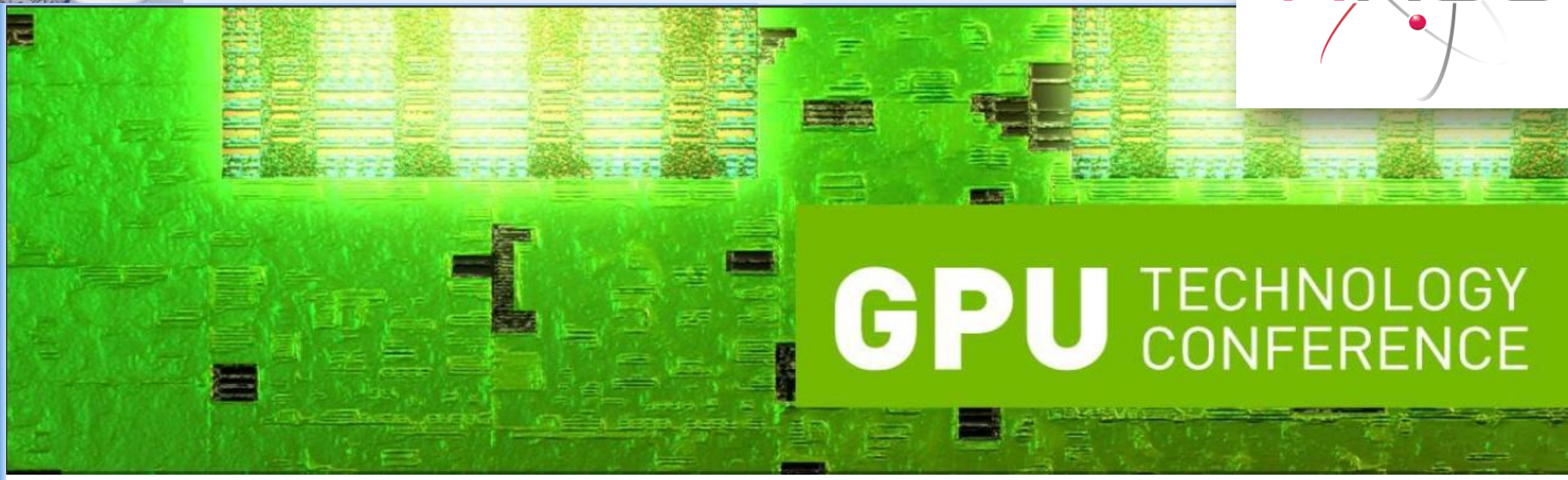
www.murex.com



Practical Methods Beyond Monte Carlo in Finance

UNDER NDA TILL CONFERENCE

September 2010

Murex

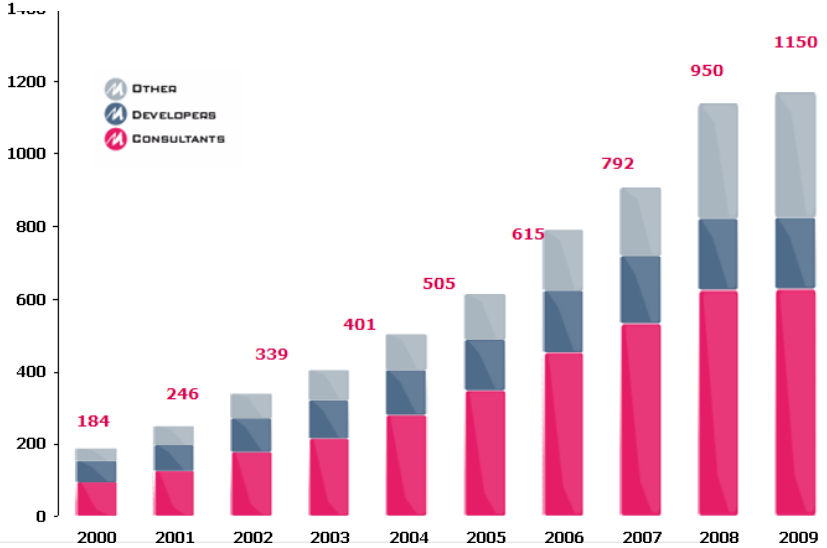
Murex develops, markets, implements and supports software platforms for the capital markets industry. 24 years of exclusive focus on capital markets, businesses and technology

Functional coverage of our solution includes trading, risk management and processing across asset classes

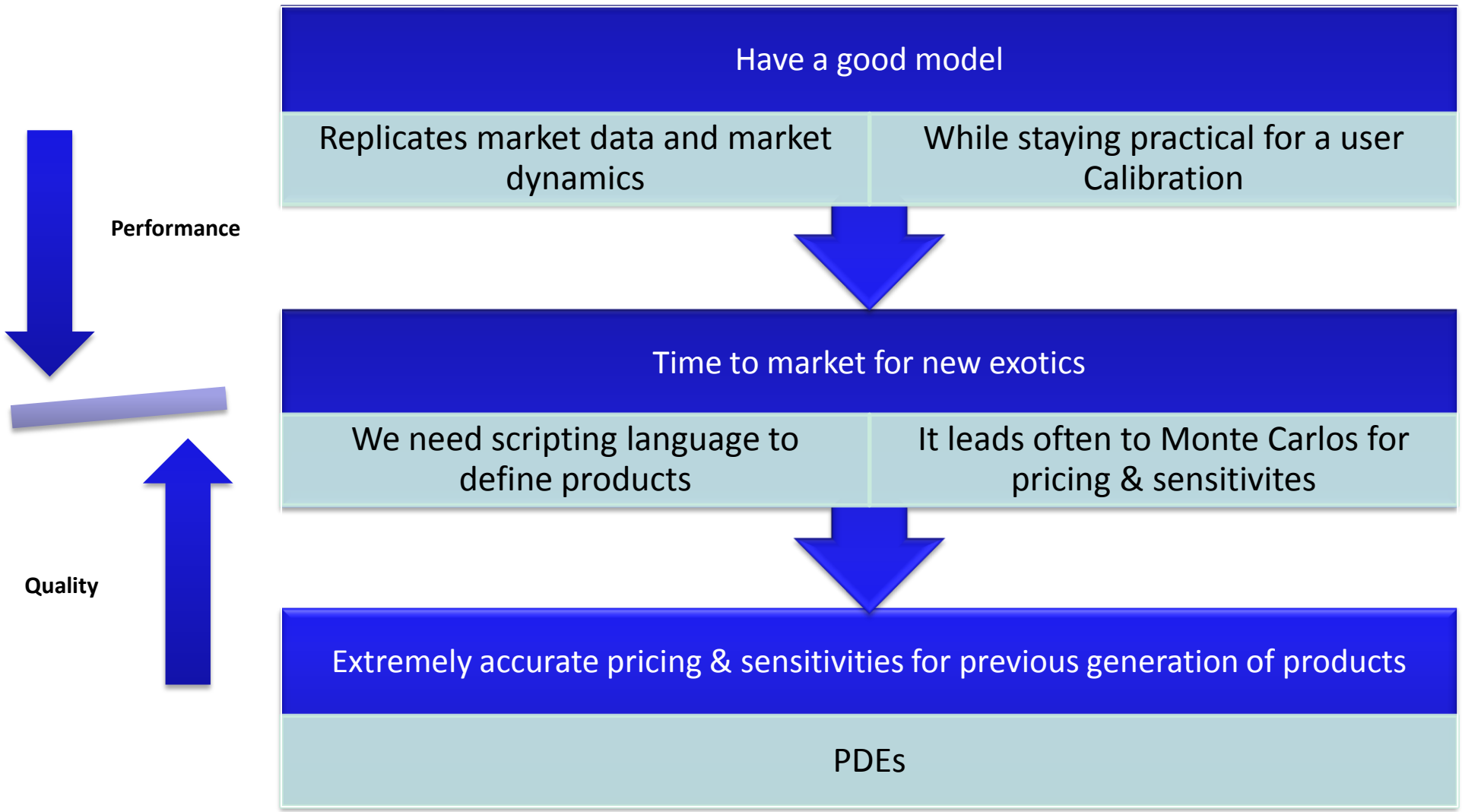
Our team combines strong experience in software design & development, and experience in capital markets. Over 1200 in 9 offices covering all time zones

