) we expect a *speedup 10x* (equal to the CFL ratio).

Validation. Lid driven 3D cubic cavity

- **Re=1000, mesh of $128 \times 128 \times 128$ (2 Mcell). Results compared with Ku et.al (JCP 70(42):439-462 (1987)).**
- **More validation and complete performance study at *Costarelli et.al, Cluster Computing (2013), DOI:10.1007/s10586-013-0329-9.***



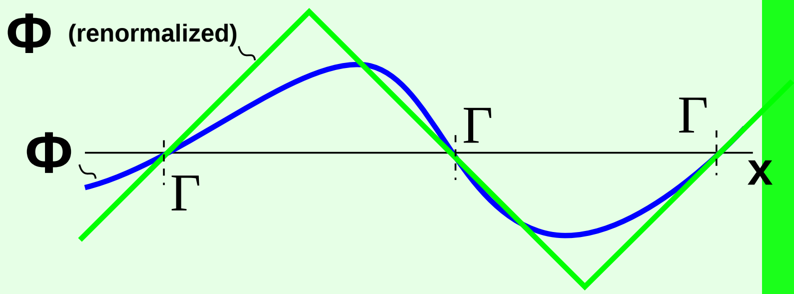
((version texstuff-1.2.1-144-g90c3aa5 Wed Mar 12 15:08:41 2014 +0100) (date Wed Mar 12 17:13:43 2014 +0100))

Renormalization

Even with a high precision, low dissipative algorithm for transporting the level set function Φ we have to *renormalize* $\Phi \to \Phi'$ with a certain frequency the level set function.

- **Requirements on the renormalization algorithm are:**
  - ▷ **Φ' must preserve as much as posible the 0 level set function (interface) Γ.**
  - ▷ **Φ' must be as regular as possible near the interface.**
  - ▷ **Φ' must have a high slope near the interface.**
  - ▷ **Usually the signed distance function is used, i.e.**

$$\Phi'(\mathbf{x}) = \mathrm{sign}(\Phi(\mathbf{x})) \min_{\mathbf{y}\in\Gamma} ||\mathbf{y} - \mathbf{x}||$$

