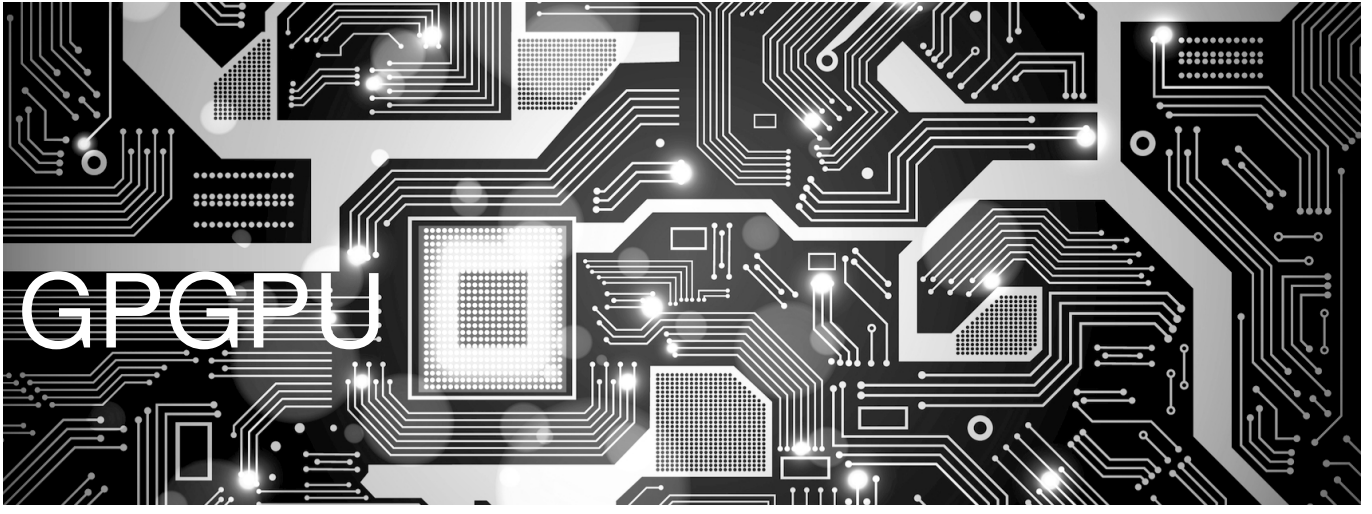
Faculté des Sciences & Techniques

Université de Limoges

Master 1ère année



Développement GPGPU

P-F. Bonnefoi

Version du 8 septembre 2023

Table des matières







Quels sont les besoins ?



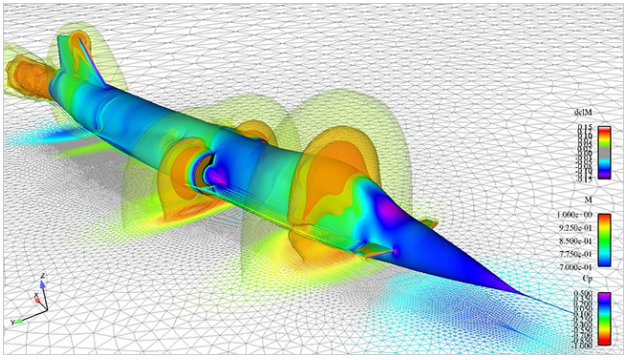


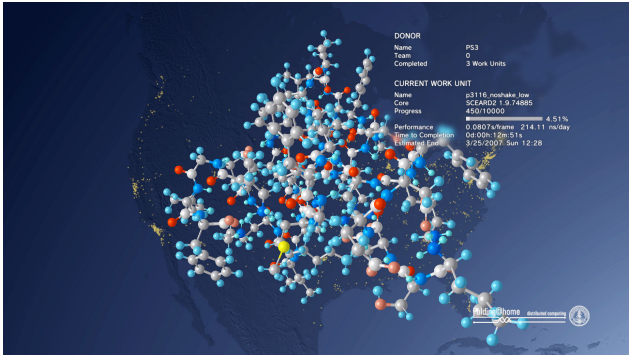


Répondre à une forte demande

En **puissance de calcul** :
- simulation, modélisation : météo, aéronautique ...
- traitement des signaux : images, sons ...
- analyse de données : génomes, fouille de données ...

Demande toujours plus importante, modèle de simulation plus complexe, obtenir des temps de calcul raisonnable, etc.

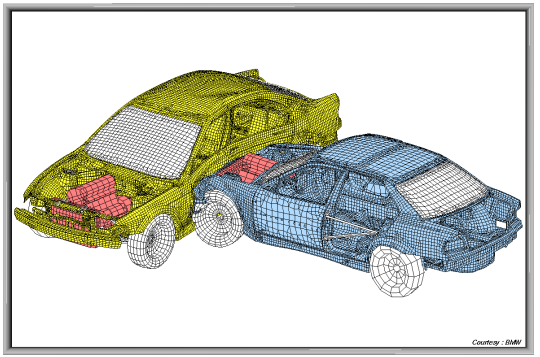


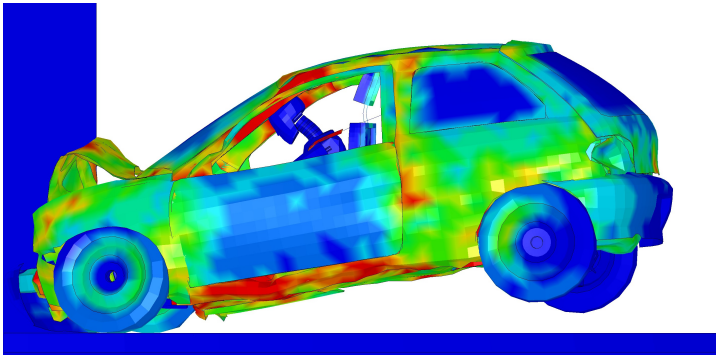


En **puissance de traitement** :
- base de données
- serveurs multimédia
- Internet

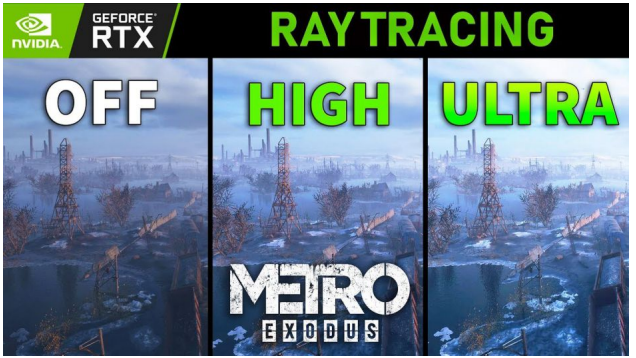
Toujours plus de données à traiter, des données plus complexes etc.

Industrie automobile

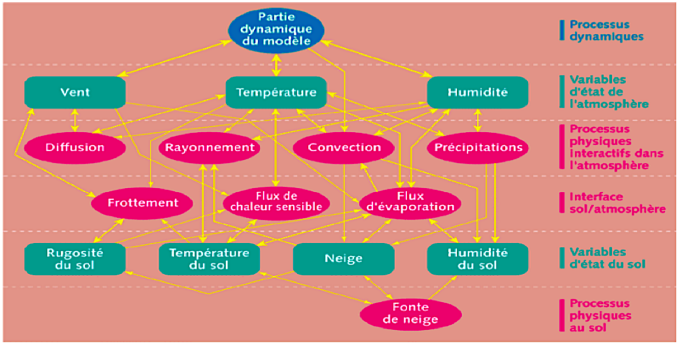




Industrie des effets spéciaux



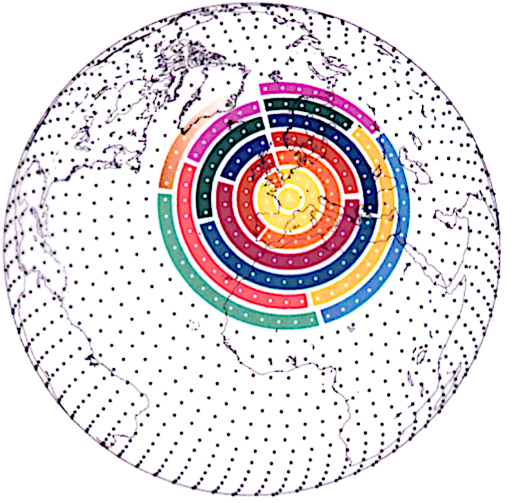
Météorologie : Modèle Arpège 1998



Découpage de l'atmosphère et répartition entre processeurs

Le nombre de variables à traiter est $Nv = 2,3.10^7$

- ▷ quatre variables à trois dimensions x 31 niveaux x 600 x 300 points sur l'horizontale ;
- ▷ une variable à deux dimensions x 600 x 300 points sur l'horizontale ;
- ▷ le nombre de calculs à effectuer pour une variable est $Nc = 7.10^3$
- ▷ le nombre de pas de temps pour réaliser une prévision à 24 heures d'échéance est $Nt = 96$ (pas de temps de 15 minutes simulées).



Puissance de calcul

Elle est exprimée en :

– **MIPS**, «*Machine Instructions Per Second*» représente le nombre d'instructions effectuées par seconde ;

– **FLOPS** «*FLoating Point Operations Per Second*» représente le nombre d'opérations en virgule flottante effectuées par seconde ;

Les multiplicatifs : Kilo = 2^{10} ; Mega = 2^{20} ; Giga = 2^{30} ; Tera 2^{40} ; Peta 2^{50} ; Exa 2^{60}

Certains processeurs vectoriels ont une puissance de calcul de 300 Mflops par exemple.

