A FFT Preconditioning Technique for the Solution of Incompressible Flow

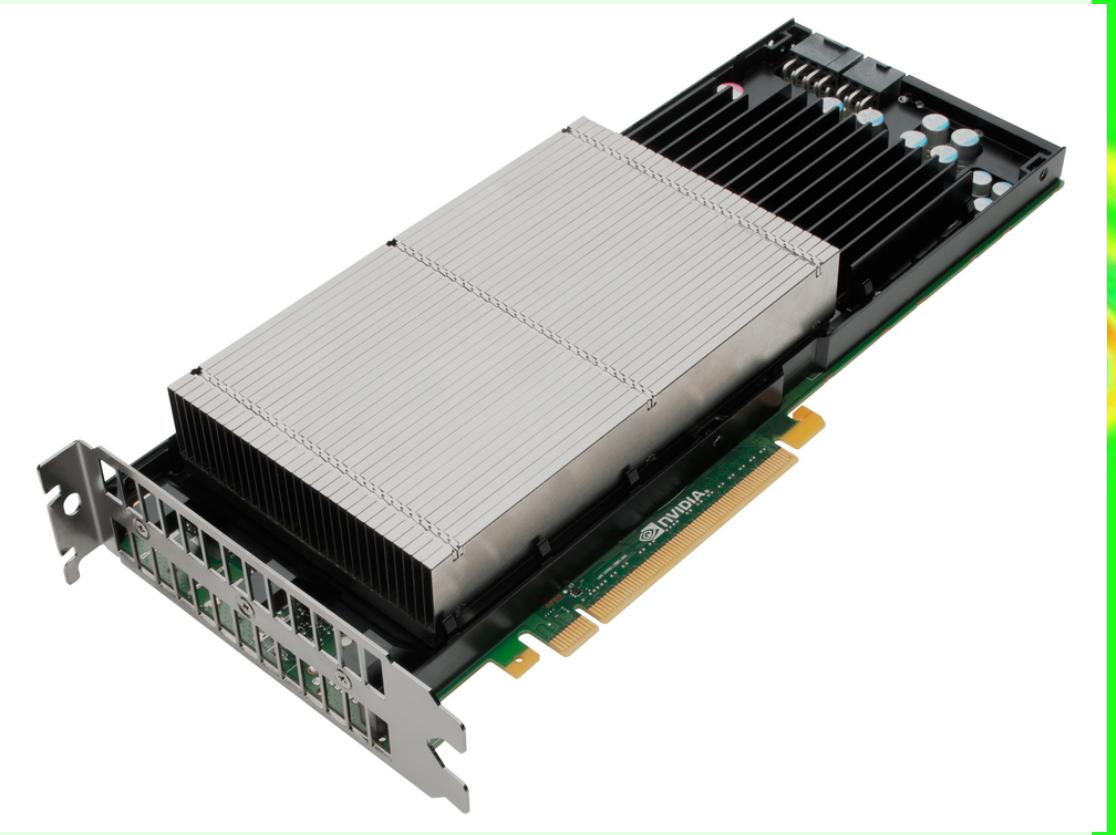
by M.Storti, S.Costarelli, R.Paz, L.Dalcin, S. Idelsohn

Centro Internacional de Métodos Computacionales
en Ingeniería - CIMEC
INTEC, (CONICET-UNL), Santa Fe, Argentina
mario.storti@gmail.com
http://www.cimec.org.ar/mstorti

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

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 GK110 K20* has *2496* cores at 745 Mhz.**
- **The *raw computing power* is in the order of *Teraflops* (3.5 Tflops in SP and 1.17 Tflops in DP for the GK110).**



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Scientific computing on GPU's (cont.)

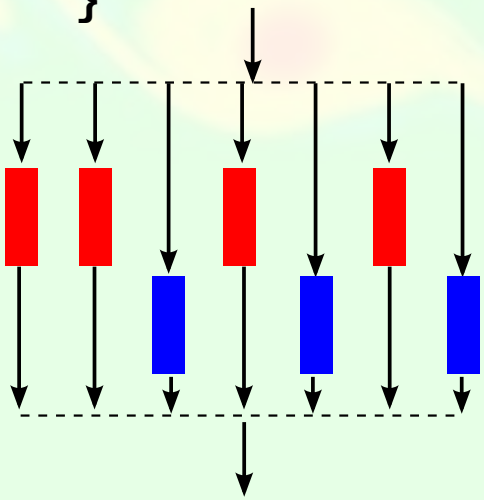
- Initially scientific researchers developed *tricks and magic* in order to convert scientific computations in terms of graphics primitives (OpenGL).
- The companies producing GPU's (Nvidia and ATI) realized this and initiated a line of GPU's for *General Purpose (GPGPU's)*.
- Today scientific computing is done with tools like *CUDA (Nvidia)* or *OpenCL* (a standard that runs on Nvidia and ATI cards, as standard multi and many-core processors).
- Nvidia started also a line completely dedicated to scientific computing named *Tesla*.
- Tesla cards have *ECC memory*, whereas the others don't.
- Initially Tesla cards had a much better *DP/SP speed ratio* w.r.t. the standard cards (1:2 vs. 1:8). Today this difference has been reduced. Also they can have more memory (up to 6GB).
- GPU cards have their own RAM memory (aka *device memory*) with *high data transfers* between the processors and the device memory. *208 GB/s* for the K20. Even so data transfer between the processors and the device memory is often a bottleneck. Normally cards have 4-8 GB of RAM.

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

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)*.

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



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

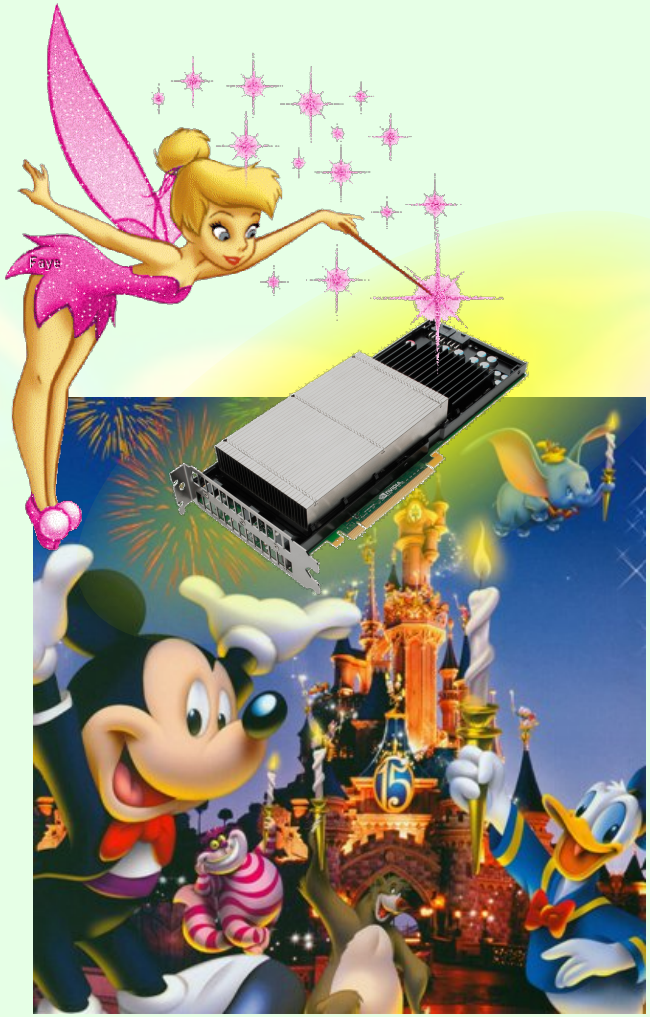
Scientific computing on GPU's (cont.)

- GPU's compete with many-core processors (e.g. Intel's Larrabee, Knights-Corner, Xeon-Phi). They would have 50 cores or more.
- Prices are $\sim$USD 500 for the GTX-580, or US 1300 for a Tesla C2075, USD 3200 for a Tesla K20.
- ***Much higher prices*** are expected for the Intel many-core processors.
- Today mainstream cards (like the GTX-580) are ***available everywhere***. Tesla cards are hard to find in Argentina.
- Companies as Microway sell tower servers with 4 GPU's.
- Many supercomputers have GPU's or Cell processors similar to those used in videogame consoles.

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

BUT WAIT... is GPU computing power REAL or a FAIRY TALE?

- 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.



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Solution of incompressible Navier-Stokes flows on GPU

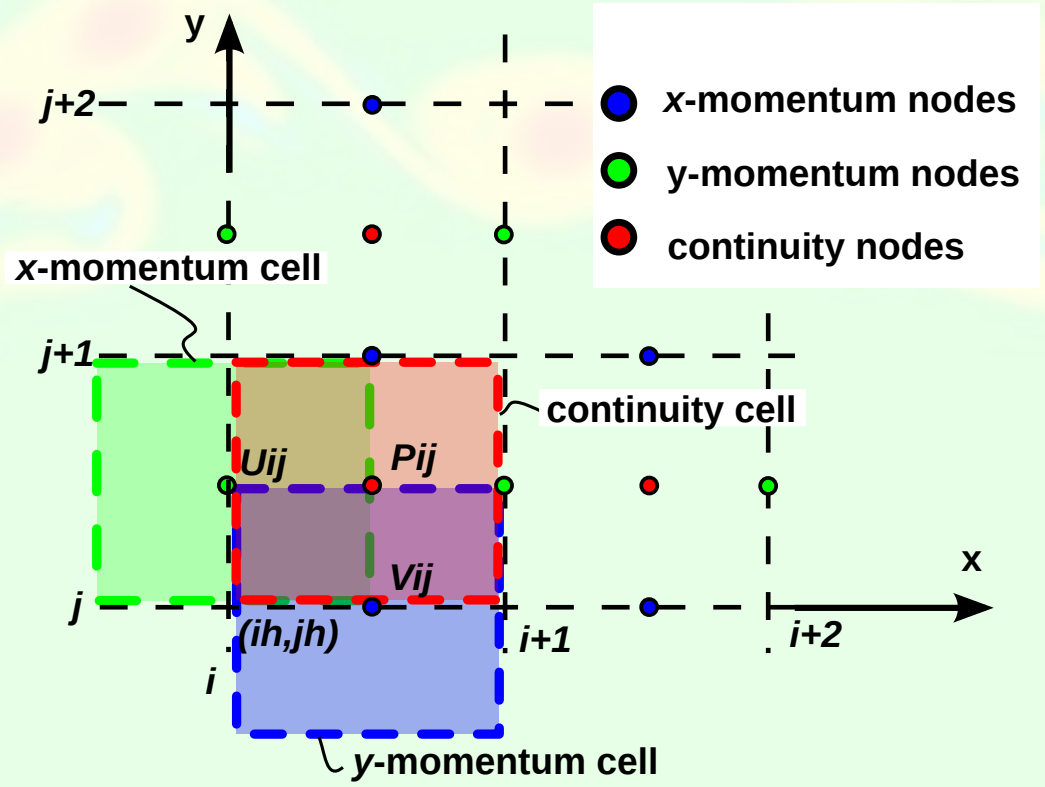
- GPU's are less efficient for algorithms that require access to the *card's (device) global memory*. Shared memory is much faster but usually *scarce* (16K per thread block in the Tesla C1060)  .
- The best algorithms are those that make computations for one cell requiring only information on that cell 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  .

