Éléments de parallélisme et parallélisation automatique pour GPU et moulticœurs

Informatique parallèle et distribuée INF560, École Polytechnique

Ronan KERYELL

HPC Project

—

9 Route du Colonel Marcel Moraine
92360 Meudon La Forêt

—

Rond Point Benjamin Franklin
34000 Montpellier

17/02/2011

http://www.par4all.org

Modéliser le monde

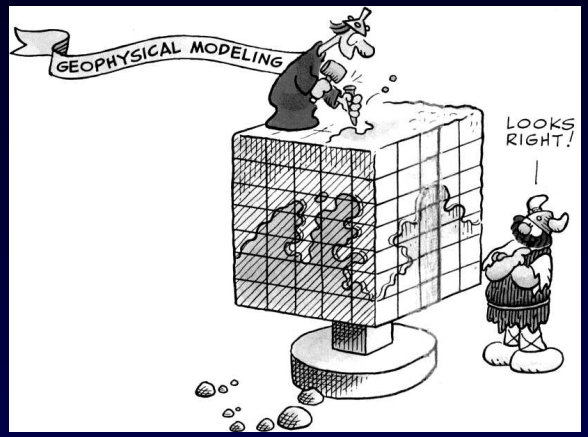
- Vieux rêve de l'humanité : mieux comprendre le monde
- Développement des mathématiques pour comprendre
- Modélisation du monde dans un formalisme mathématique pour prévoir
- Automatisation des calculs lents & immenses, sujets aux erreurs

Développement d'ordinateurs pour :

- Gros modèles : prévisions plus précises
- Automatisation de tâches (gestion)
- Nécessité de résultats rapidement

Contraintes sur les ordinateurs

- Plus d'informations à stocker (maillages + fins) ↗ mémoire



- Plus de calculs à faire, ↗ vitesse de calcul
  - Débit : beaucoup de calculs aboutissent/unité de temps
  - Latence : temps d'exécution d'une tâche
- Applications limitées par les performances
- Désirs : toujours plus !

  μηδὲν ἄγαν
- Mesures par programmes étalons (*Benchmark*)

Ordinateurs les plus rapides d'une époque (1–4 ordres de grandeur) :

supercalculateurs

Gros gains aussi sur les algorithmes...

Top 500

`http://top500.org`

- Liste 500 plus gros ordinateurs déclarés dans le monde depuis 1993
- Top 10 : crème de la crème
- Étalon : factorisation de matrice *LU* LINPACK
  - Plus de calculs que de communications
  - Cas d'école hyper régulier rarement rencontré dans la vraie vie
  -  À considérer comme une puissance crête (efficace)

Permet d'estimer les directions futures technologiques de l'informatique « standard »

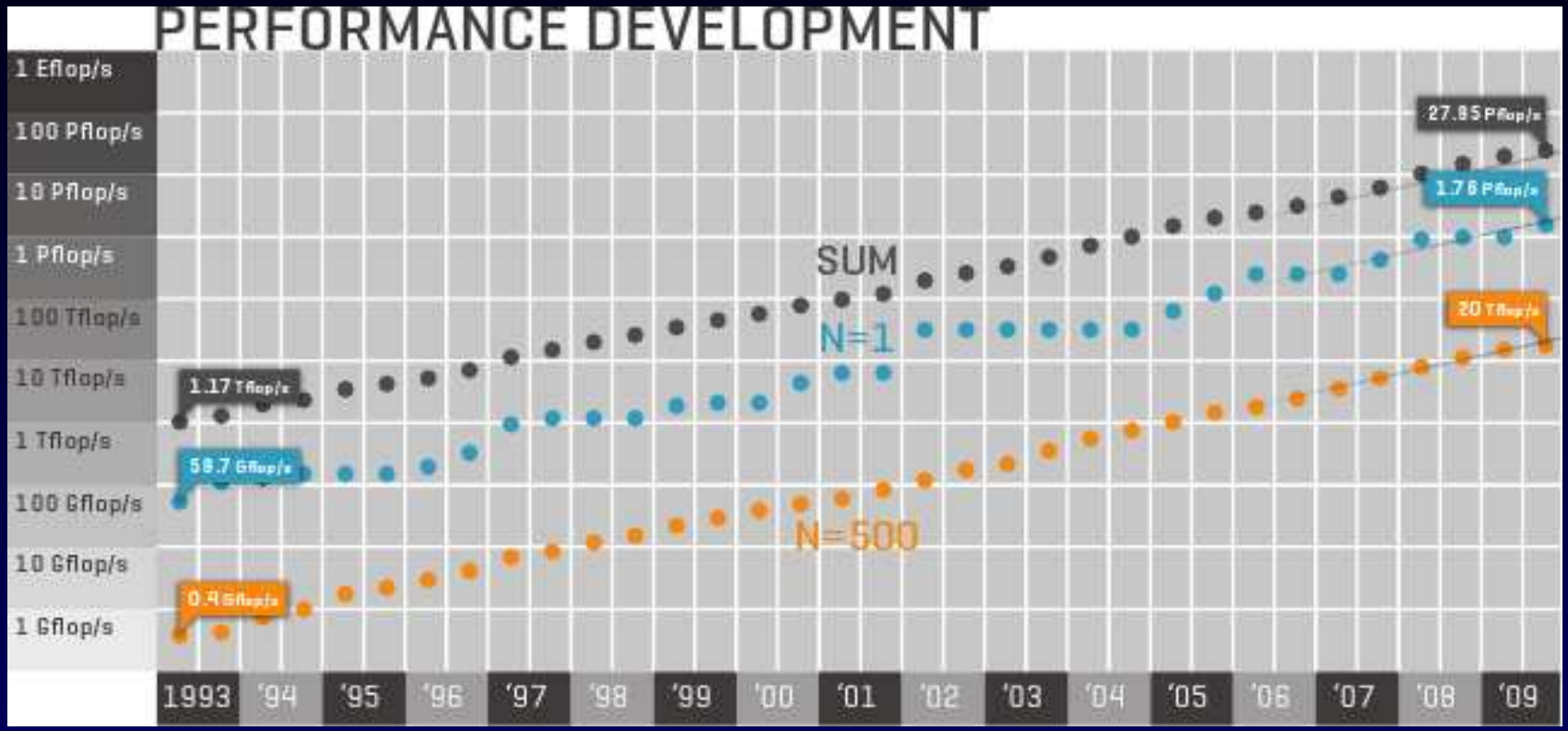
Top 10 — November 2010

TFLOPS performance

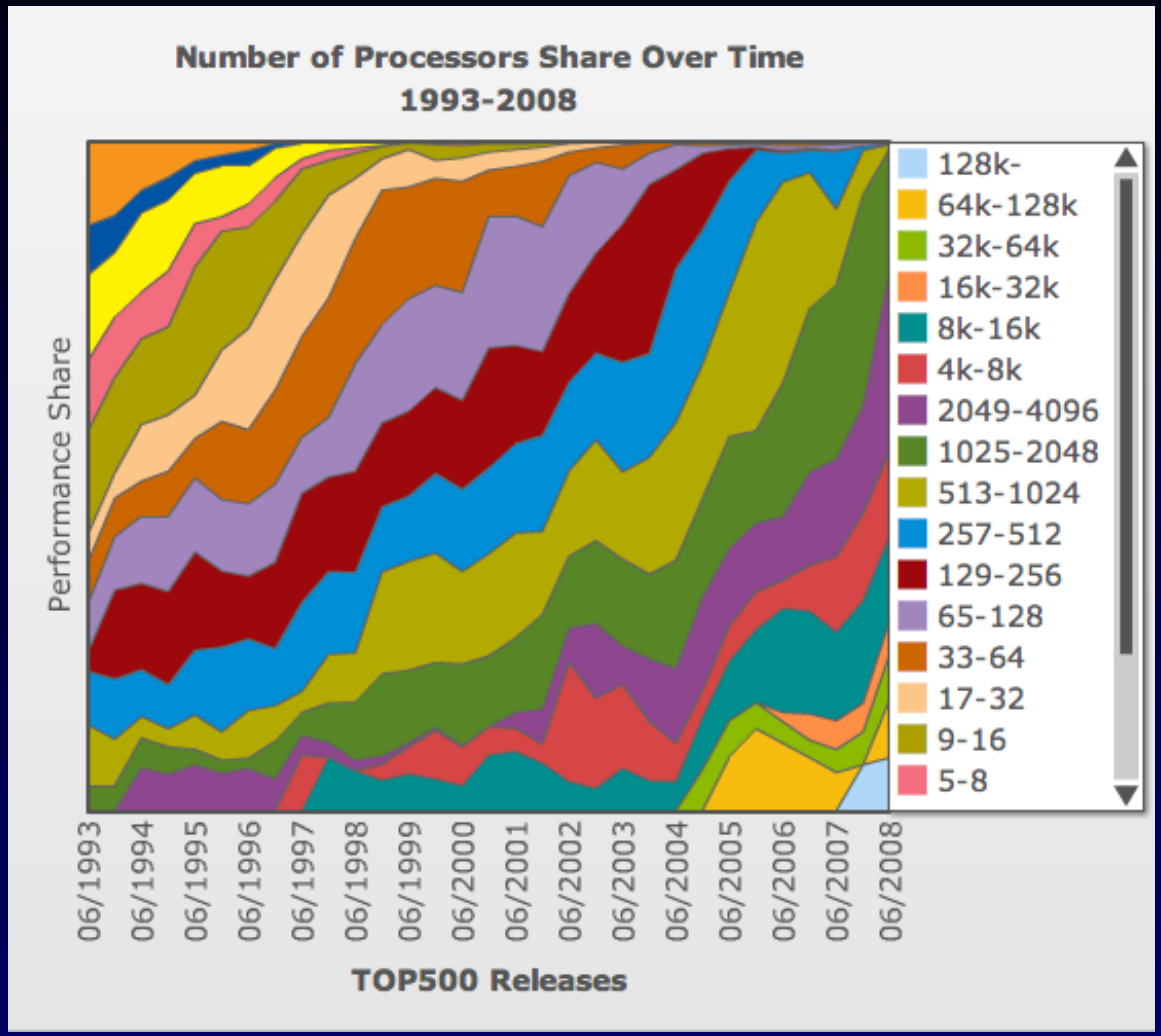
Rank	*Site*	*Computer/Year Vendor*	*Cores*	*Rmax*	*Rpeak*	Power (kW)
1	National Supercomputing Center in Tianjin (China)	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 (NUDT)	186368	2566	4701	4040
2	DOE/SC/Oak Ridge National Laboratory (United States)	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 (Cray Inc.)	224162	1759	2331	6950
3	National Supercomputing Centre in Shenzhen (NSCS) (China)	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 (Dawning)	120640	1271	2984	2580
4	GSIC Center, Tokyo Institute of Technology (Japan)	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010 (NEC/HP)	73278	1192	2287	1398
5	DOE/SC/LBNL/NERSC (United States)	Hopper - Cray XE6 12-core 2.1 GHz / 2010 (Cray Inc.)	153408	1054	1288	2910
6	Commissariat a l'Energie Atomique (CEA) (France)	Tera-100 - Bull bullx super-node S6010/S6030 / 2010 (Bull SA)	138368	1050	1254	4590
7	DOE/NNSA/LANL (United States)	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 (IBM)	122400	1042	1375	2345
8	National Institute for Computational Sciences/University of Tennessee (United States)	Kraken XT5 - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 (Cray Inc.)	98928	831	1028	3090
9	Forschungszentrum Juelich (FZJ) (Germany)	JUGENE - Blue Gene/P Solution / 2009 (IBM)	294912	825	1002	2268
10	DOE/NNSA/LANL/SNL (United States)	Cielo - Cray XE6 8-core 2.4 GHz / 2010 Cray Inc.	107152	816	1028	2950

http://www.top500.org/list/2010/11/100

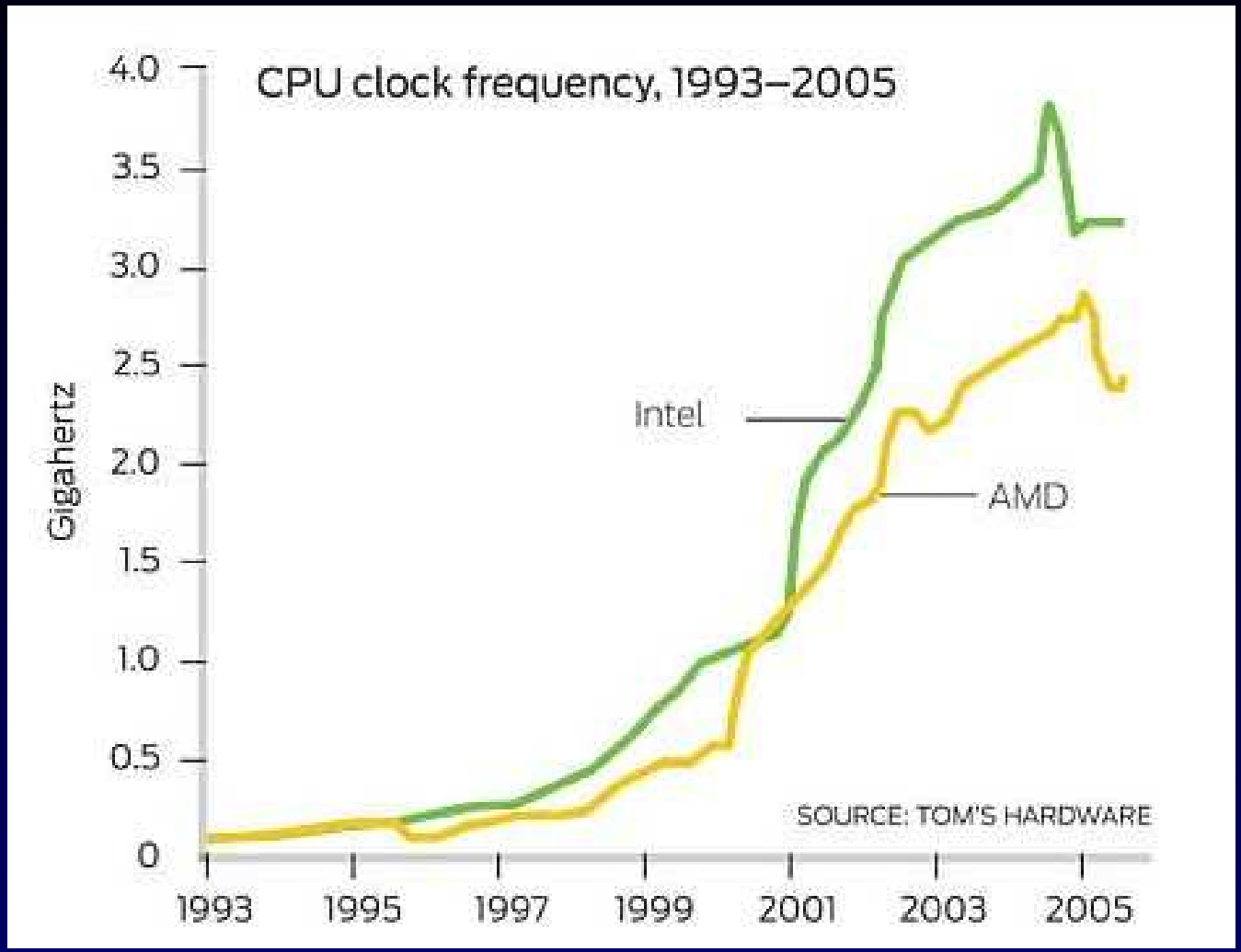
Performance totale — novembre 2009



Parallélisme massif — 06/2008



Évolution vitesse des processeurs



Green 500

- Problème de puissance dissipée dans les centres de calcul... ☹
- 6 MW $\approx$ 600 €/h, $\approx$ 5 M€/an...
- Futur aux économies d'énergie
- $\rightsquigarrow$ Classement supplémentaire
  - TOP Green500: most powerful supercomputers running the Linpack benchmark ranked by energy efficiency
  - Little Green500: most energy-efficient supercomputers achieving at least 9 tflops on the linpack benchmark
  - Open Green500: exploratory list for energy-efficient supercomputers achieving more than 9 tflops on linpack however they wish
  - HPCC Green500: exploratory lists for most energy-efficient supercomputers running the HPCC benchmark

`http://www.green500.org`

Tendances

- Fréquence des processeurs : n'évolue plus
- Loi de Moore apporte encore des transistors
  - Toujours plus de parallélisme : *many*cores
  - Architectures hétérogènes : GPGPU, Cell, vectoriel/SIMD, FPGA
- Compilateurs toujours derrière... ☹

The “Software Crisis”

Edsger DIJKSTRA, 1972 Turing Award Lecture, « The Humble Programmer »

“To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.”

`http://en.wikipedia.org/wiki/Software_crisis`

But... it was before parallelism democratization!☹

Évolution logicielle

Jusqu'à présent...

- Langages d'assemblage
- Langages de haut niveau pour machines à la von NEUMANN (Fortran, C...)
- Programmation orientée objet pour composabilité, malléabilité et maintenabilité de gros programmes
- Bibliothèques de composants, outils, patrons de conception, spécifications, modélisation, méthodologies, tests...

Hautes performance ?

Bah... Loi de MOORE ☺

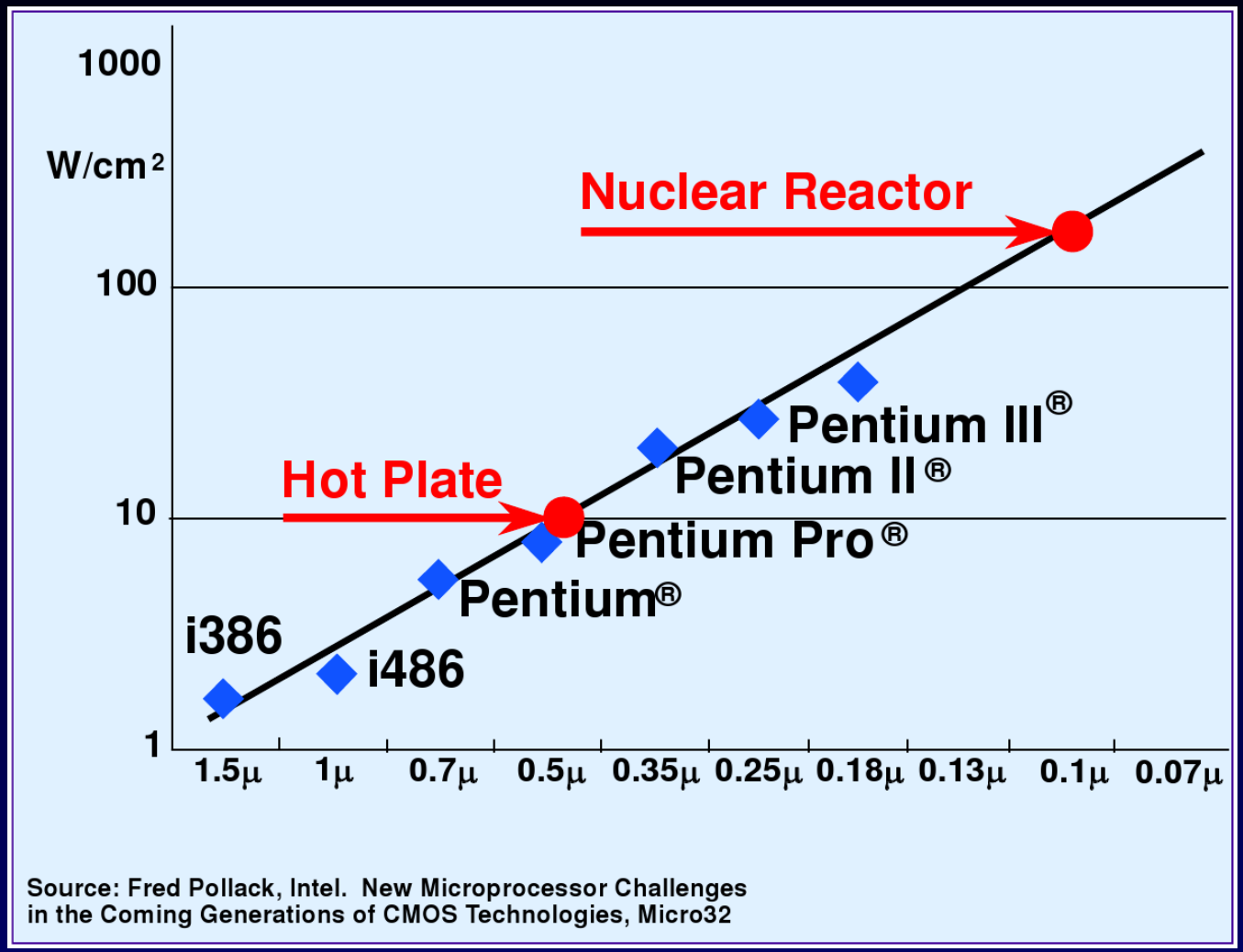
Programmeurs inconscients des processeurs...

- On ne mélange pas les ~~mouchoirs en dentelle~~softeux avec les ~~serpillières~~hardeux ☹
- Langages de haut niveau loin du matériel (encore pire avec JVM, CLI...)
- Abstraction qui a permis beaucoup de liberté créatrice aux programmeurs ☺
- L'avenir est au Web 3.0 ! ☺
- La vitesse ? Travailleurs de l'ombre (loi de MOORE...) font qu'un programme antique va beaucoup plus vite aujourd'hui sur n'importe quel processeur ! ☺

⇝ Un programmeur peut tout ignorer des processeurs

$\exists^?$ encore cours d'architecture des ordinateurs en école d'ingénieurs ? ☹

Densité de puissance



Source: Fred Pollack, Intel. New Microprocessor Challenges in the Coming Generations of CMOS Technologies, Micro32

Fin de l'augmentation des performances séquentielle

- Passe d'un facteur 2 tous les 1,5 ans à tous les ≈5 ans... ☹
- Pourtant besoin de toujours plus de performances
  - Devise de Delphes « rien de trop » μηδὲν ἄγαν
  - Données traitées plus grandes
  - Plus de fonctionnalités par €
  - Plus de fonctionnalités par W

Seule solution : faire du parallélisme...

...et garder le moral : « composabilité, malléabilité et maintenabilité, portabilité... »

Comment faire face ?

Exemple projet logiciel dans monde selon Moore

Déroulement

- 2010 : démarrage du projet logiciel : 8 cœurs par circuit intégré
- fin 2011 : sortie première version : 16 cœurs par circuit intégré
- 2013 : seconde version, 32 cœurs par circuit intégré

Serez-vous prêt(e) ?

Dans le monde des écoles d'ingénieur...

- 2010 : un élève intègre. Programmation séquentielle
- 2012 : un élève en stage. Ouch ! $\exists$ parallélisme
- 2013 : un élève dans la vraie vie... Faire du parallélisme sans formation
- 2016 : un élève passe sa thèse, 256 cœurs/circuit... 