Over 35 000 users in 65 countries either on the sell or the buy side

Murex is characterized by a high relative investment in R&D supported by a strong balance sheet > 265 Millions Euros in 2009. Early 2010 figures above 305 Millions Euros



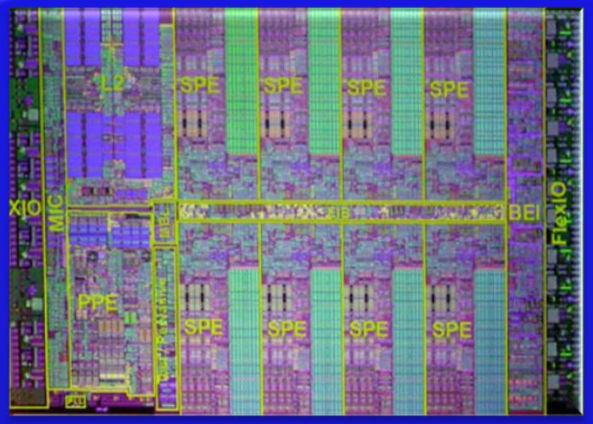
We are quants
What our users are asking for



Our journey toward GPU programming

First CUDA proof of concept on a C870 during April 2008 for a basic PFE example

- Full Monte Carlo paths generation on the CPU
- Only the swaps are evaluated on the GPU
- Already a speed up around 30



One year of mixed experiences with alternative solutions . We learned a lot

- What is a processor
- What is bandwidth
- How programming languages matter
- How large will be the investment
- Begun to rewrite our code with the future in mind

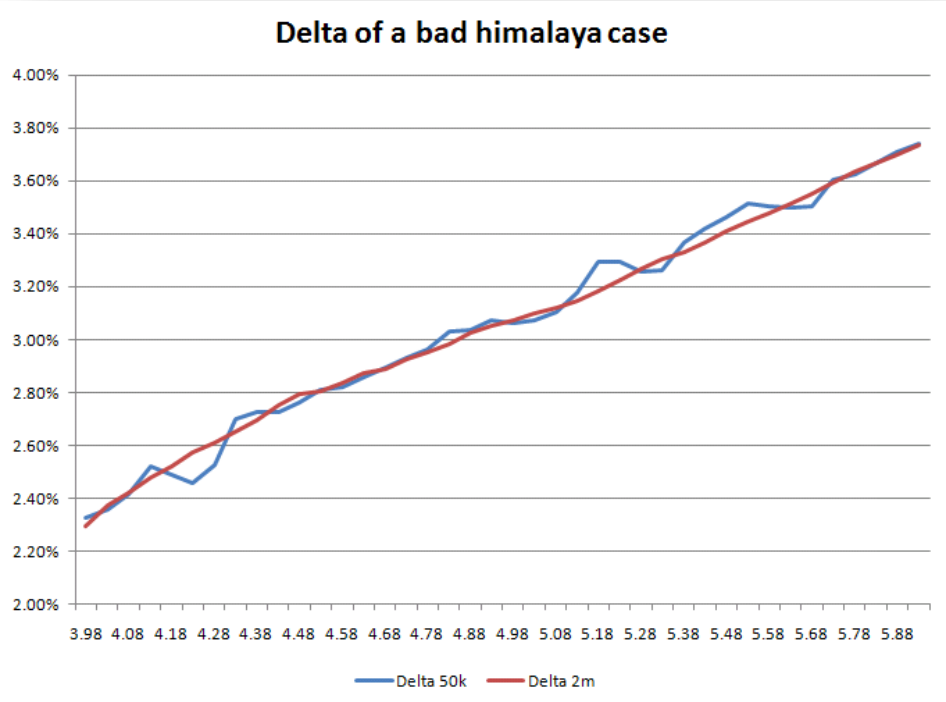
June 2009 : First usable beta drivers are available from NVIDIA

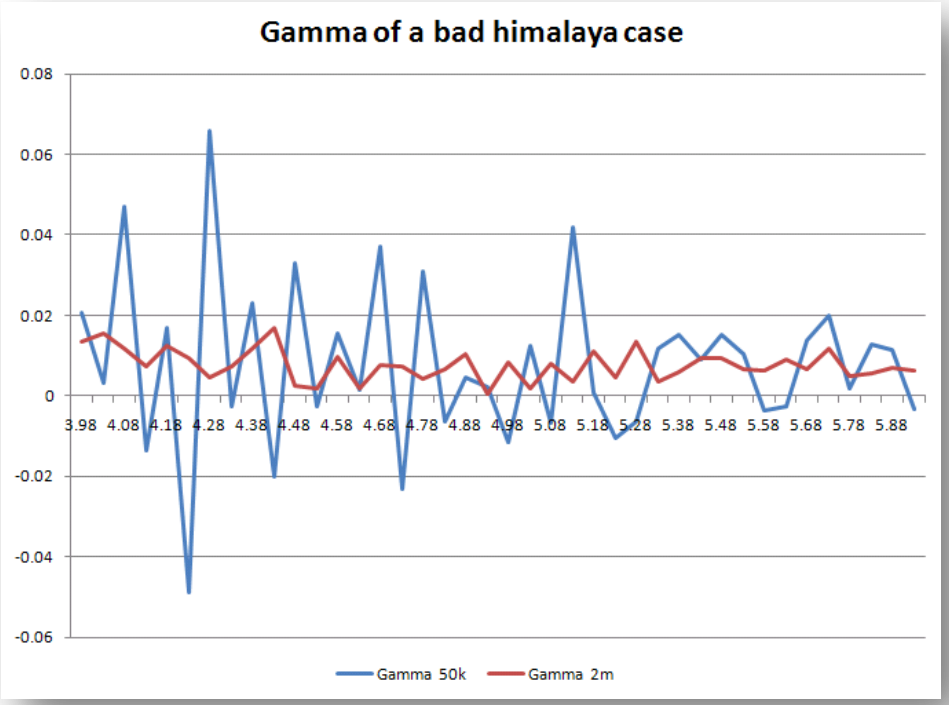
- They have worked well from the beginning and we launched the official development of a sustainable hybrid code
- Solution available for production now