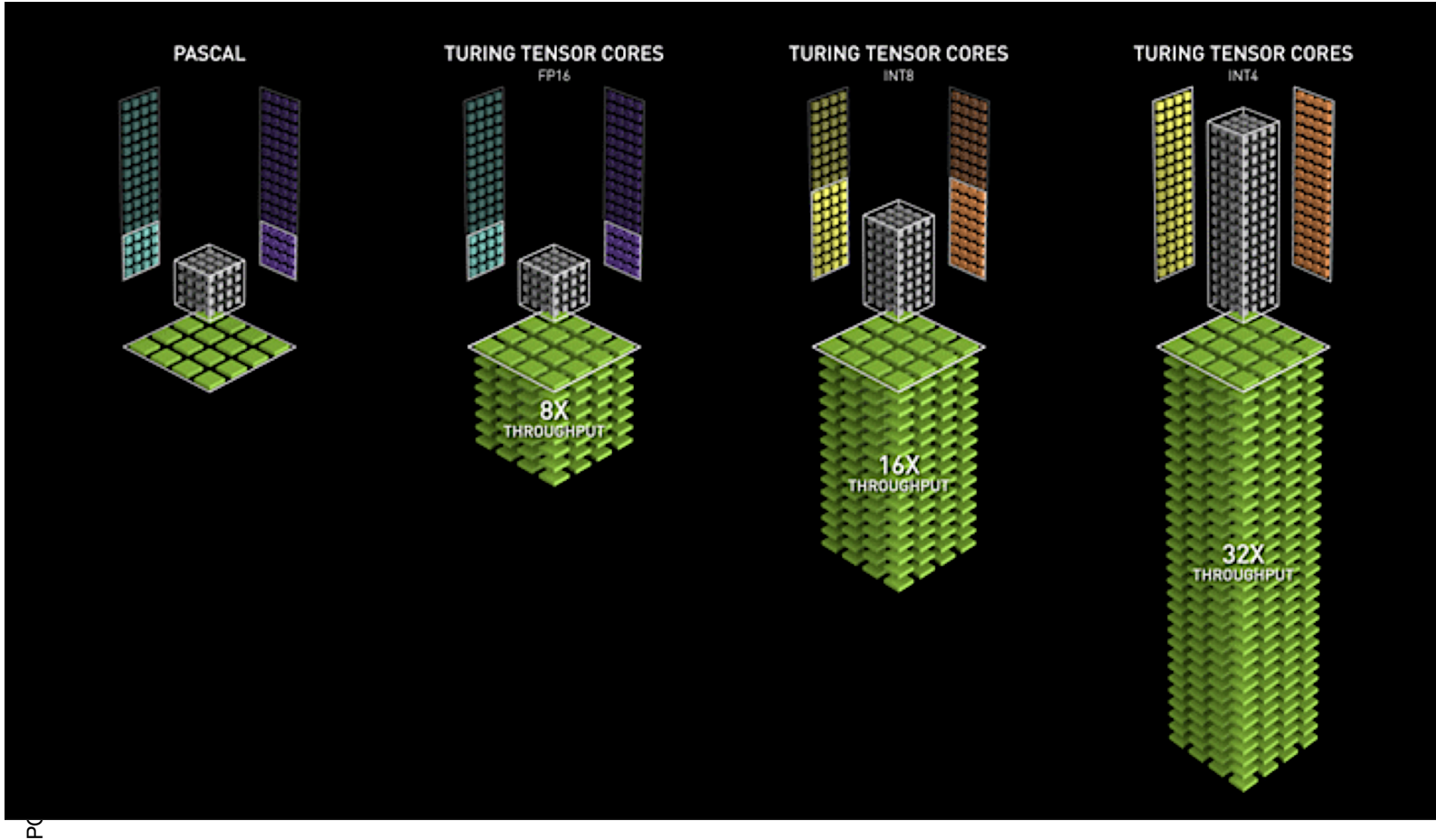
Pour en revenir à la météorologie

- 1998 **Fujitsu VPP700** crédité d'une vitesse de calcul atteignant 62 gigaflops (62 milliards d'opérations flottantes par seconde) ;
- 2003 **Fujitsu VPP5000** avec une puissance de 1,19 Téraflops ;
- 2006 **NEC SX-8** avec une puissance de 9,1 Tflops ;
- 2021 **Sequana XH2000** développée par Bull (filiale du groupe ATOS) :
  - améliorer la prévision des **phénomènes dangereux** avec un gain de 1 à 2 heures d'échéance sur les prévisions ;
  - améliorer la précision géographique et donc mieux déterminer les risques, en descendant à une **échelle infra-départementale** ;
  - prendre en compte plus d'observations et de nouveaux types d'observations tels que les **objets connectés**.








PASCAL
TURING TENSOR CORES
FP16
8X
THROUGHPUT
TURING TENSOR CORES
INT8
16X
THROUGHPUT
TURING TENSOR CORES
INT4
32X
THROUGHPUT

Et le matériel ?

Quels sont les ordinateurs parallèles ?

1950 → 1970 : les pionniers

CDC 6600, (1964) :

- unités de calcul en parallèle,
- 10MHz,
- 2Mo,
- 3 MFlops

Utilisé par Niklaus Wirth pour définir Pascal

CDC7600 (1969) :
équivalent à 7 CDC6600 : 21 MFlops

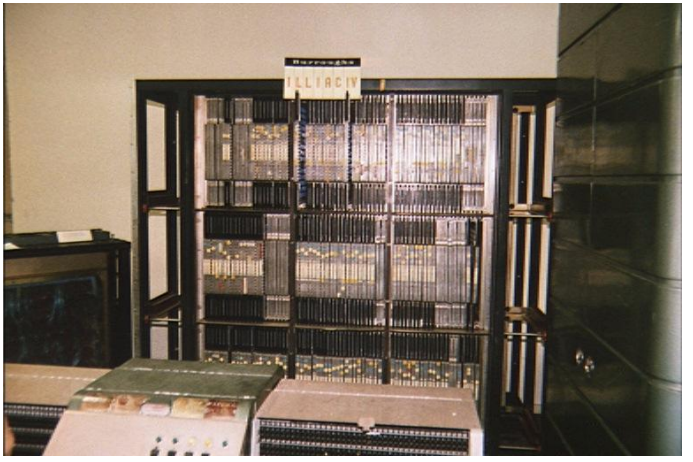
1970 → 1990 : explosion des architectures

- Cray-1 (1975), Cray X-MP (1982) : 2 à 4 processeurs, Cray-2 (1983) : 8 processeurs, Cray Y-MP (1989), Cray T3D (1993), (jusqu'à 512 processeurs, topologie : tore 3D)
- CM-5 (1992) (topologie : fat-tree).
- Hitachi S-810/820 ;
- Fujitsu VP200/VP400 ;
- Nec SX-1/2 ;
- Connection Machine 1 (65536 processeurs, topologie : hypercube) ;
- Intel iPSC/1 (128 processeurs, topologie : grille).

L'Illiac-IV 1950 à 1980

- conçu à l'Université de l'Illinois ;
- fin de construction en 1976 ;
- 64 registres de 64 bits ;
- 13MHz ;
- 1 GFlops prévu ;
- 200 MFlops obtenu ;
- Extrémement coûteux.

Des problèmes matériels : fiabilité !



Cray-1

- commercialisation ;
- utilisation du concept de pipeline ;
- 250 MFlops ;
- utilisation de micro processeurs ;
- 80 MHz.

Des problèmes de logiciel : trop difficiles !

1990 → 2000 : faillite, disparition

- fort retrait des supercalculateurs entre 1990 et 1995 ;
- **nombreuses faillites** : Thinking Machine Corporation (†), Sequent (†), Telmat (†), Archipel (†), Parsytec (†), Kendall Square Research (†), Meiko (†), BBN (†), Digital (†), IBM, Intel, CRAY (†), MasPar (), Silicon Graphics (†), Sun, Fujitsu, Nec.
- rachat de sociétés ;
- disparition des **architectures originales**.

Pourquoi ?

Manque de réalisme

- faible demande en supercalculateurs ;
- coût d'achat et d'exploitation trop élevés ;
- obsolescence rapide ;
- ratio prix/durée de vie d'une machine parallèle extrêmement élevé.

Viabilité des solutions pas toujours très étudiée

- difficultés de mise au point ;
- solutions dépassées dès leur disponibilité.

Une utilisation peu pratique

- systèmes d'exploitation propriétaires ;
- difficulté d'apprentissage.

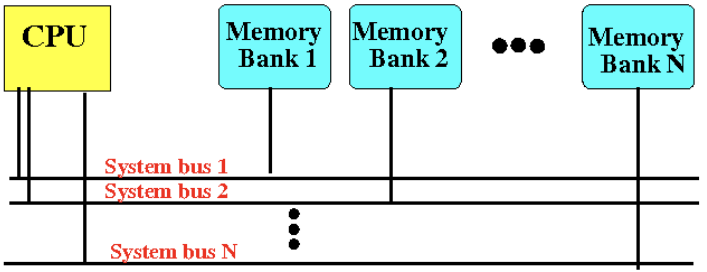
Manque ou absence d'outils

- difficulté d'exploitation.



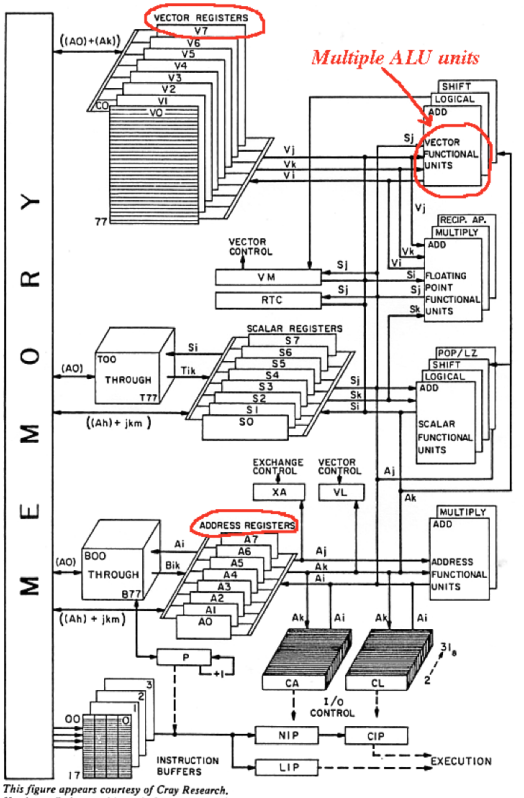



Les prémisses des futures cartes graphiques



(Cray-1 has a 64 way interleaved memory !)

plusieurs requêtes mémoires avec différentes adresses :
jusqu'à 64 transferts simultanés s'ils sont fait sur des blocs mémoires différents !



This figure appears courtesy of Cray Research, Hardware Reference Manual, CRI publication 2240004.