Heureusement la formation continue existe... ☺

Programmation parallèle

- Généralisation des machines moulticœurs & hétérogènes
  - Ordinateurs de bureaux ou portables
  - Supercalculateur (domaine d'origine)
  - Systèmes embarqués (téléphones, voitures, radars...)
- 1 carte graphique (GPU nVidia Fermi ou AMD/ATI HD 5870) $\equiv$ 2+ TFLOPS
- Futur : encore + hétérogène
- $\exists$ standards de programmation parallèle
  - OpenMP : multithread pour les nuls (?), mémoire partagée
  - MPI : passage de message pour les nuls (?), mémoire distribuée, tâches
  - OpenCL : architectures hétérogènes (CPU, GPU, accélérateurs) pour les nuls (?)

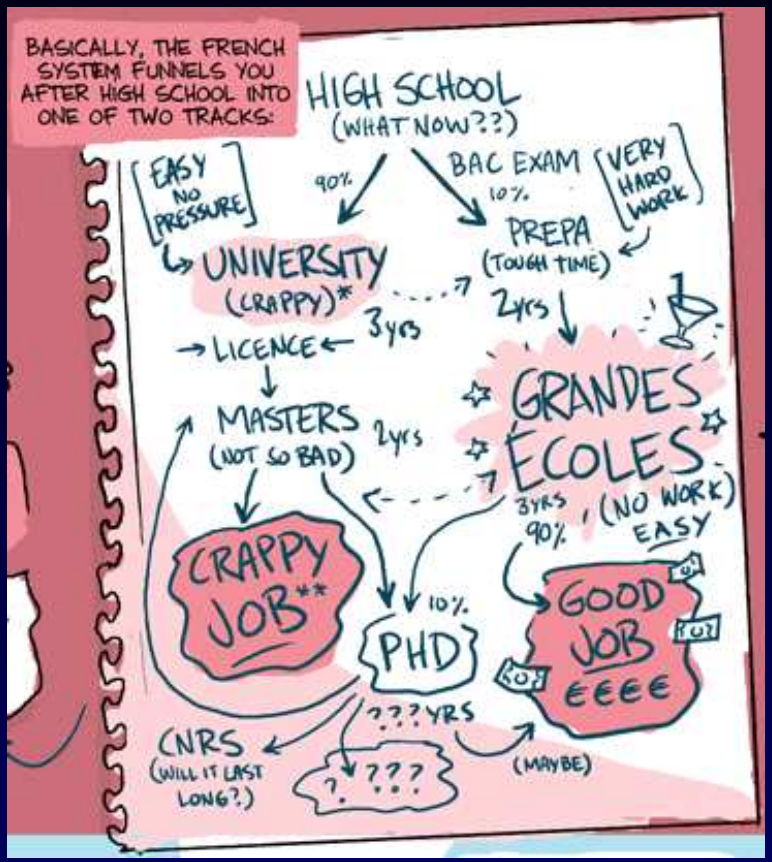
Dure réalité du parallélisme (I)

http://www.phdcomics.com/comics/archive.php?comicid=1292

- Rajoute du sucre syntaxique
- Problème de DRH
- Importance croissante des pannes...

PhDcomics du 3/15/2010, visite de Jorge CHAM à l'X

Dure réalité du parallélisme *(II)*

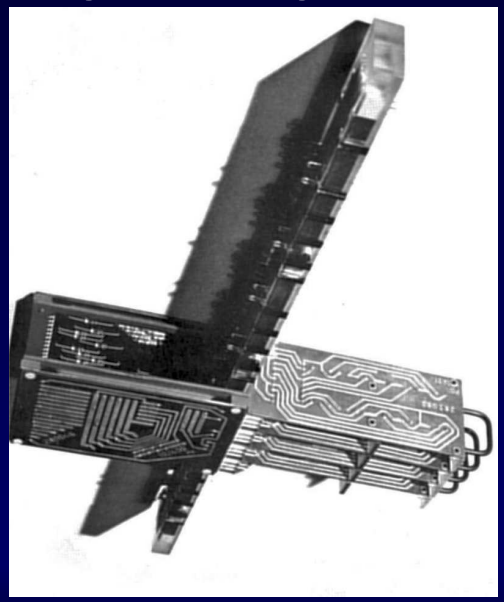


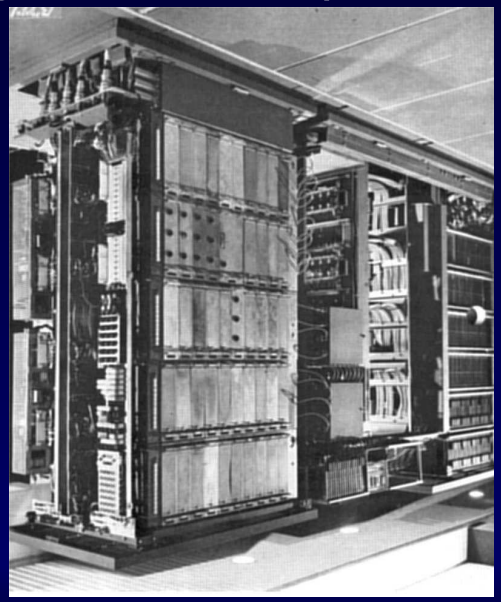
We *really* need an elevator...

Multicores strike back... (I)

Gamma 60 from Compagnie des Machines Bull

- Increase the performance
- 100 kHz memory clock
- *Heterogeneous* multicore because the memory is too... fast! ☺
- 24-bit words, 96 KiB of core memory
- Punch-cards with ECC, magnetic tapes, magnetic drums
- Highly integrated logic in 1-mm germanium bipolar lithography ☺





Multicores strike back... *(II)*

- Gamma 60 multithread programming with `SIMU` ($\approx$ fork) & `CUT` (launch a program on a functional unit) instructions
- Synchronization barrier by concurrent branching on a same target
- Scheduling of threads based on a queue per functional unit stored just... inside the code after each `CUT` instruction!
- Optional hardware critical section on subprograms (cf. `synchronized` of Java)
- Installation around 1959
- Already hard to program since the concepts were not here, at most the (grand-)parents of anyone who were to know them...

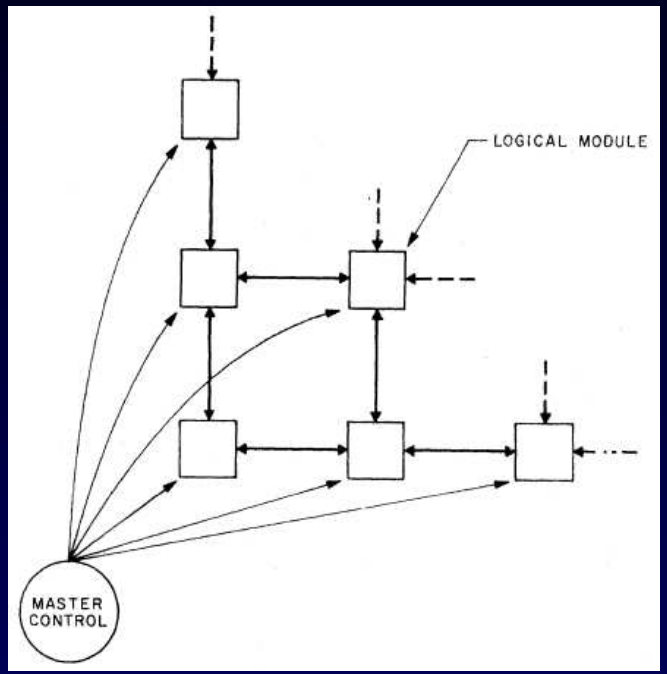
`http://www.feb-patrimoine.com/projet/gamma60/gamma_60.htm`

GPGPUs: just more integrated... *(I)*

- The “Distributed Computer”
  - Toward computing on spatial data : pattern recognition, mathematical morphology...
  - Massive parallelism to reduce the cost
  - SIMD

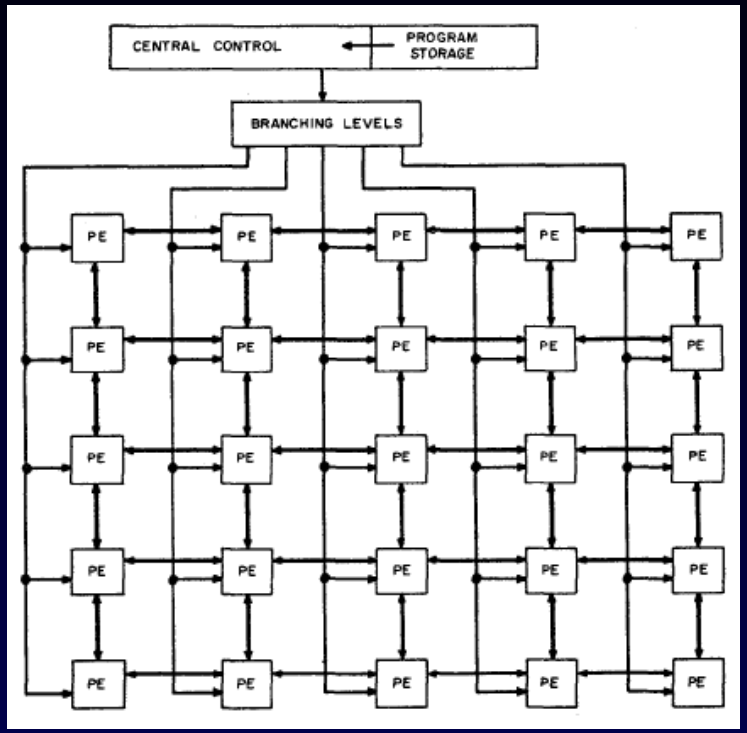
  S. H. Unger. « A Computer Oriented Toward Spatial Problems. » *Proceedings of the IRE*. p. 1744–1750. oct.

1958



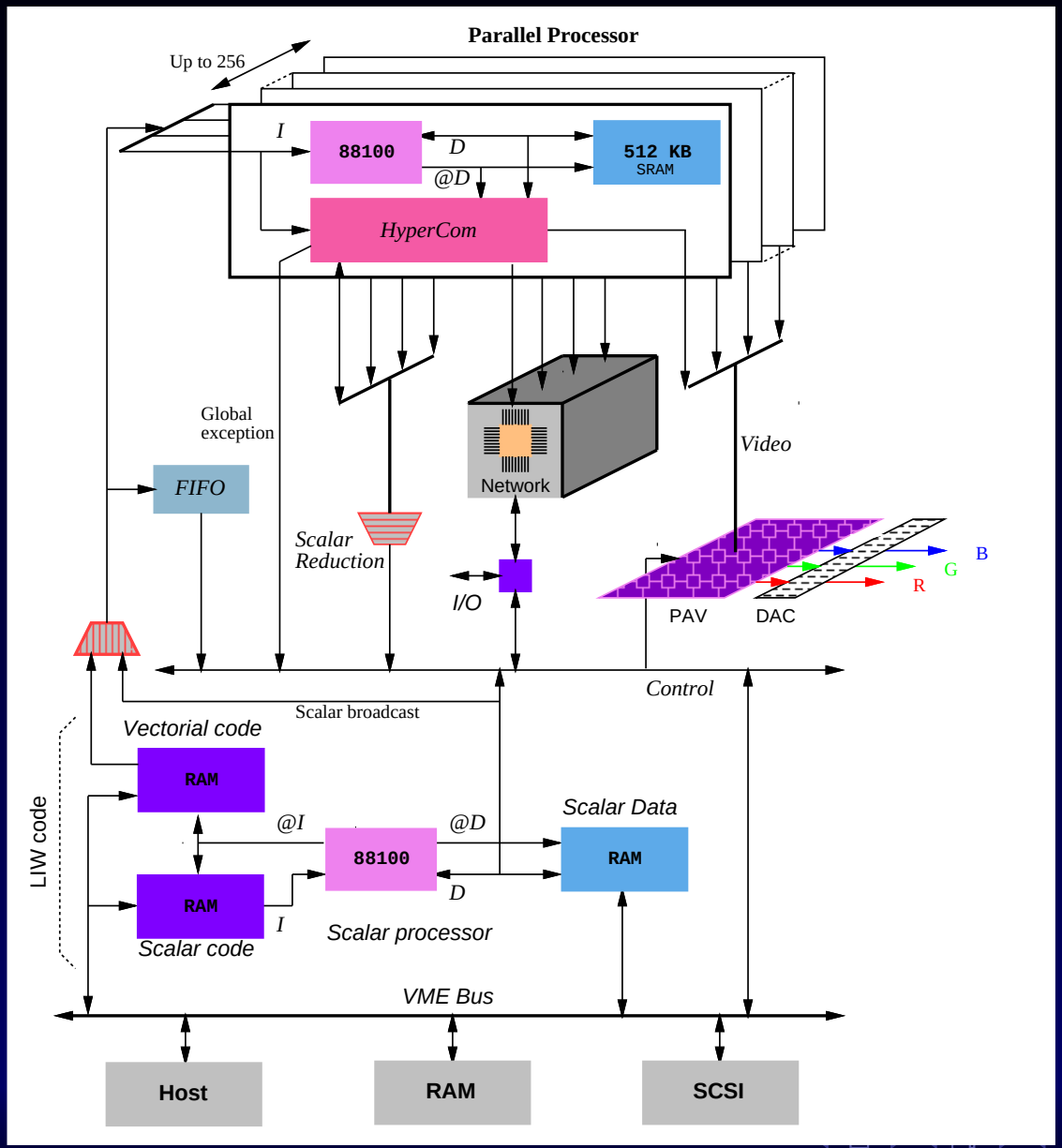
GPGPUs: just more integrated... *(II)*

- SOLOMON
  - Target application: “data reduction, communication, character recognition, optimization, guidance and control, orbit calculations, hydrodynamics, heat flow, diffusion, radar data processing, and numerical weather forecasting”
  - Diode + transistor logic in 10-pin TO5 package



Daniel L. Slotnick. « The SOLOMON computer. » *Proceedings of the December 4-6, 1962, fall joint computer conference*. p. 97–107. 1962

POMP & PompC @ LI/ENS 1987–1992



TechnoCloCgy shrinking

That was the oldies... but now:

- System... $\rightsquigarrow$ on Chip (SoC)
- Multi-Processor System... $\rightsquigarrow$ on Chip (MP-SoC)
- Network... $\rightsquigarrow$ on Chip (NoC)
- Data center... $\rightsquigarrow$ on Chip
- Heat... $\rightsquigarrow$ on Chip ☹

Present motivations

- MOORE's law there are more transistors but they cannot be used at full speed without melting ☹
- Superscalar and cache are less efficient compared to transistor budget
- Chips are too big to be globally synchronous at multi GHz ☹
- Now what cost is to move data and instructions between internal modules, not the computation!
- Huge time and energy cost to move information outside the chip

Parallelism is the only way to go...

Research is just crossing reality!

No one size fit all...

Future will be heterogeneous

More performances? Nothing but parallelism!

Good time for more startups! ☺
2 previous start-ups in //ism:

- 1986: Minitel servers with clusters of Atari 1040ST (128 users/Atari!), MIDI LAN ☺, PC+X25 cards as front-end
  50% of the total French 3614 chat at Minitel climax
- 1992: HyperParallel Technologies (Alpha processors + FPGA, 3D-torus, HyperC language) on the Saclay Plateau ☺

⇝ 2006: Time to be back in parallelism!

Yet another start-up... ☺

- People that met ≈ 1990 at the French military lab SEH/ETCA and evolved as researchers in Computer Science, CINES director, venture capital and more: ex-CEO of Thales Computer, HP marketing...
- ≈ 25 colleagues in France (Montpellier, Meudon), Canada (Montréal) & USA (Mountain View)

HPC Project hardware: WildNode from Wild Systems

Through its Wild Systems subsidiary company

- WildNode hardware desktop accelerator
  - Low noise for in-office operation
  - *x*86 manycore
  - nVidia Tesla GPU Computing
  - Linux & Windows

- WildHive
  - Aggregate 2-4 nodes with 2 possible memory views
    - Distributed memory with Ethernet or InfiniBand
    - Virtual shared memory through Linux Kerrighed for single-image system

`http://www.wild-systems.com`

HPC Project software and services

- Parallelize and optimize customer applications, co-branded as a bundle product in a WildNode (e.g. Presagis Stage battle-field simulator, WildCruncher for Scilab//...)
- Acceleration software for the WildNode
  - GPU-accelerated libraries for Scilab/Matlab/Octave/R
  - Transparent execution on the WildNode
- Remote display software for Windows on the WildNode

HPC consulting

- Optimization and parallelization of applications
- *High Performance?...* not only TOP500-class systems: power-efficiency, embedded systems, green computing...
- $\rightsquigarrow$ Embedded system and application design
- Training in parallel programming (OpenMP, MPI, TBB, CUDA, OpenCL...)

Outline

Current trends

- 2 vendors in the high-end market: nVidia & AMD/ATI
- Each vendor produce 2 versions of an architecture
  - A high end version with double precision floating point support and big memory bus, even ECC

In this talk we focus on the high end

  - A low end version, with less engines, smaller memory bus, less or no double precision floating point. Even less documented (do not target scientists ☺)
- The low end is a way to sell... broken parts! ☺ (as the IBM Cell in the Sony PS3...)

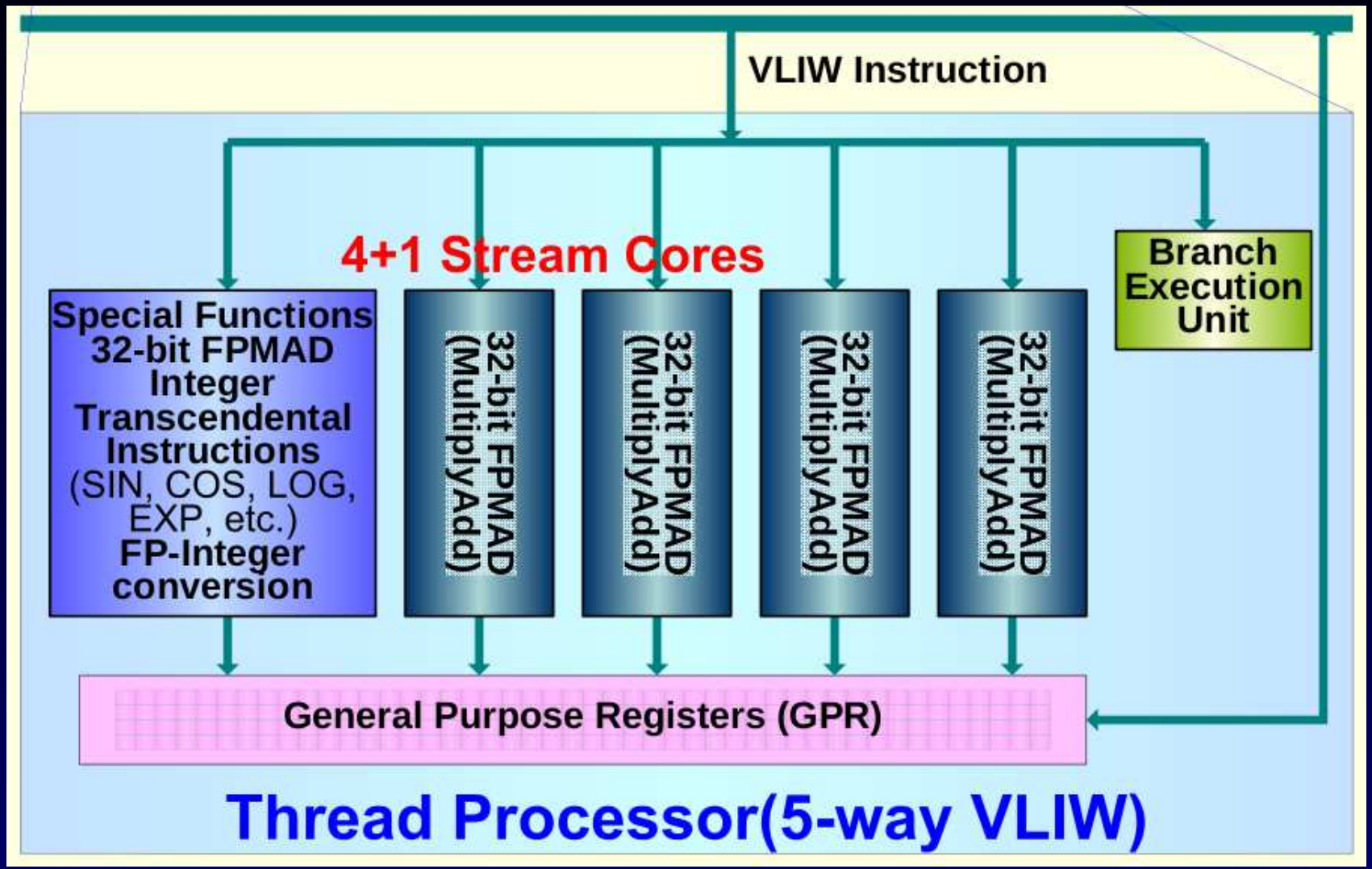
Off-the-shelf AMD/ATI Radeon HD 6970 GPU

- 2.64 billion 40nm transistors
- 1536 stream processors @ 880 MHz, 2.7 TFLOPS SP, 675 GFLOPS DP
- + External 1 GB GDDR5 memory 5.5 Gt/s, 176 GB/s, 384b GDDR5
- 250 W on board (20 idle), PCI Express 2.1 x16 bus interface
- OpenGL, OpenCL

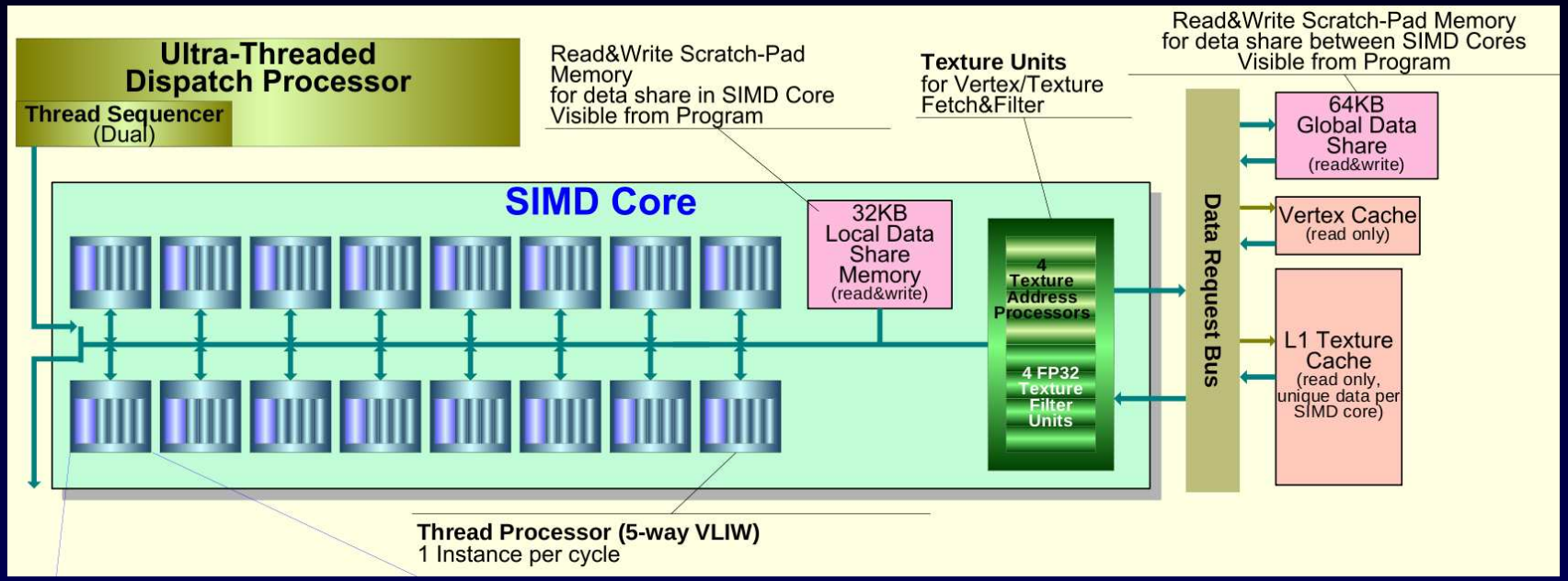
More integration:

- Llano APU (FUSION Accelerated Processing Unit) : x86 multicore + GPU 32nm, OpenCL

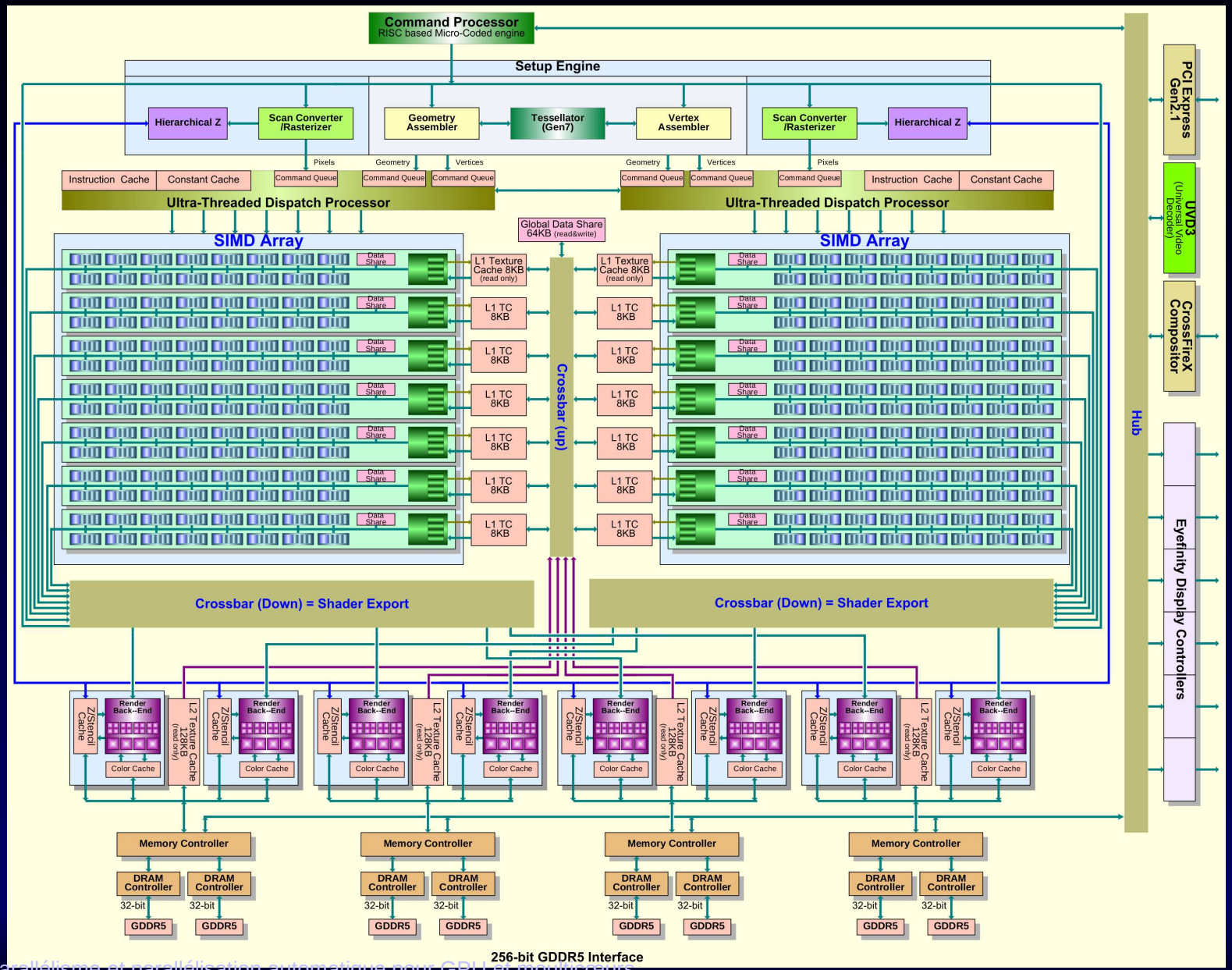
Radeon HD 6870 — thread processor



Radeon HD 6870 — SIMD core

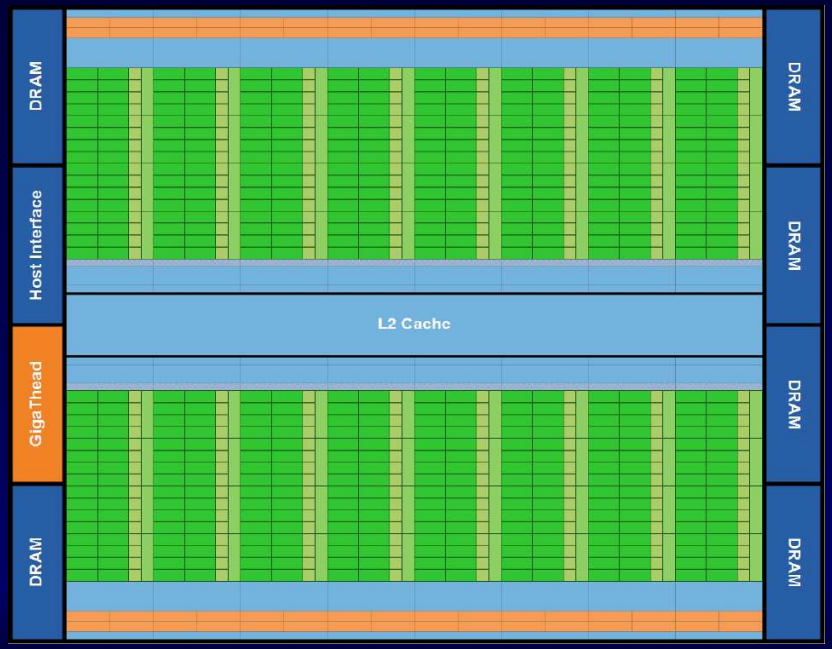


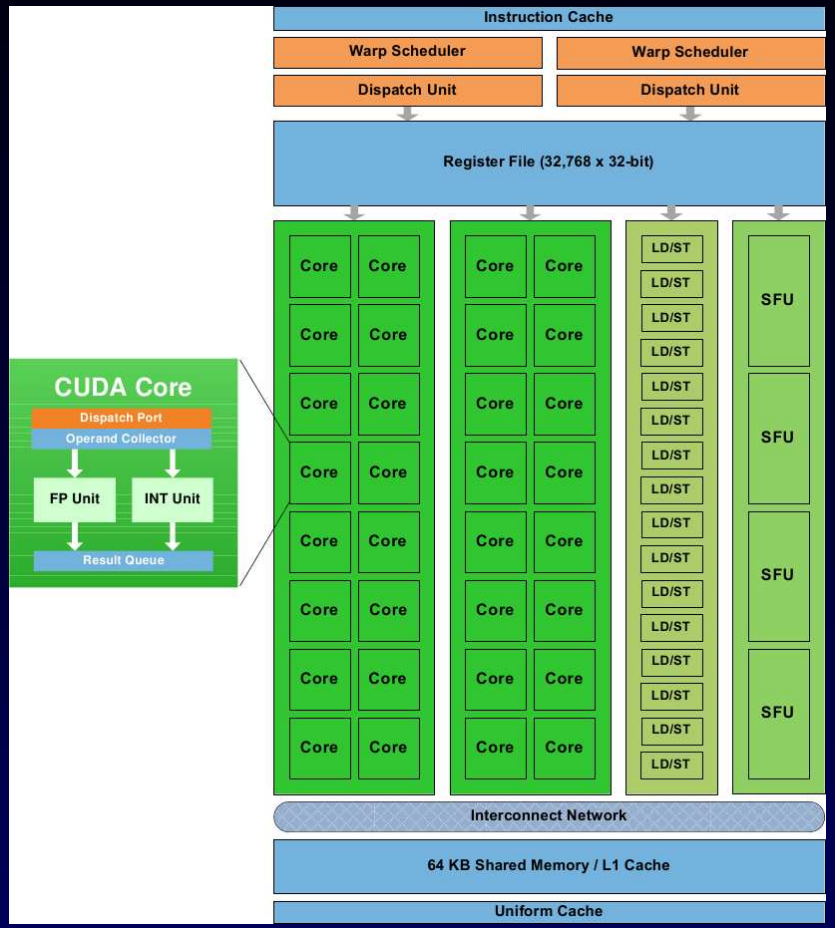
Radeon HD 6870 — big picture



Off-the-shelf nVidia Tesla Fermi

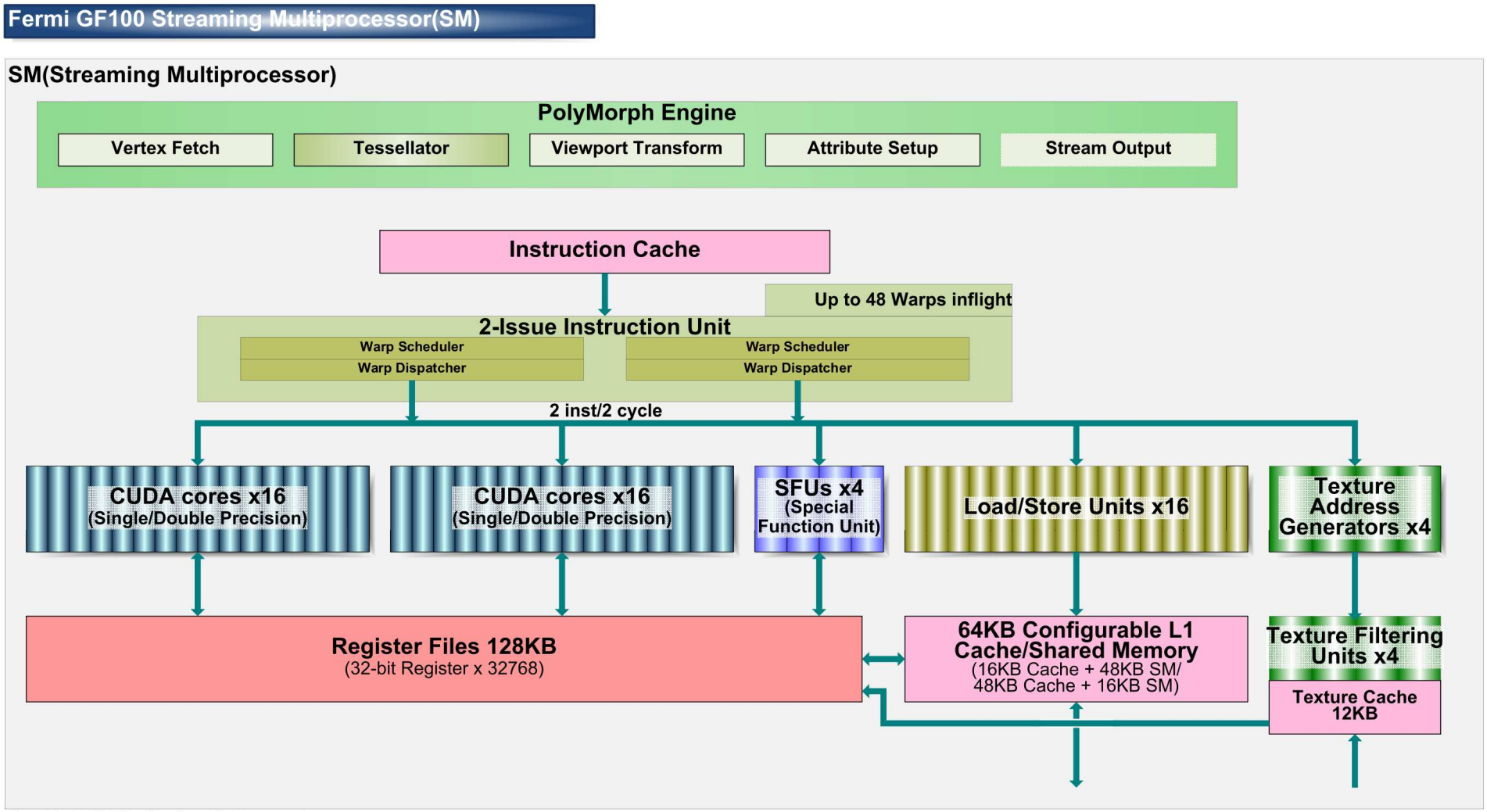
- 3 billion 40nm transistors
- 448 thread processors @ 1150 MHz, 1 TFLOPS SP, 0.5 TFLOPS DP
- + External 6 GB GDDR5 ECC memory 3 Gt/s, 144 GB/s. Less if using ECC





- 247 W on board PCI Express 2.1 x16 bus interface
- OpenGL, OpenCL, CUDA

GF100 Stream Multiprocessor

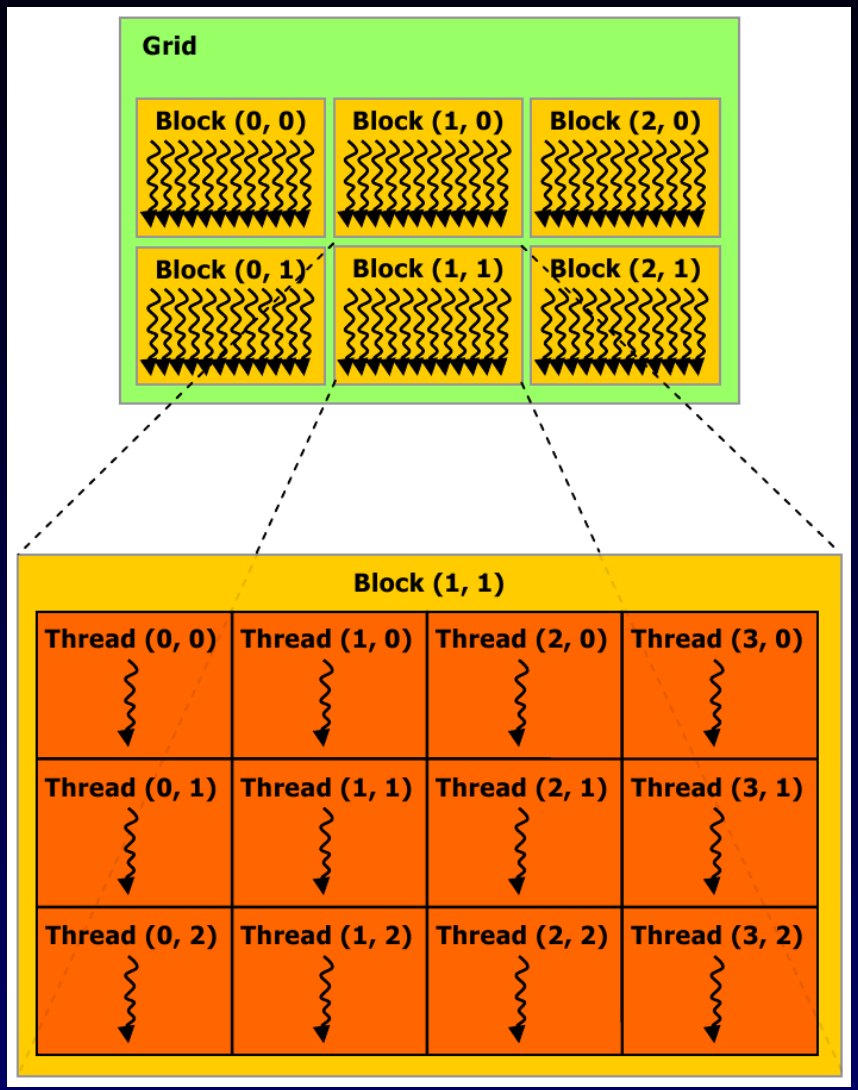


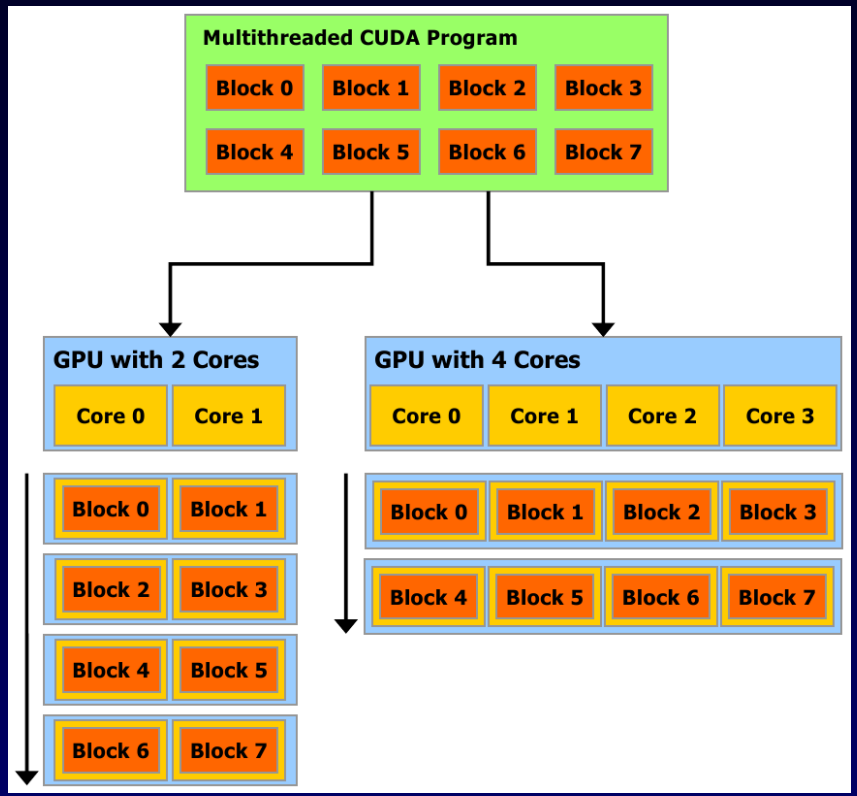
Basic GPU programming model

A sequential program on a host launches computational-intensive kernels on a GPU

- Allocate storage on the GPU
- Copy-in data from the host to the GPU
- Launch the kernel on the GPU
- The host waits...
- Copy-out the results from the GPU to the host
- Deallocate the storage on the GPU

GPU execution model





Outline

Extracting parallelism in applications...

- The implicit attitude
  - Hardware: massively superscalars processors
  - Software: auto-parallelizing compilers
- The ($\pm$) explicit attitude
  - Languages ($\pm$ extensions): OpenMP, UPC, HPF, Co-array Fortran (F- -), Fortran 2008, X10, Chapel, Fortress, Matlab, SciLab, Octave, Mapple, LabView, nVidia CUDA, AMD/ATI Stream (Brook+, Cal), OpenCL, HMPP, *insert your own preferred language here* ...
  - Libraries: application-oriented (mathematics, coupling...), parallelism (MPI, concurrency *pthreads*, SPE/MFC on Cell...), Multicore Association MCAPI, objects (parallel STL, TBB, Ct...)

... but multidimensional heterogeneity!

Welcome into Parallel Hard-Core Real Life 2.0!

- Heterogeneous execution models
  - Multicore SMP ± coupled by caches
  - SIMD instructions in processors (Neon, VMX, SSE4.2, 3DNow!, LRBni...)
  - Hardware accelerators (MIMD, MISD, SIMD, SIMT, FPGA...)
- New heterogeneous memory hierarchies
  - Classic caches/physical memory/disks
  - Flash SSD is a new-comer to play with
  - NUMA (*Non Uniform Memory Access*) : sockets-attached memory banks, remote nodes...
  - Peripherals attached to sockets : NUPA (*Non Uniform Peripheral Access*). GPU on PCIe $\times 16$ in this case...
  - If non-shared memory: remote memory, remote disks...
  - Inside GPU : registers, local memory, shared memory, constant memory, texture cache, processor grouping, locked physical pages, host memory access...
- Heterogeneous communications
  - Anisotropic networks
  - Various protocols

Several dimensions to cope with at the same time ☹

From hardware constraints to programming style (I)

- GPU computes fast but connected to CPU with slow PCI link ⇝
  - Avoid exchanging too much data between CPU & GPU or... compute on the CPU ☹
  - Possible to overlap communications with computations (more complex programing)
- Many SIMD engines (multiprocessors) ⇝ at least as much blocks of threads
- Memory hierarchy is quite complex and... visible!
  - Use (quite limited ☹) local registers by recycling local data
  - Memory is accessed in huge lines ⇝ program to use all the elements of the line
  - If not possible, try to reorganize data in the shared memory around read/write (matrix transposition...)
  - Recently added caches help too

From hardware constraints to programming style (II)

- Memory is far far away (800+ cycles) ⇝ use a lots of thread per block (but limited resources reduce block numbers) to overlap memory access with other computations
- Computing is fast, memory is slow. Rethink algorithms...

- SIMD machine, only one control flow ⇝ predicated

```
1  if (cond[i])
     b[i] = a[i];
3  else
     b[i] = -a[i] + 1;
```

- Some hardware optimizations if in a SIMD warp there is no execution ⇝ if possible sort `false`/`true` elements

Dwarfs d'applications parallèles

- `http://view.eecs.berkeley.edu/` : « The Landscape of Parallel Computing Research: A View From Berkeley »
- Essaye de capturer des exemples typiques courants qui peuvent servir à analyser et concevoir des architectures

	Dwarf	Performance Limit: Memory Bandwidth, Memory Latency, or Computation?
1	Dense Matrix	Computationally limited
2	Sparse Matrix	Currently 50% computation, 50% memory BW
3	Spectral (FFT)	Memory latency limited
4	N-Body	Computationally limited
5	Structured Grid	Currently more memory bandwidth limited
6	Unstructured Grid	Memory latency limited
7	MapReduce	Problem dependent
8	Combinational Logic	CRC problems BW; crypto problems computationally limited
9	Graph traversal	Memory latency limited
10	Dynamic Programming	Memory latency limited
11	Backtrack and Branch+Bound	?
12	Construct Graphical Models	?
13	Finite State Machine	Nothing helps!