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

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***



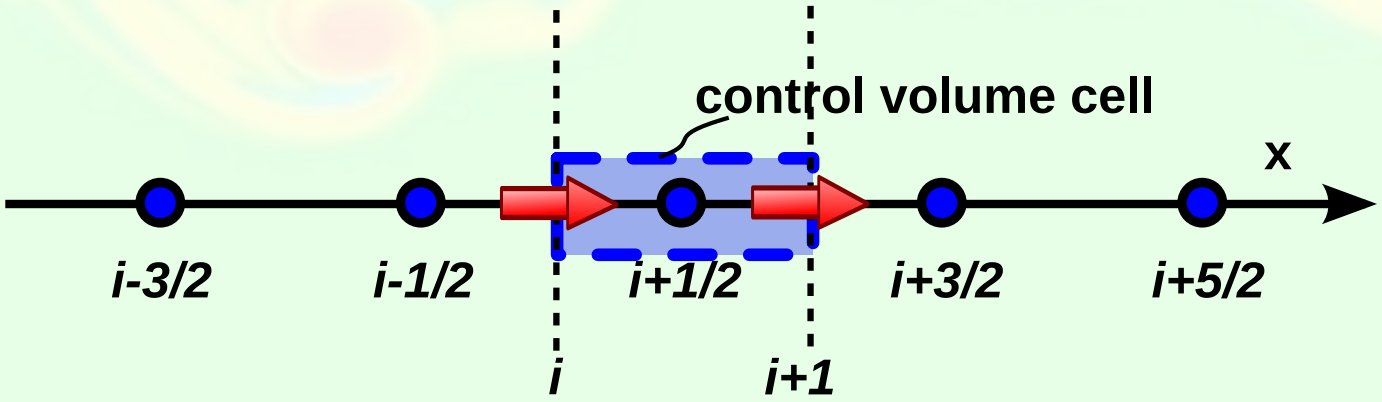
((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Quick advection scheme

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

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

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



(launch video khinstab), *(launch video khinstab-zoom)*

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Solution of the Poisson equation on embedded geometries

- Solution of the ***Poisson equation*** is, for large meshes, the more CPU consuming time stage in Fractional-Step like Navier-Stokes solvers.
- 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***  .

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

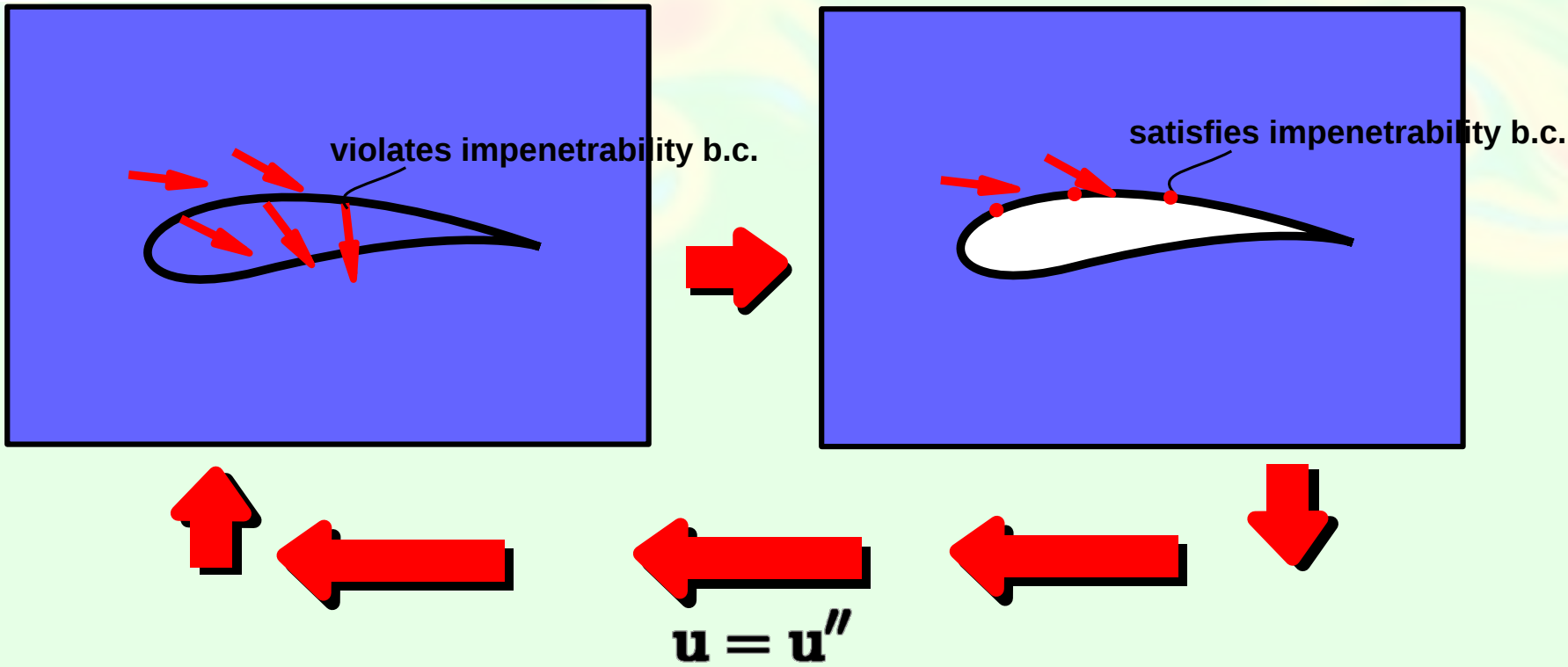
The IOP (Iterated Orthogonal Projection) method

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)

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



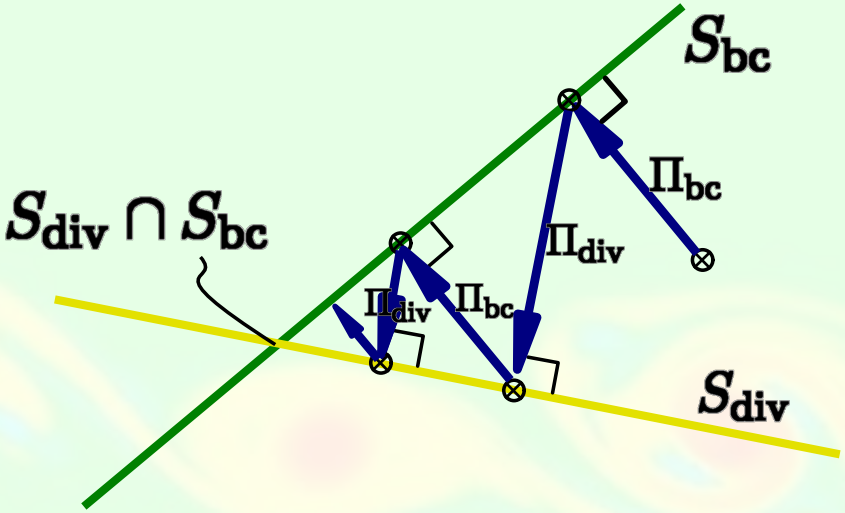
((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

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

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

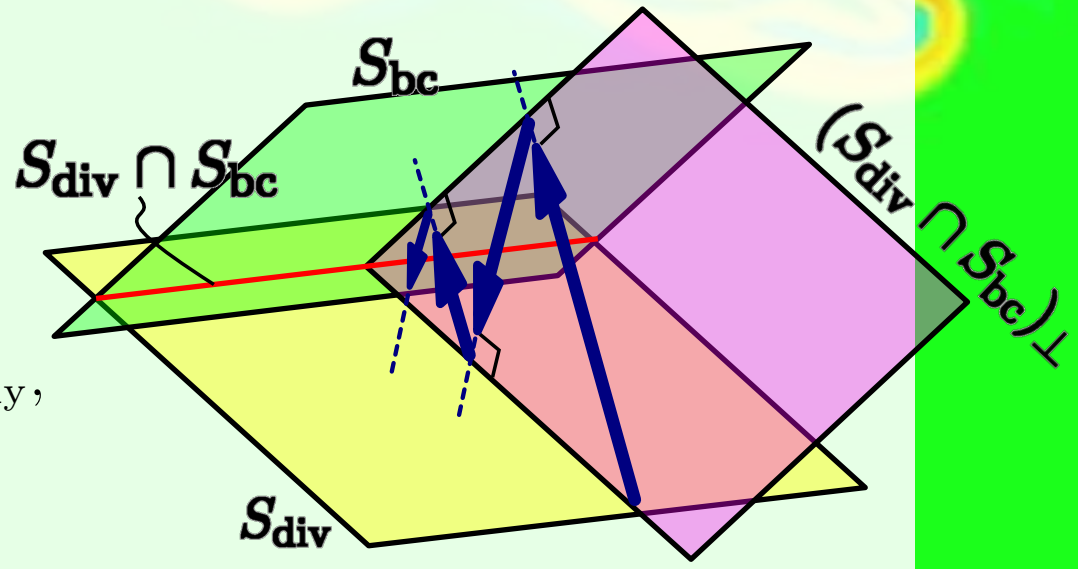
Projection on the space of *divergence-free* velocity fields:

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



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

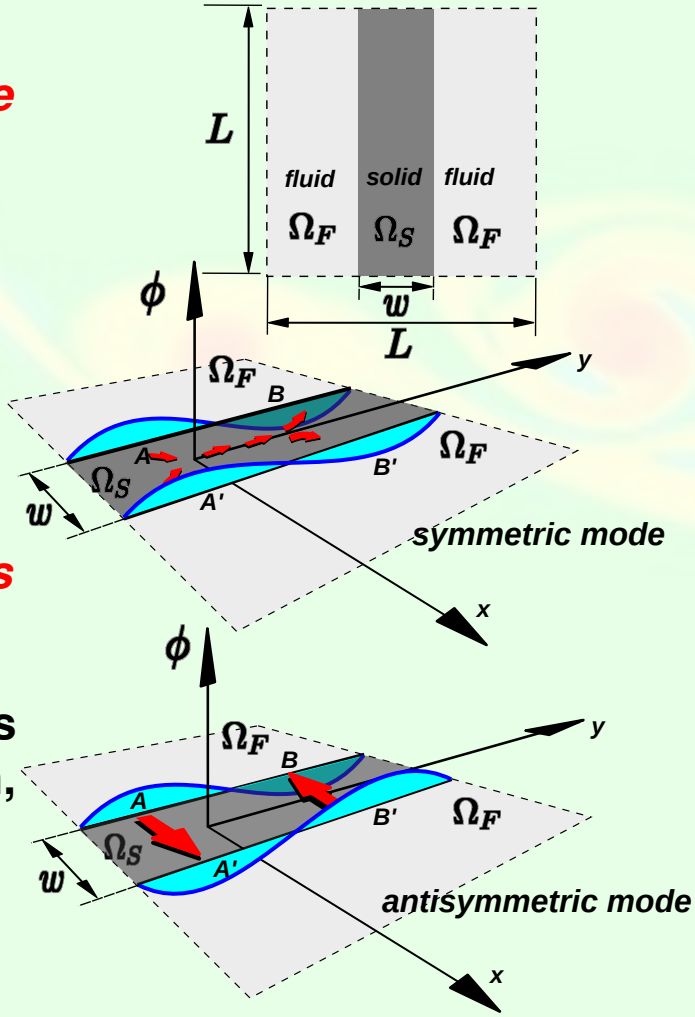
$$\mathbf{u}'' = \mathbf{\Pi}_{\mathrm{bdy}}(\mathbf{u}') \begin{cases} \mathbf{u}'' = \mathbf{u}_{\mathrm{bdy}}, \ \text{in } \Omega_{\mathrm{bdy}}, \\ \mathbf{u}'' = \mathbf{u}', \ \text{in } \Omega_{\mathrm{fluid}}. \end{cases}$$



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Convergence of IOP

- $\Pi_{\mathrm{bdy}}, \Pi_{\mathrm{div}}$ are orthogonal projection operators on $L_2 \implies$ the algorithm converges, with *linear rate of convergence*  .
- Rate of convergence is $O(1)$, i.e. *NOT depending on refinement*  . For instance for an embedded sphere, the residual is reduced to a factor of 0.1 in 3 iterations. However, the rate of convergence *degrades when thin surfaces are present*  .
- In videogame software, and special effects animation, 3 iterations are usually enough, but *for engineering purposes this is insufficient* and an algorithm with better convergence properties is needed.