Figure 10.6 CRAY-1's central processor

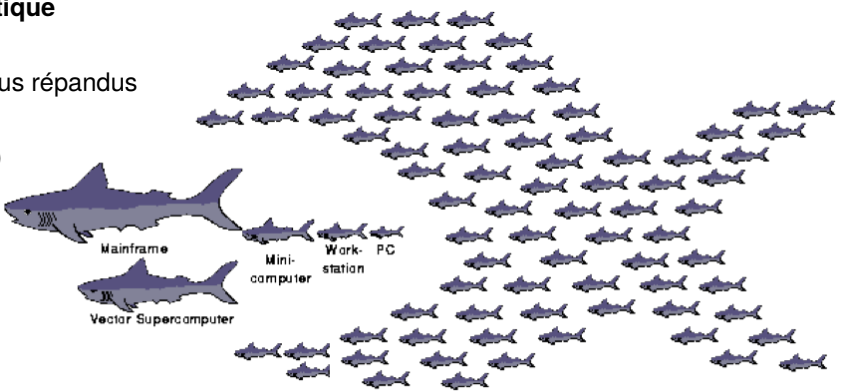
2000 : l'apparition des grilles

Améliorations apportées par la micro-informatique

– micro-processeurs rapides
– réseaux haut débit/faible latence de plus en plus répandus
– configurations PC/stations puissantes
– facilité de mise à jour (changer un composant)

Évolution du Logiciel

– bibliothèques standardisées (MPI, OpenMP)
– compilateurs parallèliseurs
– débogueurs
– système d'exploitation adapté (Beowulf)
– efforts de recherche



Disponibilité

– constat : les matériels sont la plupart du temps peu et sous-utilisés.
– Idée : utiliser ces matériels dont le nombre est énorme : meta-computing.

Grilles de calcul (metacomputing).

– Principe : des milliards de calculs indépendants effectués sur les PCs de "volontaires".
– Seti@Home : transformés de Fourier rapides,
– Folding@Home : conformation 3D de protéines.

mais...

– constat : les communications pénalisent une bonne utilisation.

Utilisation de réseaux de communication dédiés

– projet Network Of Workstation
– grappes de machines (clusters of machines)

Notions de flot de calcul et de flot de données

Sur tout type de machine, un algorithme consiste en un **flot d'instructions** à exécuter sur un **flot de données**.

On a **quatre modèles** de calcul suivant qu'il existe un ou plusieurs de ces flots :

- ▷ Modèle SISD : *Single Instruction Single Data* ;
- ▷ Modèle MISD : *Multiple Instructions Single Data* ;
- ▷ Modèle SIMD : *Single Instruction Multiple Data* ;
- ▷ Modèle MIMD : *Multiple Instruction Multiple Data*.

Classification de Flyn

Flot d'instructions \ Flot de données	Unique	Multiple
Unique	SISD (Von Neumann)	SIMD (tab de processeurs)
Multiple	MISD (pipeline)	MIMD (multiprocesseurs)

SISD

Notre ordinateur ? mais il est déjà superscalaire, multi-coeur...

MISD

Les machines vectorielles multi-processeurs :

- peut exécuter plusieurs instructions en même temps sur la même donnée (processeurs vectoriels et architectures pipelines)
- faible nombre de processeurs puissants (1 à 16)
- mémoire partagée
- limite atteinte, coût important

SIMD

Les machines **synchrones** :

- très grand nombre d 'éléments de calcul (4096 à 65536) de faible puissance avec une toute petite mémoire locale
- un programme unique : exécution d'une même instruction sur des données différentes : GPU

MIMD

Les multi-processeurs à **mémoires distribuées** :

- grand nombre de processeurs ordinaires à mémoire locale
- communication par envoi de messages à travers des réseaux de communication
- chaque processeurs a son propre programme

Les multi processeurs à **mémoire partagée** :

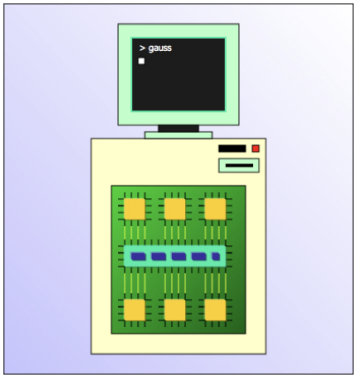
- Si le temps d'accès est égal pour chaque processeur à la mémoire, on parle de UMA, «*Uniform Memory Acess*», ou «*Symmetric Multiprocessors*» (SMP) Exemple : un Core 2 Duo ou multi-cores…
- Si le temps d'accès n'est pas le même on parle de NUMA : «*Non Uniform Memory Access*».



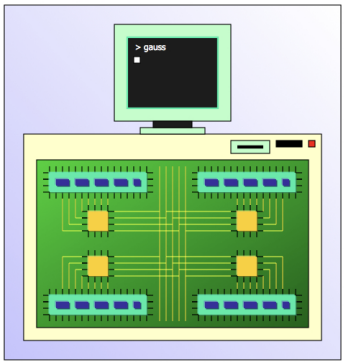



Différentes approches matérielles

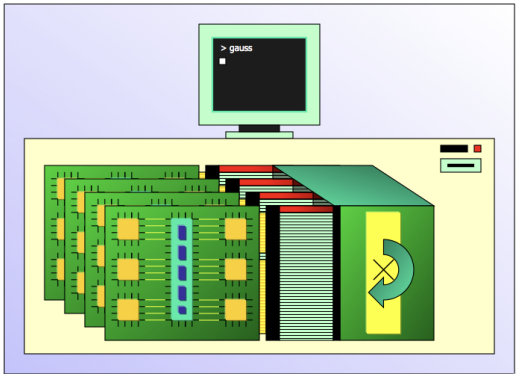
machine à **mémoire partagée**



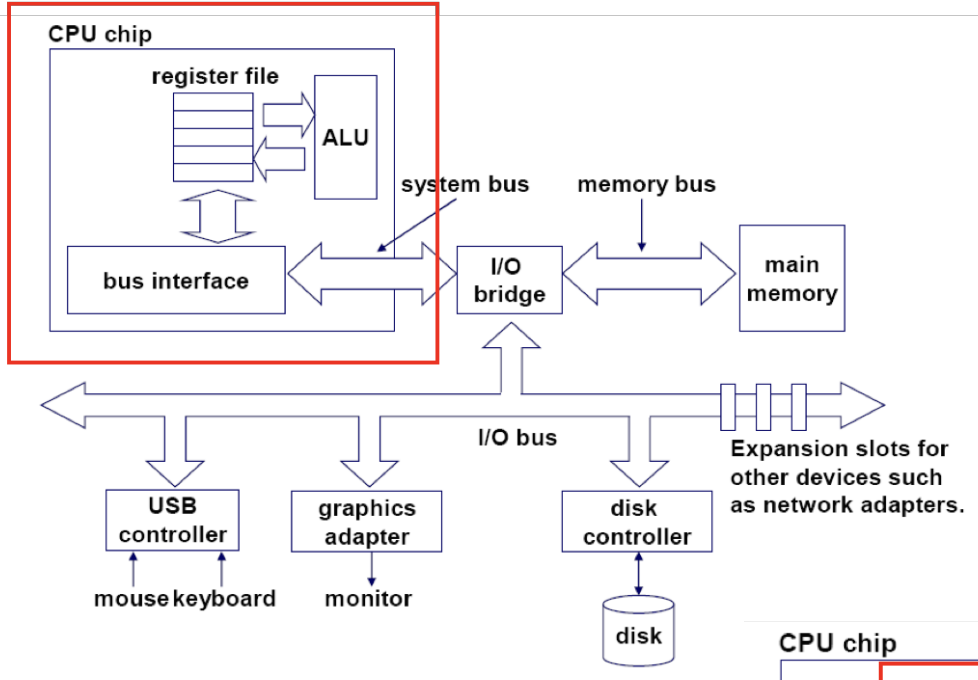
machine à **mémoire distribuée**

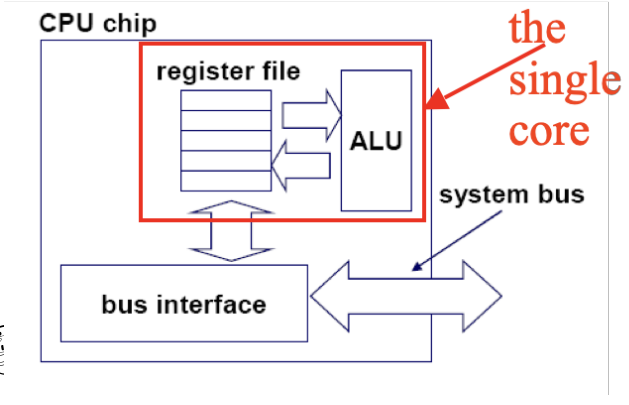


machine **hybrides** : «*Non-Uniform Memory Access*»



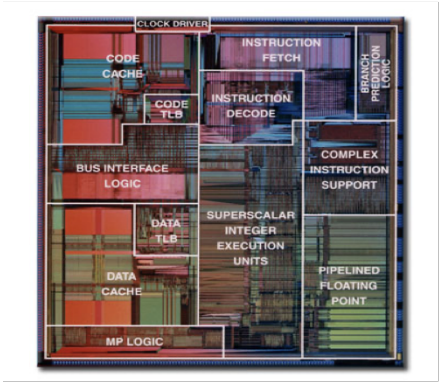
Et les multi-cores ?



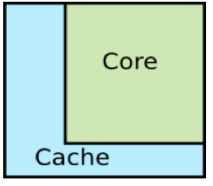


Et les multi-cores ?