Extraire du parallélisme (I)

Exemple de calcul de polynômes de vecteurs (livre « Initiation au parallélisme, concepts, architectures et algorithmes », Marc GENGLER, Stéphane UBÉDA & Frédéric DESPREZ)

pour $i = 0$ à $n - 1$ faire
$$vv[i] = a + b.v[i] + c.v[i]^2 + d.v[i]^3 + e.v[i]^4 + f.v[i]^5 + g.v[i]^6$$
fin pour

Calcul avec parallélisme de donnée (typique SIMD) ≡ faire en parallèle la même chose sur des données différentes :

pour $i = 0$ à $n - 1$ faire en parallèle
$$vv[i] = a + b.v[i] + c.v[i]^2 + d.v[i]^3 + e.v[i]^4 + f.v[i]^5 + g.v[i]^6$$
fin pour

Extraire du parallélisme *(II)*

Découpage en tâches (typique MIMD) ≡ faire des choses différentes sur des données différentes :

pour $i = 0$ à $n - 1$ faire
    tâches parallèles
        $x = a + b.v[i] + c.v[i]^2 + d.v[i]^3$
    ||
        $y = e. + f.v[i] + g.v[i]^2$
    ||
        $z = v[i]^4$
    fin tâches parallèles
    $vv[i] = x + z.y$
fin pour

Extraire du parallélisme *(III)*

Pipeline (typique systolique) $\equiv$ travail à la chaîne :

$$\begin{array}{r} v[n-1],\ldots,v[1],v[0] \rightarrow \\ 0,\ldots,0,0 \rightarrow \end{array} \boxed{\begin{array}{cc} x & x \\ y & g+xy \end{array}} \rightarrow \boxed{\begin{array}{cc} x & x \\ y & f+xy \end{array}} \rightarrow \cdots \rightarrow \boxed{\begin{array}{cc} x & x \\ y & a+xy \end{array}} \rightarrow$$

7 étages de pipeline (7 processeurs) traitant 1 flux de plusieurs données.

Type de parallélisme (I)

Parallélisme de données :

- Régularité des données
- Même calcul à des données distinctes

Parallélisme de contrôle :

- Fait des choses différentes

Parallélisme de flux : pipeline

- Régularité des données
- Chaque donnée subit séquence de traitements

Réingénierie pour le parallélisme

« Reengineering for Parallelism: An Entry Point into PLPP (Pattern Language for Parallel Programming) for Legacy Applications », Berna L. Massingill, Timothy G. Mattson, Beverly A. Sanders

`http://www.cise.ufl.edu/research/ParallelPatterns/plop2005.pdf`

- Guide de recettes pratiques lorsqu'on part d'un vieux programme...
- ... ou qu'on trouve que cela aide de concevoir d'abord un programme séquentiel (mais  ...)
- 4 espaces de conception à traverser en partant du problème, contexte et des utilisateurs
  - « Trouver de la concurrence » (*Finding Concurrency*)
  - « Structure algorithmique » (*Algorithm Structure*)
  - « Structure de support » (*Supporting Structures*)
  - « Mécanismes d'implémentation » (*Implementation Mechanisms*)

Espace de conception « trouver concurrence » (I)

- Permet de structurer problème pour exposer concurrence exploitable
- Travail niveau algorithmique de haut niveau pour exposer concurrence potentielle
- Quelques patrons de conceptions possibles dans cet espace :
  - Patrons de décomposition de problèmes en morceaux concurrents
    - Décomposition en tâches : comment un problème peut-il être décomposé en tâches qui s'exécutent de manière concurrente ?
    - Décomposition des données : comment décomposer données du problème en unités qui peuvent être traitée relativement indépendamment ?

Espace de conception « trouver concurrence » *(II)*

- Patrons d'analyse de dépendances : regroupent tâches et analyse leur dépendances
  - Groupe des tâches : comment regrouper les tâches d'un problème pour simplifier la gestion de leur dépendances ?
  - Ordonnancement des tâches : comment ordonner des groupes de tâches (provenant d'une décomposition d'un problème et d'un regroupement des tâches) pour satisfaire les inter-dépendances ?
  - Partage des données : Comment à partir d'une décomposition des données et en tâches partager des données entre tâches ?
- Évaluation de la conception : est-ce que les résultats de la phase de décomposition et d'analyse de dépendances est suffisamment bonne pour passer à l'espace de conception suivant (structure algorithmique) ou est-ce qu'on réitère conception dans cet espace ?

Espace de conception « structure algorithmique »

(I)

- Restructuration des algorithmes pour exploiter la concurrence potentielle obtenue dans l'espace précédent
- Patrons possibles de stratégies pour exploiter la concurrence :
  - Patrons pour applications centrées sur une organisation en tâche
    - Parallélisme de tâche : comment organiser un algorithme en une collection de tâches à exécution concurrente ?
    - Diviser pour régner : comment exploiter concurrence potentielle dans le cas d'un problème formulé avec stratégie « diviser pour régner » ?
  - Patrons pour applications centrées sur organisation par décomposition de données
    - Décomposition géométrique : comment organiser un algorithme autour de structures de données mises à jour par morceau de manière concurrente ?
    - Données récursives : dans le cas d'opérations sur une structure de donnée récursive (liste, arbre, graphe...) qui semblent séquentielles, comment réaliser ces opérations en parallèle ?

Espace de conception « structure algorithmique » *(II)*

- Patrons pour applications orientées flot de données
  - Pipeline : si application peut être vue comme un flux de données à travers une série d'étapes de calcul, comment exploiter cette concurrence ?
  - Coordination par événements : si application décomposée en groupe de tâches semi-indépendantes interagissant de manière irrégulière dépendant des données (donc contraintes de dépendances entre tâches aussi...), comment réaliser cette interaction pour avoir du parallélisme ?

Espace de conception « Structure de support » (I)

- Étape intermédiaire entre description algorithmique et implémentation
- Traite de la programmation mais en restant à haut niveau
- Exemples de patrons de conception :
  - Patrons représentant les approches structurant les programmes :
    - SPMD : problèmes liés aux interaction de différentes unités d'exécution. Comment structurer programmes pour gérer au mieux interactions et faciliter intégration dans programme global ?
    - Ferme de travail: Comment organiser un programme conçu avec un besoin de distribuer de manière dynamique du travail à des travailleurs ?
    - Parallélisme de boucles : comment traduire en programme parallèle un programme séquentiel dominé par de gros nids de boucles
    - Fork/Join : Si programme avec nombre de tâches concurrentes qui varient avec relations complexes entre elles, comment construire programme parallèle avec tâches dynamiques ?

Espace de conception « Structure de support » *(II)*

- Patrons représentant des structures de données courantes :
  - Données partagées : comment gérer explicitement des données partagées entre différentes tâches concurrentes ?
  - Files partagées : comment partager de manière correcte une structure de file entre différentes unités d'exécution ?
  - Tableaux distribués. Souvent, tableaux partitionnés sur plusieurs unités d'exécution. Comment faire un programme efficace et... lisible ?
- À ce niveau est aussi discuté d'autres structures comme SIMD, MPMD, client-serveur, langages parallèles déclaratifs, environnements de résolution de problèmes...

Espace de conception « Mécanismes d'implémentatio (I)

- Traite de l'adaptation des espaces de conception de haut niveau à des environnements de programmation particuliers
- Souvent correspondance directe entre choix de cet espace et élément de l'environnement de programmation cible
- Exemple de patrons
  - Gestion des unités d'exécution : parallélisme implique plusieurs entités fonctionnant simultanément qui doivent être gérées (création et destruction de processus lourds ou légers...)
  - Synchronisation : permet de respecter des contraintes d'ordonnancement d'événements sur différences unités d'exécutions (barrière, exclusion mutuelle, barrière mémoire...)
  - Communication : si pas de mémoire partagée, besoin de communications explicites pour échanger des informations entre processus

Cycle de développement

1. Analyse de complexité du problème
2. Analyse du programme disponible
   - Analyse performances, profiling (`gprof`, VTune...)
3. Conception de la parallélisation
   - Bibliothèques déjà optimisées (mathématique)
   - Objets parallèles (STL parallèles, TBB...)
   - Langages parallèles (OpenMP, UPC, Fortran 2008...)
   - Bibliothèques parallèles (MPI, threads systèmes...)
4. Mise au point
   - Tests de non régression ( non associativité flottant)
   - Débogage (`gdb`, TotalView...)
   - Correction concurrence (Intel Thread Checker, Helgrind)
5. Optimisation
   - Profiling : Intel Thread Profiler, `gprof`, VTune...

La nouvelle donne du renouveau informatique (I)

« The Landscape of Parallel Computing Research: A View from Berkeley », Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick. EECS Department University of California, Berkeley Technical Report UCB/EECS-2006-183, December 18, 2006

`http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf`

Révision des bonnes vieilles hypothèses (*conventional wisdoms*)...

1. Old CW: Power is free, but transistors are expensive.
   New CW is the “Power wall”: Power is expensive, but transistors are “free”. That is, we can put more transistors on a chip than we have the power to turn on.

La nouvelle donne du renouveau informatique *(II)*

2. Old CW: If you worry about power, the only concern is dynamic power.
   New CW: For desktops and servers, static power due to leakage can be 40% of total power.
3. Old CW: Monolithic uniprocessors in silicon are reliable internally, with errors occurring only at the pins.
   New CW: As chips drop below 65 nm feature sizes, they will have high soft and hard error rates. [Borkar 2005] [Mukherjee et al 2005]
4. Old CW: By building upon prior successes, we can continue to raise the level of abstraction and hence the size of hardware designs.
   New CW: Wire delay, noise, cross coupling (capacitive and inductive), manufacturing variability, reliability (see above), clock jitter, design validation, and so on conspire to stretch the

La nouvelle donne du renouveau informatique *(III)*

development time and cost of large designs at 65 nm or smaller feature sizes.

5. Old CW: Researchers demonstrate new architecture ideas by building chips.
   New CW: The cost of masks at 65 nm feature size, the cost of Electronic Computer Aided Design software to design such chips, and the cost of design for GHz clock rates means researchers can no longer build believable prototypes. Thus, an alternative approach to evaluating architectures must be developed.
6. Old CW: Performance improvements yield both lower latency and higher bandwidth.
   New CW: Across many technologies, bandwidth improves by at least the square of the improvement in latency. [Patterson 2004]

La nouvelle donne du renouveau informatique *(IV)*

7. Old CW: Multiply is slow, but load and store is fast.
   New CW is the “Memory wall” [Wulf and McKee 1995]: Load and store is slow, but multiply is fast. Modern microprocessors can take 200 clocks to access Dynamic Random Access Memory (DRAM), but even floating-point multiplies may take only four clock cycles.
8. Old CW: We can reveal more instruction-level parallelism (ILP) via compilers and architecture innovation. Examples from the past include branch prediction, out-of-order execution, speculation, and Very Long Instruction Word systems.
   New CW is the “ILP wall”: There are diminishing returns on finding more ILP. [Hennessy and Patterson 2007]

La nouvelle donne du renouveau informatique *(V)*

9. Old CW: Uniprocessor performance doubles every 18 months.
   New CW is Power Wall + Memory Wall + ILP Wall = Brick Wall.
   Figure 2 plots processor performance for almost 30 years. In 2006, performance is a factor of three below the traditional doubling every 18 months that we enjoyed between 1986 and 2002. The doubling of uniprocessor performance may now take 5 years.
10. Old CW: Don't bother parallelizing your application, as you can just wait a little while and run it on a much faster sequential computer.
    New CW: It will be a very long wait for a faster sequential computer (see above).
11. Old CW: Increasing clock frequency is the primary method of improving processor performance.
    New CW: Increasing parallelism is the primary method of improving processor performance.

La nouvelle donne du renouveau informatique *(VI)*

12 Old CW: Less than linear scaling for a multiprocessor application is failure.
New CW: Given the switch to parallel computing, any speedup via parallelism is a success.

Outline

Outline

Programmation CUDA (I)

- Data-parallel extension to a C++ subset
- Target nVidia GPU and *x*86 multicores
- 2-level parallelism: threads in blocks of threads + block-tiling
- In a block of threads : communication through shared memory and synchronization via `__syncthreads()`
- Complex heterogeneous memory layout (GPU...)

```
1  __global__ void
2  add_matrix_gpu(float *a, float *b, float *c, int N) {
     int i=blockIdx.x*blockDim.x+threadIdx.x;
4    int j=blockIdx.y*blockDim.y+threadIdx.y;
     int index =i+j*N;
6
     if( i < N && j < N)
8      c[index]=a[index]+b[index];
   }
10
   void main() {
12   float ha[N][N], hb[N][N], hc[N][N];
```

Programmation CUDA (II)

```
   /* Allocate array on the GPU with cudaMalloc */
14 float *a, *b, *c;
   cudaMalloc((void **) &a, sizeof(float)*N*N);
16 cudaMalloc((void **) &b, sizeof(float)*N*N);
   cudaMalloc((void **) &c, sizeof(float)*N*N);
18
   cudaMemcpy(a, ha, sizeof(float)*N*N, cudaMemcpyHostToDevice);
20 cudaMemcpy(b, hb, sizeof(float)*N*N, cudaMemcpyHostToDevice);

22 // Describe iteration tiling (2D strip-mining)
   dim3 dimBlock (blocksize,blocksize);
24 dim3 dimGrid (N/dimBlock.x,N/dimBlock.y);
   add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
26 cudaMemcpy(c, hc, sizeof(float)*N*N, cudaMemcpyDeviceToHost);
}
```

- Need some heavy code restructuring
- ∃ other version: CUDA driver, similar to OpenCL

OpenCL (I)

- Language based on a C_{99} subset
- Started by Apple to *unify* parallel use (multicores, GPGPU...)
  ⇝ similar to OpenGL & OpenAL
- Followed by AMD/ATI and nVidia
- Data-parallelism and control-parallelism (1–3-dimensions) according to targets
- Kernel oriented computations on streams
- Complex split memory model (GPGPU...)
- New types (vectors, images...)

OpenCL (II)

```
/* This kernel computes FFT of length 1024.
   The 1024 length FFT is decomposed into calls to a radix 16
   function, another radix 16 function and then a radix 4 function */

__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                          __local float *sMemx, __local float *sMemy) {
  int tid = get_local_id(0);
  int blockIdx = get_group_id(0) * 1024 + tid;
  float2 data[16];
  // starting index of data to/from global memory
  in = in + blockIdx;  out = out + blockIdx;
  globalLoads(data, in, 64); // coalesced global reads
  fftRadix16Pass(data);      // in-place radix-16 pass
  twiddleFactorMul(data, tid, 1024, 0);
  // local shuffle using local memory
  localShuffle(data, sMemx, sMemy, tid,
               (((tid & 15) * 65) + (tid >> 4)));
  fftRadix16Pass(data);                  // in-place radix-16 pass
  twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication
  localShuffle(data, sMemx, sMemy, tid,
               (((tid >> 4) * 64) + (tid & 15)));
  // four radix-4 function calls
  fftRadix4Pass(data); fftRadix4Pass(data + 4);
  fftRadix4Pass(data + 8); fftRadix4Pass(data + 12);
```

OpenCL (III)

```
25      // coalesced global writes
        globalStores(data, out, 64);
27  }
    [...]
29  // create a compute context with GPU device
    context = clCreateContextFromType(CL_DEVICE_TYPE_GPU);
31  // create a work-queue
    queue = clCreateWorkQueue(context, NULL, NULL, 0);
33  // allocate the buffer memory objects
    memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
35  CL_MEM_COPY_HOST_PTR,
                                sizeof(float)*2*num_entries, srcA);
37  memobjs[1] = clCreateBuffer(context, CL_MEM_READ_WRITE,
                                sizeof(float)*2*num_entries, NULL);
39  // create the compute program
    program = clCreateProgramFromSource(context, 1,
41                                      &fft1D_1024_kernel_src, NULL);
    // build the compute program executable
43  clBuildProgramExecutable(program, false, NULL, NULL);
    // create the compute kernel
45  kernel = clCreateKernel(program, "fft1D_1024");
    // create N-D range object with work-item dimensions
47  global_work_size[0] = n;
    local_work_size[0] = 64;
```

OpenCL (IV)

```
49  range = clCreateNDRangeContainer(context, 0, 1,
                                     global_work_size, local_work_size);
51  // set the args values
    clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem), NULL);
53  clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem), NULL);
    clSetKernelArg(kernel, 2, NULL,
55                 sizeof(float)*(local_work_size[0]+1)*16, NULL);
    clSetKernelArg(kernel, 3, NULL,
57                 sizeof(float)*(local_work_size[0]+1)*16, NULL);
     // execute kernel
59  clExecuteKernel(queue, kernel, NULL, range, NULL, 0, NULL);
```

- Need a lot of code restructuring

CUDA or OpenCL?

- Both language are small extension to C-like language
- CUDA
  - Appeared first
  - Language basis not well defined: not C indeed but C++!
  - nVidia GPU only
  - Rather limited to 2D threads
- OpenCL
  - Standard backed by many companies
  - C99 based
  - 3D threads with less constraints
  - More verbose API (kernel call...)
  - Kernel source code outside of host source ☹

Take advantage of C99 (I)

Write C99impler C99ode, easier to parallelize automatiC99ally...

- BaC99k to C99imple C99ings
- Avoiding using C99umbersome old C C99onstruC99ts C99an lead to C99leaner C99ode, more effiC99ient and more parallelizable C99ode (with Par4All...)
- C99 adds C99ympaC99etiC99 features that are unfortunately not well known:
  - MultidimenC99ional arrays with non-statiC99 size
    - Avoid `malloc()` spam and useless pointer C99onstruC99ions
    - You C99an have arrays with a dynamiC99 size in funC99tion parameters (as in Fortran)
    - Avoid useless linearizing C99omputaC99ions (`a[i][j]` instead of `a[i+n*j]`...)
      Avoid non-affine constructs that are hard to analyze for parallelization, communications...
    - Avoid most of `alloca()`

Take advantage of C99 *(II)*

- TLS (*Thread-Local Storage*) ExtenC$_{99}$ion C$_{99}$an express independent storage
- C$_{99}$omplex numbers and booleans avoid further struC$_{99}$tures or enums

⇝ C$_{99}$ is good for you!

Bad/good C programming example

A programmer want a dynamically allocated 2-d array with dynamic size

- Bad

```
1  int n, m;
2  double * t =
     malloc(sizeof(double)*n*m);
4  for(int i = 0; i < n; i++)
     for(int j = 0; j < m; m++)
6      t[i*n + j] = i + j;
   free(t);
```

- Good:

```
1  double (*t)[n][m] =
     malloc(sizeof(double (*)[n][m]));
3  for(int i = 0; i < n; i++)
     for(int j = 0; j < m; m++)
5      (*t)[i][j] = i + j;
   free(t);
```

- Good:

```
1  {
2    double t[n][m];
     for(int i = 0; i < n; i++)
4      for(int j = 0; j < m; m++)
         t[i][j] = i + j;
6  }
```

↝ Before parallelization, good sequential programming...

CUDA or OpenCL? Part 2

- CUDA
  - It is C++ like, rather C89 and not C99
  - Painful to translate C99 to C89 and keeping clean sources...
  - C++ polymorphism adds a mess
    - `log(2.0)` can be called from inside a kernel
    - `log(2)` is an integer log function that does not have a kernel implementation (bug?), so is a host function that cannot be called from a kernel ☹
  - Should have good C support at least before light cosmetic C++ support
  - ⇝ Dead end from language point of view
- OpenCL
  - C99 based ⇝ clean
  - Some standard macro wrappers will appear such as our Par4All accel runtime

Outline

Use the Source, Luke...

Hardware is moving quite (too) fast but...

What has survived for 50+ years?

Fortran programs...

What has survived for 40+ years?

IDL, Matlab, Scilab...

What has survived for 30+ years?

C programs, Unix...

- A lot of legacy code could be pushed onto parallel hardware (accelerators) with automatic tools...
- Need automatic tools for source-to-source transformation to leverage existing software tools for a given hardware
- Not as efficient as hand-tuned programs, but quick production phase

Outline

We need software tools

- HPC Project needs tools for its hardware accelerators (*Wild Nodes* from *Wild Systems*) and to parallelize, port & optimize customer applications
- Application development: long-term business $\rightsquigarrow$ long-term commitment in a tool that needs to survive to (too fast) technology change

Not reinventing the wheel... No NIH syndrome please

Want to create your own tool?

- House-keeping and infrastructure in a compiler is a huge task
- Unreasonable to begin yet another new compiler project...
- Many academic Open Source projects are available...
- ...But customers need products ☺
- ⇝ Integrate your ideas and developments in existing project
- ...or buy one if you can afford (ST with PGI...) ☺
- Some projects to consider
  - Old projects: gcc, PIPS... and many dead ones (SUIF...)
  - But new ones appear too: LLVM, RoseCompiler, Cetus...

Par4All

- ⇝ Funding an initiative to industrialize Open Source tools
- PIPS is the first project to enter the Par4All initiative

`http://www.par4all.org`

PIPS

(/)

- PIPS (Interprocedural Parallelizer of Scientific Programs): Open Source project from Mines ParisTech... 23-year old! ☺
- Funded by many people (French DoD, Industry & Research Departments, University, CEA, IFP, Onera, ANR (French NSF), European projects, regional research clusters...)
- One of the project that coined polytope model-based compilation
- ≈ 456 KLOC according to David A. Wheeler's `SLOCCount`
- ... but modular and sensible approach to pass through the years
  - ≈300 phases (parsers, analyzers, transformations, optimizers, parallelizers, code generators, pretty-printers...) that can be combined for the right purpose
  - Polytope lattice (sparse linear algebra) used for semantics analysis, transformations, code generation... to deal with big programs, not only loop-nests

PIPS (II)

- NewGen object description language for language-agnostic automatic generation of methods, persistence, object introspection, visitors, accessors, constructors, XML marshaling for interfacing with external tools...
- Interprocedural *à la* `make` engine to chain the phases as needed. Lazy construction of resources
- On-going efforts to extend the semantics analysis for C

- Around 15 programmers currently developing in PIPS (Mines ParisTech, HPC Project, IT SudParis, TÉLÉCOM Bretagne, RPI) with public `svn`, Trac, `git`, mailing lists, IRC, Plone, Skype... and use it for many projects
- But still...
  - Huge need of documentation (even if PIPS uses literate programming...)
  - Need of industrialization
  - Need further communication to increase community size

Current PIPS usage

- Automatic parallelization (Par4All C & Fortran to OpenMP)
- Distributed memory computing with OpenMP-to-MPI translation [STEP project]
- Generic vectorization for SIMD instructions (SSE, VMX, Neon, CUDA, OpenCL...) (SAC project) [SCALOPES]
- Parallelization for embedded systems [SCALOPES]
- Compilation for hardware accelerators (Ter@PIX, SPoC, SIMD, FPGA...) [FREIA, SCALOPES]
- High-level hardware accelerators synthesis generation for FPGA [PHRASE, CoMap]
- Reverse engineering & decompiler (reconstruction from binary to C)
- Genetic algorithm-based optimization [Luxembourg university+TB]
- Code instrumentation for performance measures
- GPU with CUDA & OpenCL [TransMedi@, FREIA, OpenGPU]

Outline

Challenges in automatic GPU code generation

- Find parallel kernels
- Improve data reuse inside kernels to have better compute intensity (even if the memory bandwidth is quite higher than on a CPU...)
- Access the memory in a GPU-friendly way (to coalesce memory accesses)
- Take advantage of complex memory hierarchy that make the GPU fast (shared memory, cached texture memory, registers...)
- Reduce the copy-in and copy-out transfers that pile up on the PCIe
- Reduce memory usage in the GPU (no swap there, yet...)
- Limit inter-block synchronizations
- Overlap computations and GPU-CPU transfers (via streams)

Basic GPU execution model (bis)

A sequential program on a host launches computational-intensive kernels on a GPU

- Allocate storage on the GPU
- Copy-in data from the host to the GPU
- Launch the kernel on the GPU
- The host waits...
- Copy-out the results from the GPU to the host
- Deallocate the storage on the GPU

Automatic parallelization

Most fundamental for a parallel execution

Finding parallelism!