Monte Carlo
The tipping point before Fermi

Tomorrow I'll have my MC's gammas

- Speed eagerly needed knowing the convergence weaknesses of these methods
- Embarrassingly – even data - parallel and fit very well a many core architecture
- Single precision friendly
- Not much difficulties linked to coalesced memory read
- No reduction apart final average & cash flows or events distributions
- A lot of work to cover the legacy code but it stays an easy task

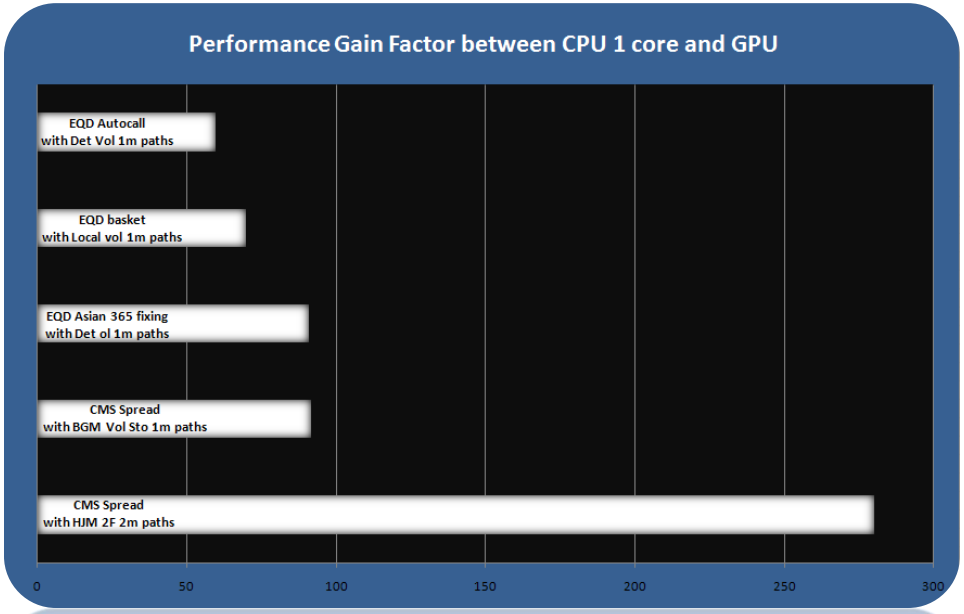


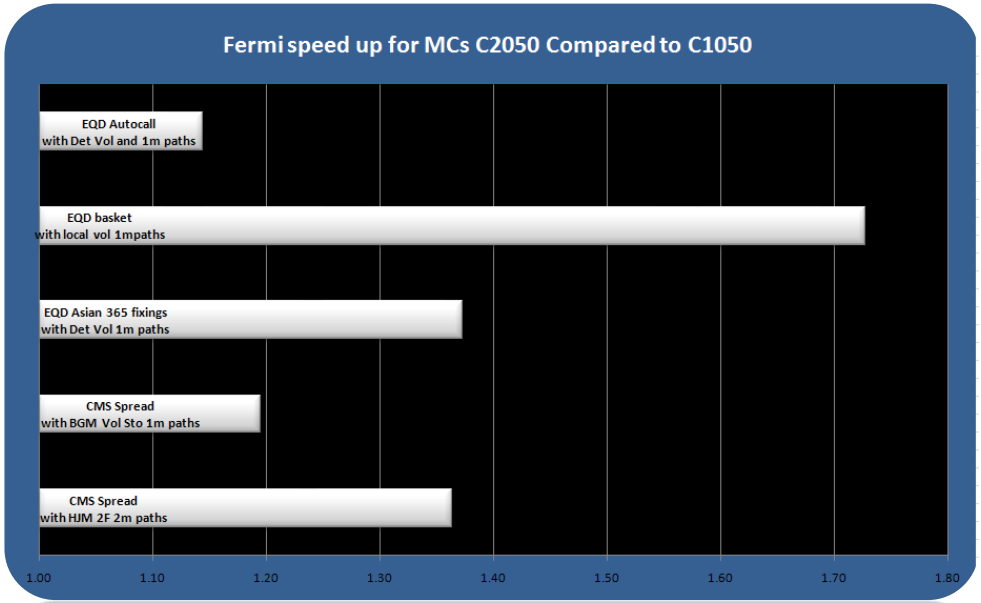


Monte Carlo
The tipping point before Fermi

Tomorrow I'll have my MC's gammas

- Speed eagerly needed knowing the convergence weaknesses of these methods
- Embarrassingly – even data - parallel and fit very well a many core architecture
- Single precision friendly
- Not much difficulties linked to coalesced memory read
- No reduction apart final average & cash flows or events distributions
- A lot of work to cover the legacy code but it stays an easy task





Monte Carlos
One difficulty the scripting language

We need them for flexibility, quality and time to market

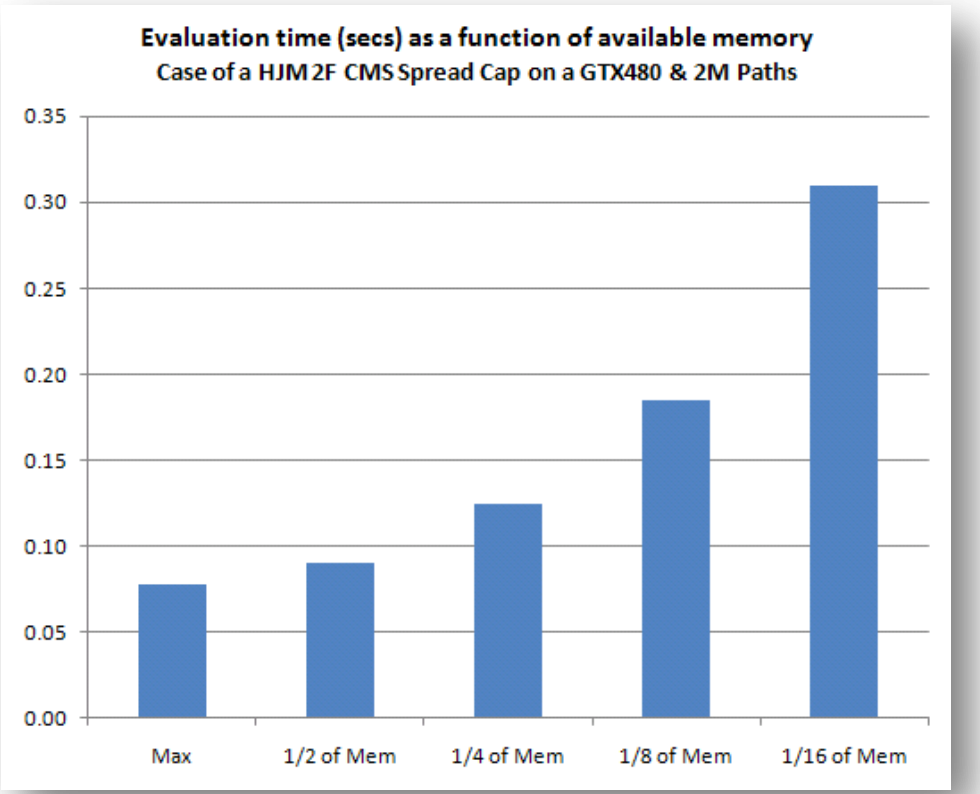
They are slow so we should avoid calling them too many times

Solution : Have your data and models vector based

Replicate calls on the GPU – Alloc, Free, +,-, Discount, Max, … –

Everything will run on the GPU apart the interpreted language

The scripting language chosen should be high level enough to handle abstract type hiding C/C++ pointers to structures containing the reference to the buffers. We have chosen python but I suspect that many other choices would have been perfect.