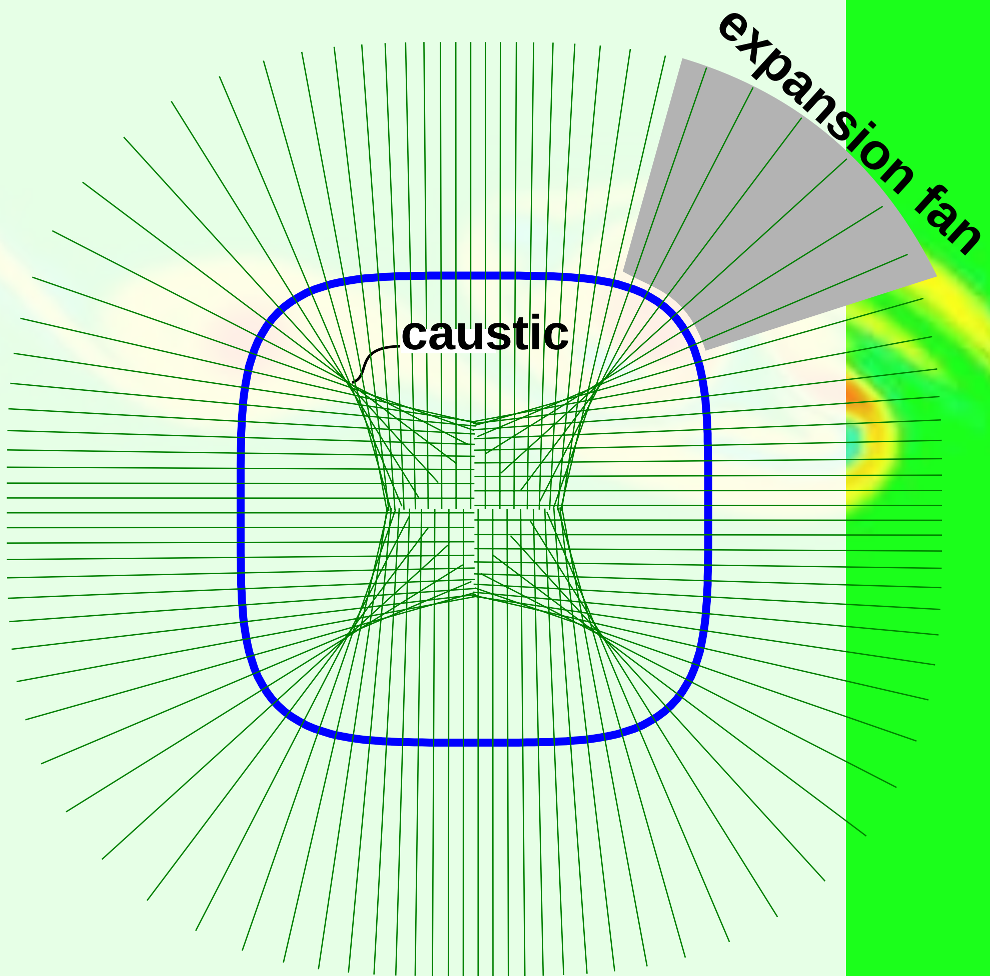


Renormalization (cont.)

- **Computing plainly the distance function is $O(N N_\Gamma)$ where N_Γ is the number of points on the interface. This scales typically $\propto N^{1+(n_d-1)/n_d}$ ($N^{5/3}$ in 3D).**
- **Many variants are based in solving the Eikonal equation**

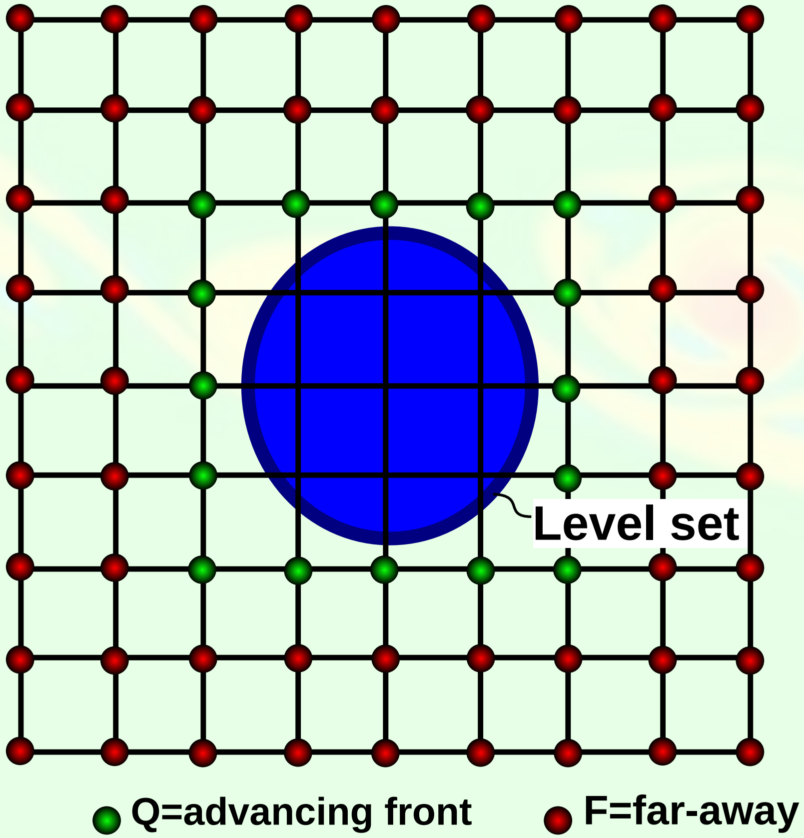
$$|\nabla\Phi| = 1,$$

- **As it is an hyperbolic equation it can be solved by a *marching* technique. The algorithm traverses the domain with an *advancing front* starting from the level set.**
- **However, it can develop *caustics* (*shocks*), and *rarefaction waves*. So, an *entropy condition* must be enforced.**