Several parallelization algorithms are available in PIPS

- For example classical Allen & Kennedy use loop distribution more vector-oriented than kernel-oriented  (or need later loop-fusion)
- Coarse grain parallelization based on the independence of array regions used by different loop iterations
  - Currently used because generates GPU-friendly coarse-grain parallelism
  - Accept complex control code without *if-conversion*

Outlining (I)

Parallel code $\rightsquigarrow$ Kernel code on GPU

- Need to extract parallel source code into kernel source code: outlining of parallel loop-nests
- Before:

```
1     for(i = 1;i <= 499; i++)
2       for(j = 1;j <= 499; j++) {
          save[i][j] = 0.25*(space[i - 1][j] + space[i + 1][j]
4                            + space[i][j - 1] + space[i][j + 1]);
        }
```

Outlining (II)

- After:

```
1  p4a_kernel_launcher_0(space, save);
   [...]
3  void p4a_kernel_launcher_0(float_t space[SIZE][SIZE],
                              float_t save[SIZE][SIZE]) {
5    for(i = 1; i <= 499; i += 1)
       for(j = 1; j <= 499; j += 1)
7        p4a_kernel_0(i, j, save, space);
   }
9  [...]
   void p4a_kernel_0(float_t space[SIZE][SIZE],
11                   float_t save[SIZE][SIZE],
                     int i,
13                   int j) {
     save[i][j] = 0.25*(space[i-1][j]+space[i+1][j]
15                      +space[i][j-1]+space[i][j+1]);
   }
```

From array regions to GPU memory allocation (I)

- Memory accesses are summed up for each statement as *regions* for array accesses: integer polytope lattice
- There are regions for write access and regions for read access
- The regions can be exact if PIPS can prove that only these points are accessed, or they can be inexact, if PIPS can only find an over-approximation of what is really accessed

From array regions to GPU memory allocation (II)

Example

```
1 for(i = 0; i <= n-1; i += 1)
2   for(j = i; j <= n-1; j += 1)
      h_A[i][j] = 1;
```

can be decorated by PIPS with write array regions as:

```
1 // <h_A[PHI1][PHI2]-W-EXACT-{0<=PHI1, PHI2+1<=n, PHI1<=PHI2}>
    for(i = 0; i <= n-1; i += 1)
3 // <h_A[PHI1][PHI2]-W-EXACT-{PHI1==i, i<=PHI2, PHI2+1<=n, 0<=i}>
      for(j = i; j <= n-1; j += 1)
5 // <h_A[PHI1][PHI2]-W-EXACT-{PHI1==i, PHI2==j, 0<=i, i<=j, 1+j<=n}>
        h_A[i][j] = 1;
```

- These read/write regions for a kernel are used to allocate with a `cudaMalloc()` in the host code the memory used inside a kernel and to deallocate it later with a `cudaFree()`

Communication generation (I)

Conservative approach to generate communications

- Associate any GPU memory allocation with a copy-in to keep its value in sync with the host code
- Associate any GPU memory deallocation with a copy-out to keep the host code in sync with the updated values on the GPU

-  But a kernel could use an array as a local (private) array
- ...PIPS does have many privatization phases ☺
-  But a kernel could initialize an array, or use the initial values without writing into it or use/touch only a part of it or...

Communication generation (II)

More subtle approach

PIPS gives 2 very interesting region types for this purpose

- In-region abstracts what really needed by a statement
- Out-region abstracts what really produced by a statement to be used later elsewhere

- In-Out regions can directly be translated with CUDA into
  - copy-in

```
1  cudaMemcpy(accel_address, host_address,
2             size, cudaMemcpyHostToDevice)
```

  - copy-out

```
1  cudaMemcpy(host_address, accel_address,
2             size, cudaMemcpyDeviceToHost)
```

Loop normalization

- Hardware accelerators use fixed iteration space (thread index starting from 0...)
- Parallel loops: more general iteration space
- Loop normalization

Before

```
1 for(i = 1;i < SIZE - 1; i++)
2   for(j = 1;j < SIZE - 1; j++) {
      save[i][j] = 0.25*(space[i - 1][j] + space[i + 1][j]
4                        + space[i][j - 1] + space[i][j + 1]);
    }
```

After

```
1 for(i = 0;i < SIZE - 2; i++)
    for(j = 0;j < SIZE - 2; j++) {
3     save[i+1][j+1] = 0.25*(space[i][j + 1] + space[i + 2][j + 1]
                           + space[i + 1][j] + space[i + 1][j + 2]);
5   }
```

From preconditions to iteration clamping (I)

- Parallel loop nests are compiled into a CUDA kernel wrapper launch
- The kernel wrapper itself gets its virtual processor index with some `blockIdx.x*blockDim.x + threadIdx.x`
- Since only full blocks of threads are executed, if the number of iterations in a given dimension is not a multiple of the `blockDim`, there are incomplete blocks ☹
- An incomplete block means that some index overrun occurs if all the threads of the block are executed 

From preconditions to iteration clamping *(II)*

- So we need to generate code such as

```
1  void p4a_kernel_wrapper_0(int k, int l,...)
2  {
     k = blockIdx.x*blockDim.x + threadIdx.x;
4    l = blockIdx.y*blockDim.y + threadIdx.y;
     if (k >= 0 && k <= M - 1 && l >= 0 && l <= M - 1)
6      kernel(k, l, ...);
   }
```

  But how to insert these guards?

- The good news is that PIPS owns *preconditions* that are predicates on integer variables. Preconditions at entry of the kernel are:

```
1  //  P(i,j,k,l) {0<=k, k<=63, 0<=l, l<=63}
```

- Guard $\equiv$ directly translation in C of preconditions on loop indices that are GPU thread indices

Complexity analysis (I)

- Launching a GPU kernel is expensive
  - so we need to launch only kernels with a significant speed-up (launching overhead, memory CPU-GPU copy overhead...)
- Some systems use `#pragma` to give a go/no-go information to parallel execution

```
1  #pragma omp parallel if(size>100)
```

- ∃ phase in PIPS to symbolically estimate complexity of statements
- Based on preconditions
- Use a SuperSparc2 model from the '90s... ☺
- Can be changed, but precise enough to have a coarse go/no-go information
- To be refined: use memory usage complexity to have information about memory reuse (even a big kernel could be more efficient on a CPU if there is a good cache use)

Optimized reduction generation

- Reduction are common patterns that need special care to be correctly parallelized

$$s = \sum_{i=0}^{N} x_i$$

- Reduction detection already implemented in PIPS
- Efficient computation on GPU needs to create local reduction trees in the thread-blocks
  - Use existing libraries but may need several kernels?
  - Inline reduction code?
- Not yet implemented in Par4All

Communication optimization

- Naive approach : load/compute/store
- Useless communications if a data on GPU is not used on host between 2 kernels... ☹
- ⇝ Use static interprocedural data-flow communications
  - ▶ Fuse various GPU arrays : remove GPU (de)allocation
  - ▶ Remove redundant communications

⇝ New `p4a --com-optimization` option

Fortran to C-based GPU languages

- Fortran 77 parser available in PIPS
- CUDA & OpenCL are C++/C99 with some restrictions on the GPU-executed parts
- Need a Fortran to C translator (`f2c`...)?
- Only one internal representation is used in PIPS
  - Use the Fortran parser
  - Use the... C pretty-printer!
- But the IO Fortran library is complex to use... and to translate
  - If you have IO instructions in a Fortran loop-nest, it is not parallelized anyway because of sequential side effects ☺
  - So keep the Fortran output everywhere but in the parallel CUDA kernels
  - Apply a memory access transposition phase `a(i,j)` ⇝ `a[j-1][i-1]` inside the kernels to be pretty-printed as C
- Compile and link C GPU kernel parts + Fortran main parts
- Quite harder than expected... Use Fortran 2003 for C interfaces...

Par4All accel runtime (I)

- CUDA or OpenCL can not be directly represented in the internal representation (IR, abstract syntax tree) such as `__device__` or `<<< >>>`
- PIPS motto: keep the IR as simple as possible
- Use some calls to intrinsics functions that can be represented directly
- Intrinsics functions are implemented with (macro-)functions
  - `p4a_accel.h` has indeed currently 2 implementations
    - `p4a_accel-CUDA.h` than can be compiled with CUDA for nVidia GPU execution or emulation on CPU
    - `p4a_accel-OpenMP.h` that can be compiled with an OpenMP compiler for simulation on a (multicore) CPU
- Add CUDA support for complex numbers

Par4All accel runtime *(II)*

- On-going support of OpenCL written in C/CPP/C++
- Can be used to simplify manual programming too (OpenCL...)
  - Manual radar electromagnetic simulation code @TB
- OpenMP emulation for almost free
  - Use Valgrind to debug GPU-like and communication code !

Big picture — p4a-generated code (I)

```
1  int main(int argc, char *argv[]) {
[...]
3    float_t (*p4a_var_space)[SIZE][SIZE];
     P4A_accel_malloc(&p4a_var_space, sizeof(space));
5    P4A_copy_to_accel(space, p4a_var_space, sizeof(space));

7    float_t (*p4a_var_save)[SIZE][SIZE];
     P4A_accel_malloc(&p4a_var_save, sizeof(save));
9    P4A_copy_to_accel(save, p4a_var_save, sizeof(save));

11   for(t = 0; t < T; t++)
       compute(*p4a_var_space, *p4a_var_save);

13   P4A_copy_from_accel(space, p4a_var_space, sizeof(space));

15   P4A_accel_free(p4a_var_space);
     P4A_accel_free(p4a_var_save);
17 [...]
19 }
   void compute(float_t space[SIZE][SIZE],
21              float_t save[SIZE][SIZE]) {
[...]
23    p4a_kernel_launcher_0(space, save);
[...]
```

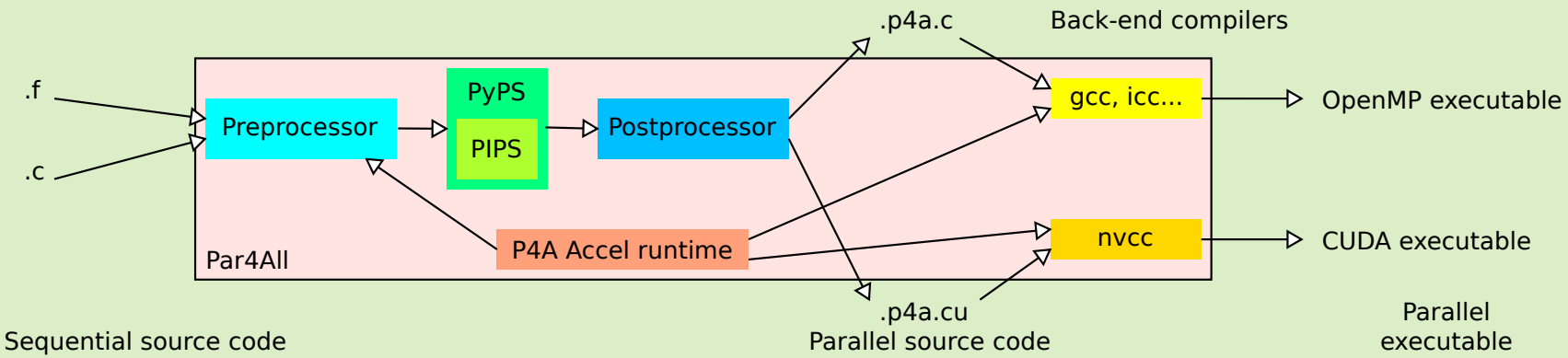
Big picture — p4a-generated code *(II)*

```
25 }
   void p4a_kernel_launcher_0(float_t space[SIZE][SIZE],
27                            float_t save[SIZE][SIZE]) {
     P4A_call_accel_kernel_2d(p4a_kernel_wrapper_0, SIZE, SIZE,
29                            space, save);
   }
31 P4A_accel_kernel_wrapper void
   p4a_kernel_wrapper_0(float_t space[SIZE][SIZE],
33                      float_t save[SIZE][SIZE]) {
     int j;
35   int i;
     i = P4A_vp_0;
37   j = P4A_vp_1;
     if (i >= 1 && i <= SIZE - 1 && j >= 1 && j <= SIZE - 1)
39     p4a_kernel_0(space, save, i, j);
   }
41 P4A_accel_kernel void p4a_kernel_0(float_t space[SIZE][SIZE],
                                      float_t save[SIZE][SIZE],
43                                    int i,
                                      int j) {
45   save[i][j] = 0.25*(space[i-1][j]+space[i+1][j]
                      +space[i][j-1]+space[i][j+1]);
47 }
```

Par4All ≡ PyPS scripting in the backstage (I)

- Up to now PIPS was scripted with a special shell-like language: `tpips`
- Not enough powerful (not a programming language)
- Add a SWIG Python interface to PIPS phases and interface
  - All the power of a wide-spread real language
  - Add introspection in the compiling phase
  - Easy to add any glue, pre-/post-processing to generate target code

Overview



Par4All ≡ PyPS scripting in the backstage (II)

⇝ p4a script as simple as

- `p4a --openmp toto.c -o toto`
- `p4a --cuda toto.c -o toto`

- p4a sample code

```
1   def parallelize(self, fine = False, filter_select = None, filter_exclude = None):
        all_modules = self.filter_modules(filter_select, filter_exclude)
3
        # Try to privatize all the scalar variables in loops:
5       all_modules.privatize_module()

7       if fine:
            # Use a fine-grain parallelization à la Allen & Kennedy:
9           all_modules.internalize_parallel_code()
        else:
11          # Use a coarse-grain parallelization with regions:
            all_modules.coarse_grain_parallelization()
13
    def gpuify(self, filter_select = None, filter_exclude = None):
15      all_modules = self.filter_modules(filter_select, filter_exclude)

17      # First, only generate the launchers to work on them later. They are
        # generated by outlining all the parallel loops:
19      all_modules.gpu_ify(GPU_USE_WRAPPER = False,
                            GPU_USE_KERNEL = False,
21                          concurrent=True)
```

Par4All ≡ PyPS scripting in the backstage (III)

```
23      # Select kernel launchers by using the fact that all the generated
        # functions have their names beginning with "p4a_kernel_launcher":
25      kernel_launcher_filter_re = re.compile("p4a_kernel_launcher_.*[^!]$")
        kernel_launchers = self.workspace.filter(lambda m: kernel_launcher_filter_re.match(m.name))
27
        # Normalize all loops in kernels to suit hardware iteration spaces:
29      kernel_launchers.loop_normalize(
            # Loop normalize for the C language and GPU friendly
31          LOOP_NORMALIZE_ONE_INCREMENT = True,
            # Arrays start at 0 in C, so the iteration loops:
33          LOOP_NORMALIZE_LOWER_BOUND = 0,
            # It is legal in the following by construction (..Hmmm to verify)
35          LOOP_NORMALIZE_SKIP_INDEX_SIDE_EFFECT = True,
            concurrent=True)
37
        # Unfortunately the information about parallelization and
39      # privatization is lost by the current outliner, so rebuild
        # it... :-( But anyway, since we've normalized the code, we
41      # changed it so it is to be parallelized again...
        #kernel_launchers.privatize_module()
43      kernel_launchers.capply("privatize_module")
        #kernel_launchers.coarse_grain_parallelization()
45      kernel_launchers.capply("coarse_grain_parallelization")

47      # In CUDA there is a limitation on 2D grids of thread blocks, in
        # OpenCL there is a 3D limitation, so limit parallelism at 2D
49      # top-level loops inside parallel loop nests:
        kernel_launchers.limit_nested_parallelism(NESTED_PARALLELISM_THRESHOLD = 2, concurrent=True)
51
        #kernel_launchers.localize_declaration()
53      # Does not work:
```

Par4All ≡ PyPS scripting in the backstage *(IV)*

```
        #kernel_launchers.omp_merge_pragma()
55
57      # Add iteration space decorations and insert iteration clamping
        # into the launchers onto the outer parallel loop nests:
59      kernel_launchers.gpu_loop_nest_annotate(concurrent=True)

61      # End to generate the wrappers and kernel contents, but not the
        # launchers that have already been generated:
63      kernel_launchers.gpu_ify(GPU_USE_LAUNCHER = False,
                                 concurrent=True)
65
        # Add communication around all the call site of the kernels:
67      kernel_launchers.kernel_load_store(concurrent=True)

69      # Select kernels by using the fact that all the generated kernels
        # have their names of this form:
71      kernel_filter_re = re.compile("p4a_kernel_\\d+$")
        kernels = self.workspace.filter(lambda m: kernel_filter_re.match(m.name))
73
        # Unfortunately CUDA 3.0 does not accept C99 array declarations
75      # with sizes also passed as parameters in kernels. So we degrade
        # the quality of the generated code by generating array
77      # declarations as pointers and by accessing them as
        # array[linearized expression]:
79      kernels.array_to_pointer(ARRAY_TO_POINTER_FLATTEN_ONLY = True,
                                 ARRAY_TO_POINTER_CONVERT_PARAMETERS = "POINTER")
```

Outline

Scilab language

- Interpreted scientific language widely used like Matlab
- Free software
- Roots in free version of Matlab from the 80's
- Dynamic typing (scalars, vectors, (hyper)matrices, strings...)
- Many scientific functions, graphics...
- Double precision everywhere, even for loop indices (now)
- Slow because everything decided at runtime, garbage collecting
  - Implicit loops around each vector expression
    - Huge memory bandwidth used
    - Cache thrashing
    - Redundant control flow
- Strong commitment to develop Scilab through Scilab Enterprise, backed by a big user community, INRIA...
- HPC Project WildNode appliance with Scilab parallelization
- Reuse Par4All infrastructure to parallelize the code

Scilab & Matlab (I)

- Scilab/Matlab input : *sequential* or array syntax
- Compilation to C code
- Parallelization of the generated C code
- Type inference to guess (crazy ☹) semantics
  - Heuristic: first encountered type is forever
- Speedup $>$ 1000 ☺
- WildCruncher: *x*86+GPU appliance with nice interface
  - Scilab — mathematical model & simulation
  - Par4All — automatic parallelization
  - //Geometry — polynomial-based 3D rendering & modelling

Outline

Hyantes (I)

- Geographical application: library to compute neighbourhood population potential with scale control
- Example given in `par4all.org` distribution
- WildNode with 2 Intel Xeon X5670 @ 2.93GHz (12 cores) and a nVidia Tesla C2050 (Fermi), Linux/Ubuntu 10.04, gcc 4.4.3, CUDA 3.1
  - Sequential execution time on CPU: 30.355s
  - OpenMP parallel execution time on CPUs: 3.859s, speed-up: 7.87
  - CUDA parallel execution time on GPU: 0.441s, speed-up: 68.8
- With single precision on a HP EliteBook 8730w laptop (with an Intel Core2 Extreme Q9300 @ 2.53GHz (4 cores) and a nVidia GPU Quadro FX 3700M, 16 multiprocessors, 128 cores, architecture 1.1) with Linux/Debian/sid, gcc 4.4.3, CUDA 3.1:
  - Sequential execution time on CPU: 38s
  - OpenMP parallel execution time on CPUs: 18.9s, speed-up: 2.01
  - CUDA parallel execution time on GPU: 1.57s, speed-up: 24.2

Hyantes (II)

Original main C kernel:

```
1  void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, data_t range,
2    town pt[rangex][rangey], town t[nb])
{
4      size_t i,j,k;

6      fprintf(stderr,"begin computation ...\n");

8      for(i=0;i<rangex;i++)
           for(j=0;j<rangey;j++) {
10             pt[i][j].latitude =(xmin+step*i)*180/M_PI;
               pt[i][j].longitude =(ymin+step*j)*180/M_PI;
12             pt[i][j].stock =0.;
               for(k=0;k<nb;k++) {
14                 data_t tmp = 6368.* acos(cos(xmin+step*i)*cos( t[k].latitude )
                       * cos((ymin+step*j)-t[k].longitude)
16                     + sin(xmin+step*i)*sin(t[k].latitude));
                   if( tmp < range )
18                     pt[i][j].stock += t[k].stock  / (1 + tmp) ;
               }
20         }
       fprintf(stderr,"end computation ...\n");
22 }
```

Hyantes (III)

Generated GPU code:

```
1  void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, data_t range,
2    town pt[290][299], town t[2878])
{
4     size_t i, j, k;
     //PIPS generated variable
6     town (*P_0)[2878] = (town (*)[2878]) 0, (*P_1)[290][299] = (town (*)[290][299]) 0;

8     fprintf(stderr, "begin computation ...\n");
     P4A_accel_malloc(&P_1, sizeof(town[290][299])-1+1);
10    P4A_accel_malloc(&P_0, sizeof(town[2878])-1+1);
     P4A_copy_to_accel(pt, *P_1, sizeof(town[290][299])-1+1);
12    P4A_copy_to_accel(t, *P_0, sizeof(town[2878])-1+1);

14    p4a_kernel_launcher_0(*P_1, range, step, *P_0, xmin, ymin);
     P4A_copy_from_accel(pt, *P_1, sizeof(town[290][299])-1+1);
16    P4A_accel_free(*P_1);
     P4A_accel_free(*P_0);
18    fprintf(stderr, "end computation ...\n");
}
20
void p4a_kernel_launcher_0(town pt[290][299], data_t range, data_t step, town t[2878],
22    data_t xmin, data_t ymin)
{
24    //PIPS generated variable
     size_t i, j, k;
26    P4A_call_accel_kernel_2d(p4a_kernel_wrapper_0, 290,299, i, j, pt, range,
                              step, t, xmin, ymin);
28 }

30 P4A_accel_kernel_wrapper void p4a_kernel_wrapper_0(size_t i, size_t j, town pt[290][299],
```

Hyantes (IV)