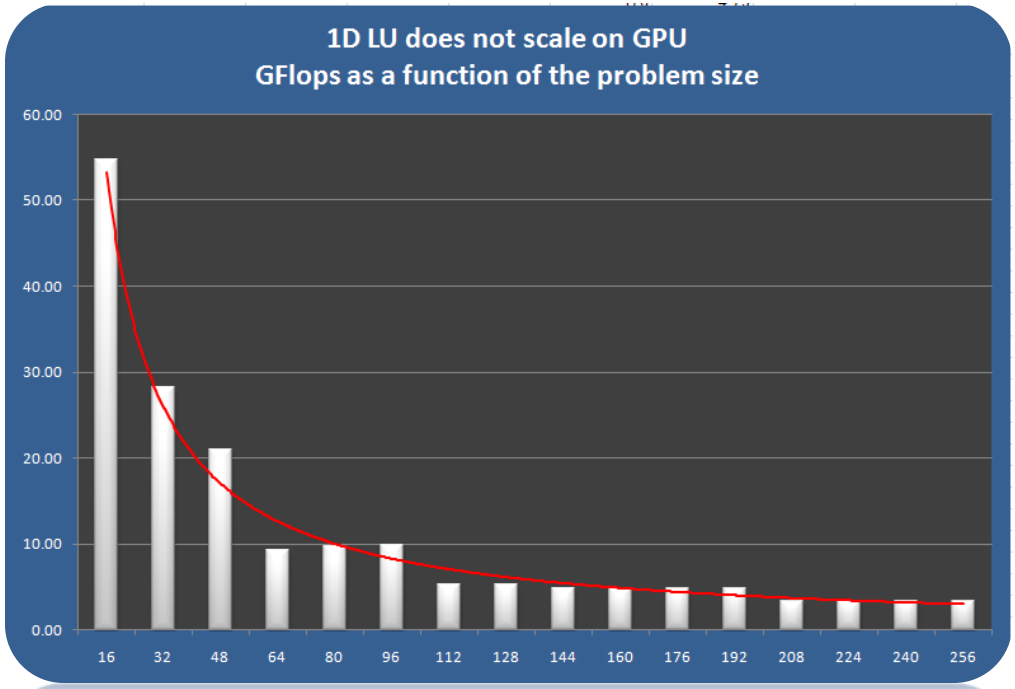


Time to come back to PDEs
1D Crank–Nicolson First

Already fast on the CPU since it is cache friendly thanks to the small size of the problem

Explicit step is easy – see CUDA sample on Cox Tree –

Implicit step is non parallel since we are using Gaussian elimination - Thomas algorithm since Tridiag -



We can theoretically use only one thread per option

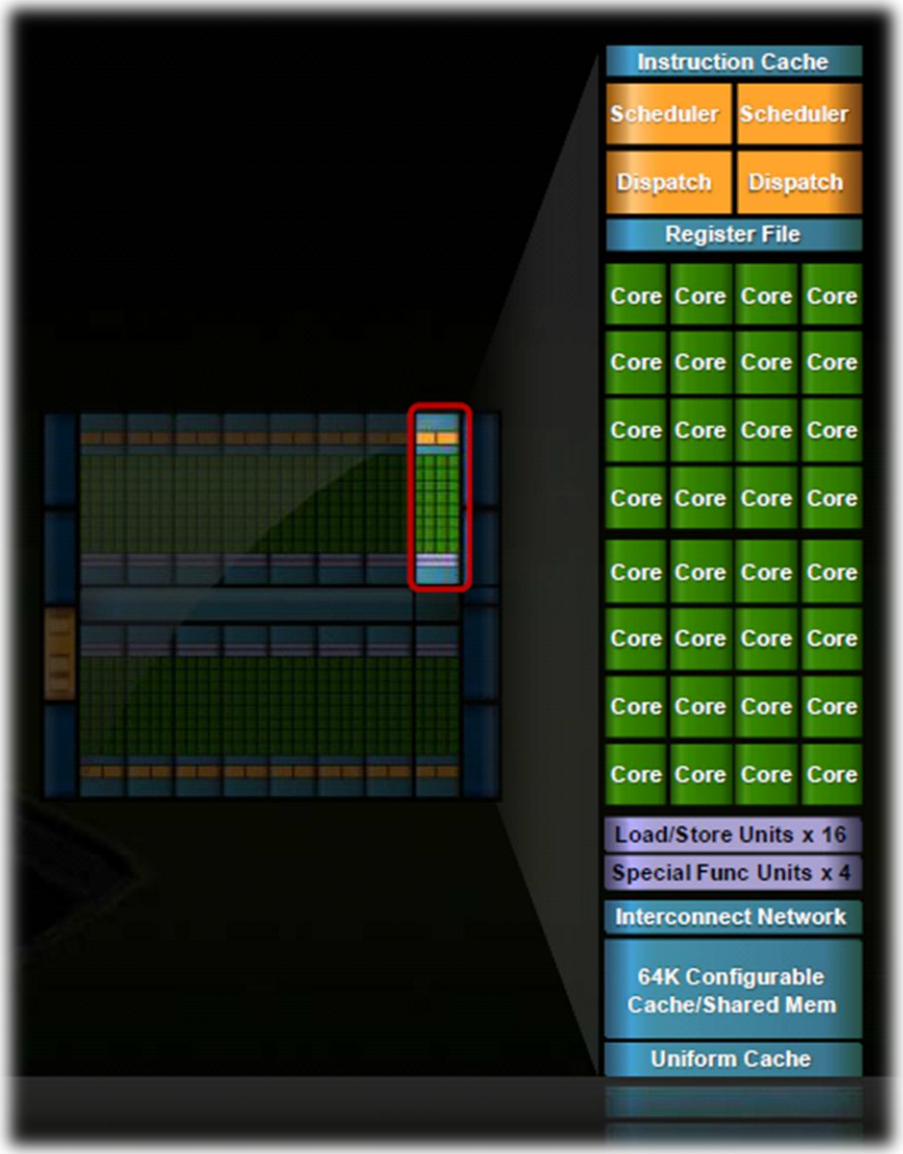
By pricing several options stored in the shared memory at the same time we have a first workaround but it does not scale with spot steps number. GFlops are going down tremendously with the problem size

Time to come back to PDEs
A quick look back at what is a GPU

We should consider the GPU as a set of streaming processors and not as a huge set of cores

Cores in a SM can share data and synchronize

We need to find a solution to solve a tridiagonal system on a SM to use more than one core



Time to come back to PDEs
Parallel cyclic reduction

Use PCR – Zhang, Cohen & Al - instead of standard LU solver

- Divide and conquer like method. Reduce the system by 2 at each step
- More calculations -N x Log2 N- and more memory usage , but we can use as many threads as discretizations
- Can handle dimension big enough for our financial problems >> 1000 on Fermi thanks to enlarge shared memory
- Performance in GFLops increases with the size of the problem and goes above 30Gflops in our current implementation.
- Small for a GPU but big for very sparse matrices