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Using IOP/AGP with the FFT transform

- When solving the projection problem $\mathbf{u}' = \mathbf{\Pi}_{\mathrm{div}}(\mathbf{u})$ for IOP or the preconditioning for AGP, we have to solve a ***Poisson problem on the whole (fluid+solid) domain***. This is normally done with a ***Geometric Multigrid*** solver which has a complexity $O(N \log \epsilon)$ (N=nbr of grid cells, ϵ=tolerance). It is an ***iterative solver***.
- On the other hand, FFT solves the same problem in $O(N \log N)$. It is a ***direct solver***.

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Accelerated Global Preconditioning (AGP)

- The IOP algorithm iterates on the ***velocity*** u state.
- A method based on ***pressure*** would be more efficient, and in particular in the GPGPU, due to a better use of the ***shared memory***  .
- In addition, IOP is a stationary method (with linear rate of convergence)  . We look for an ***accelerated Krylov space*** algorithm (CG)  .
- The proposed ***AGP algorithm*** is to solve the fluid pressure problem with ***PCG (Preconditioned Conjugate Gradient)*** with the solution on the ***whole (fluid+solid) domain***.
- It can be shown that the ***condition number*** of the preconditioned matrix is also $O(1)$  .
- It is an ***accelerated method***, so convergence is much better than IOP; for the sphere with three iterations we have a reduction of 1e-3 in the residual (while IOP gives a reduction of 0.1)  .
- Conditioning degrades also for ***thin surfaces***  .

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Accelerated Global Preconditioning (AGP) (cont.)

To solve:

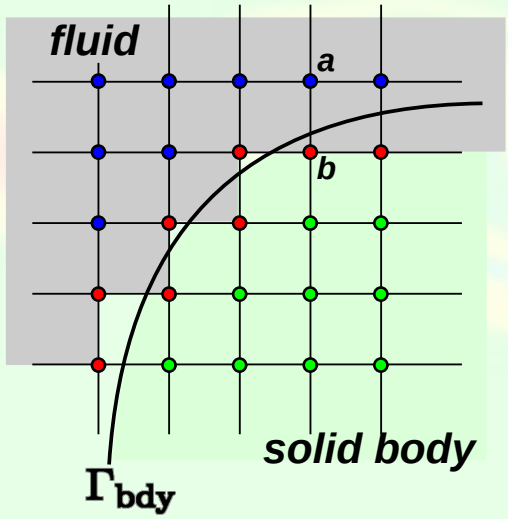
$$\begin{bmatrix} \mathbf{A}_{FF} & \mathbf{A}_{FB} \\ \mathbf{A}_{BF} & \mathbf{A}_{BB} \end{bmatrix} \begin{bmatrix} \mathbf{x}_F \\ \mathbf{x}_B \end{bmatrix} = \begin{bmatrix} \mathbf{b}_F \\ \mathbf{b}_B \end{bmatrix}$$

AGP Preconditioning:

$$\mathbf{P}_{AGP}\mathbf{x}_{FB} = \mathbf{y}_{FB}$$

defined by

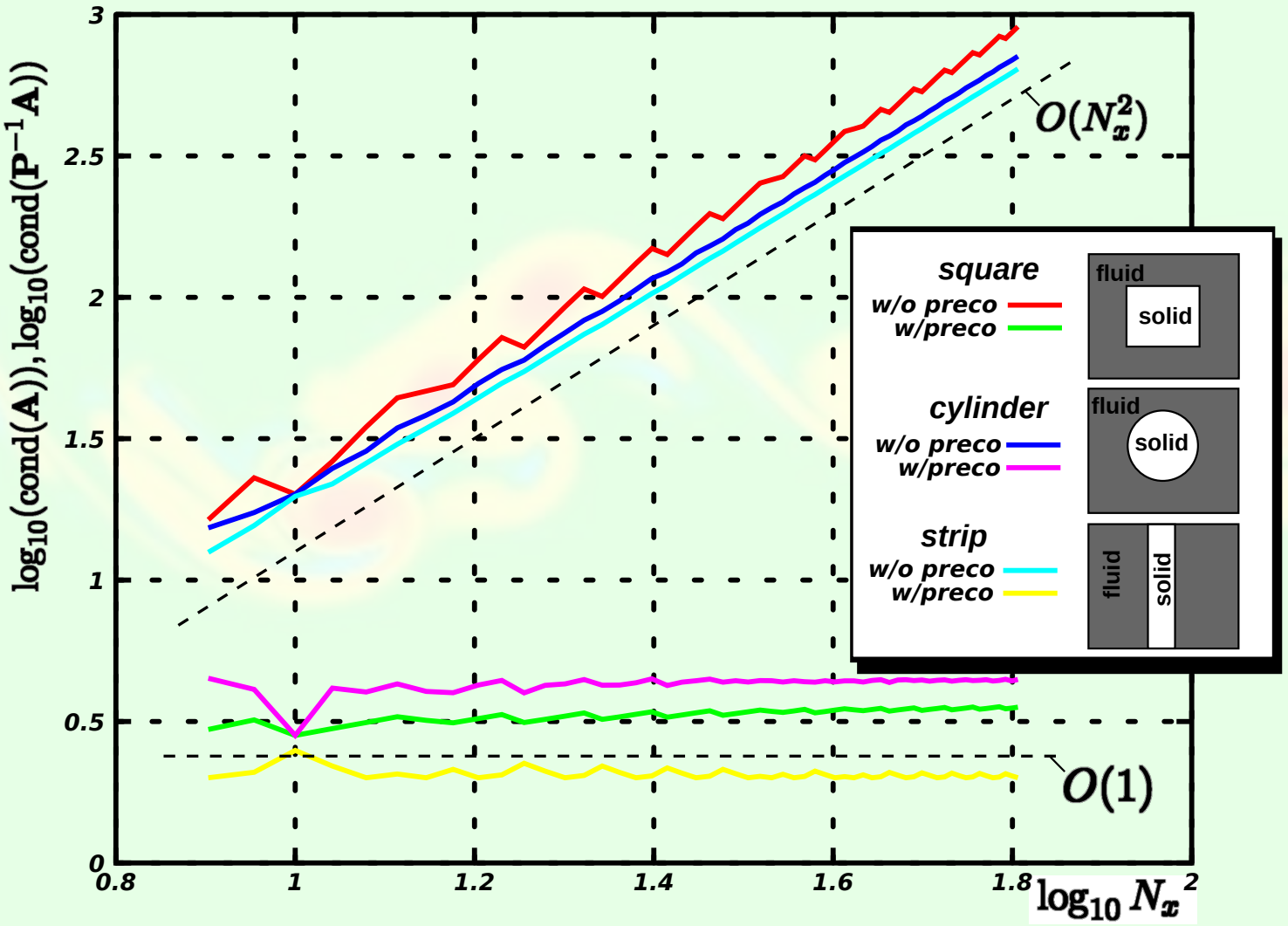
$$\begin{bmatrix} \mathbf{A}_{FF} & \mathbf{A}_{FB} & \mathbf{0} \\ \mathbf{A}_{BF} & \tilde{\mathbf{A}}_{BB} & \mathbf{A}_{BG} \\ \mathbf{0} & \mathbf{A}_{GB} & \mathbf{A}_{GG} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{FB} \\ \mathbf{x}_G \end{bmatrix} = \begin{bmatrix} \mathbf{y}_{FB} \\ \mathbf{0}_G \end{bmatrix}.$$



- *(F) fluid node*
- *(B) boundary node*
- *(G) ghost node*

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Accelerated Global Preconditioning (AGP) (cont.)



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Spectral decomposition of Stekhlov operators

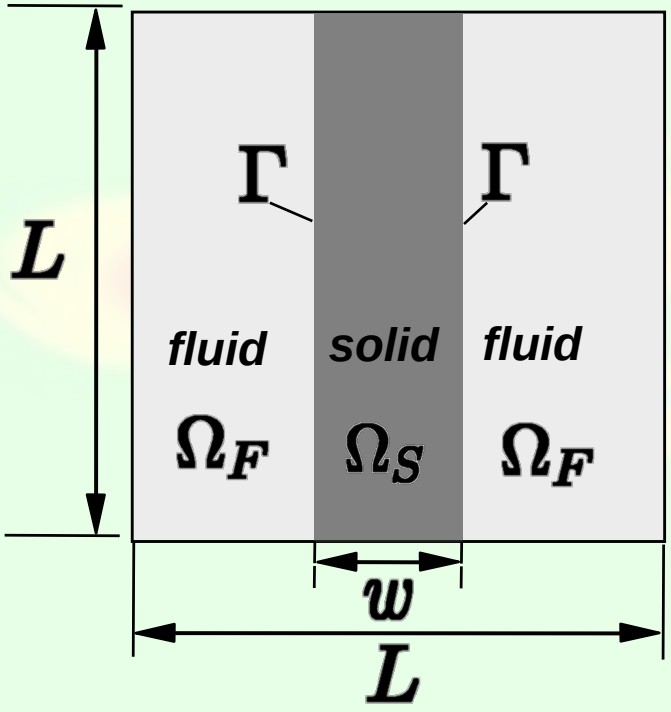
Stekhlov operator $\mathcal{S}_F$ for the fluid domain is defined by: $w = \mathcal{S}_F(v)$, if

$$\Delta\phi = 0, \text{ in } \Omega_F$$

$$\phi_\Gamma = v$$

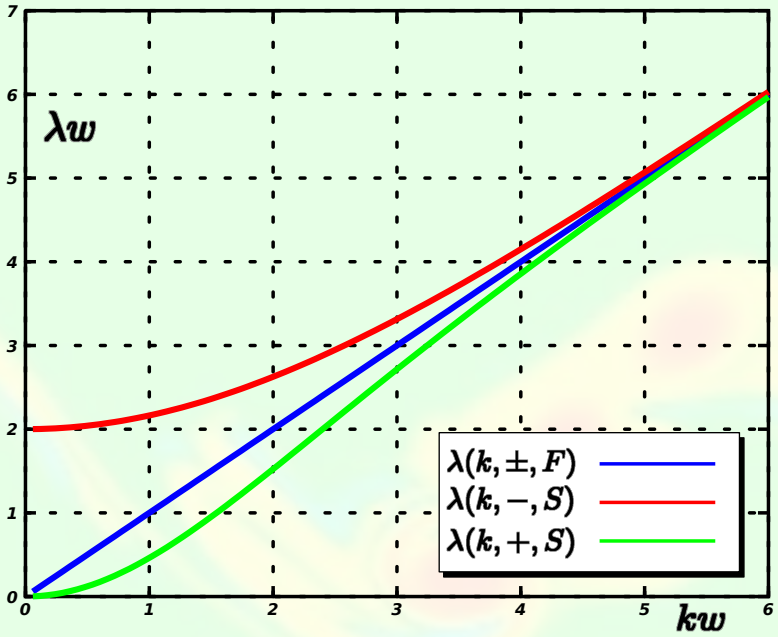
then $w = (\partial\phi/\partial n)|_\Gamma$