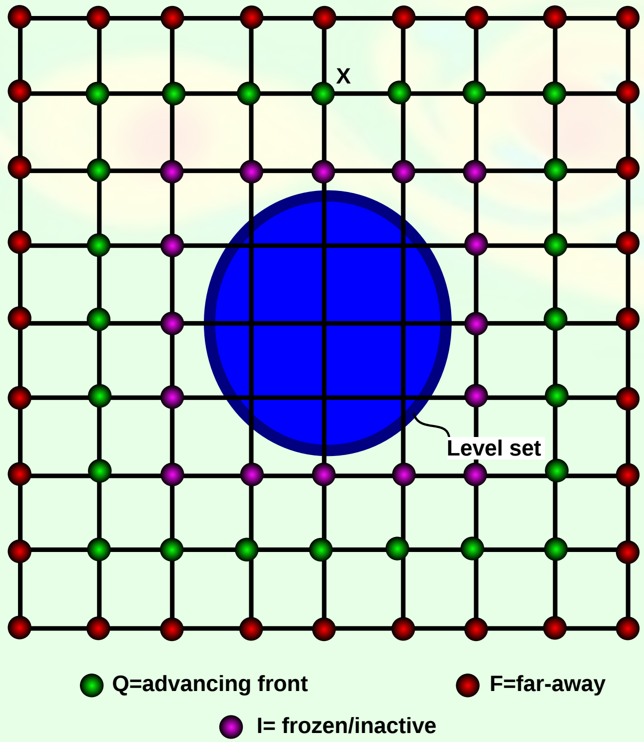
Renormalization (cont.)

- The ***Fast Marching*** algorithm proposed by **Sethian (Proc Nat Acad Sci 93(4):1591-1595 (1996))** , is a ***fast*** (near optimal) algorithm based on ***Dijkstra's algorithm*** for computing minimum distances in graphs from a source set. (Note: the original Dijkstra's algorithm is $O(N^2)$, not fast. The fast version using a priority queue is due to **Fredman and Tarjan (ACM Journal 24(3):596-615, 1987)**, and the complexity is $O(N\log(|Q|)) \sim O(N\log(N))$.



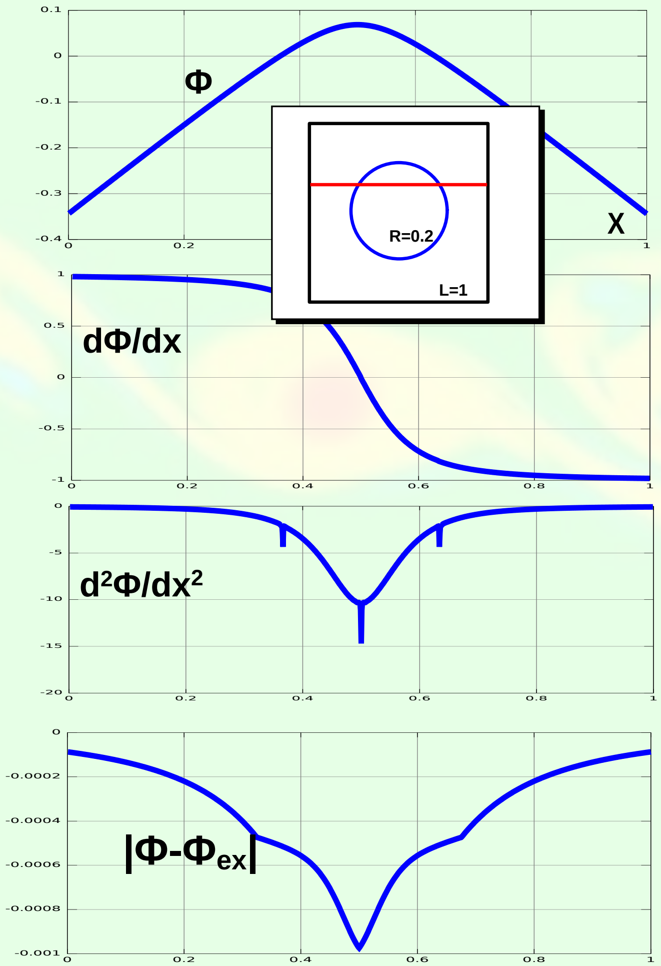
The Fast Marching algorithm

- **We explain for the positive part $\Phi > 0$. Then the algorithm is reversed for $\Phi < 0$.**
- **All nodes are in either: Q=*advancing front*, F=*far-away* , I=*frozen/inactive*. The advancing front sweeps the domain starting at the level set and converts F nodes to I.**
- **Initially $Q = \{$nodes that are in contact with the level set$\}$. Their distance to the interface is computed for each cut-cell. The rest is in $F =$*far-away*.**
- ***loop*: Take the node X in Q closest to the interface. Move it from $Q \to I$.**
- **Update all distances from neighbors to X and move them from $F \to Q$.**
- **Go to *loop*.**
- **Algorithm ends when $Q = \emptyset$.**