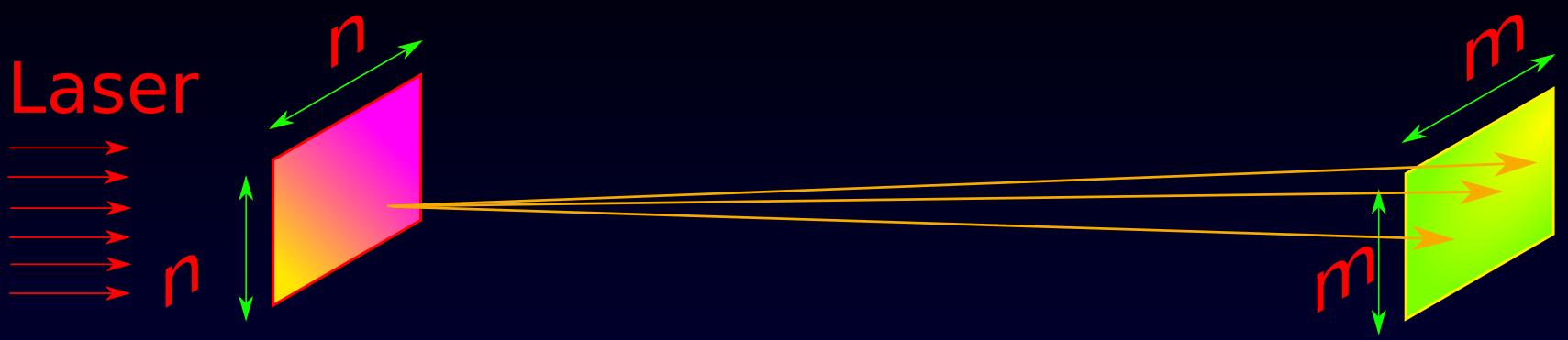
```
  data_t range, data_t step, town t[2878], data_t xmin, data_t ymin)
{
   // Index has been replaced by P4A_vp_0:
   i = P4A_vp_0;
   // Index has been replaced by P4A_vp_1:
   j = P4A_vp_1;
   // Loop nest P4A end
   p4a_kernel_0(i, j, &pt[0][0], range, step, &t[0], xmin, ymin);
}

P4A_accel_kernel void p4a_kernel_0(size_t i, size_t j, town *pt, data_t range,
  data_t step, town *t, data_t xmin, data_t ymin)
{
   //PIPS generated variable
   size_t k;
   // Loop nest P4A end
   if (i<=289&&j<=298) {
      pt[299*i+j].latitude = (xmin+step*i)*180/3.14159265358979323846;
      pt[299*i+j].longitude = (ymin+step*j)*180/3.14159265358979323846;
      pt[299*i+j].stock = 0.;
      for(k = 0; k <= 2877; k += 1) {
         data_t tmp = 6368.*acos(cos(xmin+step*i)*cos((*(t+k)).latitude)*cos(ymin+step*j
               -(*(t+k)).longitude)+sin(xmin+step*i)*sin((*(t+k)).latitude));
         if (tmp<range)
            pt[299*i+j].stock += t[k].stock/(1+tmp);
      }
   }
}
```

Stars-PM

- *Particle-Mesh* N-body cosmological simulation
- C code from Observatoire Astronomique de Strasbourg
- Use FFT 3D
- Example given in `par4all.org` distribution

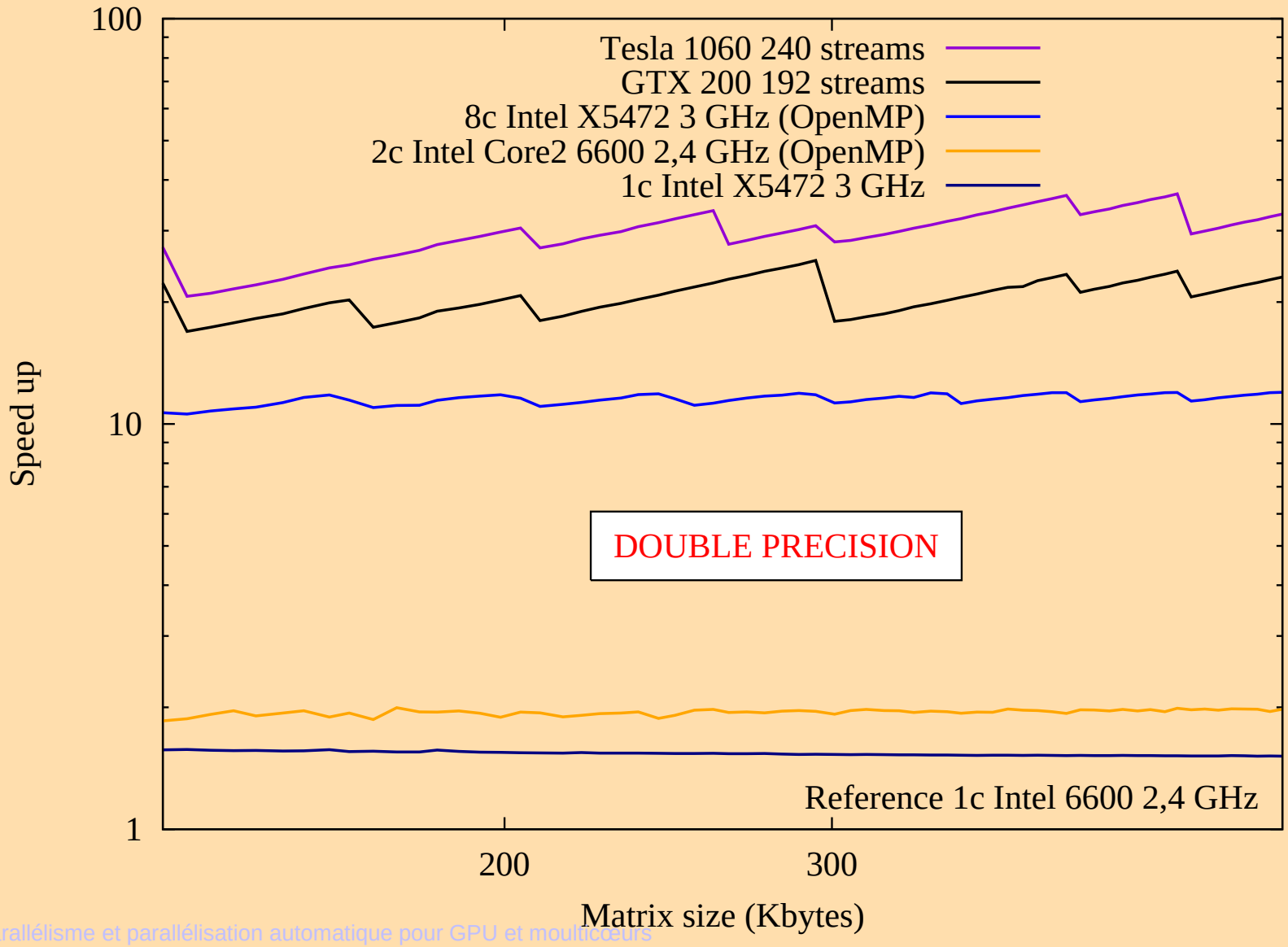
Results on a customer application



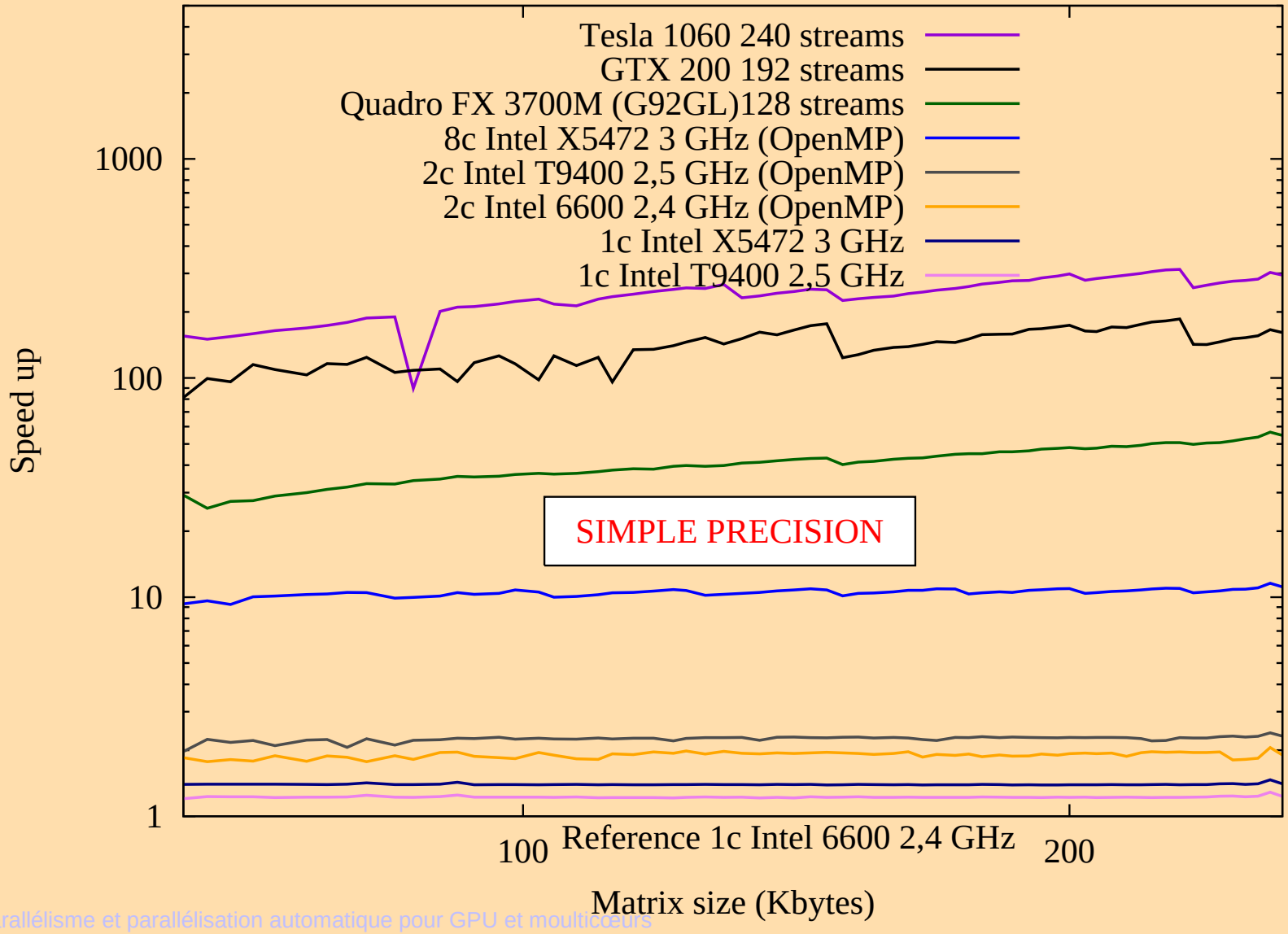
- Holotetrix's primary activities are the design, fabrication and commercialization of prototype diffractive optical elements (DOE) and micro-optics for diverse industrial applications such as LED illumination, laser beam shaping, wavefront analyzers, etc.
- Hologram verification with direct Fresnel simulation
- Program in C
- Parallelized with
  - Par4All CUDA and CUDA 2.3, Linux Ubuntu x86-64
  - Par4All OpenMP, gcc 4.3, Linux Ubuntu x86-64
- Reference: Intel Core2 6600 @ 2.40GHz

`http://www.holotetrix.com`

Comparative performance



Keep it simple (precision)



Stars-PM time step

```
1  void iteration(coord pos[NP][NP][NP],
2                 coord vel[NP][NP][NP],
                  float dens[NP][NP][NP],
4                 int data[NP][NP][NP],
                  int histo[NP][NP][NP]) {
6    /* Découpe l'espace tridimensionnel
        selon une grille régulière */
8  discretisation(pos, data);
     /* Calcul de la densité sur la grille */
10 histogram(data, histo);
     /* Calcul du potentiel sur la grille
12      dans l'espace de Fourier */
   potential(histo, dens);
14   /* Calcul dans chaque dimension de la force
        et application à la vitesse des particules */
16 forcex(dens, force);
   updatevel(vel, force, data, 0, dt);
18 forcey(dens, force);
   updatevel(vel, force, data, 1, dt);
20 forcez(dens, force);
   updatevel(vel, force, data, 2, dt);
22   /* Déplacement des particules */
   updatepos(pos, vel);
24 }
```

Stars-PM & Jacobi results with `p4a 1.0.5` *(I)*

- 2 Xeon Nehalem X5670 (12 cores @ 2,93 GHz)
- 1 GPU nVidia Tesla C2050
- Automatic call to CuFFT instead FFTW
- 150 iterations of Stars-PM

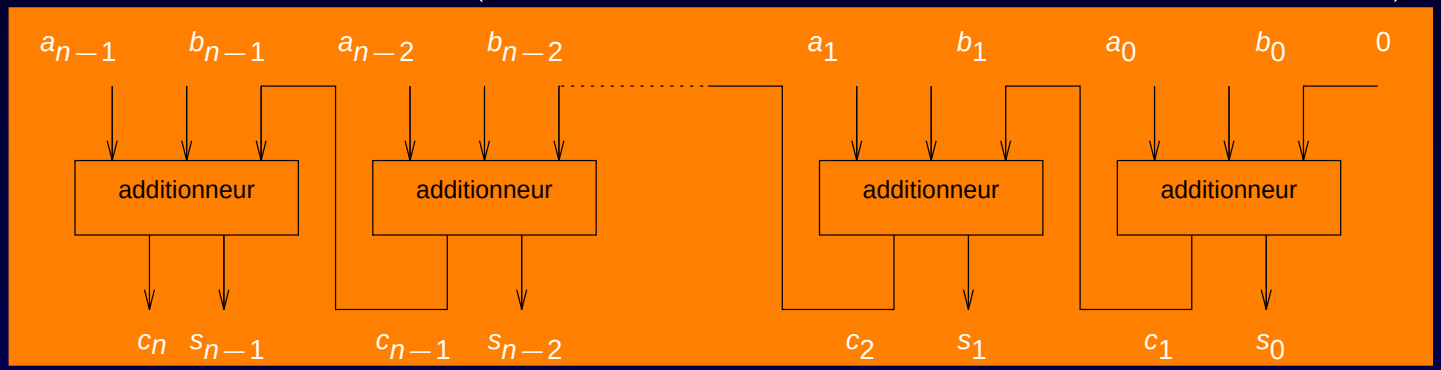
Temps d'exécution	`p4a`	Simulation Cosmo. 32^3	Simulation Cosmo. 64^3	Simulation Cosmo. 128^3	Jacobi
Séquentiel	(`gcc -O3`)	0,68	6,30	98,4	24,5
OpenMP 6 *threads*	`--openmp`	0,16	1,28	16,6	13,8
CUDA base	`--cuda`	0,88	5,21	31,4	67,7
Comm. optimisées	`--cuda --com-opt.`	0,20	1,17	8,9	6,5
Optimisation manuelles	(`gcc -O3`)	0,05	0,26	1,7	

Current limitation for Stars-PM with `p4a`: histogram is not parallelized...

Outline

Parallélisme opérateur : addition entière *(I)*

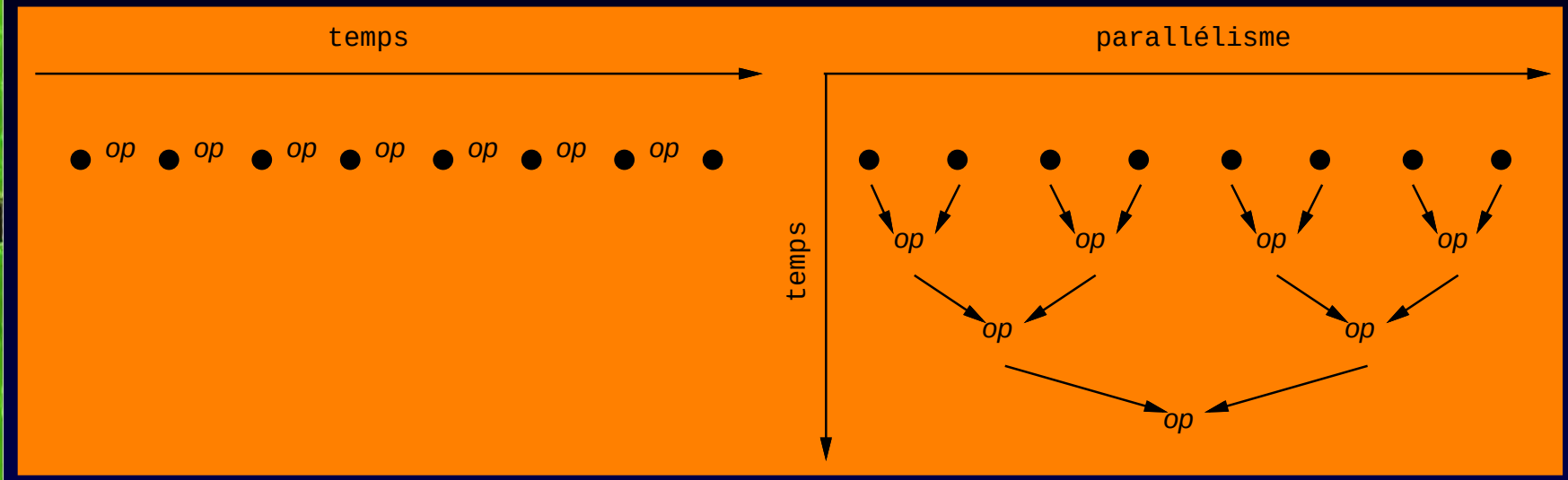
Additionneur 1 bit : $(c, s) = \text{add}_1(a, b) = (a \wedge b, a \oplus b)$ (4 transistors)
Additionneur complet (3 entrées) :
$(c_o, s) = \text{add}(a, b, c_i) = \big((a \wedge b) \vee (a \wedge c_i) \vee (b \wedge c_i), a \oplus b \oplus c_i\big)$



Temps et complexité en $\mathcal{O}(n)$ pour additionneur n bits
Motivation des ▸ Opérations parallèles préfixes

Réduction (I)

Réduction $r = \bigoplus_{i=1}^{n} a_i$



	Séquentiel	Parallèle
Temps	$n-1$	$\lceil \log_2 n \rceil$
Opérateurs	1	$\lfloor \frac{n}{2} \rfloor$ ou $n-1$
Efficacité	1	$\frac{n-1}{\lfloor \frac{n}{2} \rfloor \lceil \log_2 n \rceil}$ ou $\frac{1}{\lceil \log_2 n \rceil}$

Parallélisme : gâcher utile !

Environments with reductions

Available in various languages and libraries

- APL (1962 !): $+/$, $*/$, $\lceil/$, $\lfloor/$...
- Scilab & Matlab: `sum`, `prod`...
- MPI: `MPI_SUM`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`...
- Fortran: `SUM`, `PRODUCT`, `MINVAL`, `MAXVAL`...
- OpenMP

```
1       #pragma omp parallel for reduction(+:sum)
2       for(i = 0; i <= 99; i += 1)
           sum += i+k;
```

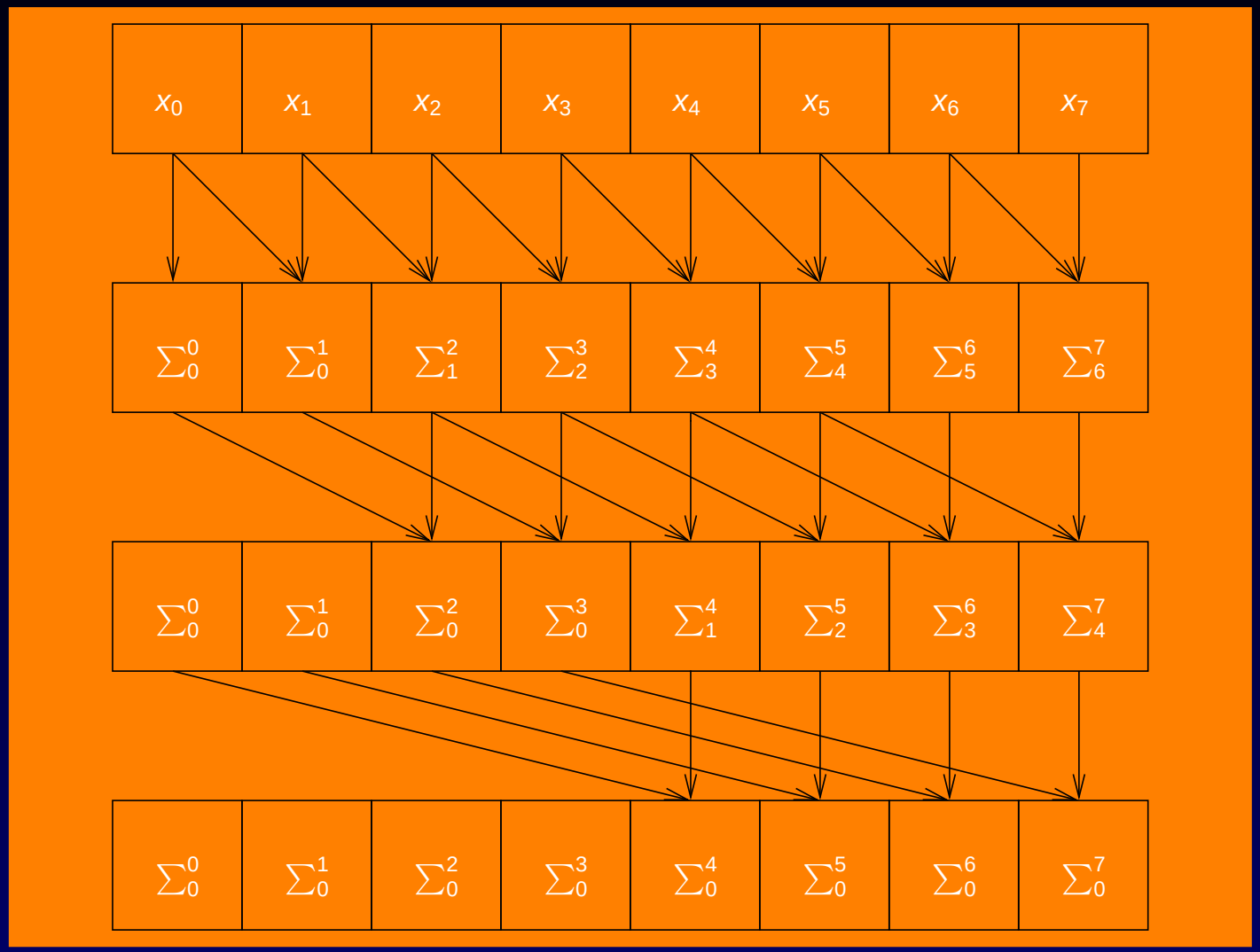
- C++ TBB `tbb::parallel_reduce()`
- Libraries for GPU: CuPP...