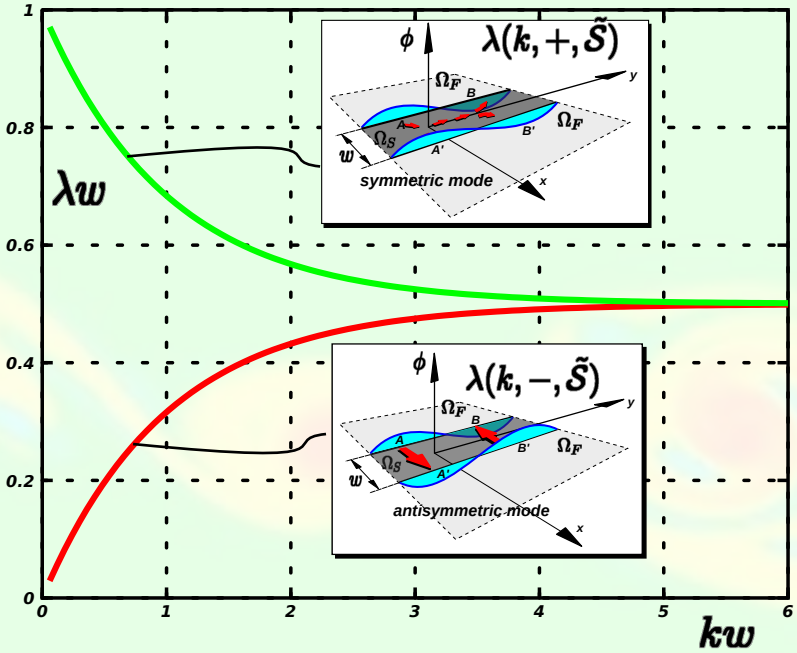
In the same way the Stekhlov operator $\mathcal{S}_S$ for the fluid domain can be defined. It turns out to be that the preconditioned matrix corresponds to $\mathbf{P}^{-1}\mathbf{A} \rightarrow (\mathcal{S}_F + \mathcal{S}_S)^{-1}\mathcal{S}_F$.



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Spectral decomposition of Stekhlov operators (cont.)





$$\lambda(k,\pm,F) = |k|,$$
$$\lambda(k,+,S) = k\tanh(kw/2),$$
$$\lambda(k,-,S) = k\coth(kw/2).$$

$$\lambda(k,\pm,\tilde{\mathcal{S}}) = \frac{|k|}{|k| + k\left\{\begin{matrix}\tanh\\ \coth\end{matrix}\right\}(kw/2)},$$
$$\kappa(\tilde{\mathcal{S}}) = 1/\lambda_{\min}(\tilde{\mathcal{S}}) = L/(\pi w)$$

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

FFT Solver

- We have to solve a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$
- The Discrete Fourier Transform (DFT) is an orthogonal transformation $\tilde{\mathbf{x}} = \mathbf{O}\mathbf{x} = \mathrm{fft}(\mathbf{x})$.
- The inverse transformation $\mathbf{O}^{-1} = \mathbf{O}^T$ is the inverse Fourier Transform $\mathbf{x} = \mathbf{O}^T\tilde{\mathbf{x}} = \mathrm{ifft}(\tilde{\mathbf{x}})$.
- 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

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

- 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.

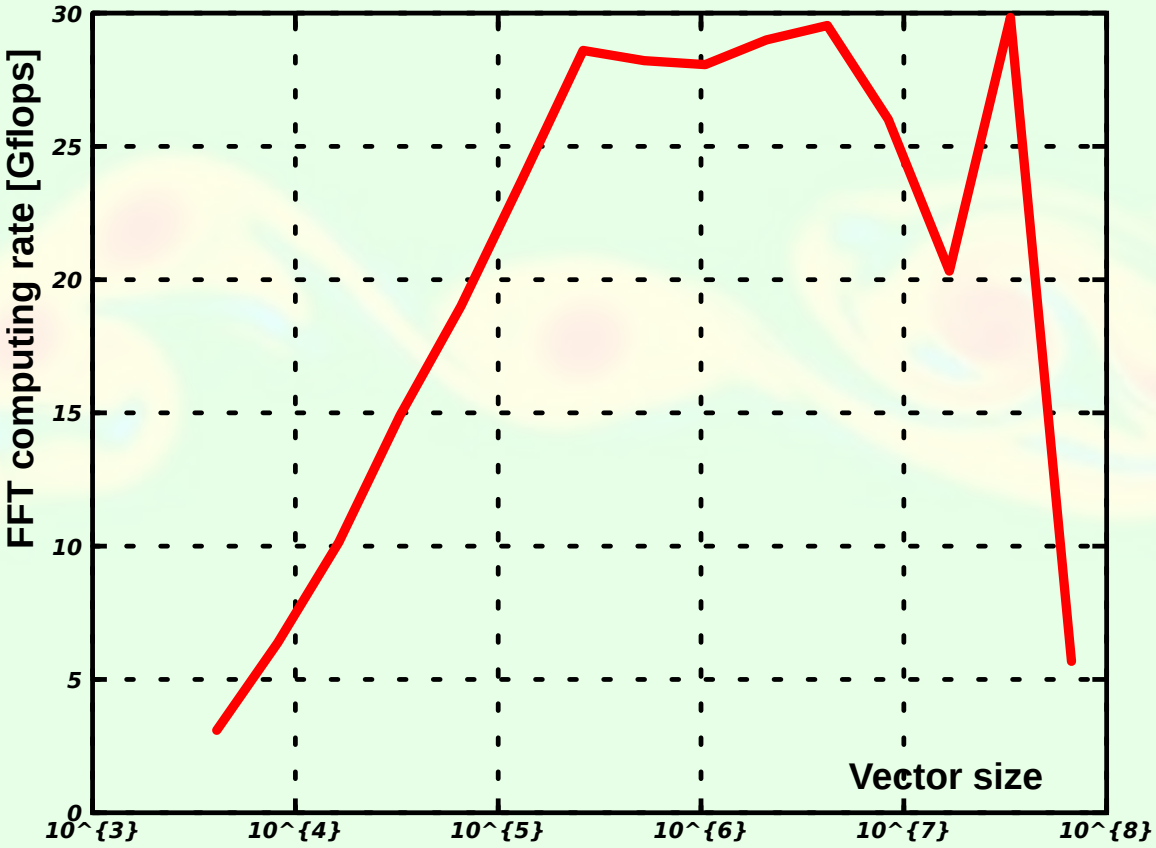
((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

FFT Solver (cont.)

- **So the following algorithm computes the solution of the system in $O(N\log(N))$ ops.**
  - ▷ $\tilde{\mathbf{b}} = \mathrm{fft}(\mathbf{b})$, **(transform r.h.s)**
  - ▷ $\tilde{\mathbf{x}} = \mathbf{D}^{-1}\tilde{\mathbf{b}}$, **(solve diagonal system $O(N)$)**
  - ▷ $\mathbf{x} = \mathrm{ifft}(\tilde{\mathbf{x}})$, **(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 $\tilde{\mathbf{z}} = \mathrm{fft}(\mathbf{z})$, $\tilde{\mathbf{y}} = \mathrm{fft}(\mathbf{y})$,**
  - ▷ $D_{jj} = \tilde{y}_j/\tilde{z}_j$.

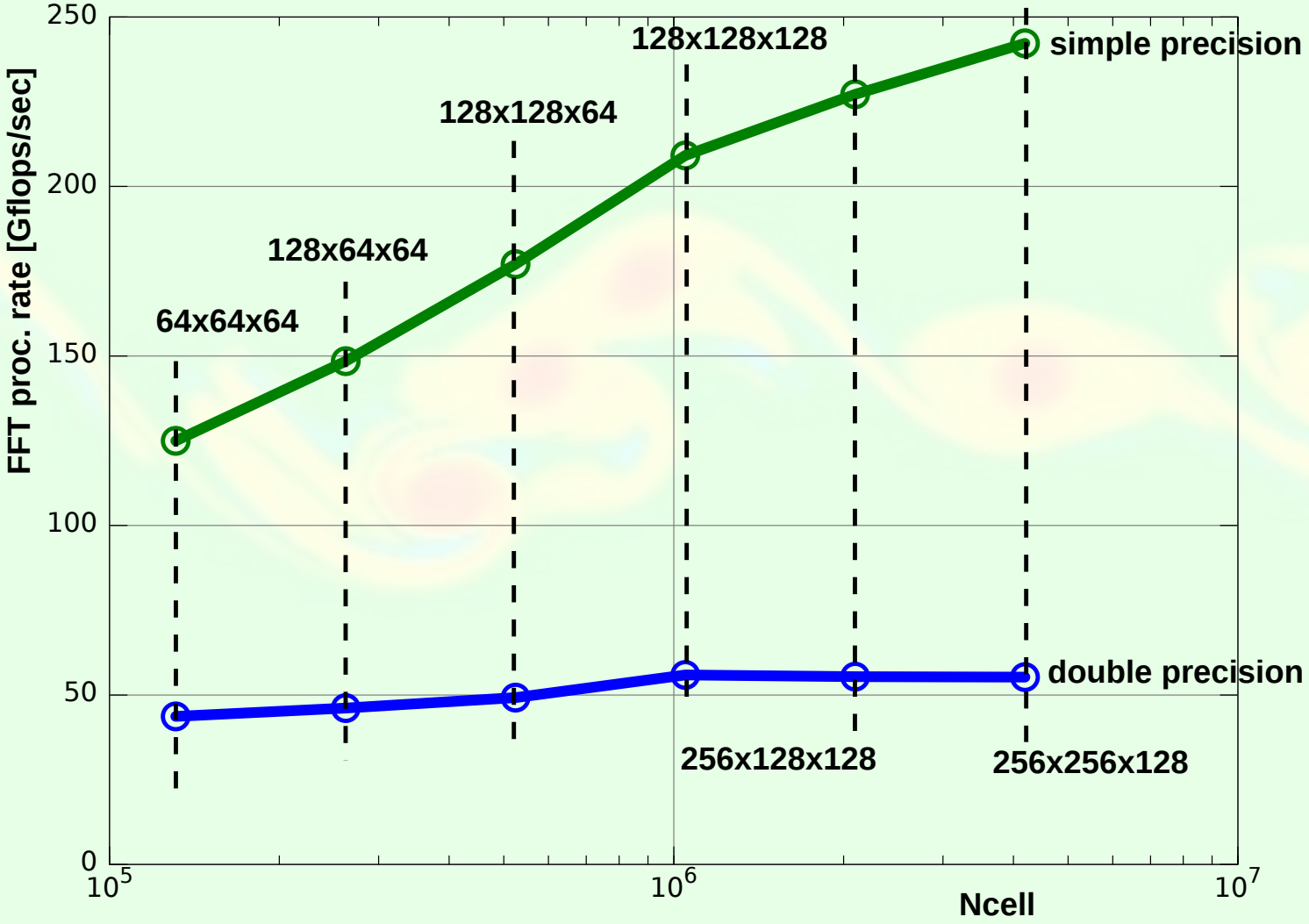
Implementation details on the GPU

- 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 $5N\log_2(N)$.