Basic sample focusing on the variable d for a 7 variables heat equation

$$
\begin{array}{rrrrrrrcll}
2a & - & b & & & & & = 1 & & x1/2 \\
-1a & +2b & - & 1c & & & & = 1 & & x1 \\
 & - & b & +2c & - & d & & = 1 & x1/2 & x1/2 \\
 & & & -1c & +2d & -1e & & = 1 & x1 & \\
 & & & & -d & +2e & -1f & = 1 & x1/2 & x1/2 \\
 & & & & & -1e & +2f -1g & = 1 & & x1 \\
 & & & & & & -1f +2g & = 1 & & x1/2
\end{array}
$$

Suppress variables of odd indices from equations of even indices

$$
\begin{array}{rrrcll}
+1 & b & -1/2\,d & = & 2 & x1/2 \\
-1/2\,b & +1 & d -1/2\,f & = & 2 & x1 \\
 & -1/2\,d & +1 & f = & 2 & x1/2
\end{array}
$$

3 variables are remaining and we redo the same thing

$$1/2\,d = 4$$

Only d remains

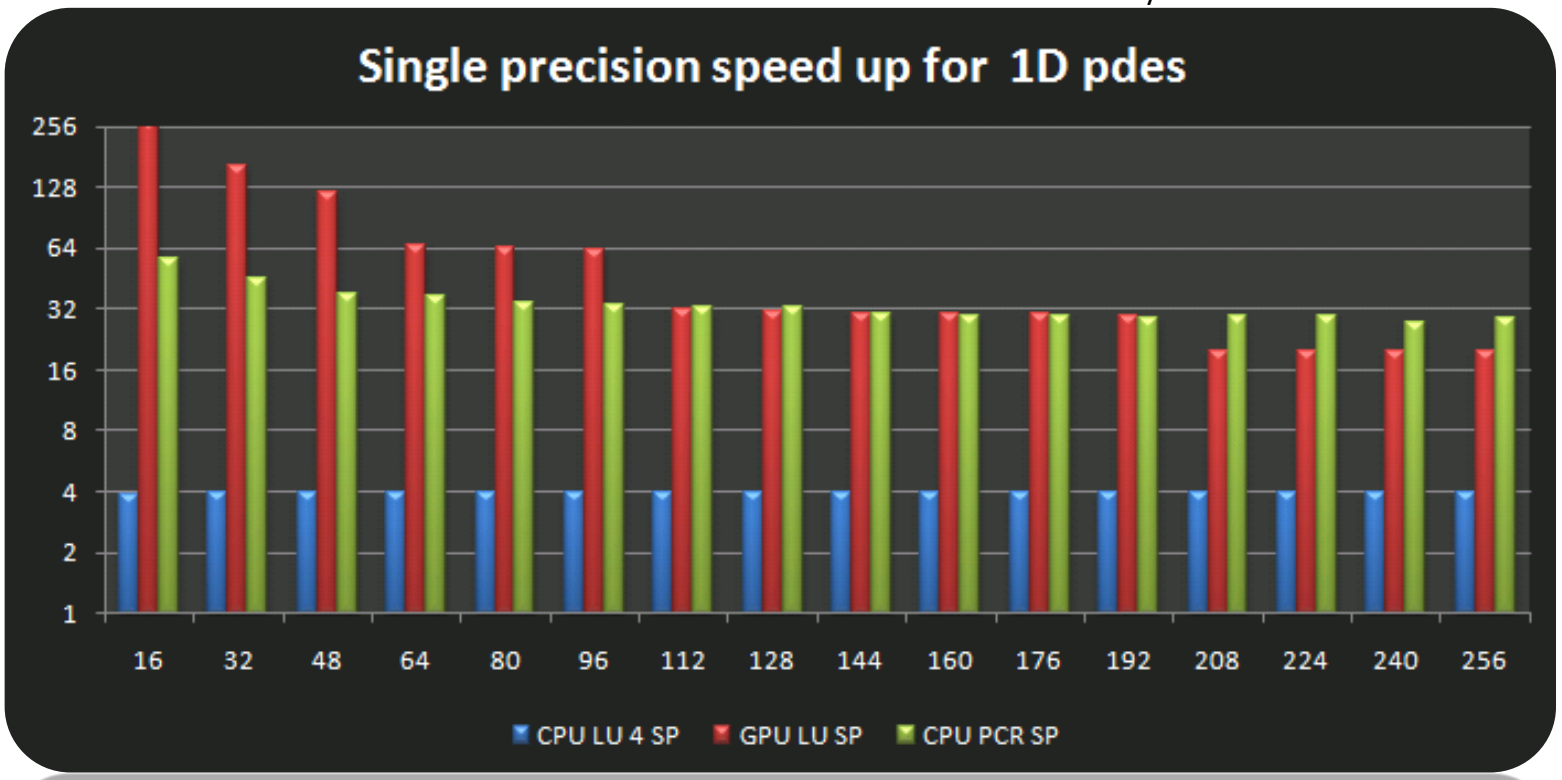
Time to come back to PDEs
Parallel cyclic reduction

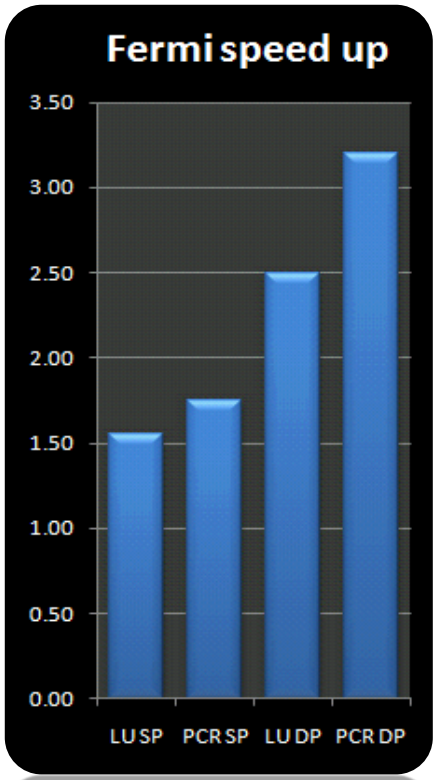
We need to price several options at the same time to feed all SM and obtain peak performances for small problems

- Hardcoded American option screening is easy to adapt
- Payoff & sensitivities calculation have to be handled in parallel for scripted payoffs