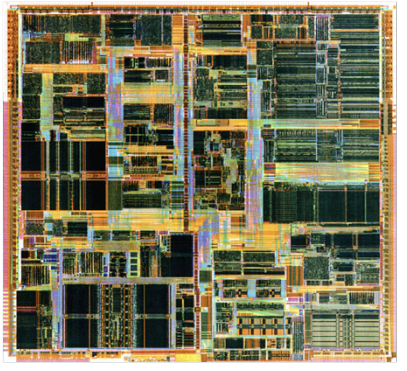
Pentium I

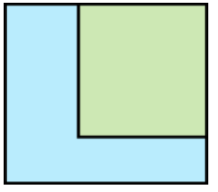


Chip area breakdown

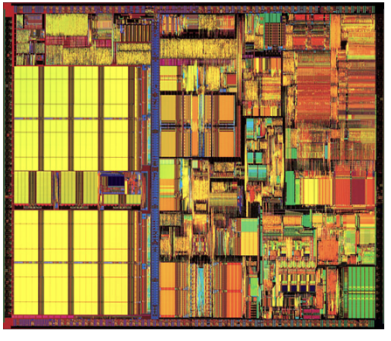


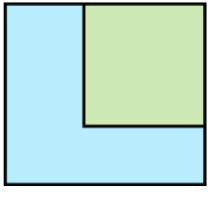
Pentium II



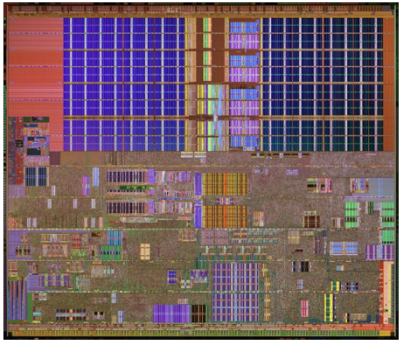


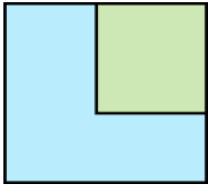
Pentium III





Pentium IV

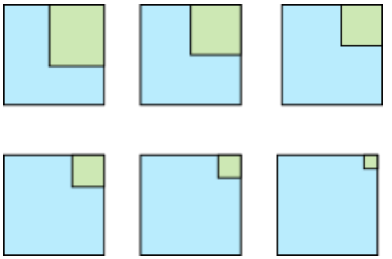




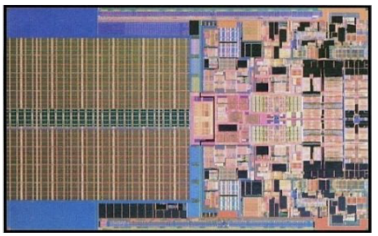
Et les multi-cores ?

L'avenir ?

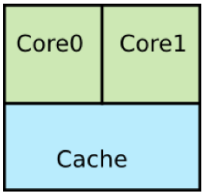
Non ! le multi-cœurs :



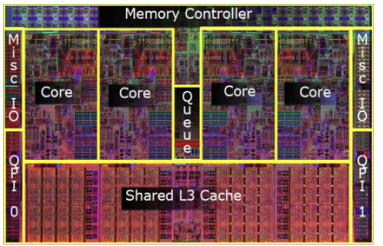
Penryn

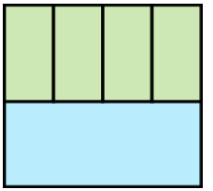


Chip area breakdown

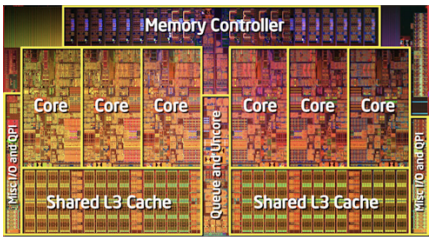


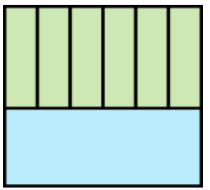
Bloomfield



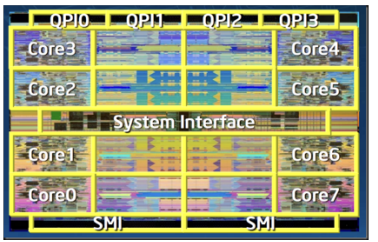


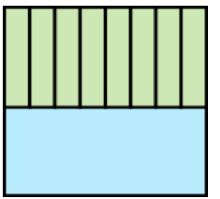
Gulftown

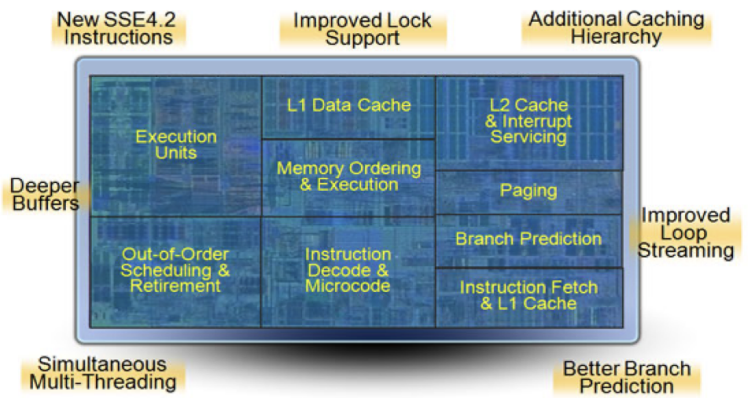


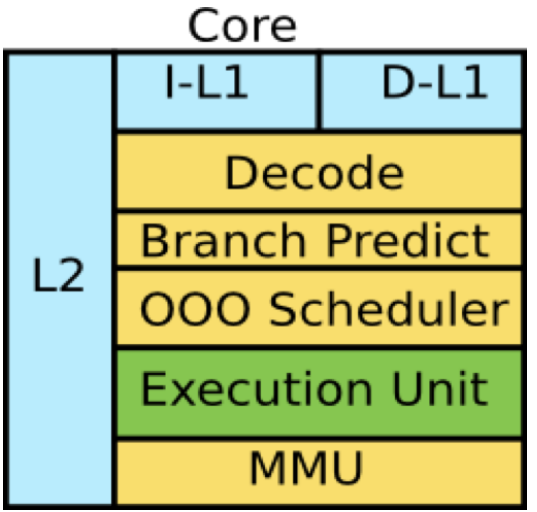


Beckton





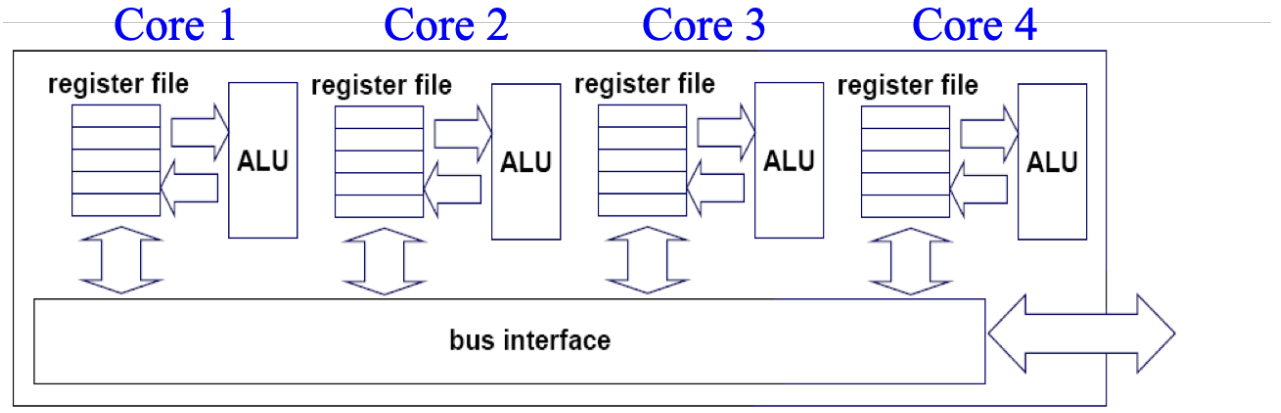




Moins de 10%de la surface sert à l'exécution rélle

OOO : «Out Of Order»

Et les «*multi-cores*» ?

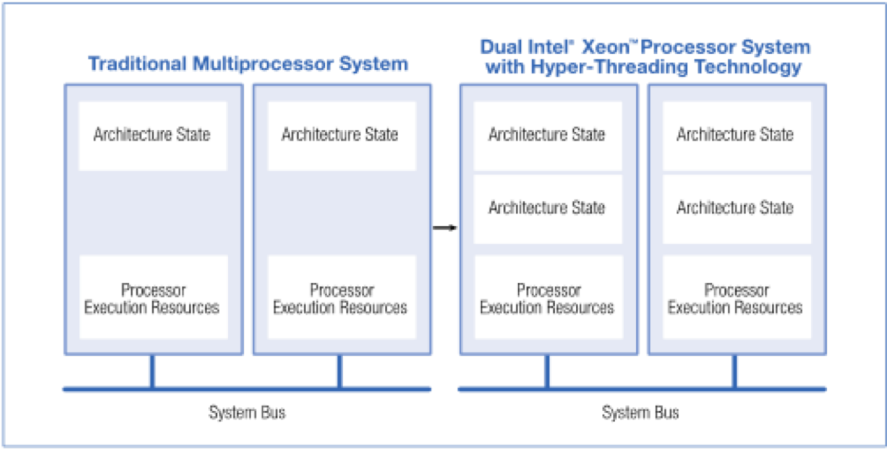


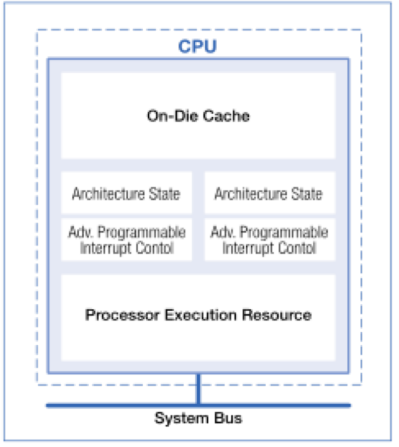
Multi-core CPU chip

- On «*grave*» plusieurs processeurs sur le même support.
- Chaque cœur est vu par le système d'exploitation comme un **processeur séparé**.

Avantages

- On augmente moins la cadence du processeur (échauffement, consommation, difficultés de conception)
- On va vers vers plus de parallélisme (bien !)