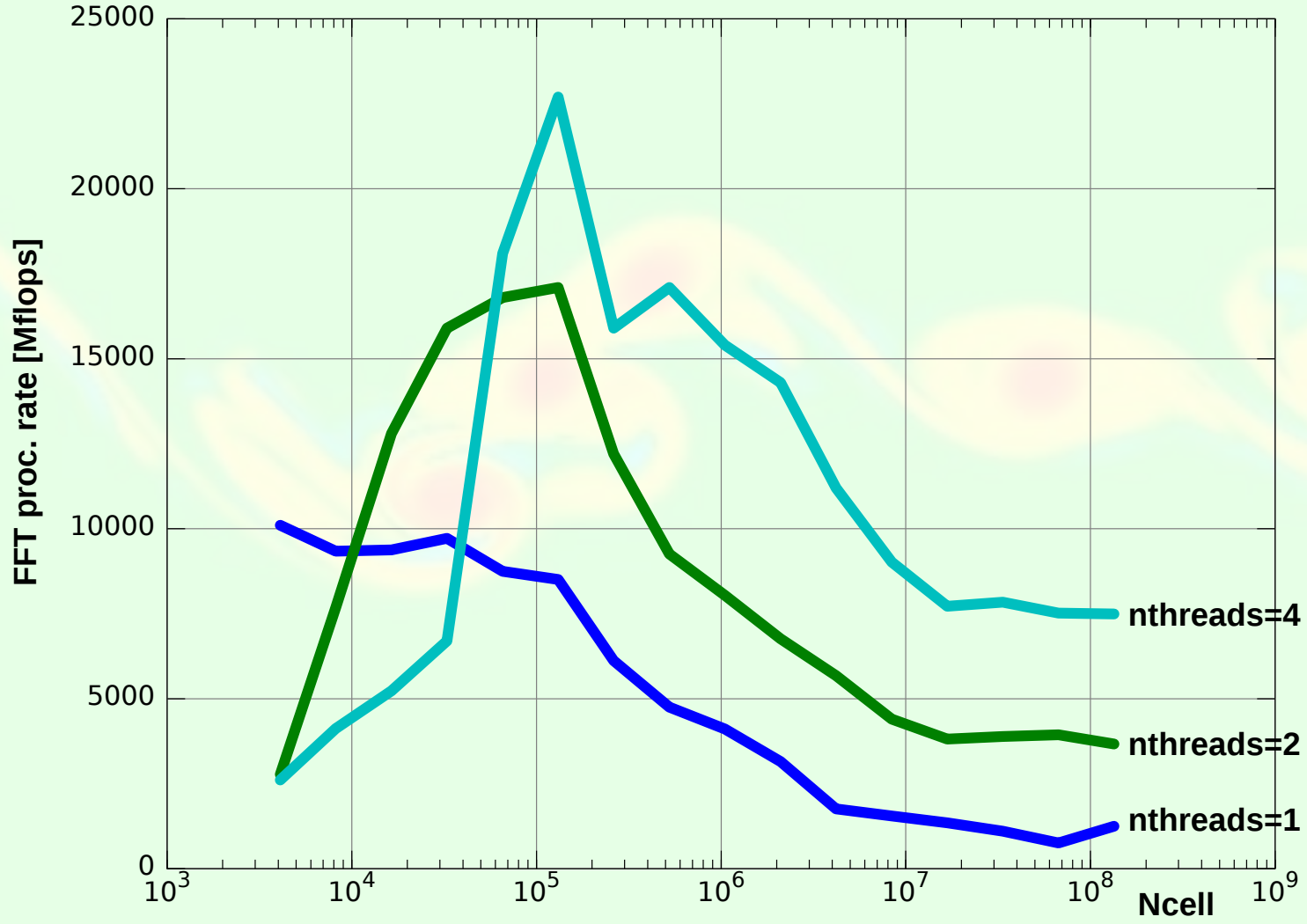
((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