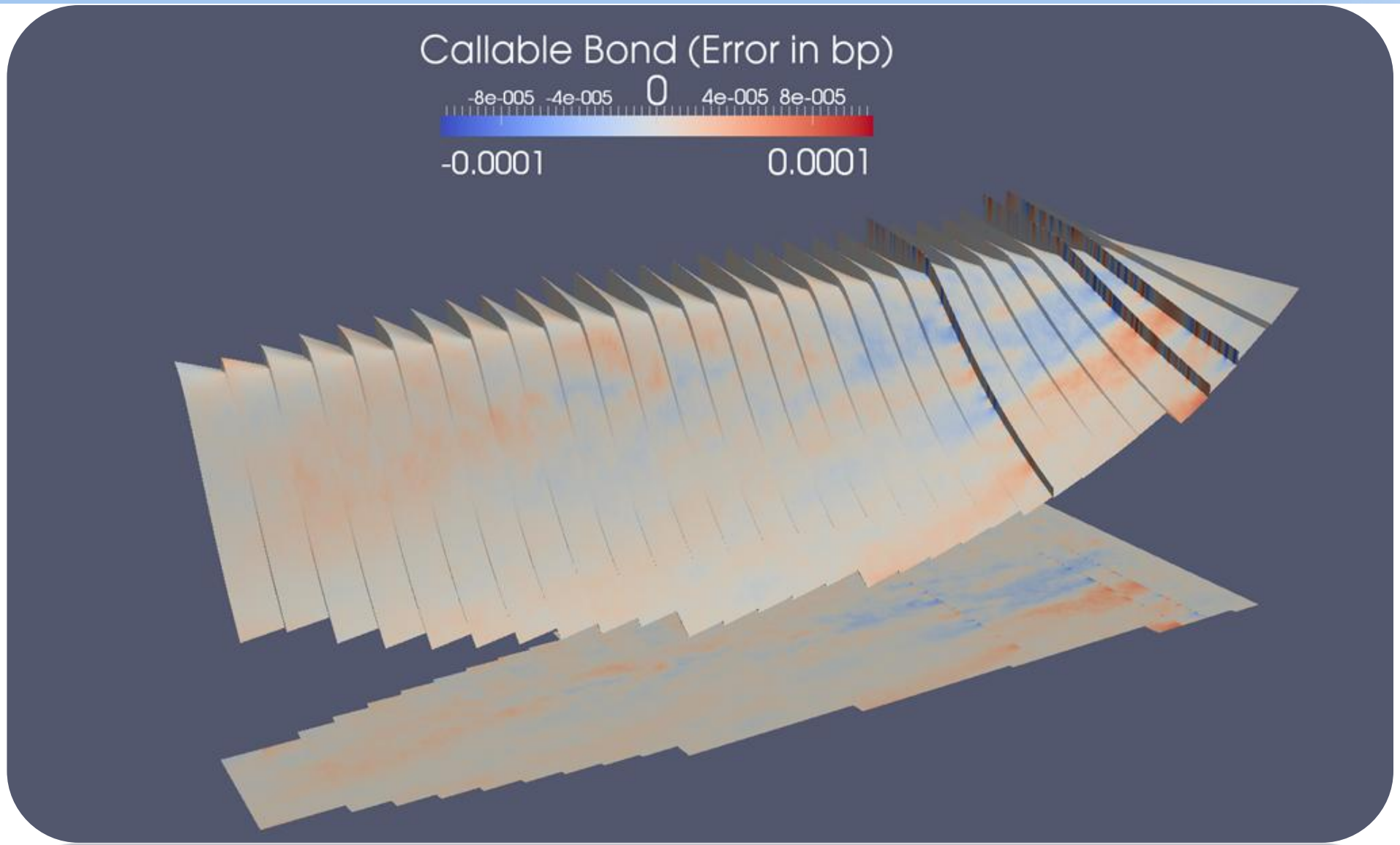
We have now a good solver which enable us to take advantage of fast analytics inside the payoff function without any PCI express communication

- CMS calculation
- Payoff smoothing
- Discount prices when available only with Gaussian quadrature like for markov functional
- And many others





Time to come back to PDEs
Some remarks on a sample case



Time to come back to PDEs
Then 2D

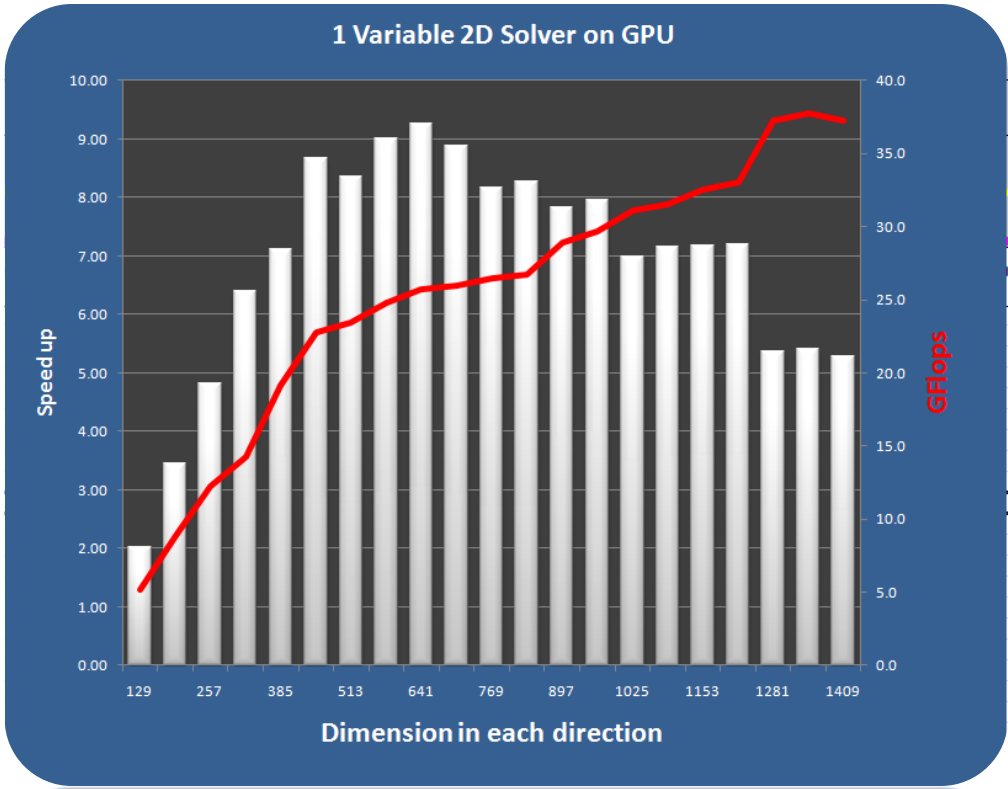
This problem lies in between compute intensive & bandwidth intensive ones

We often use ADI or Vector splitting techniques which are **approximation techniques** that transform a 2D problem into a set of 1D problem

PCR applies then automatically

Can parallelize even **single variable** thanks to GPU thread switch speed which is nearly impossible with openmp on standard processor

Same methodology applies for 3D



Learn from supercomputer
Iterative methods

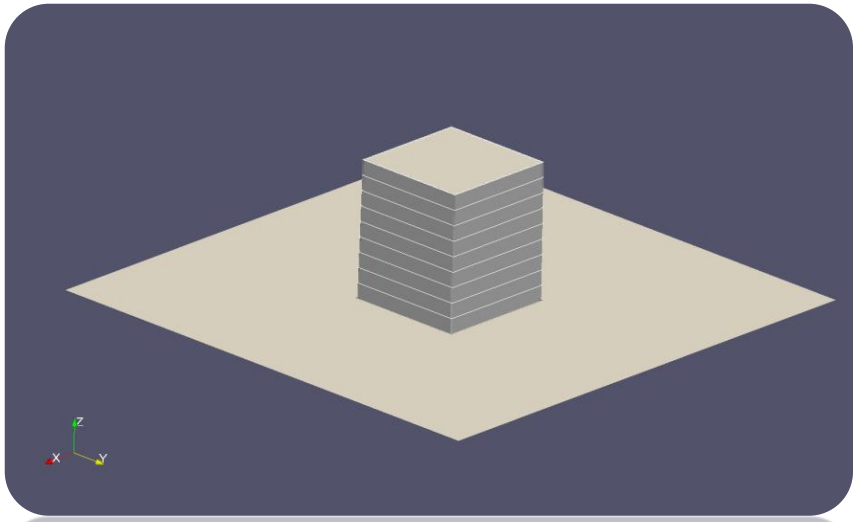
We have focussed on speed up first

Why not use a faster processor for more precision since ADI is only a very good approximation