Traditional Multiprocessor System
Dual Intel® Xeon™ Processor System with Hyper-Threading Technology
Architecture State
Architecture State
Architecture State
Architecture State
Architecture State
Architecture State
Processor Execution Resources
Processor Execution Resources
Processor Execution Resources
Processor Execution Resources
System Bus
System Bus


CPU
On-Die Cache
Architecture State
Architecture State
Adv. Programmable Interrupt Contol
Adv. Programmable Interrupt Contol
Processor Execution Resource
System Bus

Un processeur logique

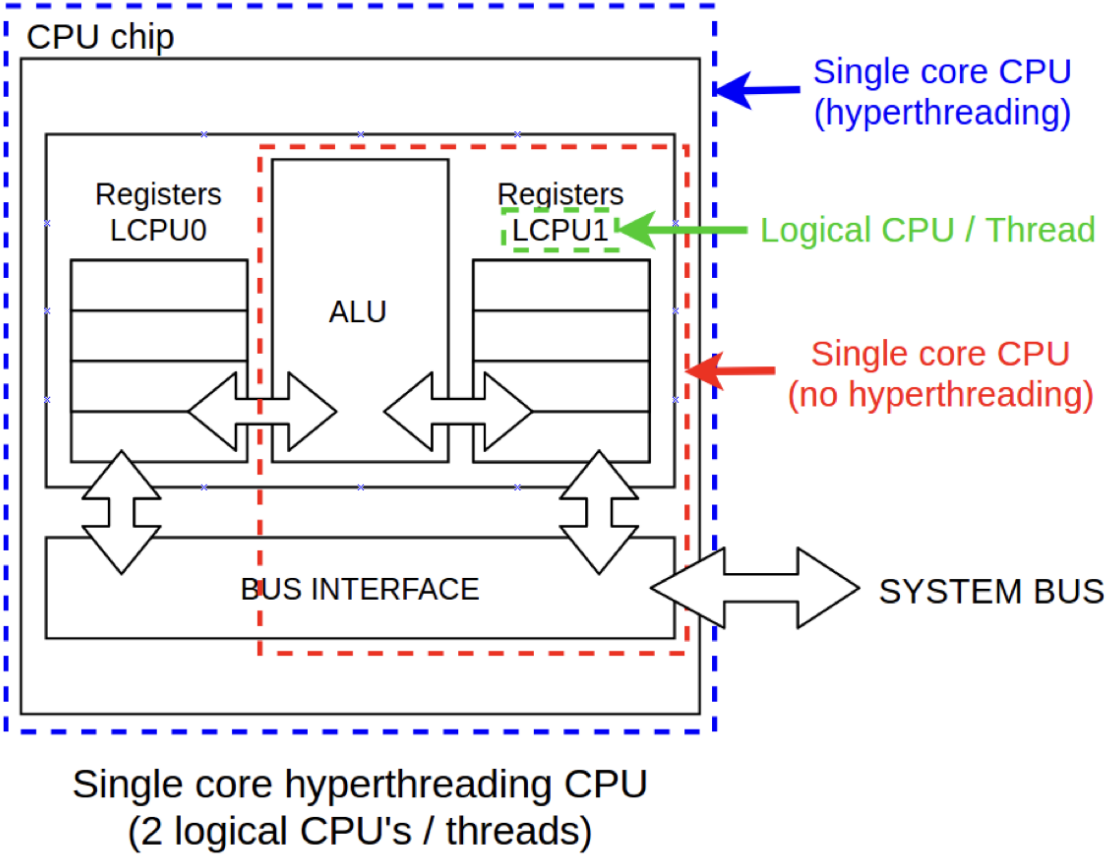
- un «*architecture state*», c-à-d un état matériel : registres, RI, CO, PSW, Interruptions ;
- son propre **flot d'instruction** ;
- peut être interrompu et stoppé **indépendamment**.

Tous les **processeurs logiques** partagent la partie «*exécution*» :

- les caches mémoires ;
- les bus mémoires ;
- le CPU, ALU, FPU, *etc*.

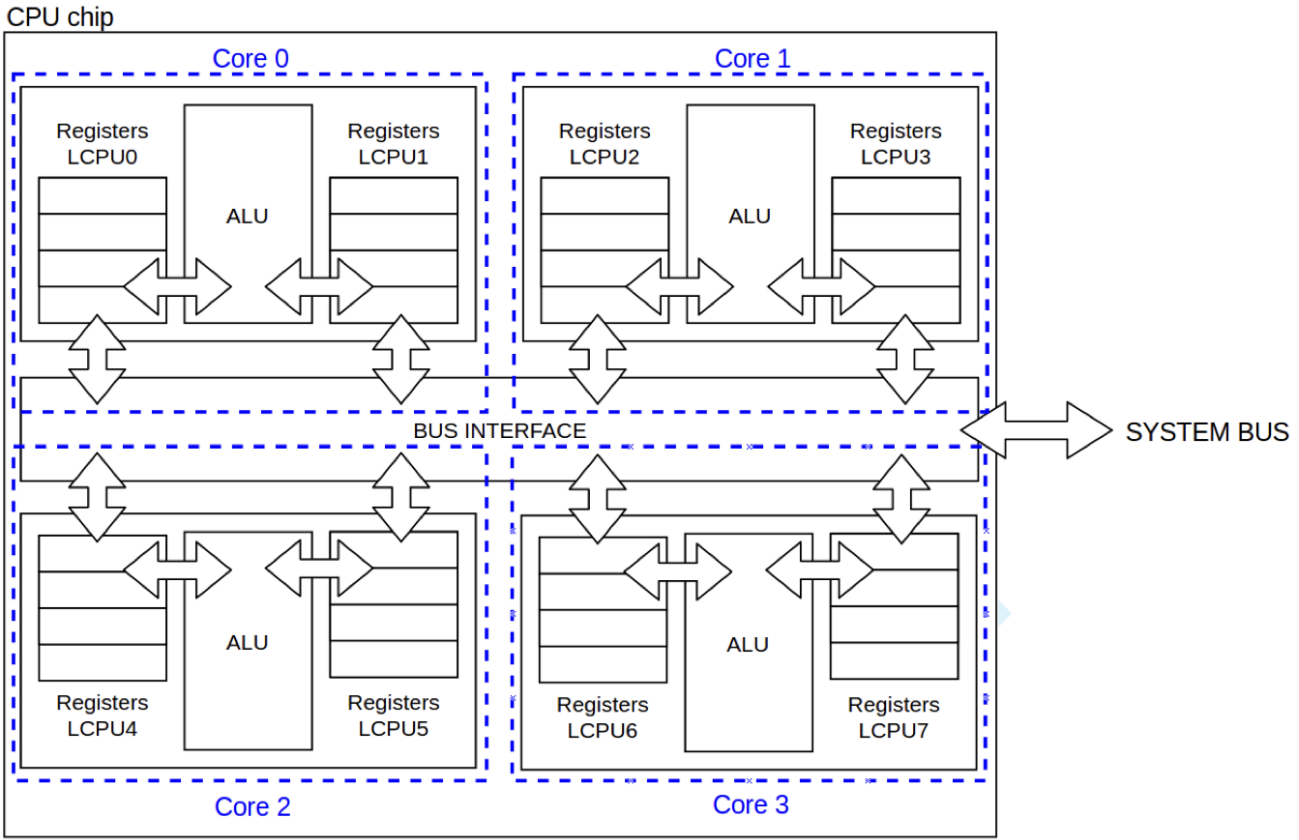
D'après la documentation d'Intel

Each logical processor maintains a complete set of the architecture state. The **architecture state** consists of registers including the general-purpose registers, the control registers, the advanced programmable interrupt controller (APIC) registers and some machine-state registers.
From a software perspective, once the architecture state is duplicated, the processor appears to be two processors. The number of transistors to store the architecture state is an extremely small fraction of the total.
Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic and buses.



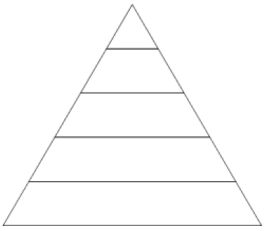
Single core hyperthreading CPU
(2 logical CPU's / threads)

Hyperthreading



Quad-core hyperthreading CPU

Hiérarchie mémoire



	Size (Byte)	Energy (pJ)	Delay (cycles)	Bandwidth (GB/s)
Reg	1K	10	1	1000
L1	32K	20	5	100
L2	256K	100	10	100
L3	8M	200	50	100
Off-chip	4G	2000	100	10

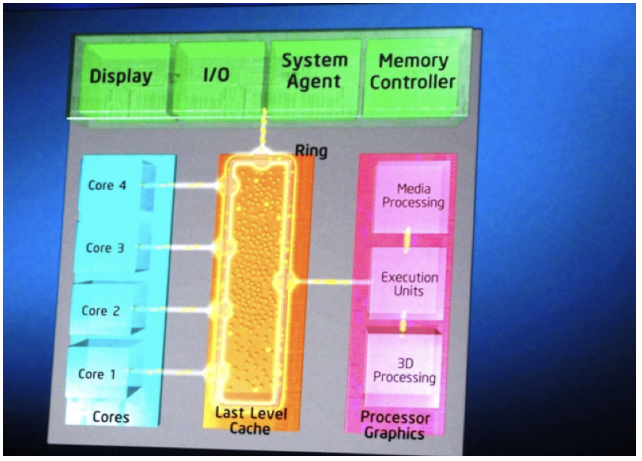
Consommation d'énergie à $45nm$:

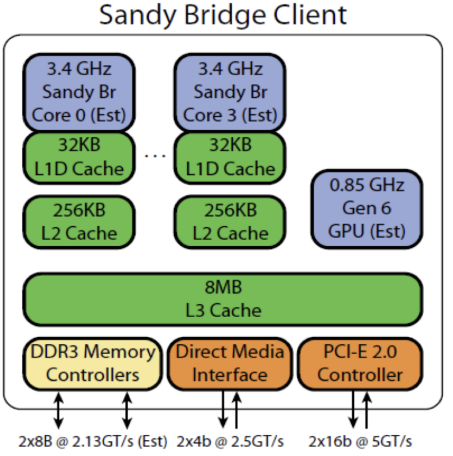
- 64bits Int ADD consomme $1pJ$;
- 64bits FP FMA consomme $200pJ$;

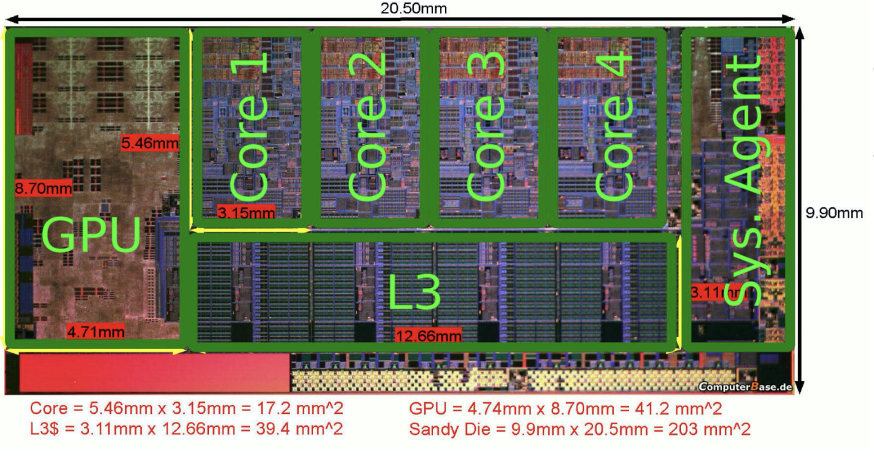
Difficile d'augementer la densité des processeurs : $7nm$ en 2023