FFT computing rates in GPGPU. GTX-580

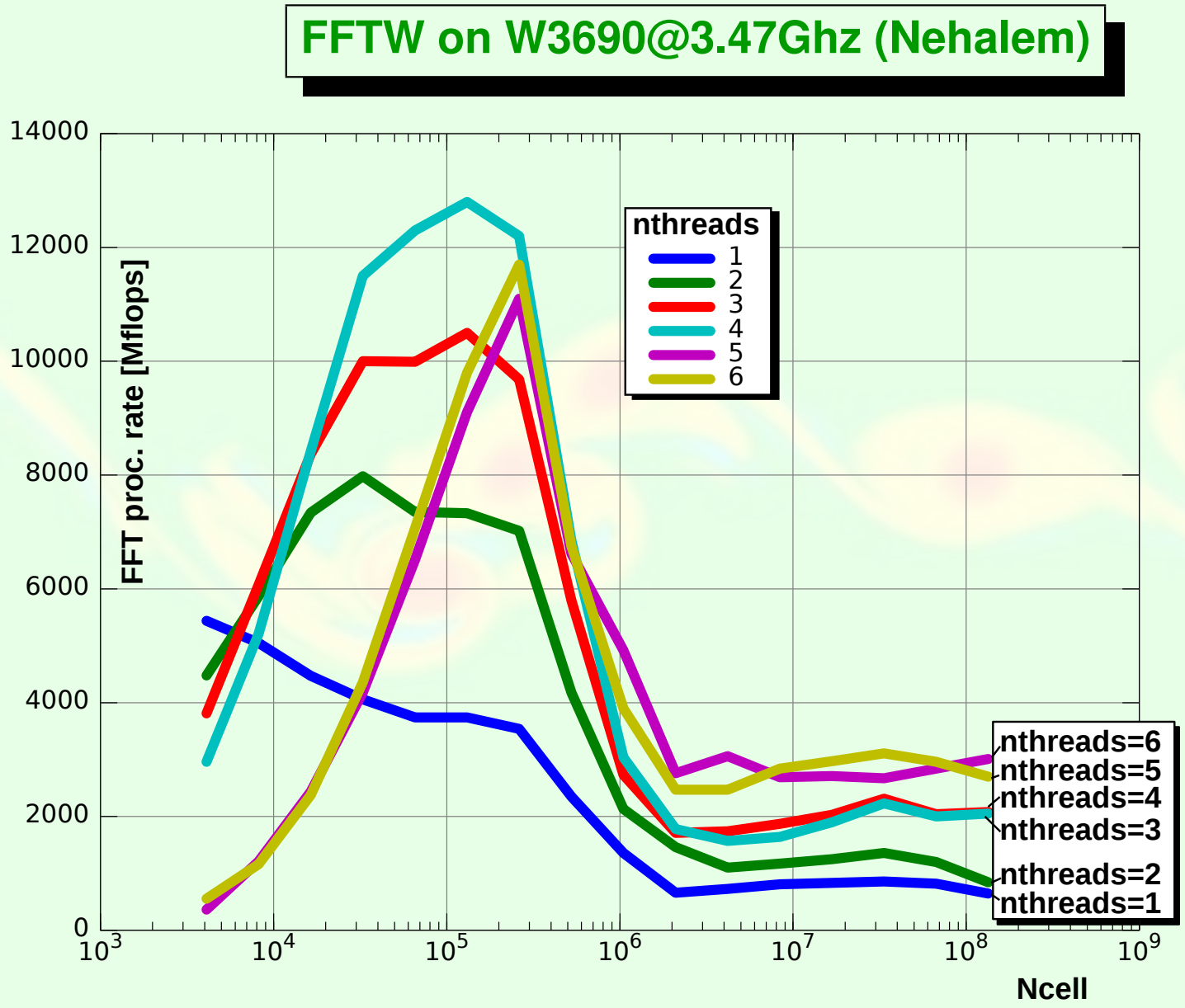


((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

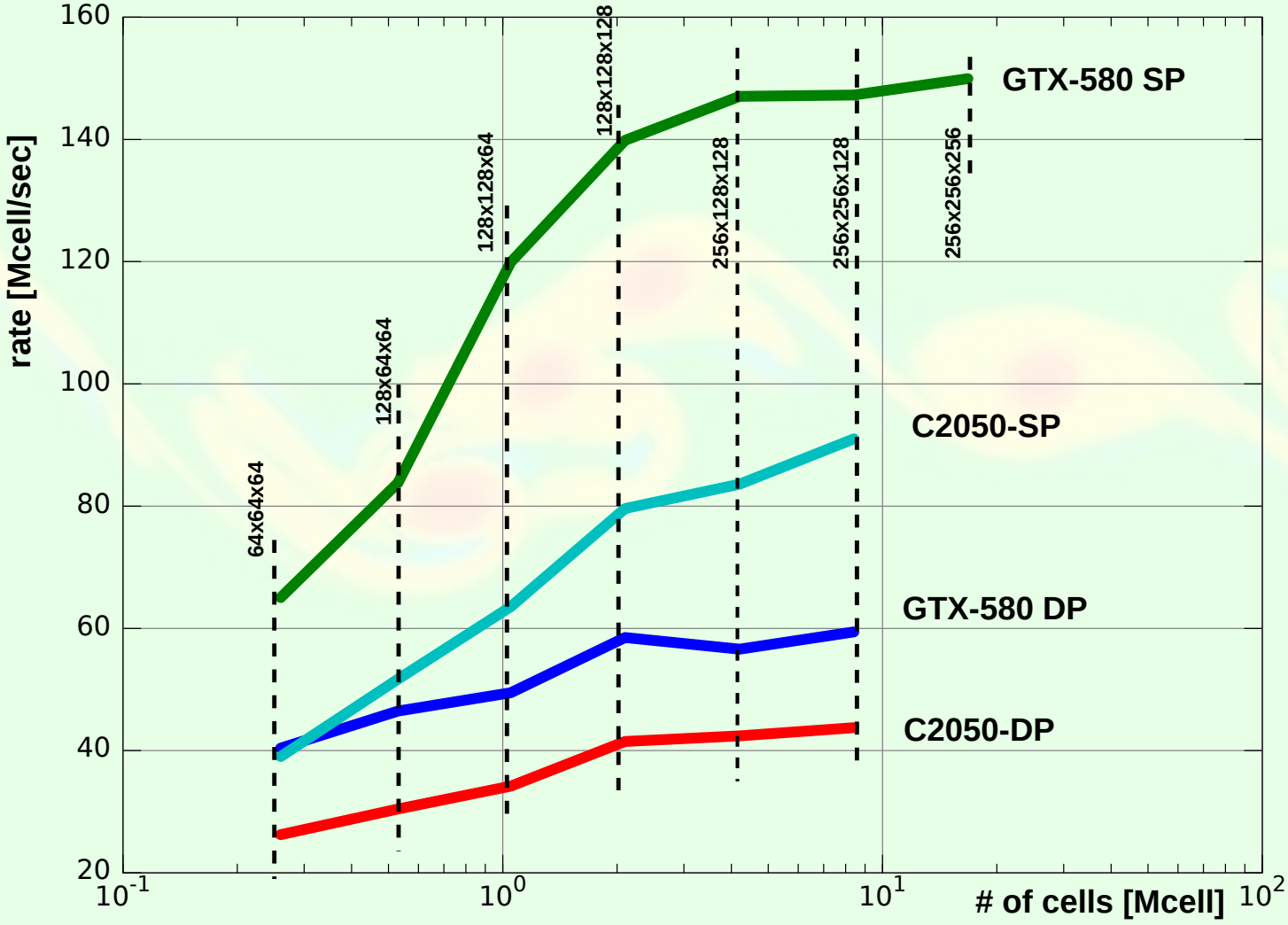
FFTW on i7-3820@3.60Ghz (Sandy Bridge)



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))


FFTW on W3690@3.47Ghz (Nehalem)
FFT proc. rate [Mflops]
Ncell
nthreads
1
2
3
4
5
6
nthreads=6
nthreads=5
nthreads=4
nthreads=3
nthreads=2
nthreads=1
0
2000
4000
6000
8000
10000
12000
14000
10^3
10^4
10^5
10^6
10^7
10^8
10^9

NSFVM Computing rates in GPGPU. Scaling



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

NSFVM Computing rates in CPU

- i7-3820@3.60Ghz (Sandy Bridge), 1 core (sequential): 1.7 Mcell/sec
- i7-950@3.07 (Nehalem), 1 core (sequential): 1.51 Mcell/sec
- Cellrates with nthreads$>$1, and W3690@3.47Ghz not available at this time.
- BUT, we expect at most 7 to 10 Mcell/secs, so there is speedup factor of 8 to 10, with respect to the GPGPU (GTX-580, DP).

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 $L = 1, u = 1, h = 1/128, \Delta t = 0.5h/u = 0.004\ [\mathrm{sec}]$, 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), *(launch video NSFVM-64-64-64-Simple)*, *(launch video NSFVM-2-128-128-Simple)*.

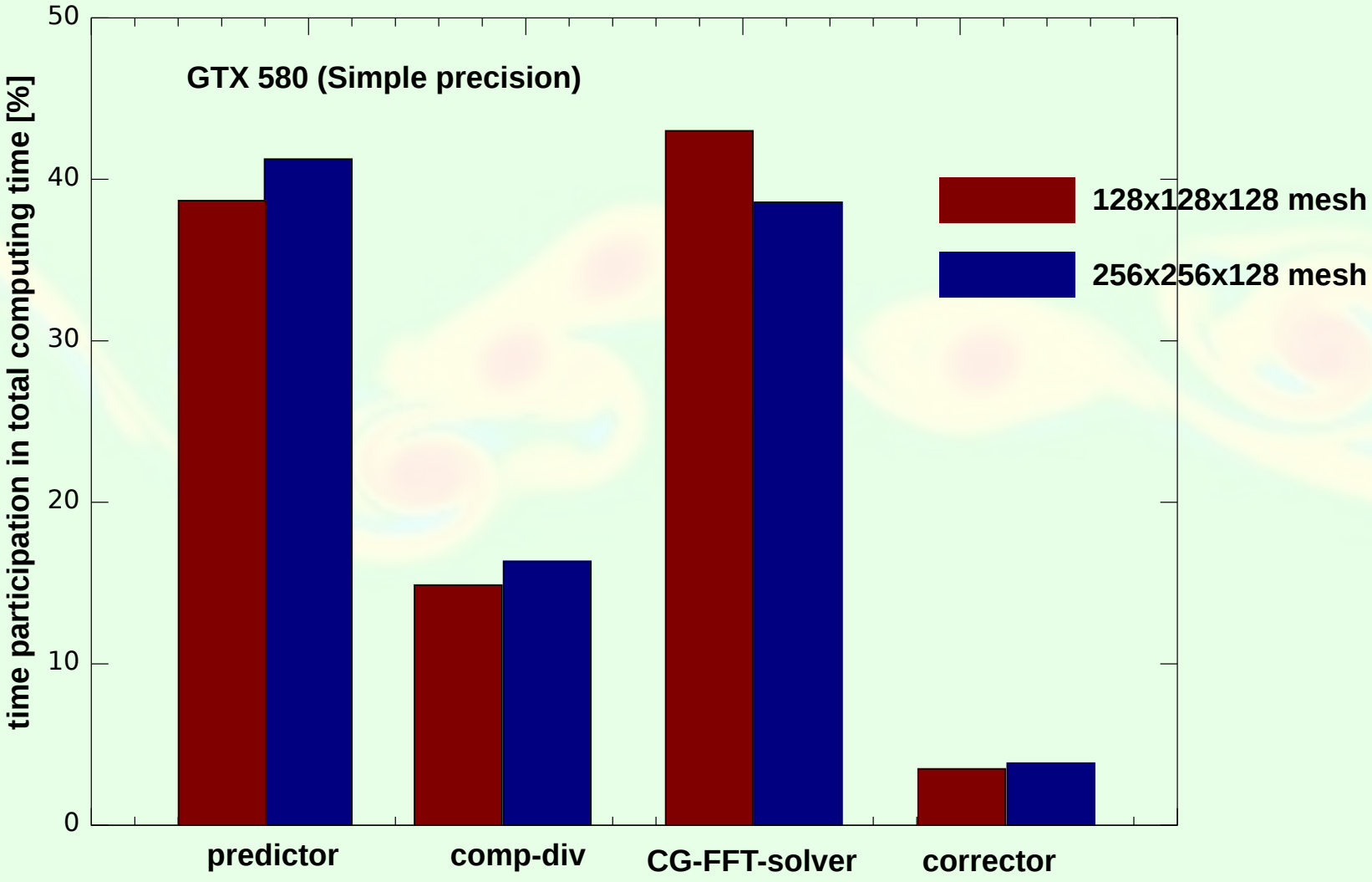
((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

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-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Computing times in GPGPU. Fractional Step components



LBM and FVM

- This algorithm competes with the popular Lattice Boltzmann Method.
- Both are CA (Cellular Automata) algorithms
- Both are fast (measured in cellrates) on GPGPU's with structured meshes.
- LBM doesn't solve a Poisson equation, so it's partially compressible, and then there is a CFL penalization factor (CFL $\propto \mathrm{Mach}_{\mathrm{art}}$).
- Both can be nested refined near surfaces, or other interest zones.
- Higher order treatment of BC's on body surfaces may be better improved in FVM.

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Current work

Current work is done in two main 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 (see Peskin, Acta numerica 11.0 (2002): 479-517)**

MOC+BFECC

- QUICK 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 disipation scheme with a more compact stencil.
- The *Method Of Characteritics (MOC)* is a method that has a null disipation for a constant velocity field and integer CFL number.
- Disipation is non-null for non-integer CFL and maximum for semi-integer CFL.
- Combination of MOC with the BFECC reduces dissipation and gives a compact stencil.

MOC+BFECC (cont.)

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

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

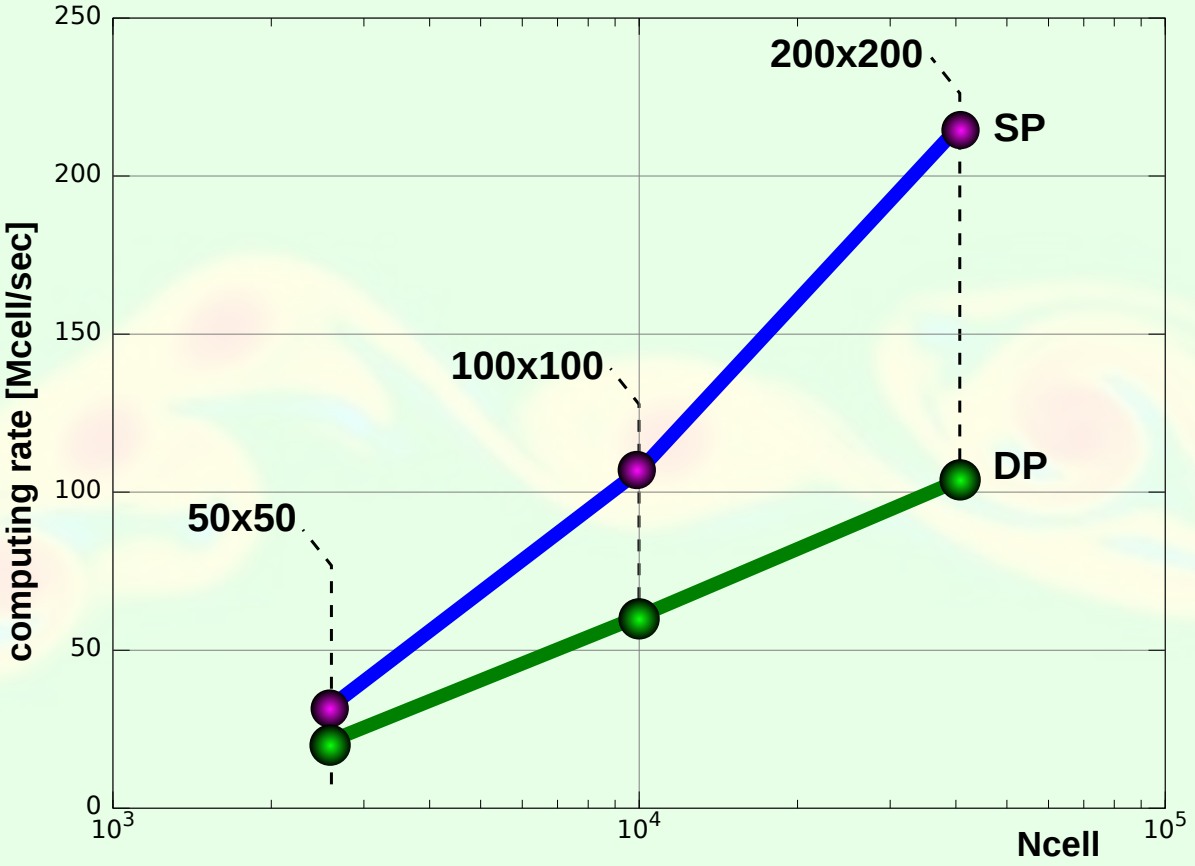
((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

MOC+BFECC (cont.)