Opération préfixe parallèle (scan) *(I)*

Autre algorithmique parallèle classique (ILLIAC IV, 1968) :

$$\forall i \in [0, n-1], \quad S_i = \bigoplus_{j=0}^{i} x_j$$

Opération préfixe parallèle (scan) *(II)*



Opération préfixe parallèle (scan) *(III)*

Si réutilisation des opérateurs :

	Séquentiel	Parallèle
Temps	$n-1$	$\lceil \log_2 n \rceil$
Opérateurs	1	$n-1$
Efficacité	1	$\frac{1}{\lceil \log_2 n \rceil}$

non associativité des opérations flottantes... Changement de l'ordre d'évaluation $\rightsquigarrow$ changement du résultat

Environments with parallel prefix/suffix scans

Available in various languages and libraries

- APL (1962 !): $+\backslash$, $*\backslash$, $\lceil\backslash$, $\lfloor\backslash$...
- Scilab & Matlab: `cumsum`, `cumprod`...
- MPI: `MPI_SCAN`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`
- Fortran: `SUM_PREFIX`/`SUM_SUFFIX`, `PRODUCT_PREFIX`/`PRODUCT_SUFFIX`, `MINVAL_PREFIX`/`MINVAL_SUFFIX`, `MAXVAL_PREFIX`/`MAXVAL_SUFFIX`...
- C++ TBB `tbb::parallel_scan()`
- Libraries for GPU: CuDPP `http://code.google.com/p/cudpp`

suffix : begin at the end, reverse order

Parallel prefix variants

- Direction: prefix (left-to-right) or suffix (right-to-left)
- Can exclude the local element or not from the computation
  - $\text{scan}(+, (1, 2, 3, 4)) = (1, 3, 6, 10)$
  - $\text{scan}_{\text{excl}}(+, (1, 2, 3, 4)) = (0, 1, 3, 6)$
- Segmentation

x	1	2	3	4	5	6	7	8
s	?	t	f	t	t	f	f	t
$\text{scan}_{\text{seg}}(+, v, s)$	1	2	5	4	5	11	18	8

Additionneur *carry-lookahead* (I)

Retenue = facteur limitant $\rightsquigarrow$ changements de variable :

$$\left.\begin{array}{r} g_i = a_i b_i \\ p_i = a_i \vee b_i \end{array}\right\} \Longrightarrow c_{i+1} = g_i \vee p_i c_i$$

- si g_i est vrai alors $c_i + 1$ l'est : *génération* de la retenue
- si p_i est vrai alors si c_i est vrai elle est *propagée* à c_{i+1}.

$$\begin{aligned} c_{i+1} \quad = \quad & g_i \vee p_i g_{i-1} \vee p_i p_{i-1} g_{i-2} \vee p_i p_{i-1} p_{i-2} g_{i-3} \\ & \vee \cdots \vee p_i p_{i-1} \cdots p_1 g_0 \vee p_i p_{i-1} \cdots p_1 p_0 c_0 \end{aligned}$$

Appliquer une opération parallèle préfixe : temps en $\mathcal{O}(\log n)$ et espace en $\mathcal{O}(n \log n)$.
Autres méthodes : *carry skip*, *carry select*,...
Soustraction : inverser une des entrée ($\overline{a}$ ou $\overline{b}$) et $c_0 = 1$

Compressing a vector (I)

- If lot of 0, useless to compute or store trivial values
- Store and compute useful values only: sparse representation & computations

Compress x into c according to validity v

x	42	?	7	8	?	?	3	?
v	1	0	1	1	0	0	1	0
$\text{scan}_{\text{excl}}(+, v)$	0	0	1	2	2	2	3	3

```
1  s = scan_add_exclude(x);
2  if (v[i])
       c[s[i]] = x[i];
```

c	42	7	8	3	?	?	?	?

Computing FIBONACCI suite (I)

$$u_{n+2} = u_{n+1} + u_n$$
$$u_1 = 1$$
$$u_0 = 0$$

- Naive recursion: $\mathcal{O}(2^n)$
- Naive recursion with memoization or loop: $\mathcal{O}(n)$
- Reduce the recursion distance

$$\begin{pmatrix} u_{n+2} \\ u_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} u_{n+1} \\ u_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

- Matrix exponentiation after diagonalization $\rightsquigarrow$ gold number appears, use KNUTH exponentiation algorithm $\mathcal{O}(\log n)$

Computing FIBONACCI suite *(II)*

- Inspect:

$$\begin{pmatrix} a+c & b+d \\ a & b \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Identify:

$$\begin{pmatrix} u_{n+2}+u_{n+1}=u_{n+3} & u_{n+1}+u_n=u_{n+2} \\ u_{n+2} & u_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} u_{n+2} & u_{n+1} \\ u_{n+1} & u_n \end{pmatrix}$$

Recursion:

$$\begin{pmatrix} u_{n+1} & u_n \\ u_n & u_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

⇝ Apply matrix multiplication reduction and keep the upper left element: $\mathcal{O}(\log n)$ without diagonalization and floating point error computation!

Parsing a regular language (I)

« Update to "data parallel algorithms" », W. Daniel Hillis & Guy L. Steele, Jr. Communications of the ACM, Volume 30 Issue 1, Jan. 1987

How to parse `if x<=n then print("x = ", x);` into

`if` | `x` | `<=` | `n` | `then` | `print` | `(` | `"x = "` | `,` | `x` | `)` | `;` ?

Use finite-state automaton with transition from current state to new one according to character class

- N: initial state
- A: start of alphabetic token
- Z: continuation of alphabetic token
- *: single-special-character token
- <: < or > character
- =: = following < or >
- Q: double quote starting a string

Parsing a regular language *(II)*

- S: character within a string
- E: double quote ending a string

Old state	Character read										
	A	...	Z	+	-	*	<	>	=	"	space/new line
N	A	...	A	*	*	*	<	<	*	Q	N
A	Z	...	Z	*	*	*	<	<	*	Q	N
Z	Z	...	Z	*	*	*	<	<	*	Q	N
*	A	...	A	*	*	*	<	<	*	Q	N
<	A	...	A	*	*	*	<	<	=	Q	N
=	A	...	A	*	*	*	<	<	*	Q	N
Q	S	...	S	S	S	S	S	S	S	E	S
S	S	...	S	S	S	S	S	S	S	E	S
E	A	...	A	*	*	*	<	<	*	S	N

Parsing a regular language *(III)*

- Consider characters as function mapping an automaton state into an other one ☺: NY is character Y applied to state N and produce state A
- N`x<=y` = A`<=y` = `<=y` = `=y` = A
- The composition operation is associative...
- The character function can be represented by an array indexed by state with produced state as value
- Perform parallel prefix with combining functions of the string
- Use initial automaton state to index into *all* these arrays: every character has been replaced by the state the automaton would have after that character

Can be generalized to any function that can be reasonably represented with a look-up-table

CUDA CuDPP

Libraries for GPU: CuDPP `http://code.google.com/p/cudpp`

- RadixSort
- Array compaction
- Scan with C++ template operator
- Segmented scan
- Sparse matrix-vector multiply

Use plans *à la* FFTW

Multiplication entière (I)

Méthode moyenâgeuse avec table de carrés (mémoire morte, place en $\mathcal{O}(n)$ au lieu de $\mathcal{O}(n^2)$) :

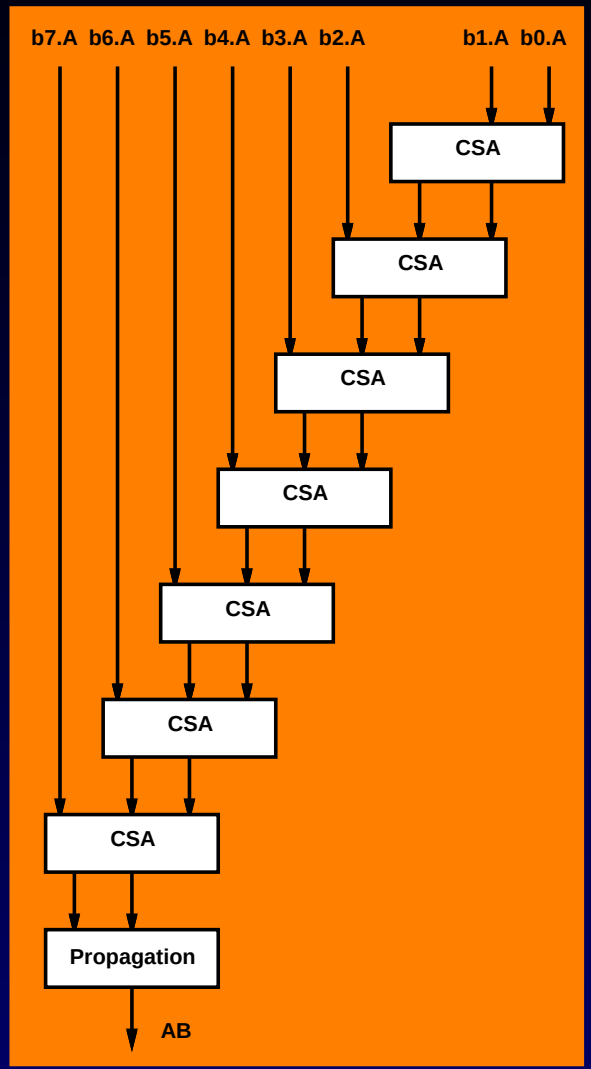
$$a \times b = \frac{(a+b)^2 - a^2 - b^2}{2}$$

Bonne vieille méthode manuelle : n additions :

- Sauter les bits à 0 (temps de multiplication non constant)
- Propagation de la retenue lente $\rightsquigarrow$ ne pas propager et garder toutes les retenues (*Carry Save Adder*) et propager lors de la dernière addition
  Utilisation d'un codage redondant ($2 \times n$ bits, chiffres dans 0–3) sauf pour le dernier résultat :

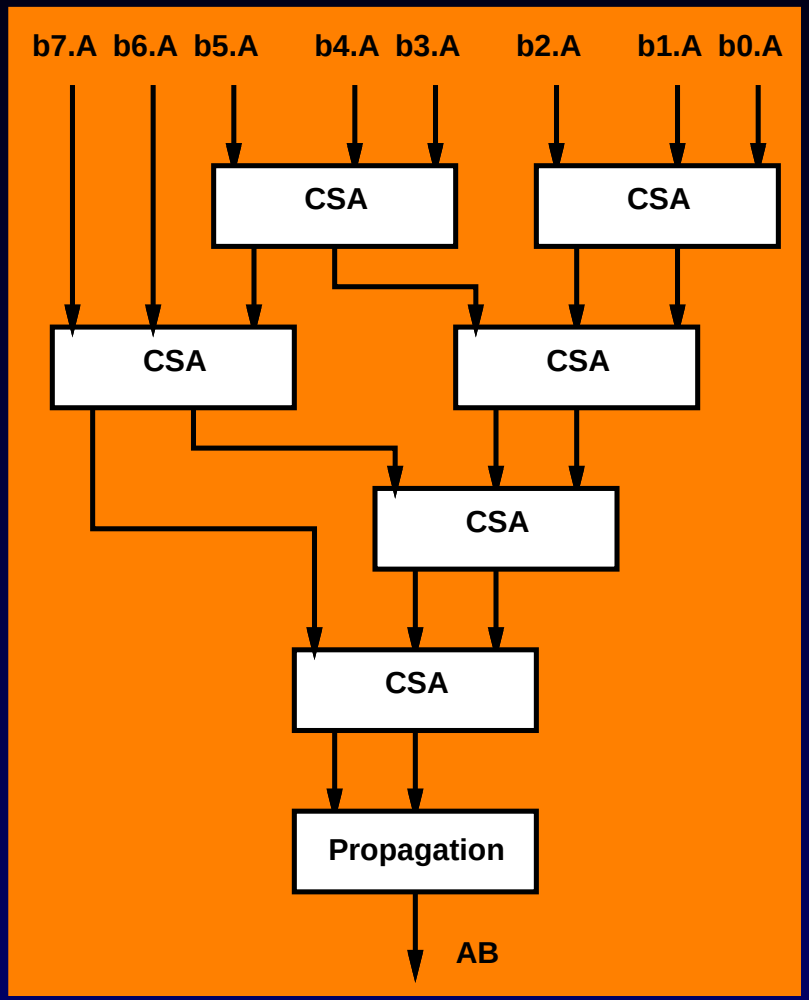
$$(s_i, c'_i) = (\lfloor (a_i + b_i + c_i)/2 \rfloor, \lfloor (a_i + b_i + c_i) \mod 2 \rfloor)$$

Multiplication entière (II)



Multiplication par arbre de Wallace

Réduction ⇝ arbre :



Retour à l'addition entière

Outline

Nombres flottants (I)

- Besoin de représenter des valeurs plus « continues » et plus de dynamique que types entiers
- Sous-ensemble de $\mathbb{D}(\subset \mathbb{R})$
- Représentation souvent au format IEEE 754-1985

$$f = (-1)^{\mathcal{S}} \times \mathcal{M} \times 2^{\mathcal{E}}$$

  - $\mathcal{S}$: bit de signe
  - $\mathcal{M}$: mantisse (entier positif)
  - $\mathcal{E}$: exposant
- Plusieurs tailles de flottants
  - Simple précision (**float**)
  - Double précision (**double**)
  - Précision étendue (**long double**)

Nombres flottants (II)

- Codage en mémoire

Format	Taille en bit			
	Totale	Signe	Exposant	Mantisse
Simple	32	1	8	24
Double	64	1	11	53
Étendu	≥ 80	1	≥ 15	≥ 64

  - $1 + 8 + 24 = 33 \neq 32$: voir plus tard...
  - Exposant codé en biaisé : $\mathcal{E}_{\text{réel}} = \mathcal{E}_{\text{stocké}} - \mathcal{E}_{\text{biais}}$
  - Tri lexicographique sur bits compatible avec tri flottant ! Même si on ne gère pas le flottant on sait trier ☺

Format	Minimum en dénormalisé	Minimum en normalisé	Maximum fini	2^{-N} (grain)	Chiffres significatifs
Simple	$1,4 \cdot 10^{-45}$	$1,2 \cdot 10^{-38}$	$3,4 \cdot 10^{38}$	$5,96 \cdot 10^{-8}$	6–9
Double	$4,9 \cdot 10^{-324}$	$2,2 \cdot 10^{-308}$	$1,8 \cdot 10^{308}$	$1,11 \cdot 10^{-16}$	15–17
Étendu	$\leq 3,6 \cdot 10^{-4951}$	$\leq 3,4 \cdot 10^{-4932}$	$\geq 1,2 \cdot 10^{4932}$	$\leq 5,42 \cdot 10^{-20}$	$\geq$ 18–21

Nombreux paramètres définis dans `<float.h>`

Nombres flottants *(III)*

- Possibilité de déclencher exceptions (division par 0, débordement,...) (fonction exécutée sur événement)
- Rajout de quantités symboliques (déclarées dans `<math.h>`)
  - $+0$ et -0. Néanmoins $+0 = -0$ est vrai
  - $+\infty$ et $-\infty$ (par exemple $\frac{1}{+0}$ et $\frac{1}{-0}$) (`HUGE_VAL`...)
  - NaN (*Not a Number*) pour $\frac{0}{0}$ ou $\sqrt{-1}$
    Seul cas ou $x \neq x$ lorsque x vaut NaN. Existe en signé et en version déclenchant exception (SNaN)

  $\rightsquigarrow$ Peuvent simplifier programmation et calcul si bien géré (éviter tests cas particuliers...)
- $\exists$ Nombreux choix d'arrondi (plus proche, +, -, vers 0,...)
- `http://grouper.ieee.org/groups/754` En cours de révision `http://en.wikipedia.org/wiki/IEEE_754r` si vous voulez participer ☺

Plein de subtilités !

Conversion flottants→entiers à l'arrache (I)

- En temps normal `int i = (int) f` doit faire le travail
- ∃ nombreuses fonctions de conversion et d'arrondi dans la bibliothèque mathématique (`rint()`, `round()`...)

La bibliothèque du C permet de le faire tout seul mais pas toujours disponible (cf. microcontrôleur Coupe de Robotique 2008). Pour hackers :

```
1  int ftoi(float f)
2  {
       // Récupère les bits du flottant dans un entier :
4      uint32_t dw = *((uint32_t *) &f);
       // La valeur spéciale où tous les bits sont à 0 code 0 :
6      if (dw == 0)
           return 0;
8      // Récupère l'exposant codé sur 8 bits et compense le biais :
       char exp = (dw >> 23) - 127; // Suppose un char de 8 bits
10     if (exp < 0 || exp > 23)
           /* Si l'exposant est négatif, de toute manière, le nombre vaut
12            moins que 1, donc arrondi à 0. Si c'est supérieur à 23, le nombr
              est supérieur à 2^{24} en on décide de le jeter et de répondre
```

Conversion flottants→entiers à l'arrache *(II)*

```
14              arbitrairement 0. Mmmm... On pourrait gérer jusqu'à 2^{32} mais
                faudrait corriger le code ci-après */
16          return 0;
        /* Construit le nombre avec le 1 de poids fort qui est économisé dans
18          la norme IEEE-754, puis les 23 autres bits de la mantisse cadrés
            en fonction de l'exposant : */
20      int val = (1 << exp) + ((dw & 0x7FFFFF) >> (23 - exp));
        // En fonction du bit de signe, inverse le résultat :
22      if (dw & 0x80000000)
            return -val;
24      else
            return val;
26  }
```

Bon, évidemment, ceci ne gère pas toute la norme, le dénormalisé, les infinis, etc.

Nombres flottants $\neq$ réels ! *(I)*

« What Every Computer Scientist Should Know About Floating-Point Arithmetic », David GOLDBERG, Computing Surveys, mars 1991, ACM

- Nombres flottants $\equiv$ pale imitation de $\mathbb{R}$ et même de $\mathbb{D}$ ☹
- Nombreuses approximations
-  Propriétés algébriques de $\mathbb{R}$ non vérifiées : non associatif

$$(1 \oplus 10^{40}) \ominus 10^{40} = 0$$
$$1 \oplus (10^{40} \ominus 10^{40}) = 1$$

- Changement des résultats possibles selon optimisations... ☹
- Notion d'équivalence séquentielle de programme entre différentes versions

Nombres flottants $\neq$ réels ! *(II)*

  - Forte : le programme obtenu donne le même résultat
  - Faible : le programme obtenu donne le même résultat modulo les problèmes numériques précédents
- Choisir programmation prenant en compte ces caractéristiques
  - TEX écrit en virgule fixe 16+16 bits pour portabilité multi-plateforme ☺
  - Compromis entre performances & précision
-  Compilateurs devraient en tenir compte (pas optimisations sauvages)
- Exemples
  - $(x-y)(x+y)$ plus précis (voire plus rapide) que x^2-y^2
  - Algorithme somme de flottants

Algorithme de sommation de flottants *(I)*

- Solution triviale

```
1  double x[N];
2  double s = 0;
   for(int i = 0; i < N; i++)
4    s += x[i];
```

$$s = \sum_{i=0}^{N-1} x_i(1+\delta_i)$$

avec $|\delta_i| < (N-i)\epsilon$

Algorithme de sommation de flottants *(II)*

- Version KAHAN

```
1  double x[N];
2  double s = x[0];
   double c = 0;              // Erreur d'arrondi
4  for(int i = 1; i < N; i++) {
     double y = x[i] - c;     // Compense erreur précédente
6    double t = s + y;        // Nouvelle somme
     c = (t - s) - y;         // Estime l'erreur arrondi
8    s = t;
   }
```

$$s = \sum_{i=0}^{N-1} x_i(1+\delta_i) + \mathcal{O}(N\epsilon^2 \sum_{i=0}^{N-1} |x_i|) \quad \text{avec} \quad |\delta_i| \leq 2\epsilon$$

Optimisations incontrôlées du programme fait des ravages ici... car revient à algorithme trivial ! ☹

Vers des nombres flottants normalisés (I)

- Possible de représenter des nombres de plusieurs manières

$$\mathcal{M}' = 2^{-a}\mathcal{M}$$
$$\mathcal{E}' = \mathcal{E} + a$$

- Problème des codages redondants : comparaisons difficiles ☹
- Idée 1 : normaliser ! Exemple : choisir le $\mathcal{M}$ le plus grand pouvant loger dans les bits alloués pour la mantisse
- Idée 2
  - $\forall \mathcal{M} \neq 0$: commence toujours par 1 en binaire
  - $\rightsquigarrow$ Ne pas stocker ce 1 évident...

$\rightsquigarrow$ Flottant normalisé : gagne 1 bit de précision pour la mantisse ! ☺

Pourquoi des nombres flottants dénormalisés ? *(I)*

- Soustraction de 2 nombres normalisés, par exemple en simple précision

$$a = 2,05 \cdot 10^{-37}$$
$$b = 2,03 \cdot 10^{-37}$$
$$a - b = 2 \cdot 10^{-39}$$
$$a \ominus b = 0$$
$$a \neq b$$

  Seule solution car $\mathcal{M}$ ne peut pas commencer par 1... ☹
- Idée : rajouter mode dénormalisé pour très petits nombres où $\mathcal{M}$ peut ne pas commencer par un 1
- Permet *underflow* (dépassement de capacité par le bas) progressif

Pourquoi des nombres flottants dénormalisés ? *(II)*

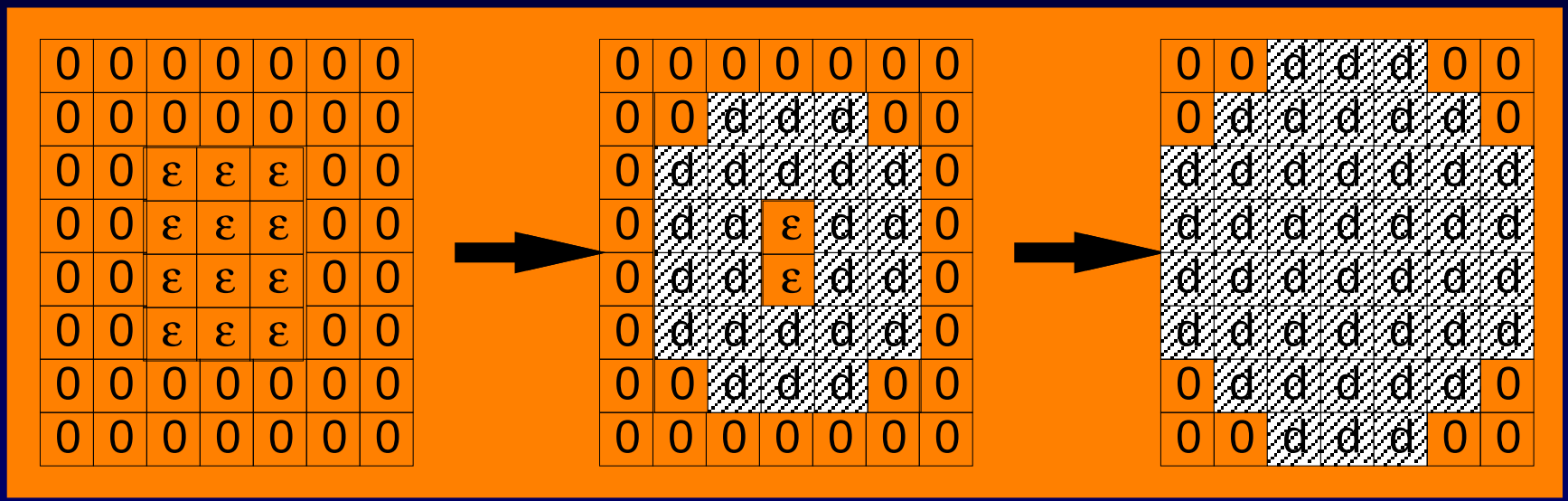
-  Si flottant dénormalisé non géré directement en matériel : génère exception et calculs terminés par... système d'exploitation $\rightsquigarrow$ performances $\searrow\searrow$ ☹
-  Parfois autorisation exception $\Longrightarrow$ suppression pipeline (DEC Alpha) ☹

Dénormalisation flottante (I)

- Parfois exception IEEE-754 générée lors de la dénormalisation
- Typiquement un programme de différences finie avec un domaine avec de petites valeur ϵ entouré de 0:

$$x^n_{i,j} = \frac{x^v_{i-1,j} + x^v_{i+1,j} + x^v_{i,j-1} + x^v_{i,j+1}}{4}$$

$\rightsquigarrow$ Propagation d'ondes de dénormalisation



Dénormalisation flottante *(II)*

- À pleurer sur machine parallèle si ordonnancement statique : tous les processeurs attendent le plus lent ! ☹
- Rajout d'un biais pour ne plus être au voisinage de 0. Mais perte de dynamique... Compromis

$$y^n_{i,j} = x^n_{i,j} + b \quad (1)$$

$$y^n_{i,j} = \frac{y^v_{i-1,j} + y^v_{i+1,j} + y^v_{i,j-1} + y^v_{i,j+1}}{4} \quad (2)$$

- Bonne nouvelle : GPU gèrent les nombres dénormalisés en matériel sans pénalité

Outline

Applications codage binaire : multispin-coding (I)

- Exploitation du parallélisme en bits
- Ranger plusieurs petites données par mot machine
- 4 opérations sur 64 bits/cycle ≡ 256 opérations sur 1 bit/cycle !
- Opérations binaire style `^`, `&`, `|`, `~` sans problème
- Jeux d'instructions :
  - 1 Alpha 21164 à 600 MHz ≡ 76,8 GIPS 1 bit, 9,6 GIPS 8 bits
  - 1 Pentium 4 SSE3 à 4 GHz : 2 opérations 128 bits/cycle ≡ 1 TIPS (10^{12} opérations par secondes) 1 bit
- Idée : plutôt que de résoudre 1 problème à la fois, éclate problème en binaire pour calculer 256 tranches de problèmes binaires à la fois

Exemple : bibliothèques de cassage de codes cryptographiques (détection mots de passe faibles avec John the Ripper), traitement d'image, traitement du signal, codage, optimisation de programmes...