Unreleased Intel Mainstream Desktop CPU Series Specs					
VideoCardz.com	Rocket Lake-S	Alder Lake-S	Raptor Lake-S	Meteor Lake-S	Lunar Lake-S
Launch Date	March 30, 2021	Q4 2021	2022	2023 (?)	2024 (?)
Fabrication Node	14nm	10nm Enhanced SuperFin	10nm Enhanced SuperFin (?)	7nm Enhanced SuperFin (?)	TBC
Core µArch	Cypress Cove	Golden Cove + Gracemont	Golden Cove + Gracemont (?)	Redwood Cove + Gracemont (?)	TBC
Graphics µArch	Gen12.1	Gen12.2	Gen12.2	Gen 12.7	Gen 13
Max Core Count	up to 8 cores	up to 16 (8+8)	up to 16 (8+8)	TBC	TBC
Socket	LGA1200	LGA1700	LGA1700	LGA1700	TBC
Memory Support	DDR4	DDR4/DDR5	DDR5	DDR5	DDR5
PCIe Gen	PCIe 4.0	PCIe 5.0	PCIe 5.0	PCIe 5.0	PCIe 5.0
Intel Core Series	11th Gen Core-S	12th Gen Core-S	13th Gen Core-S	14th Gen Core-S	14th Gen Core-S
Motherboard Chipsets	Intel 500 (Z590)	Intel 600 (eg. Z690)	TBC	TBC	TBC

Intel Sandy Bridge





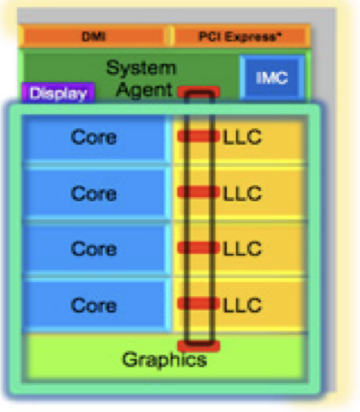


Highlight

- reconfigurable shared L3 cache for CPU and GPU
- ring bus

Sandy Bridge LLC Sharing

- **LLC shared** among all Cores, Graphics and Media
  - Graphics driver controls **which streams** are cached/coherent
  - **Any agent** can access all data in the LLC, independent of who allocated the line, after **memory range checks**
- Controlled LLC **way allocation** mechanism to prevent thrashing between Core/graphics
- Multiple coherency domains
  - **IA Domain** (*Fully coherent via cross-snoops*)
  - **Graphic domain** (*Graphics virtual caches, flushed to IA domain by graphics engine*)
  - **Non-Coherent domain** (*Display data, flushed to memory by graphics engine*)



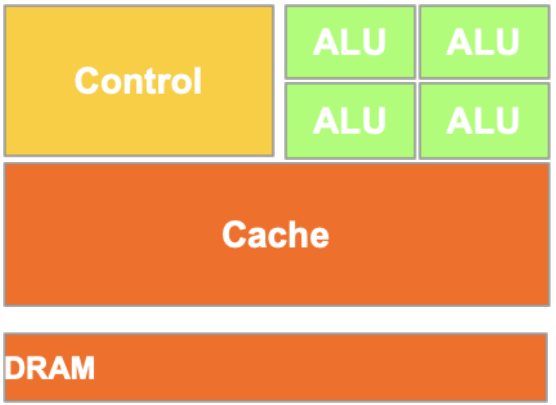
Much higher Graphics performance, DRAM power savings, more DRAM BW available for Cores

IDF2010

d'après l'article "Intel's Sandy Bridge Architecture Exposed", from Anandtech.

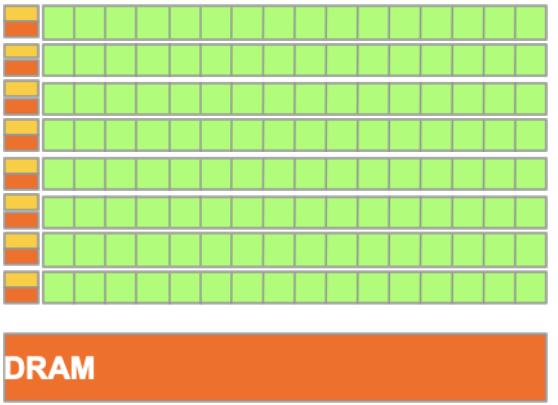
Et les GPUs ?

Prendre de la place pour des cœurs et du cache… Et si on prenait toute la place pour des CPUs ?



CPU

- ▷ Accès mémoire irréguliers ;
- ▷ Plus de cache et contrôle ;
- ▷ Cherche la **performance** par thread.

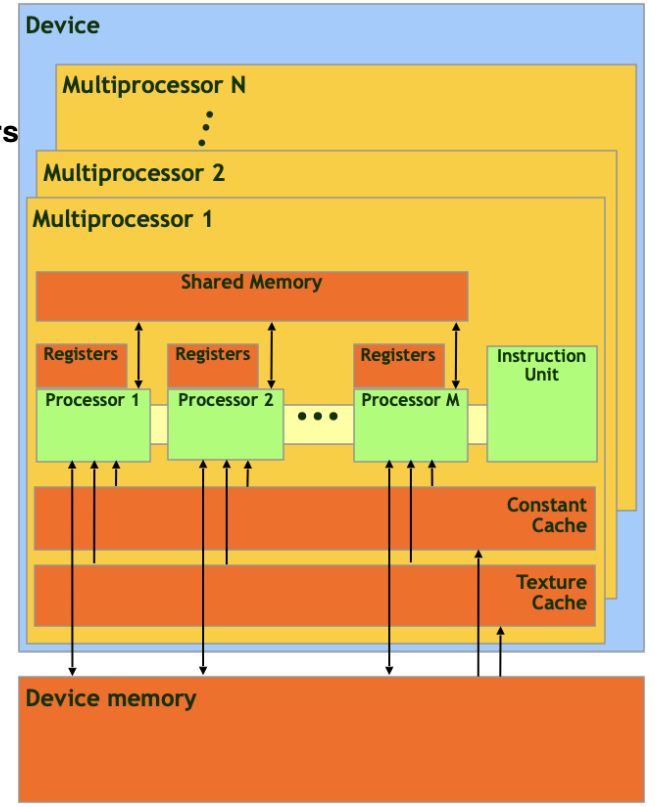


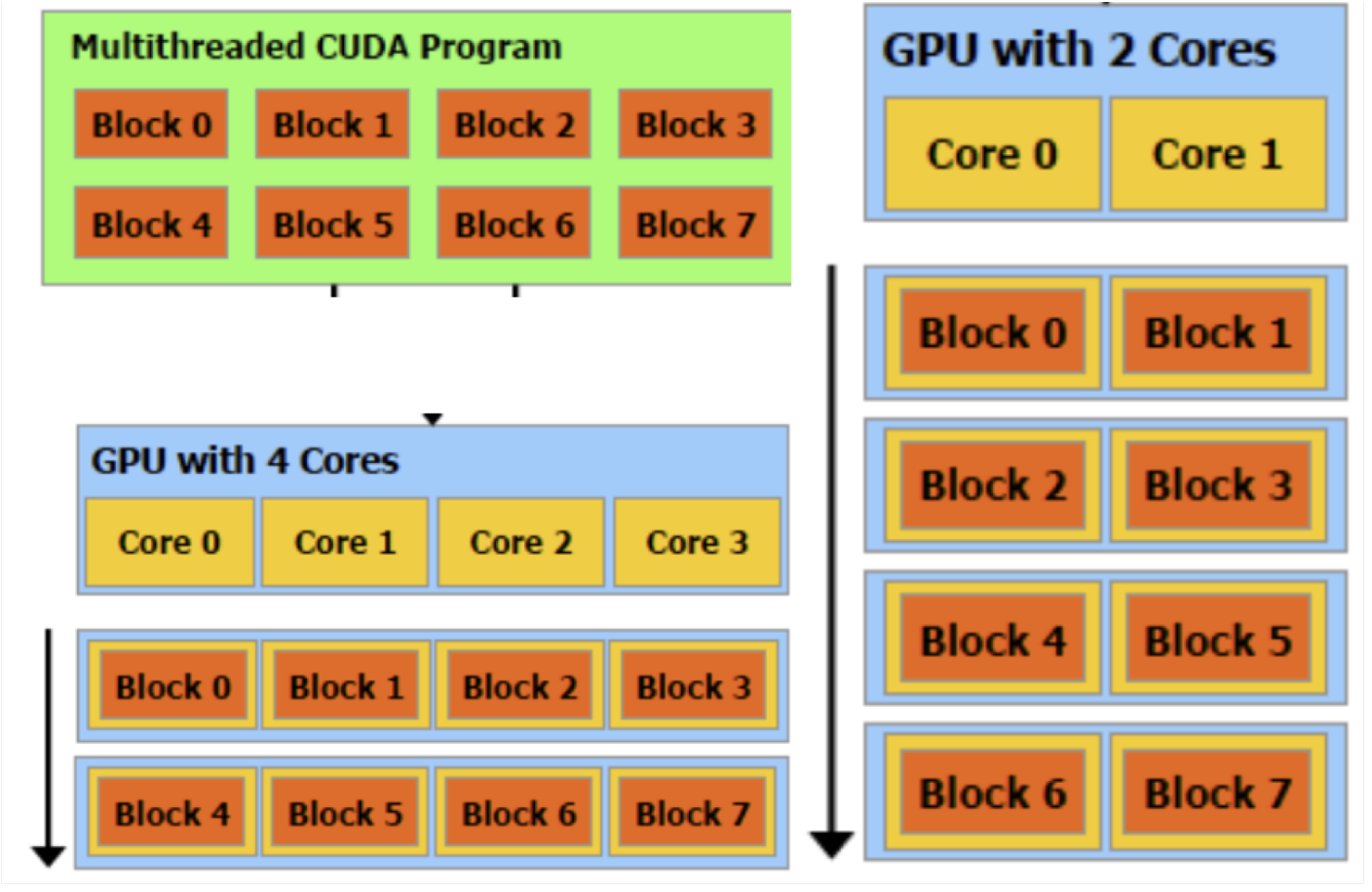
GPU

- ▷ Accès mémoire réguliers ;
- ▷ Plus d'ALUs et massivement parallèle ;
- ▷ Maximiser le **débit**.