Comparing computing rates with QUICK and MOC-BFECC:

- **NSFVM+QUICK 7 ms/Mcell**
- **Predictor step (QUICK): 3 ms/Mcell**
- **MOC-BFECC (Scalar) 5 ms/Mcell**

BUT: NSFVM+QUICK advances at CFL=0.5, while MOC-BFECC could advance at CFL=4.9.



Zalezak's disk. Computing rates for the MOC-BFECC on a GPGPU Nvidia GTX580. CFL=4.9. Scalar advection.

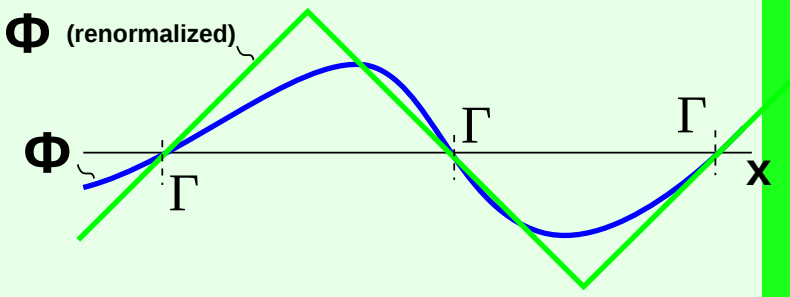
((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Renormalization

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

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



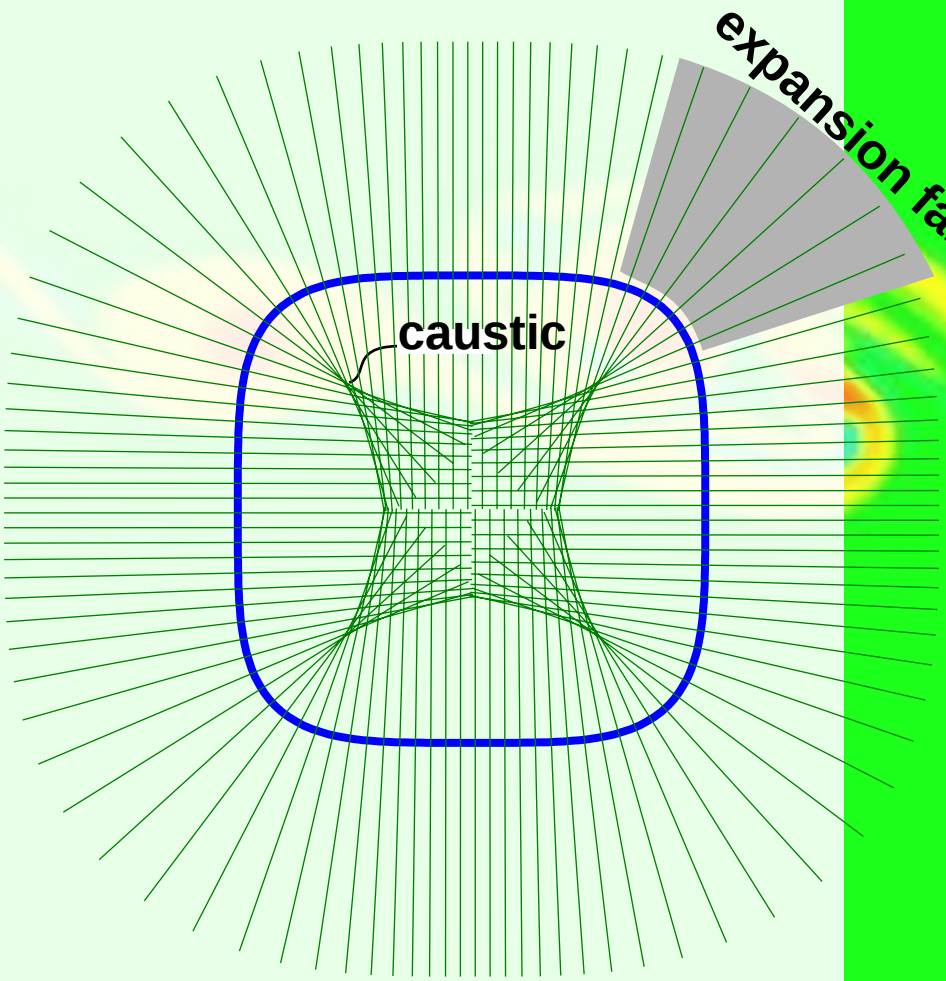
((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Renormalization (cont.)

- Computing plainly the distance function is $O(NN_\Gamma)$ 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

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

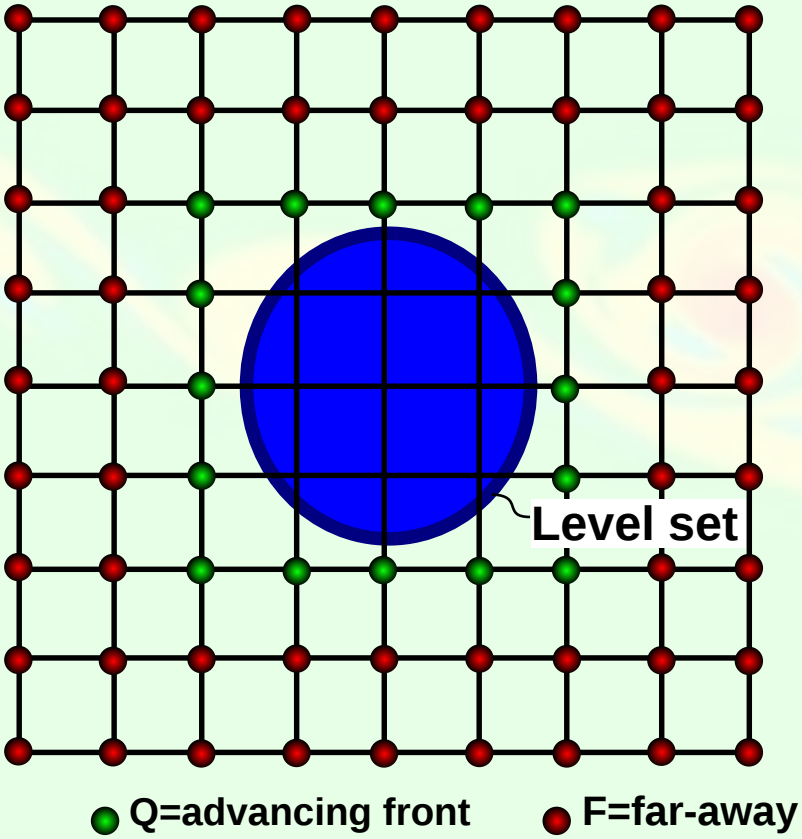
- As it is an hyperbolic equation it can be solved by a *marching* technique. The algorithm traverses the domain with an *advancing front* starting from the level set.
- However, it can develop *caustics* (*shocks*), and *rarefaction waves*. So, an *entropy condition* must be enforced.



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

Renormalization (cont.)

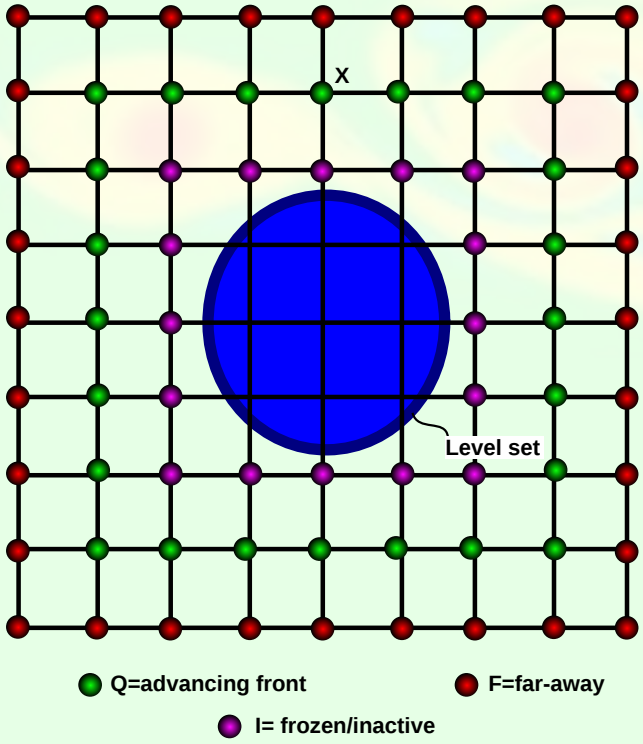
- The ***Fast Marching*** algorithm proposed by Sethian (Proc Nat Acad Sci 93(4):1591-1595 (1996)) , is a ***fast*** (near optimal) algorithm based on ***Dijkstra's algorithm*** for computing minimum distances in graphs from a source set. (Note: the original Dijkstra's algorithm is $O(N^2)$, not fast.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



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

The Fast Marching algorithm

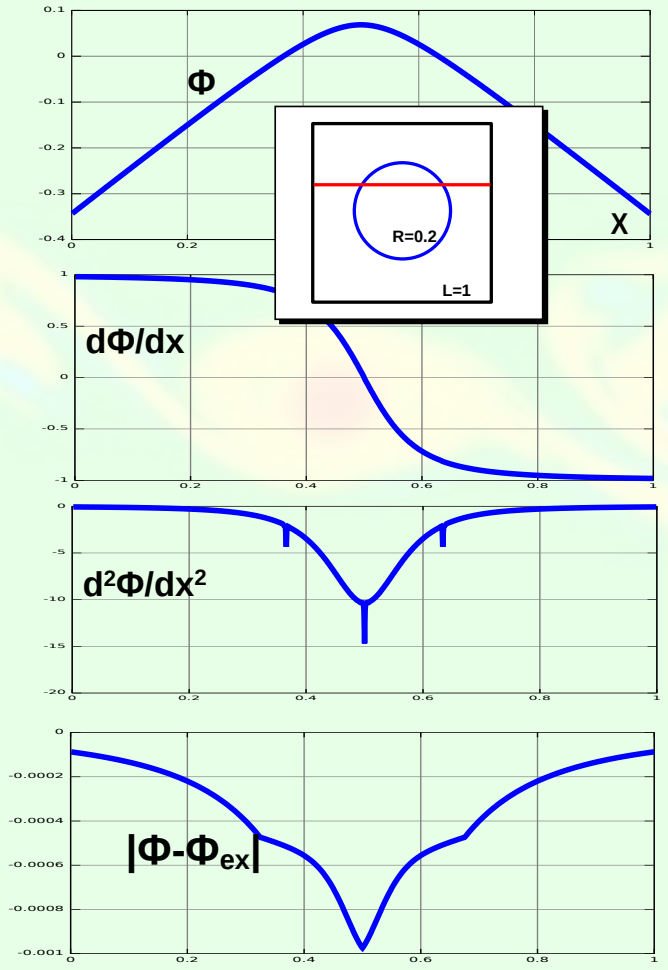
- We explain for the positive part $\Phi > 0$. Then the algorithm is reversed for $\Phi < 0$.
- All nodes are in either: Q=*advancing front*, F=*far-away* , I=*frozen/inactive*. The advancing front sweeps the domain starting at the level set and converts F nodes to I.
- Initially $Q = \{$nodes that are 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 $Q \rightarrow I$.
- Update all distances from neighbors to X and move them from $F \rightarrow Q$.
- Go to *loop*.
- Algorithm ends when $Q = \emptyset$.