Solve exactly the implicit step Ax = B using an iterative method

1. Multigrid method

2. Additive Schwarz method with minimal overlapping

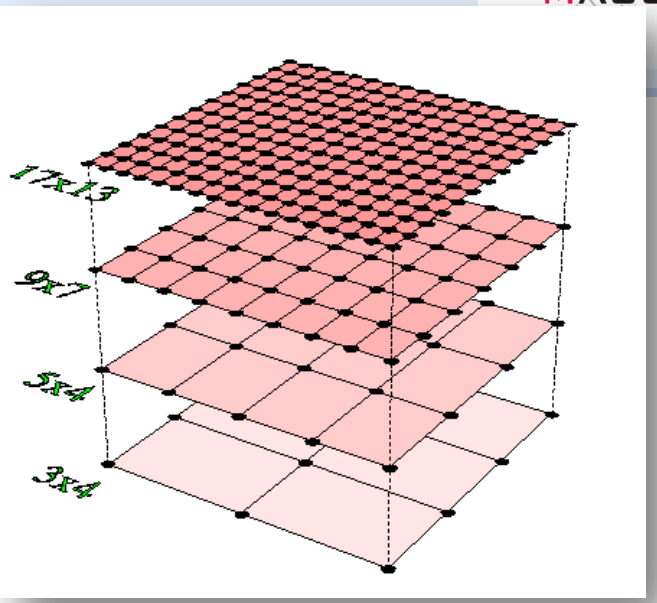


Learn from supercomputer
Multigrid : Treat the GPU as a big SMP

Basic smoother like highly parallel red black Gauss Seidel are converging too slowly when the size of the problem increases

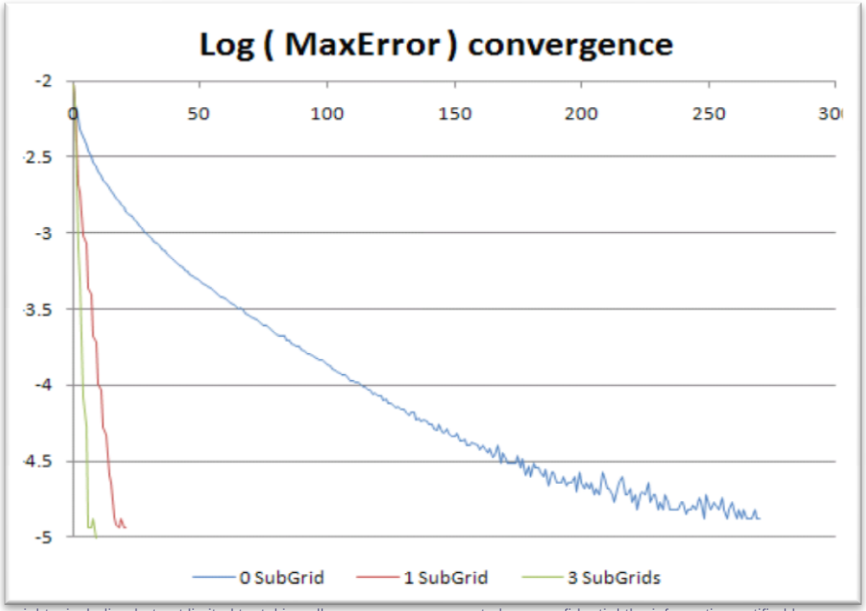
Use smaller grids (coarse) to solve the problem of a big (fine) grid

Benefits from GPUs high bandwidth and texture cache



Number of smoothing steps per level
for different number of sub-grids and 2D problem

Grids Size	0	1	2	3	4
513	**271**	**21**	**11**	**9**	**9**
257		40	20	16	16
129			20	16	16
65				16	16
33					13



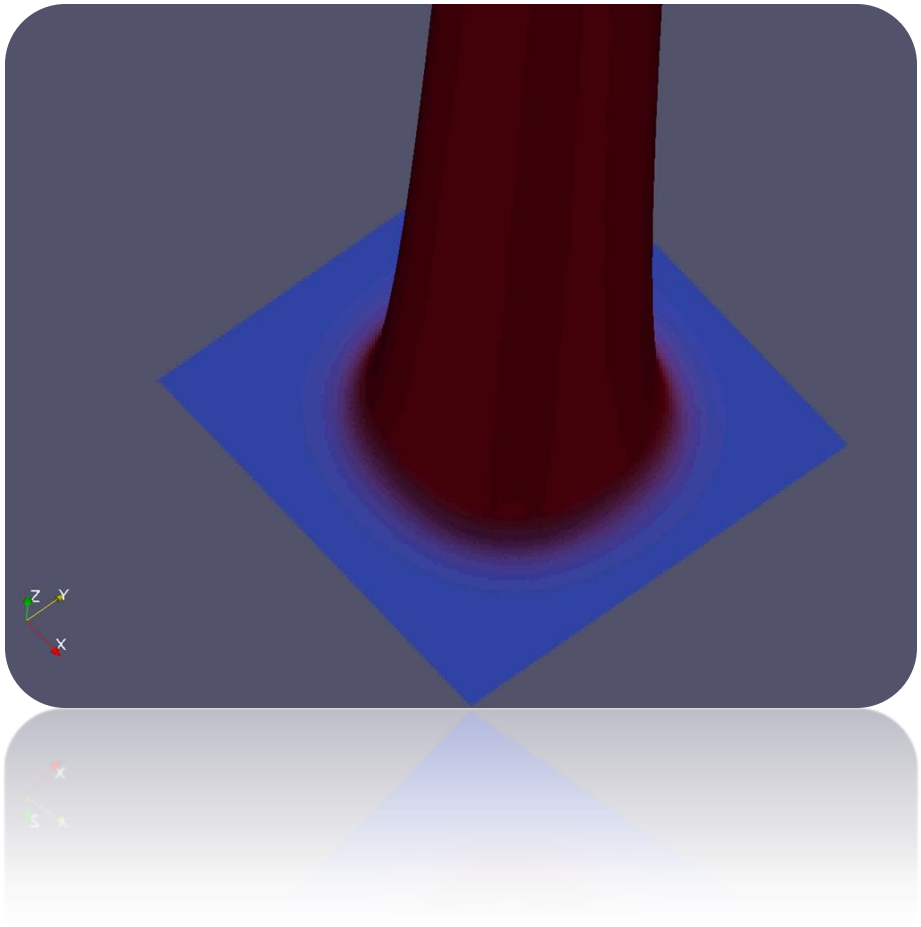
Learn from supercomputer
Schwarz : Treat the GPU as cluster

It is a method used heavily on cluster - MPI - to handle domain decomposition and reduce network communication