FastMarch: error and regularity of the distance function

- **Numerical example shows regularity of computed distance function in a mesh of 100x100.**
- **We have a LS consisting of a circle $R = 0.2$ inside a square of $L = 1$.**
- **Φ is shown along the $x = 0.6$ cut of the geometry, also we show the first and second derivatives.**
- **Φ deviates less than 10^{-3} from the analytical distance.**
- **Small spikes are observed in the second derivative.**
- **The error $\Phi - \Phi_{\rm ex}$ shows the discontinuity in the slope at the LS.**

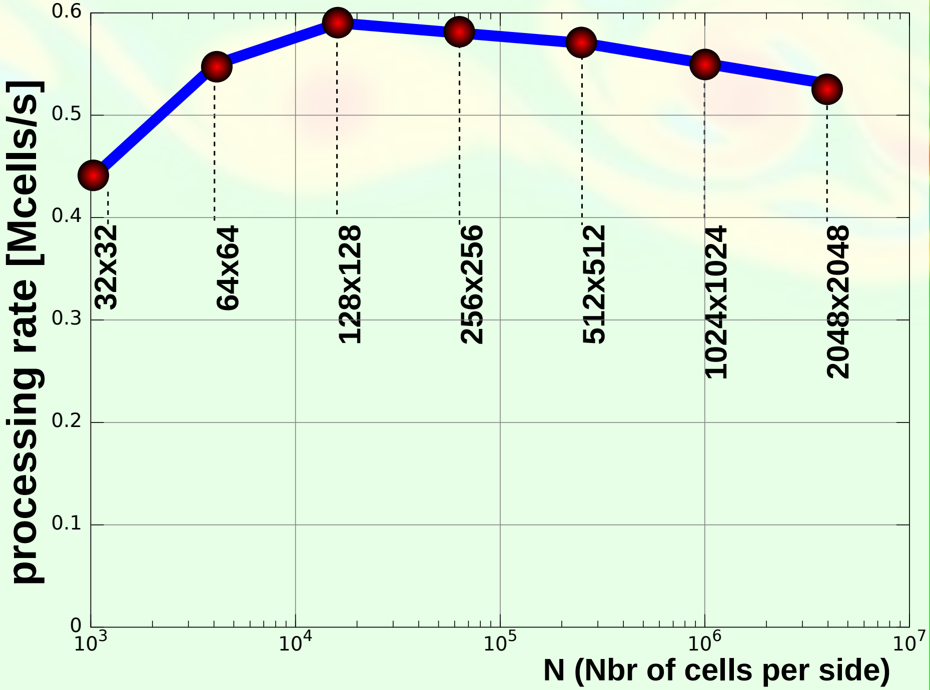


FastMarch: implementation details

- **Complexity is $O(N)\times$ the cost of finding the node in Q closest to the level set.**
- **This can be implemented in a very efficient way with a *priority queue* implemented in top of a *heap*. In this way finding the closest node is $O(\log|Q|)$. So the total cost is $O(N\log|Q|) \leq O(N\log(N^{(n_d-1)/n_d})) = O(N\log N^{2/3})$ (in 3D).**
- **The standard C++ class `priority_queue<>` is not appropriate because don't give access to the elements in the queue.**
- **We implemented the heap structure on top of a `vector<>` and an `unordered_map<>` (hash-table based) that tracks the Q-nodes in the structure. The hash function used is very simple.**

FastMarch renorm: Efficiency

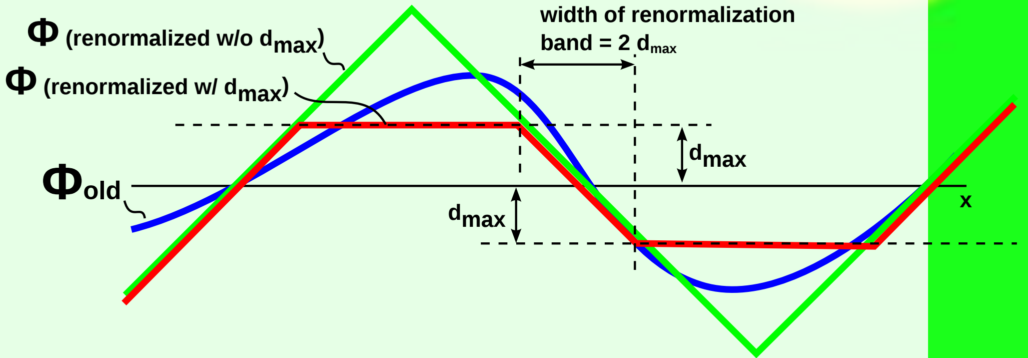
- The ***Fast Marching*** **algorithm is** $O(N \log |Q|)$ **where** N **is the number of cells and** $|Q|$ **the size of the advancing front.**
- **Rates were evaluated in an Intel i7-950@3.07 (Nehalem).**
- **Computing rate is practically constant and even decreases with high** N.
- **Since the rate for the NS-FVM algorithm is** $>$**100 [Mcell/s], renormalization at a frequency greater than 1/200 steps would be too expensive.**
- **Cost of renormalization step is reduced with** ***band renormalization*** **and** ***parallelism (SMP)*****.**