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

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 $\Phi - \Phi_{\text{ex}}$ shows the discontinuity in the slope at the LS.



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

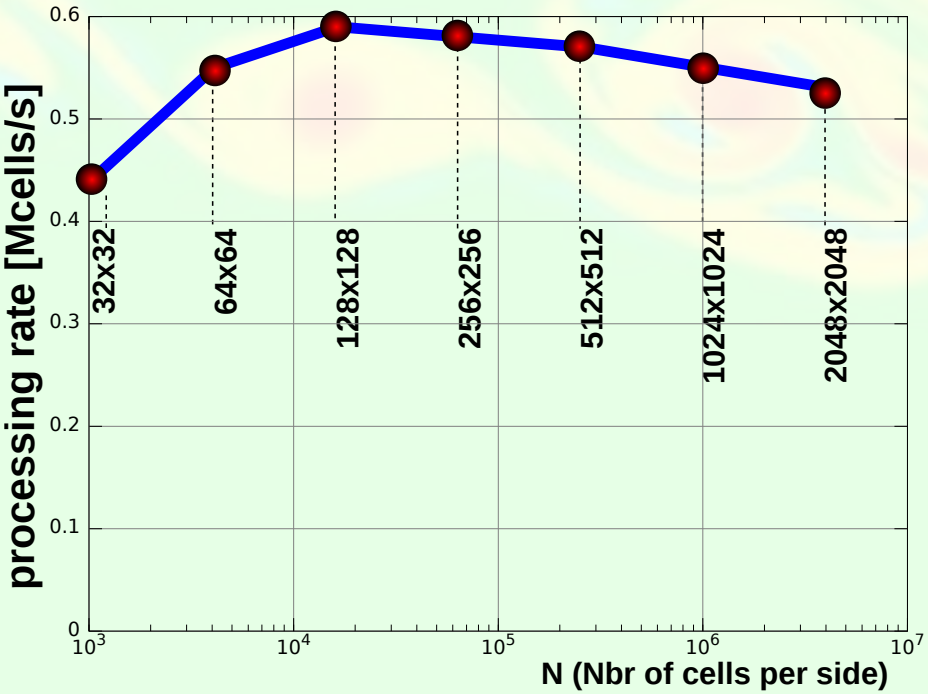
FastMarch: implementation details

- 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.

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

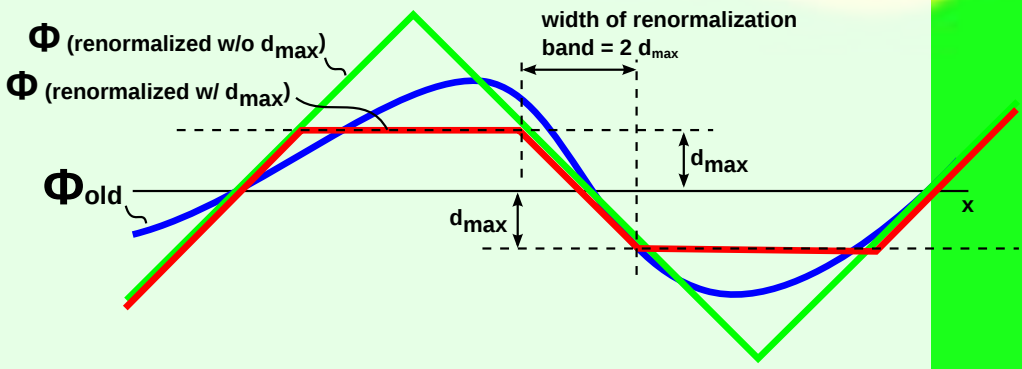
FastMarch renorm: Efficiency

- 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)***.



FastMarch renorm: band renormalization

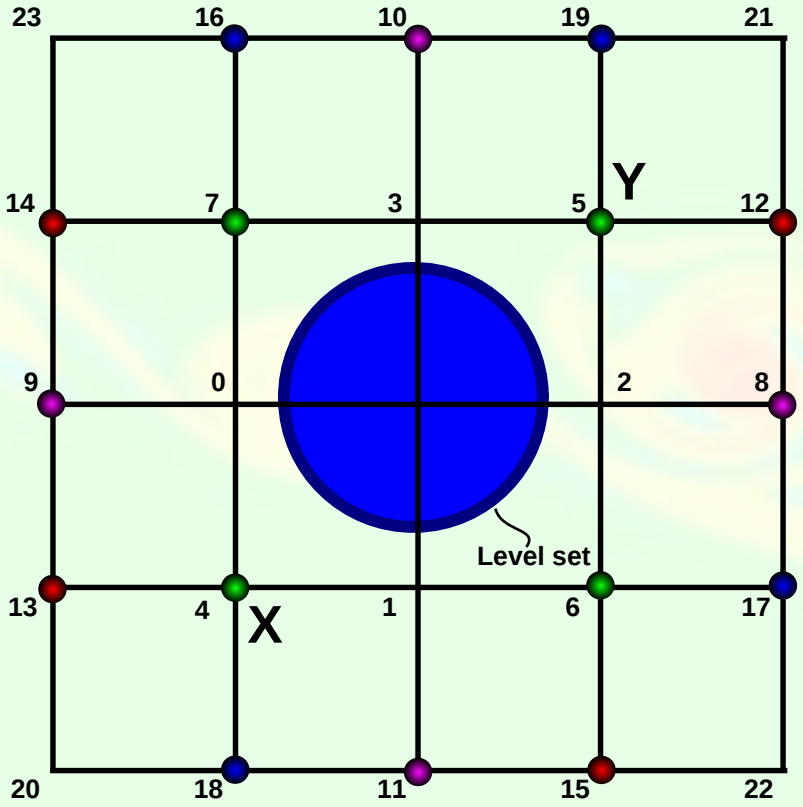
- The renormalization algorithm doesn't need to cover the whole domain. Only a band around the level set (interface) is needed.
- The algorithm is modified simply: set distance in far-away nodes to $d = d_{\max}$.
- Cost is proportional to the volume of the band, i.e.: $V_{\text{band}} = S_{\text{band}}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



((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

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.

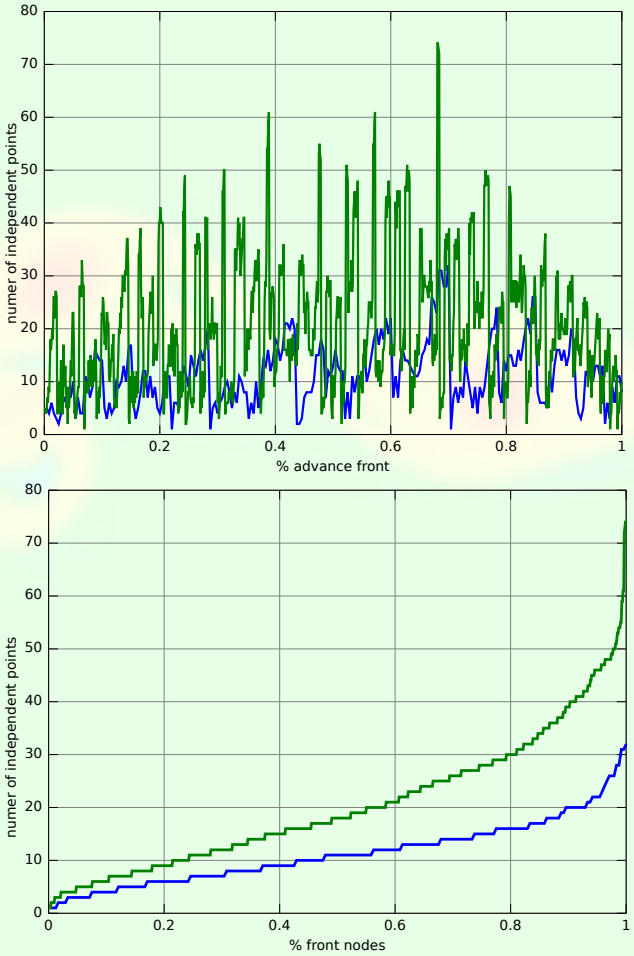


All nodes of the same color can be processed at the same time.

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

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-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

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 *Accelerated Global Preconditioning (AGP)* algorithm for the solution of the Poisson equation specially oriented to the solution of Navier-Stokes equations on GPU hardware was presented. It shares some features with the well known *IOP* iteration scheme. As a summary of the comparison between both methods, the following issues may be mentioned

- Both solvers are based on the fact that an efficient preconditioning that consists in solving the Poisson problem on the global domain (fluid+solid). Of course, this represents more computational work than solving the problem only in the fluid, but this can be faster in a structured mesh with some fast solvers as Multigrid or *FFT*.
- Both solvers have their convergence governed by the spectrum of the $\mathcal{S}^{-1}\mathcal{S}_F$, however

((version texstuff-1.2.0-118-g8afc74d Wed Nov 14 23:40:27 2012 -0300) (date Thu Nov 15 08:13:22 2012 -0300))

- **IOP is a *stationary method* and its limit rate of convergence is given by**

$$\|\mathbf{r}^{n+1}\| \leq \gamma_{\mathrm{IOP}} \|\mathbf{r}^{n}\|$$
$$\gamma_{\mathrm{IOP}} = 1 - \lambda_{\min}, \tag{3}$$
$$\lambda_{\min} = \min(\mathrm{eig}(\mathcal{S}^{-1}\mathcal{S}_F)).$$

- **AGP is a preconditioned *Krylov space method* and its convergence is governed by the condition number of $\mathcal{S}^{-1}\mathcal{S}_F$, i.e.**

$$\kappa(\mathbf{A}^{-1}\mathbf{A}_F) = \frac{1}{\min(\mathrm{eig}(\mathcal{S}^{-1}\mathcal{S}_F))} = \frac{1}{\lambda_{\min}}, \tag{4}$$

- **It has been shown that $\lambda_{\min} = O(1)$, i.e. it *does not degrade with refinement*, so that *IOP* has a linear convergence with limit rate $O(1)$.**
- **By the same reason, the condition number for *AGP does not degrade with refinement*.**
- ***IOP* iterates over both the velocity and pressure fields, whereas *AGP* iterates only on the pressure vector (which is better for implementation on GPU's).**
- **The *MOC+BFECC* scheme is an efficient solver for the advection equation.**

It gives high computing rates with large CFL numbers.

- The ***Fast-Marching*** renormalization technique is a good candidate for doing renormalization on the CPU and having times competitive with those of NS-FVM on the GPU.

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.