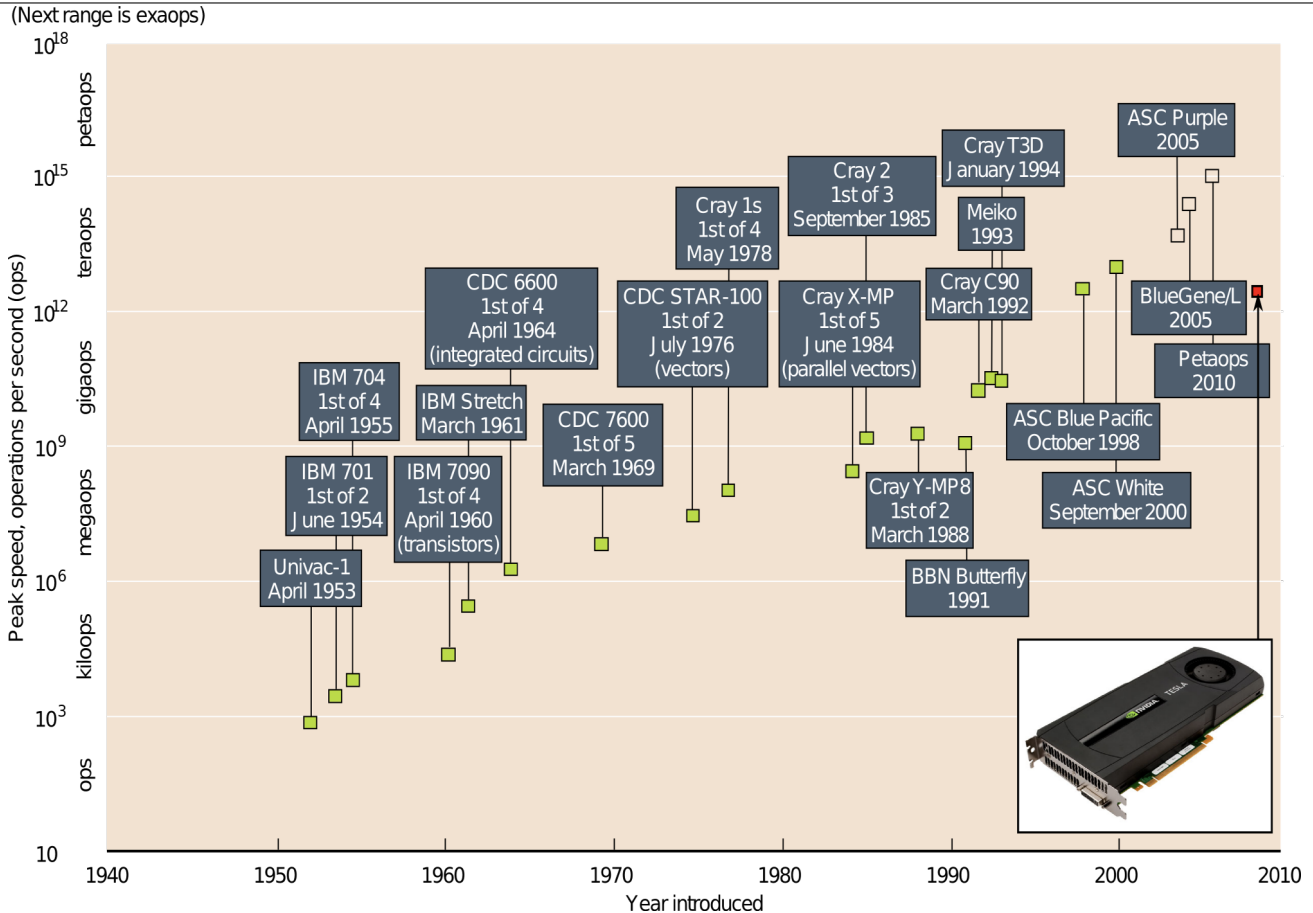
Une architecture complexe

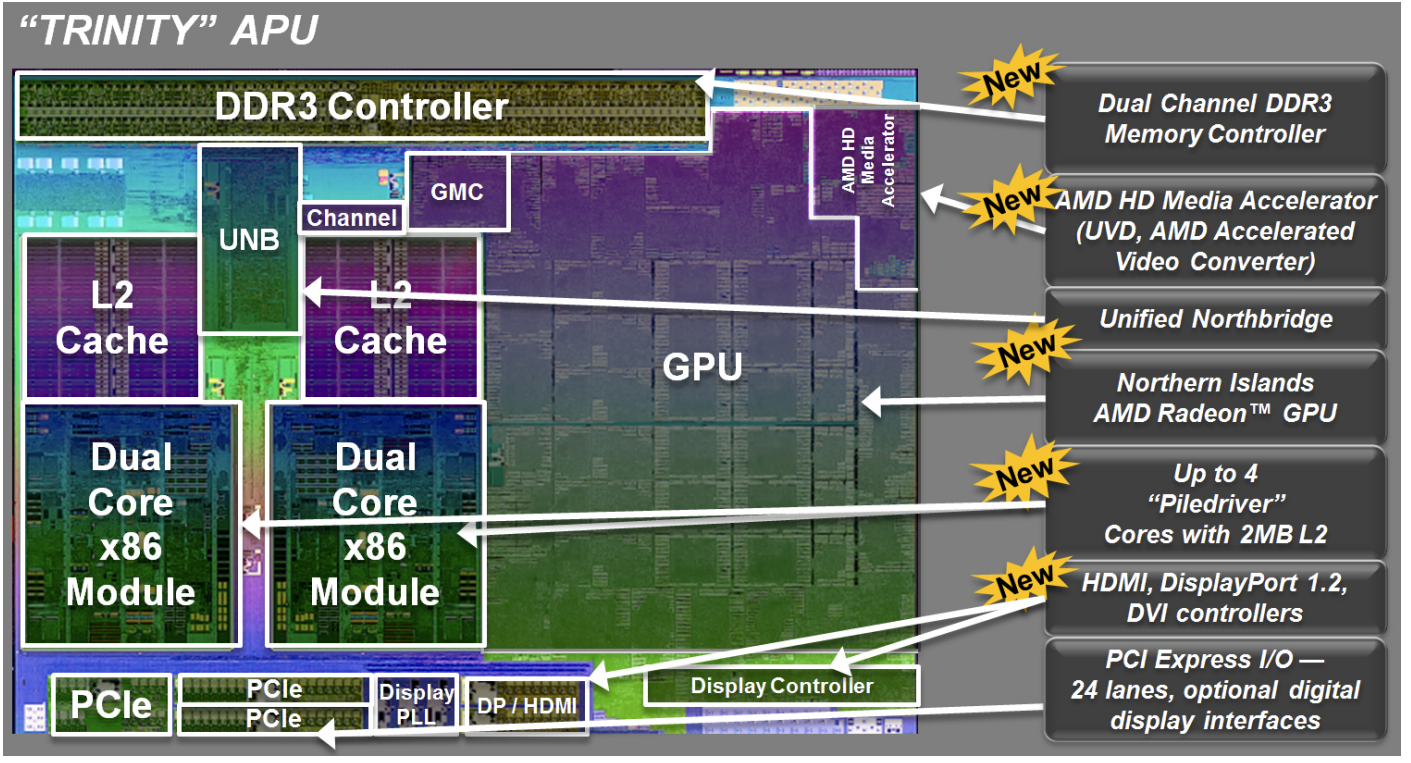
- CUDA, «*Compute Unified Device Architecture*» ;
- **Architecture hiérarchique** ;
  - Une carte contient **plusieurs multiprocesseurs**
  - Plusieurs «*cuda cores*» par multiprocesseur (32 en général)
  - Une unité de contrôle unique.
- **Différents espaces mémoires**
  - Mémoire de la carte : GDDR
    - Beaucoup de mémoire avec un bus rapide vers le multiprocesseur
  - Registres sur la puce : environ 16k
  - Mémoire partagée sur la puce :
    - Partagée entre les différents cores
    - Faible latence et organisée en bloc
  - Mémoire constante (en accès lecture uniquement) et de texture ;
    - En lecture seule et avec du cache.

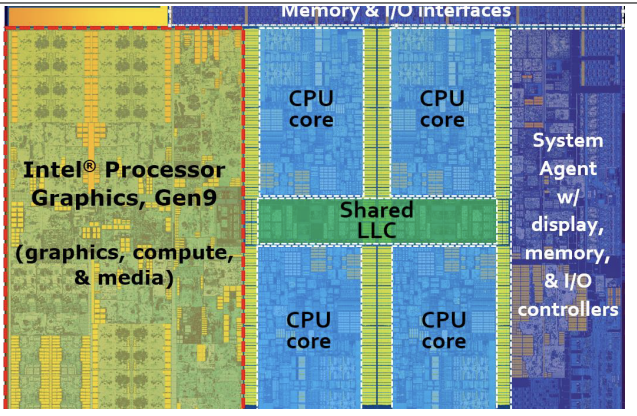


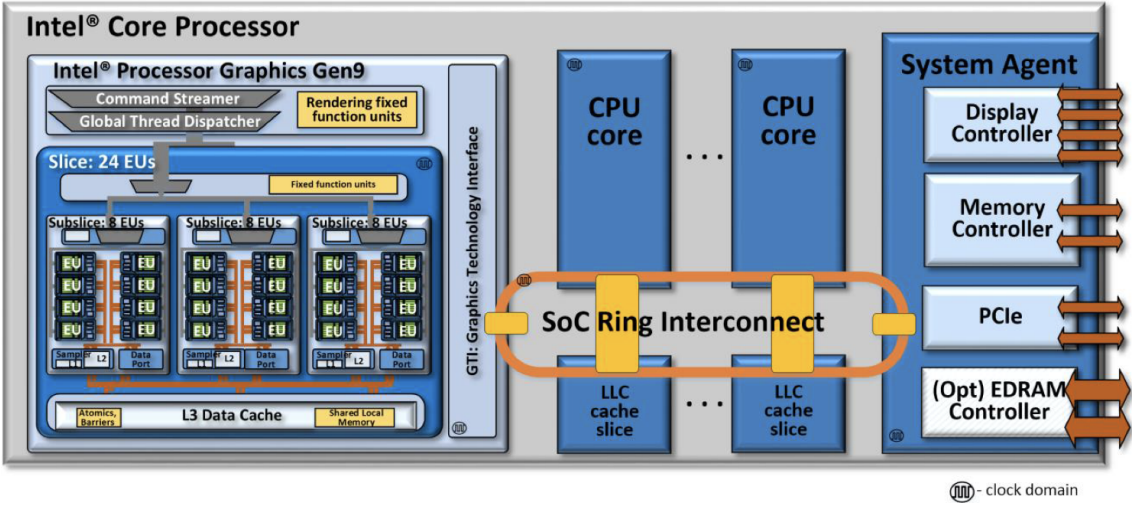

Multithreaded CUDA Program
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 2 Cores
Core 0
Core 1
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 4 Cores
Core 0
Core 1
Core 2
Core 3
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7