((version texstuff-1.2.1-144-g90c3aa5 Wed Mar 12 15:08:41 2014 +0100) (date Wed Mar 12 17:13:43 2014 +0100))

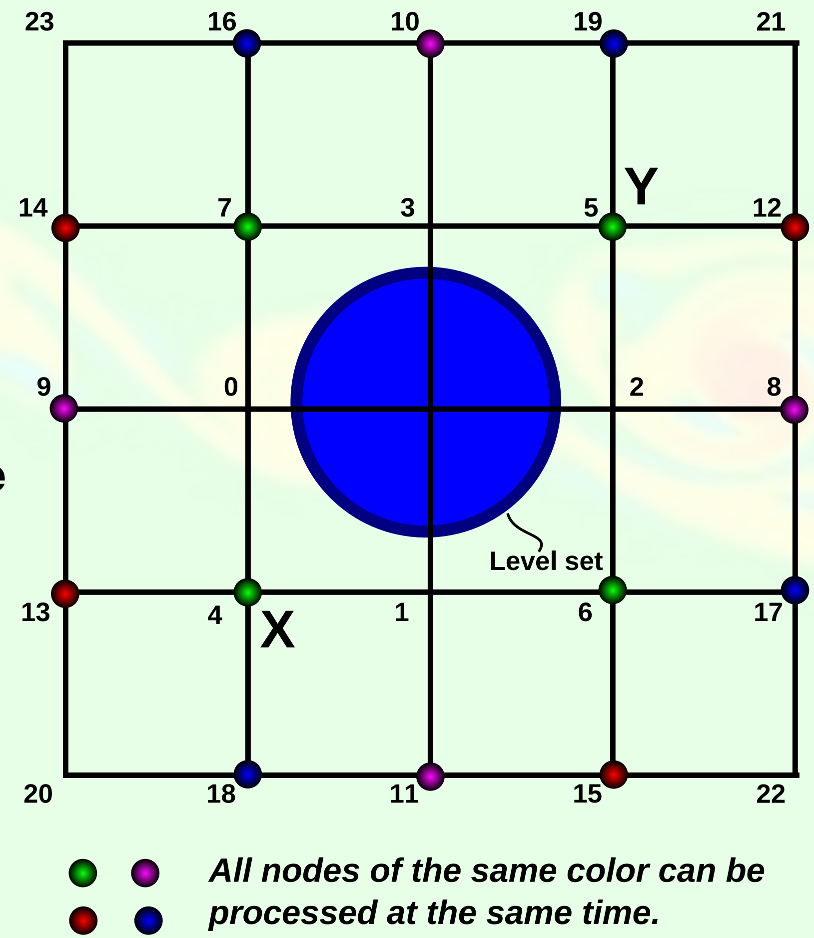
FastMarch renorm: band renormalization

- **The renormalization algorithm doesn't need to cover the whole domain. Only a band around the level set (interface) is needed.**
- **The algorithm is modified simply: set distance in far-away nodes to $d = d_{\rm max}$.**
- **Cost is proportional to the volume of the band, i.e.: $V_{\rm band} = S_{\rm band} \times 2d_{\rm max} \propto d_{\rm max}$.**
- **Low $d_{\rm max}$ reduces cost, but increases the probability of forcing a new renormalization, and thus increasing the renormalization frequency.**