Application utilisant des additions 9 et 6 bits *(I)*

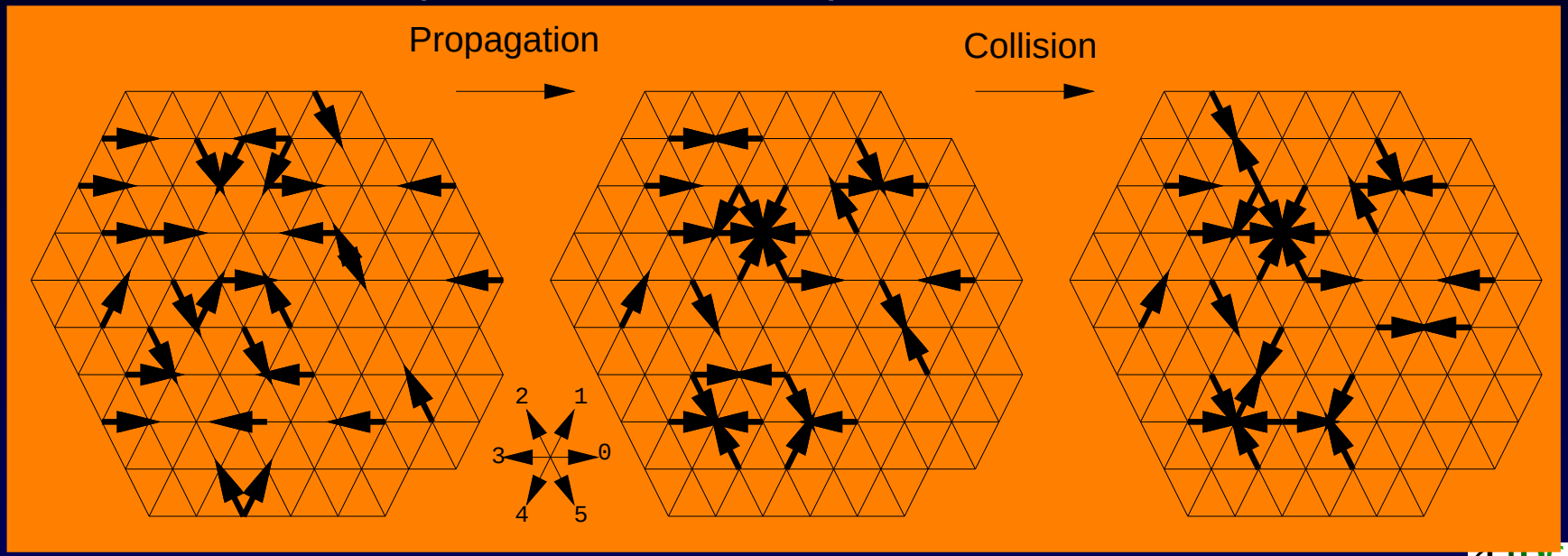
- Compactage dans 32 bits `a_xxs_yys` :

`quality`	`q_a`	0	`q_x`	0	`q_y`
8	6	1	8	1	8

- `q_a` sur 6 bits, `q_x` et `q_y` sur 8 bits
- Opérations sur `q_x` et `q_y` sur 9 bits $\rightsquigarrow$ stockage sur 9 bits aussi (évite l'extraction)
- Garde des 0 délimiteurs absorbant les retenues (`& masque`)
- Besoin de tester $(q_x, q_y) \in [-128, 127]^2$:
  - Changement repère (biais $+128$) $\rightsquigarrow$ $(q'_x, q'_y) \in [0, 255]^2$
  - Test de `a_xxs_yys & ((1<<8) + (1<<18)) == 0` : 1 instruction !

Gaz sur réseau (I)

- Méthode *lattice* BOLTZMAN
- Sites contenant des particules se déplaçant quantiquement
- Interactions entre particules sur chaque site



- Tableau de sites contenant 1 bit de présence d'1 particule allant dans 1 direction
- Symétrie triangulaire

Gaz sur réseau (II)

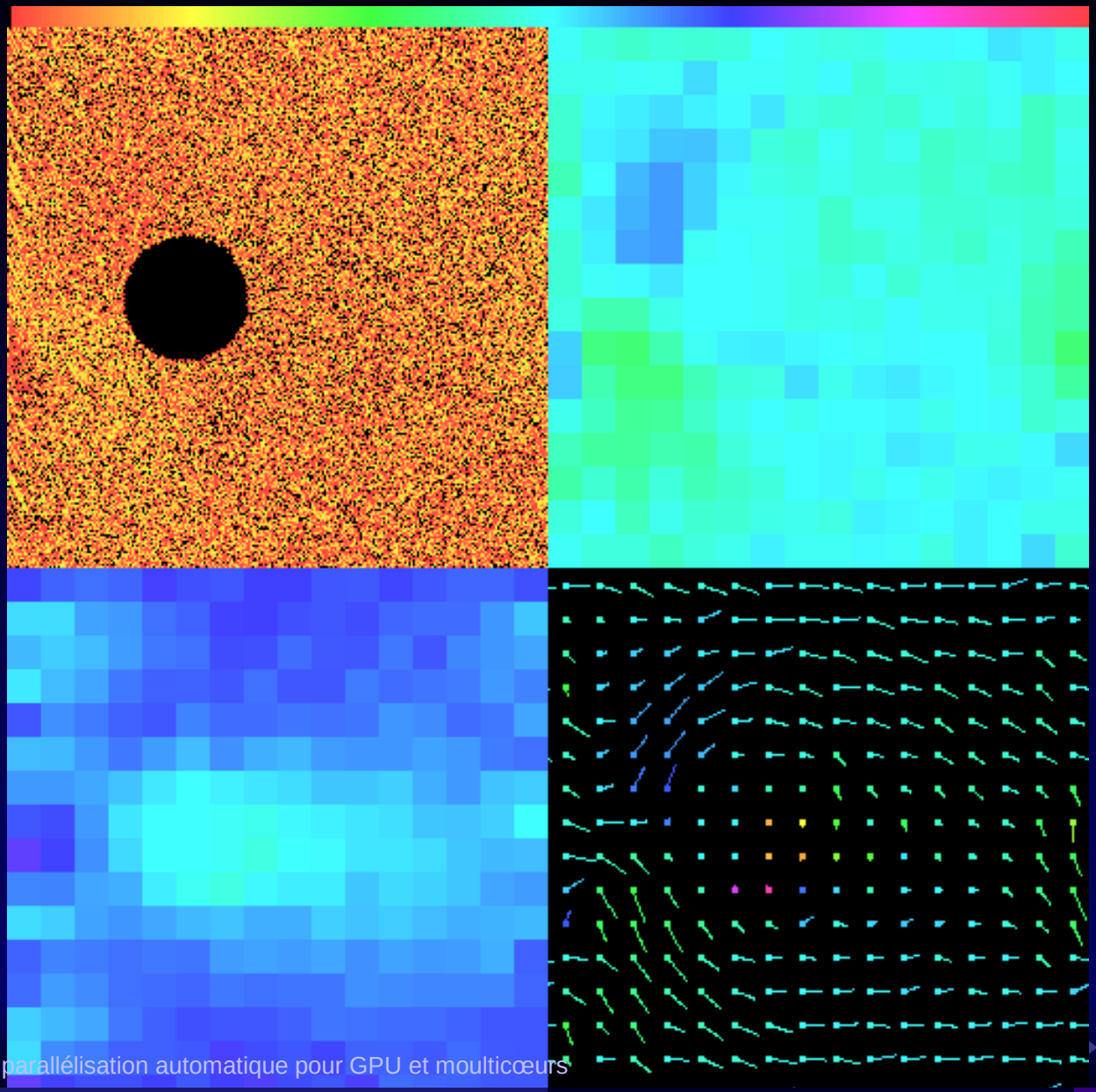
- Compactage de 32 ou 64 sites/`int` par direction

```
1  a = lattice[RIGHT];
2  b = lattice[TOP_RIGHT];
3  c = lattice[TOP_LEFT]; // Particules qui montent à gauche
4  d = lattice[LEFT]; // Particules qui vont à gauche
5  e = lattice[BOTTOM_LEFT];
6  f = lattice[BOTTOM_RIGHT];
7  s = solid; // Une condition limite
8  ns = ~s;
9  r = lattice[RANDOM]; // Un peu d'aléa
10 nr = ~r;
       /* A triplet ? */
11 triple = (a^b)&(b^c)&(c^d)&(d^e)&(e^f);
       /* Doubles ? */
12 double_ad = (a&d&~(b | c | e | f));
13 double_be = (b&e&~(a | c | d | f));
14 double_cf = (c&f&~(a | b | d | e));
       /* The exchange of particles : */
15 change_ad = triple | double_ad | (r&double_be) | (nr&double_cf);
```

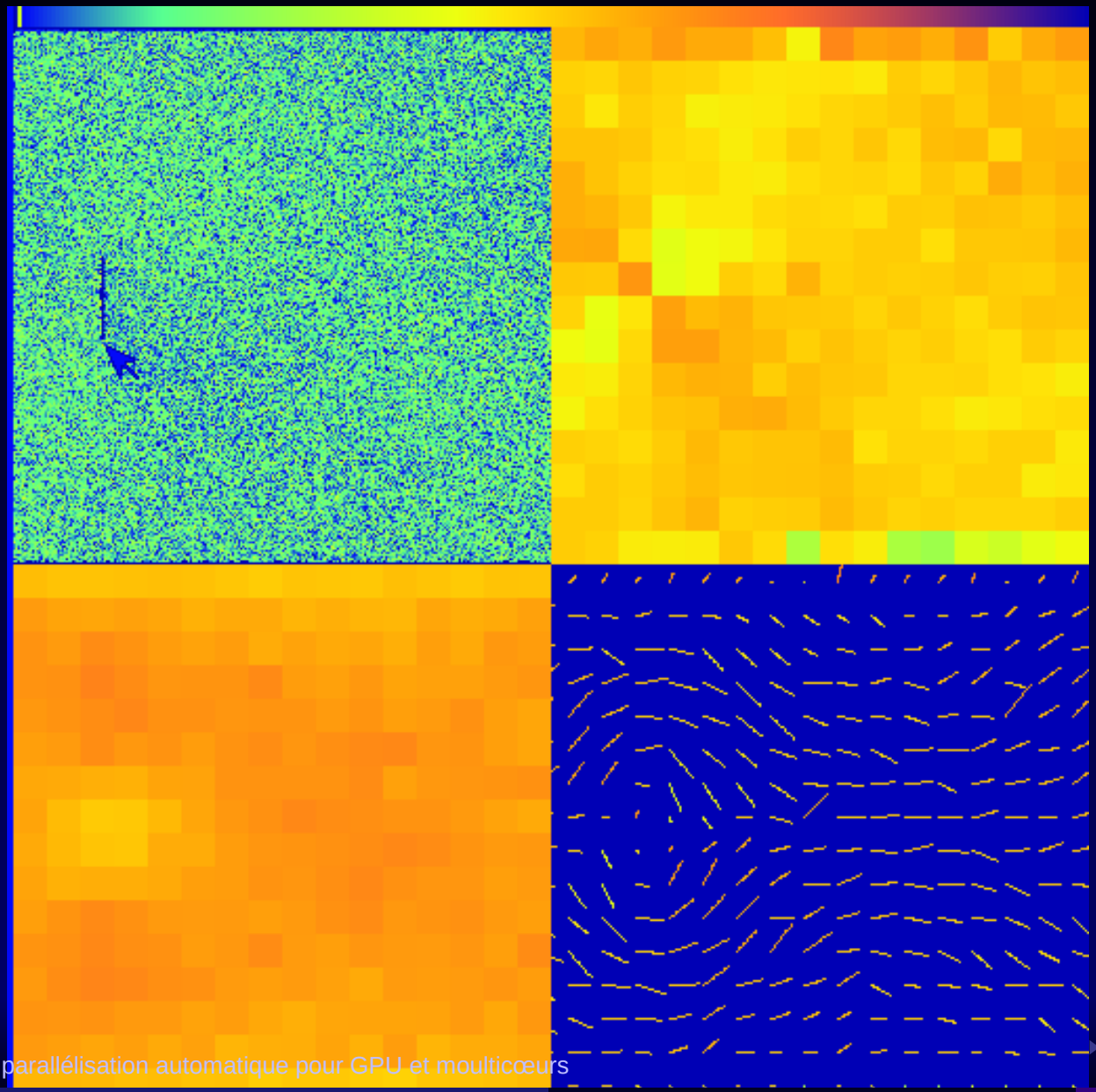
Gaz sur réseau (III)

```
change_be = triple | double_be | (r&double_cf) | (nr&double_ad);
change_cf = triple | double_cf | (r&double_ad) | (nr&double_be);
    /* Where there is blowing, collisions are no longer valuable : */
bl = blow[N_DIR];
s &= ~bl;
ns &= ~bl;
    /* Effects the exchange where it has to do according the solid : *
lattice[RIGHT] = (((a^change_ad)&ns) | (d&s))
    | bl&blow[RIGHT];
lattice[TOP_RIGHT] = (((b^change_be)&ns) | (e&s))
    | bl&blow[TOP_RIGHT];
lattice[TOP_LEFT] = (((c^change_cf)&ns) | (f&s))
    |bl&blow[TOP_LEFT];
lattice[LEFT] = (((d^change_ad)&ns) | (a&s))
    | bl&blow[LEFT];
lattice[BOTTOM_LEFT] = (((e^change_be)&ns) | (b&s))
    | bl&blow[BOTTOM_LEFT];
lattice[BOTTOM_RIGHT] = (((f^change_cf)&ns) | (c&s))
    | bl&blow[BOTTOM_RIGHT];
```

Gaz sur réseau — Cylindre



Gaz sur réseau — Instabilités de Von KARMAN



Conclusion sur multispin coding

- Méthodes de simulation permet de modéliser de nombreux phénomènes (agrégations, gaz + fluides, solides, biologie...)
- Marche avec n'importe quelle architecture : parallélisme de bits des opérations binaires
- Passage en AVX ou LRBni : 256 ou 512 sites traités par cycles
- Bien pour cartes graphiques aussi
- Autre méthode par table de collision mais problème débit mémoire/cache (cf scatter/gather des processeurs vectoriels)
  - Bien pour cartes graphiques si utilisation de caches possible

Outline

Synchronisation (I)

- Exemple de 2 processus producteurs qui produisent 1 produit à chaque fois et 1 variable globale pour compter ce qu'on produit

P_1 :

$$\texttt{produits} = \texttt{produits} + 1$$

P_2 :

$$\texttt{produits} = \texttt{produits} + 1$$

Supposons que `produits` = 42 et que P_1 et P_2 s'exécutent en même temps, on peut avoir P_1 et P_2 qui lisent 42 et réécrivent... 43 au lieu d'avoir 44 ☹

⇝ Besoin d'*atomicité* ou d'avoir de l'exclusion mutuelle: protéger `produits` contre des accès chaotiques

- Besoin de coordonner différents processus pour coopérer sans erreur

Synchronisation (II)

- Sinon, on a des situations de compétitions (*race conditions*)

∃ nombreux moyens techniques d'assurer ce genre de protection

Sémantique mémoire parfois étrange...

Shared memory semantics (I)

- Sequential classical assumption in a multiprocessor: memory access are... in order (causality)
- May not be true for multiprocessor!
- Read and write access done through queues for efficiency, that may reverse some access
- The sequential approximation is the basis of some synchronization algorithms
- $\rightsquigarrow$ Need for a precise semantics about global memory behaviour...
- ... But processor behaviour not always well defined
- « *Memory Models: A Case for Rethinking Parallel Languages and Hardware* », Sarita V. Adve, Hans-J. Boehm. *Communications of the ACM* Vol. 53 No. 8, August 2010, pages 90-101

Shared memory semantics *(II)*

- « *x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors* », Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, Magnus O. Myreen. *Communications of the ACM* Vol. 53 No. 7, July 2010, pages 89-97

DEKKER's mutual exclusion algorithm (I)

- First known correct solution to mutual exclusion in $\approx$ 1960

```
1  // Scoreboard to mark work interest per process. Initial state:
   bool volatile flag[2] = {false, false};
3  // The priority if any conflict:
   int volatile turn = 0;
5  // From process p = 0 or 1
   void acquire(int p) {
7       flag[p] = true; // Warn that I want to work
        while (flag[1 - p]) { // While the other process wants to work
9           if (turn != p) { // If it is not my turn
                flag[p] = false; // Give up and be polite...
11              while (turn != p) { // Wait the other one to finish
                }
13              flag[p] = true; // Warn again I want to work and retry
            }
15      }

17 // Some critical section in between...

19 void leave(int p)
        turn = 1 - p; // Be polite: give the other one a try on next conflict
21      flag[p] = false; // My work is finished
   }
```

DEKKER's mutual exclusion algorithm *(II)*

- Do not need any special instruction to work (test-and-set, atomic memory swap...)
- But need to warn the compiler that some optimizations such as constant propagation is forbidden in this case to avoid this optimization:

```
1  ...
2  int register constant = flag[1 - p]; // Loop invariant hoisting
   while (constant) { // Infinite loop!
4  ...
   }
6  ...
```

↝ Need volatile keyword to warn about external world 

Sequential consistency... from theory to practice *(I)*

- Often considered implicitly true by programmers
- In DEKKER's algorithm, with initial state x =flag[0] = 0 and y =flag[1] = 0

Processor 0	Processor 1
mov (x)←1 mov eax←(y)	mov (y)←1 mov ebx←(x)

should not produce $eax_{p0} = 0 \wedge ebx_{p1} = 0$ in... theory (2 process in the critical section)

- Theoretical sequential-consistent interleaving of instructions (LAMPORT 1979), eventually considered as atomic

(x)←1 eax←(y) (y)←1 ebx←(x)	(x)←1 (y)←1 eax←(y) ebx←(x)	(x)←1 (y)←1 ebx←(x) eax←(y)	(y)←1 (x)←1 eax←(y) ebx←(x)	(y)←1 (x)←1 ebx←(x) eax←(y)	(y)←1 ebx←(x) (x)←1 eax←(y)
eax=0 ebx=1	eax=1 ebx=1	eax=1 ebx=1	eax=1 ebx=1	eax=1 ebx=1	eax=1 ebx=0

Sequential consistency... from theory to practice (II)

We cannot have $eax_{p0} = 0 \wedge ebx_{p1} = 0$ in... theory

- But in practice
  - Read are write are done through processor-local queues for memory latency pipelining
  - Since x and y have different addresses, there is no *local* causality violation
  - But write commit to global memory or cache may be delayed... breaking global causality ☹

  $\rightsquigarrow$ In practice we may have $eax_{p0} = 0 \wedge ebx_{p1} = 0$ $\rightsquigarrow$ 2 process in critical section  ☹
- Hardware architects may not envision all programming consequences of some hardware optimizations...
- Modern shared multiprocessors have no sequential memory semantics 
- ∃ Benchmarks to detect some of this ill-behaviours (Litmus...)

Data-race-free memory model (I)

- Many different memory model exists: ADA, OpenMP, Java, C++,
- Need a model understandable at least by... expert programmers! ☹
- At least, for programs without data race, sequential model should stand: *data-race-free memory model*
  - A data race occurs when 2 threads share data with at least one write
  - Only care about parts with data race
  - Parts without data race should behave with a sequential memory semantics
-  Changing a memory model on an architecture means
  - Changing programming 
  - Breaking memory compatibility  
- Well defined semantic for SPARC processors (old parallel machines ☺)

Data-race-free memory model *(II)*

- More precise models for some *fence* instructions enforcing some memory ordering have been added to recent AMD (2007, 2009) and Intel (2007, 2008, 2009) specifications

Processor 0	Processor 1
mov (x)←1	mov (y)←1
fence	fence
mov eax←(y)	mov ebx←(x)

- ∃ old `LOCK` prefix instruction on *x*86 that locks a global memory lock and flush the local write buffer, prevent other write buffer flush, but quite slow (memory latency and global big lock)...

Processor 0	Processor 1
lock mov (x)←1	lock mov (y)←1
mov eax←(y)	mov ebx←(x)

- ⇝ Synchronization stuff should be implemented by specialists in language constructs and libraries for the programming masses...

Data-race-free memory model *(III)*

- Java `volatile`
- `atomic` in next C++ and C version
- OpenMP `flush` pragma

- Safe language issues
  - Safe languages allow some safe behaviour by construction: Java with sand-boxed execution for untrusted code
  - What if a multithread untrusted code has a wicked race condition *by design*?
  - Is it practically possible to build security breach with race conditions? 

Go back to cache coherence protocols

Outline

Conclusion (I)

- GPU (and other heterogeneous accelerators): impressive peak performances and memory bandwidth, power efficient
- Domain is maturing: any languages, libraries, applications, tools... Just choose the good one ☺
- Real codes are often not well written to be parallelized... even by human being ☹
- At least writing clean C99/Fortran/Scilab... code should be a prerequisite
- Take a positive attitude. . . Parallelization is a good opportunity for deep cleaning (refactoring, modernization. . .) ⇝ improve also the original code
- Open standards to avoid sticking to some architectures
- Need software tools and environments that will last through business plans or companies

Conclusion (II)

- Open implementations are a warranty for long time support for a technology (cf. current tendency in military and national security projects)
- p4a motto: keep things simple
- Open Source for community network effect
- Easy way to begin with parallel programming
- Source-to-source
  - Give some programming examples
  - Good start that can be reworked upon
-  Entry cost
-  Exit cost! ☹
  - Do not loose control on *your* code and *your* data !

Par4All is currently supported by...

- HPC Project
- Institut TÉLÉCOM/TÉLÉCOM Bretagne
- MINES ParisTech
- European ARTEMIS SCALOPES project
- European ARTEMIS SMECY project
- French NSF (ANR) FREIA project
- French NSF (ANR) MediaGPU project
- French Images and Networks research cluster TransMedi@ project (finished)
- French System@TIC research cluster OpenGPU project
- French System@TIC research cluster SIMILAN project