Here our cluster is the GPU and each SM is a node

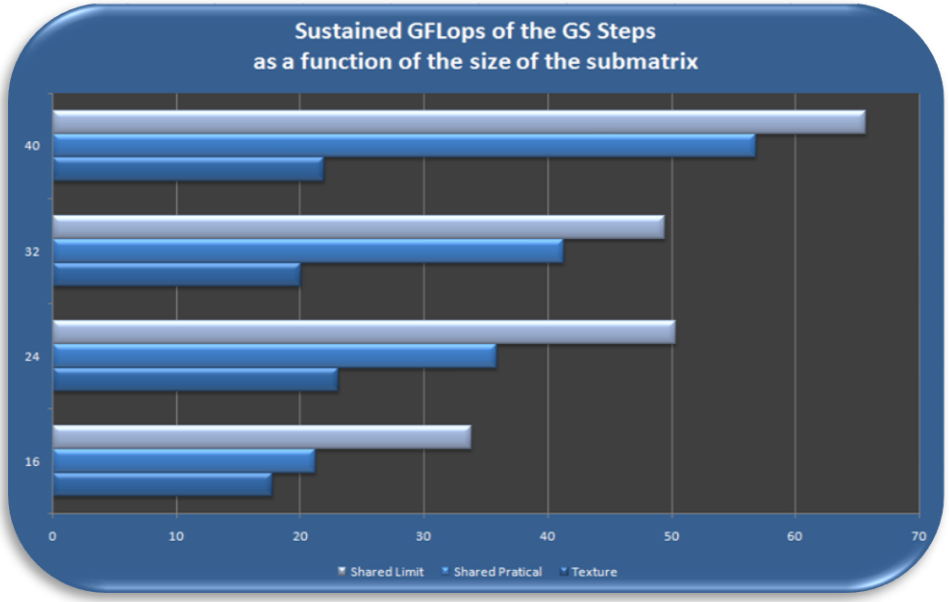
The problem is split into sub-matrices which memory stays on each node of the cluster and the following algorithm is followed

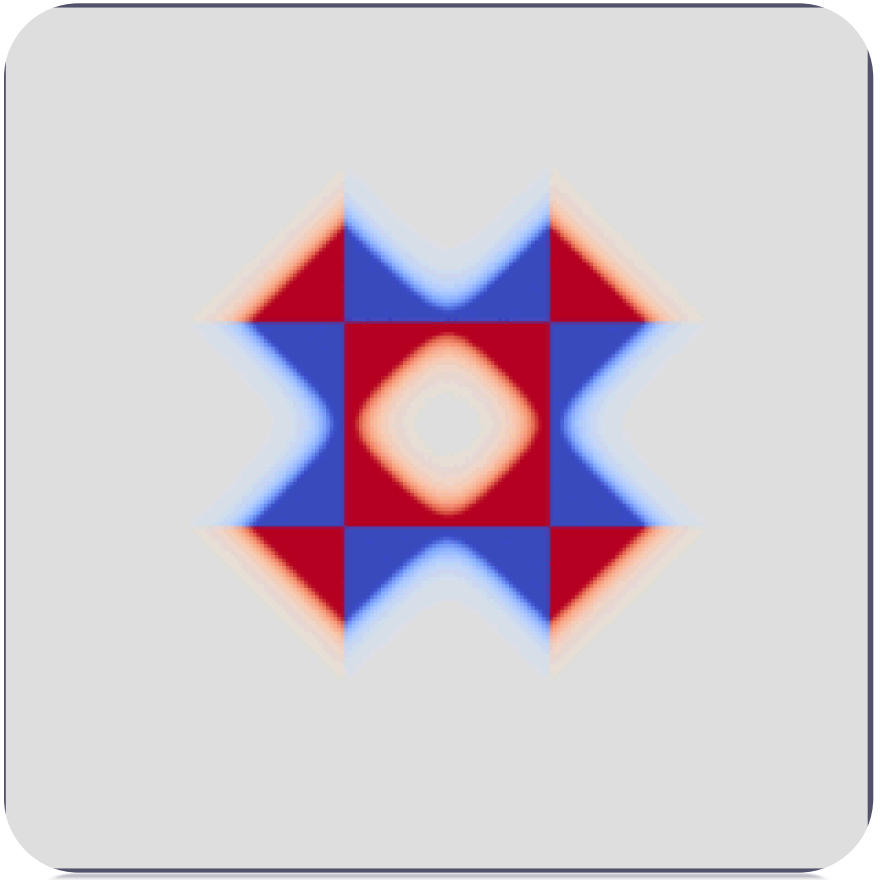
- Independent solves on the system error are run on each sub matrices imposing either Dirichlet or Von Neuman conditions. Sub matrices are sized in a way to stay solely in shared memory
- Each Node send its border result only to its neighbours
- Each matrix updates its boundary condition by at least averaging its value and the one of its neighbours.
- Loop till convergence



Learn from supercomputer
Treat the GPU as cluster

Since we have plenty of GFlops when our data are stored in shared memory we can choose a relatively slow parallel algorithm like multicolor Gauss Seidel on each submatrix.





The method will benefit from additional SM when available

The iterative method is single precision friendly but mixed precision or double precision can be added quickly if needed.

The previous ADI can be used jointly to initialize the inputs of the method or a coarse schwarz level – aka amg – can be added to speed up convergence

Can be used to apply multigrid on multiple GPUs

Final remarks

7. CONCLUSION

In this paper, we analyzed the performance of an important set of throughput computing kernels on Intel Core i7-960 and Nvidia GTX280. We show that CPUs and GPUs are much closer in performance (2.5X) than the previously reported orders of magnitude difference. We believe many factors contributed to the reported large gap in performance, such as which CPU and GPU are used and what optimizations are applied to the code. Optimizations for CPU that contributed to performance improvements are: multithreading, cache blocking, and reorganization of memory accesses for SIMD-ification. Optimizations for GPU that contributed to performance improvements are: minimizing global synchronization and using local shared buffers are the two key techniques to improve performance. Our analysis of the optimized code on the current CPU and GPU platforms led us to identify the key hardware architecture features for future throughput computing machines – high compute and bandwidth, large caches, gather/scatter support, efficient synchronization, and fixed functional units. We plan to perform power efficiency study on CPUs and GPUs in the future.

There is some sense in what Intel says. We do not have always a x100 speed up with GPUs and optimization matters

GPUs are more difficult to program than CPUs but not so much more

But from our experience with CUDA C and OpenCL which are languages far closer to the machine, GPU code is far easier to optimize than CPU code and GPU performance scales far better from one generation to another.

All reviews of our legacy code to port it to GPU led to faster CPU code.

Thank you. Question & answers

References

[1]	Dominik Göddeke : Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaftler
[2]	Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, pages 127–136, January 2010. DOI: 10.1145/1693453.1693472
[3]	A Multigrid Tutorial by William L. Briggs
[4]	MUDPACK: Multigrid Software for Elliptic Partial Differential Equations
[5]	Schwarz Domain Decomposition Methods in the Course of Time Martin J. Gander University of Geneva February 2009
[6]	Debunking the 100X GPU vs. CPU Myth : An Evaluation of Throughput Computing on CPU and GPU Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher ,Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy,Srinivas Chennupaty, Per Hammarlund, Ronak Singhal and Pradeep Dubey
[7]	Tipping point: The performance benefits of GPUs put an elephant on the scales. Murex Risk magazine September 2010