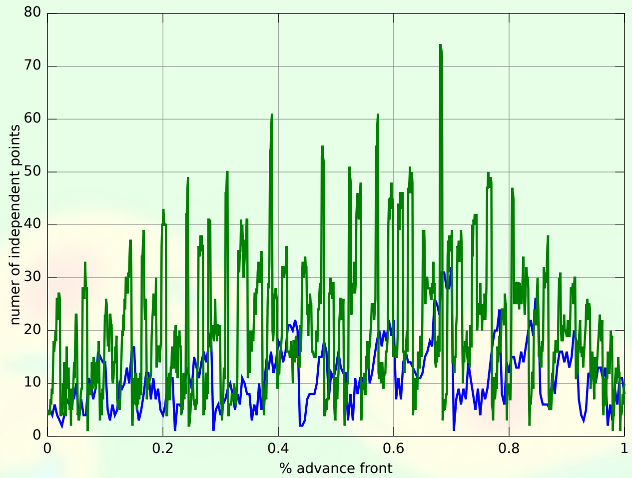
FastMarch renorm: Parallelization

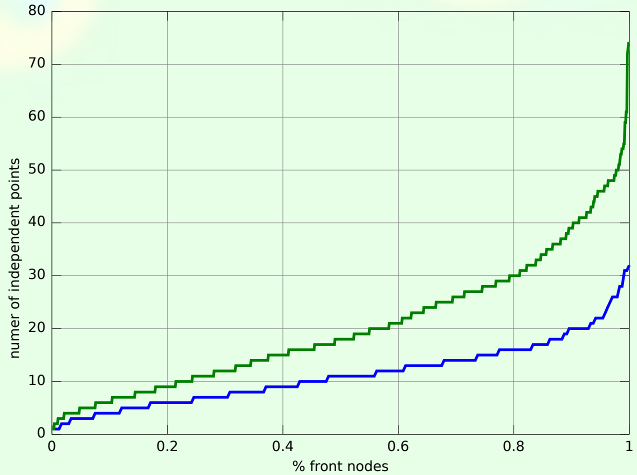
How to parallelize *FastMarch*? We can do *speculative parallelism* that is while processing a node X at the top of the heap, we can process in parallel the following node Y, speculating that most of the time node Y will be far from X and then can be processed independently. This can be checked afterwards, using *time-stamps* for instance.



FastMarch renorm: Parallelization (cont.)

- ***How much*** **nodes can be processed concurrently? It turns out that the *simultaneity* (number of nodes that can be processed simultaneously) grows linearly with refinement.**
- **Average simultaneity is**
  16x16: 11.358
  32x32: 20.507
- **Percentage of times simultaneity is $\geq$4:**
  16x16: 93.0%
  32x32: 98.0%





((version texstuff-1.2.1-144-g90c3aa5 Wed Mar 12 15:08:41 2014 +0100) (date Wed Mar 12 17:13:43 2014 +0100))

FastMarching: computational budget

- With *band renormalization* and *SMP parallelization* we expect a rate of 20 Mcell/s.
- That means that a 128^3 mesh (2 Mcell) can be done in *100 ms*.
- This is 7x times the time required for one time step (*14 ms*).
- Renormalization will be amortized if the *renormalization frequency* is more than 1/20 time steps.
- Transfer of the data to and from the processor through the PCI Express 2.0 x 16 channel ($\sim$4 GB/s transfer rate) is in the order of *10 ms*.
- BTW: note that transfers from the CPU to/from the card are amortized if they are performed each 1:10 steps or so. *Such transfers can't be done all time steps*.

Conclusions

- The NS-FVM implementation reaches high computing rates in GPGPU hardware (O(140 Mcell/s)).
- It can represent complex moving bodies without meshing.
- Surface representation of bodies can be made second order (not implemented yet).
- Solution of the Poisson problem is currently a significant part of the computing time. This is reduced by using the AGP preconditioner and MOC-BFECC combination.
- MOC+BFECC has lower computing rates than QUICK (4x slower) but may reach CFL=5 (versus CFL=0.5 for QUICK). So we get a speedup of 2.5x.
- Speedups may be higher if lower tolerances are required for the Poisson stage (more Poisson iters).

Acknowledgments

This work has received financial support from

- Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET, Argentina, PIP 5271/05),
- Universidad Nacional del Litoral (UNL, Argentina, grant CAI+D 2009-65/334),
- Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT, Argentina, grants PICT-1506/2006, PICT-1141/2007, PICT-0270/2008), and
- European Research Council (ERC) Advanced Grant, Real Time Computational Mechanics Techniques for Multi-Fluid Problems (REALTIME, Reference: ERC-2009-AdG, Dir: Dr. Sergio Idelsohn).

The authors made extensive use of Free Software as GNU/Linux OS, GCC/G++ compilers, Octave, and *Open Source* software as VTK among many others. In addition, many ideas from these packages have been inspiring to them.