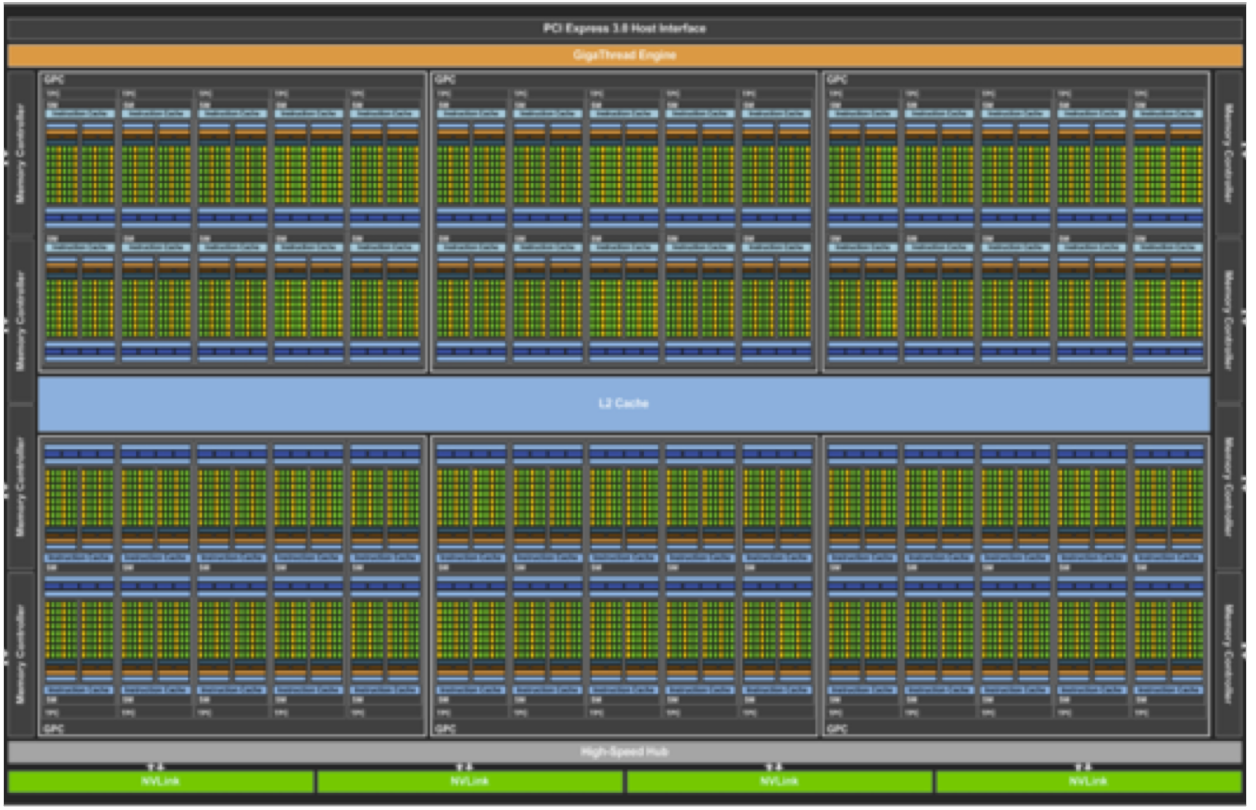
Et les GPUs ? des Résultats !




"TRINITY" APU
DDR3 Controller
UNB
Channel
GMC
L2 Cache
L2 Cache
GPU
AMD HD Media Accelerator
Dual Core x86 Module
Dual Core x86 Module
PCIe
PCIe
PCIe
Display PLL
DP / HDMI
Display Controller
New
Dual Channel DDR3 Memory Controller
New
AMD HD Media Accelerator (UVD, AMD Accelerated Video Converter)
Unified Northbridge
New
Northern Islands AMD Radeon™ GPU
New
Up to 4 "Piledriver" Cores with 2MB L2
New
HDMI, DisplayPort 1.2, DVI controllers
PCI Express I/O — 24 lanes, optional digital display interfaces


Memory & I/O Interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers


Intel® Core Processor
Intel® Processor Graphics Gen9
Command Streamer
Global Thread Dispatcher
Rendering fixed function units
Slice: 24 EUs
Fixed function units
Subslice: 8 EUs
Subslice: 8 EUs
Subslice: 8 EUs
EU
Sampler
L2
Data Port
Atomics, Barriers
L3 Data Cache
Shared Local Memory
GTI: Graphics Technology Interface
CPU core
CPU core
SoC Ring Interconnect
LLC cache slice
LLC cache slice
System Agent
Display Controller
Memory Controller
PCIe
(Opt) EDRAM Controller
- clock domain


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
L2 Cache
High-Speed Hub
NVLink

Et du point de vue du logiciel ?




L'image de la course de voiture

Plusieurs véhicules veulent aller d'un point A à un point B le plus vite possible, ils peuvent :

- ▷ faire la course sur la route et finir par :
  - ◇ soit se **suivre** les uns les autres ;
  - ◇ soit essayer de se **voler** mutuellement leurs positions respectives ;
  - ◇ soit avoir un **accident** !
- ▷ rouler sur différentes voies parallèles et arrivés ensemble s**ans entrer en collision** ;
- ▷ emprunter des **routes différentes** pour aller de A à B.

Et le parallélisme ?

- ▷ Plusieurs tâches à réaliser : chaque voiture à acheminer ;
- ▷ Chacune de ses tâches peut s'exécuter :
  - ◇ une à la fois sur un **seul processeur** : une **seule route** ;
  - ◇ en paralléle sur **plusieurs processeurs** : **plusieurs voies** sur la même route ;
  - ◇ de manière **distribuées** sur plusieurs processeurs : des **routes séparées**.
- ▷ Ces tâches nécessitent souvent d'être **synchronisées** pour éviter les collisions ou de **s'arrêter** à des feux de trafic ou bien à des panneaux de signalisation (Stop).

On peut imaginer que :

- ☐ les voitures sont des **processus** ou **threads** ;
- ☐ les routes qu'elles veulent emprunter sont des **applications** ;
- ☐ la carte des routes correspond au **matériel** ;
- ☐ et le code de la route, aux **communications** et aux **synchronisations**.

Objectif

L'objectif du parallélisme est de :

- ◇ obtenir de **meilleures performances** par rapport aux calculateurs séquentiels et vectoriels (effet pipeline essentiellement).
- ◇ traiter plus vite des **problèmes plus gros** (les machines à mémoire distribuées permettent de traiter des problèmes plus gros).

De manière informelle, une **machine parallèle** est composée de :

- ⋆ un ensemble **d'unités de calcul** (processeur) ;
- ⋆ une **mémoire** (unité de stockage) disponible :
  - ◇ soit de manière **partagée** ;
  - ◇ soit de manière **distribuée**.

Méthodologie

Un **problème original** devra être **découpé** en un certain nombre de **sous problèmes indépendants** :

- ▷ résolus **simultanément** (en parallèle)
- ▷ dont les solutions **seront combinées** pour avoir la **solution du problème original**.

Remarques

- □ La méthode est proche de celle «*diviser pour résoudre*» ⟹ algorithme **récursif**, entités indépendantes.
- □ La combinaison pose le **problème des échanges** entre unités de calcul.

Définition

Un **programme concurrent** peut contenir deux ou plus processus qui travaillent ensemble pour réaliser une tâche. *Chaque processus est un programme séquentiel ou séquence d'instructions qui sont exécutées les unes après les autres.*

- ▷ Un «*programme séquentiel*» correspond à un **seul fil de contrôle** : «*one thread of control*» ;
- ▷ Un «*programme concurrent*» possèdent **plusieurs fils de contrôle** : «*multi-threaded*» ;

Thread ou processus ?

Un processus est un programme s'exécutant au niveau d'un OS.
Une **thread** est un programme s'exécutant dans un autre programme qui est considéré comme un processus pour l'OS qui l'exécute : on parle de processus de poids léger «*lightweight processus*».

Comment travailler ensemble ?

Les processus dans un programme concurrent travaillent ensemble en communiquant les uns avec les autres.
Ces communications sont réalisées par :

- ▷ **variables partagées** : un processus écrit dans une variable qui est lue par un autre ;
- ▷ **échange de message** : un processus envoi un message qui est reçu par un autre.

Comment communiquer ?

Quelque soit la forme de communication choisie, les processus ont **besoin de se synchroniser** les uns avec les autres.



Comment se synchroniser ?

- **exclusion mutuelle** : c'est le problème de **garantir que des instructions en section critique** ne peuvent s'exécuter simultanément ;
- **synchronisation conditionnelle** : c'est le problème de **retarder un processus** jusqu'à ce qu'une condition soit vraie.

Exemple

Modèle du «*Producteur/Consommateur*» qui communiquent au travers d'une variable partagée (buffer partagé) :

- ▷ le **producteur** écrit dans le buffer ;
- ▷ le **consommateur** lit depuis le buffer.

L'exclusion mutuelle est nécessaire pour assurer que le producteur et le consommateur **n'accède pas en même temps**, permettant par exemple qu'un message écrit partiellement soit lu prématurément.

La synchronisation conditionnelle est utilisée pour **garantir qu'un message n'est pas lu** par le consommateur avant qu'il ne soit entièrement écrit par le producteur.



Vers 1960...

L'histoire de la **programmation concurrente** est liée à celle des ordinateurs.

Son émergence est liée à celle des **OS** et à l'invention des **contrôleurs de périphériques** («*device controllers*») :
- ils fonctionnent **indépendamment** du processeur central ;
- ils permettent d'effectuer des opérations d'E/S en **concurrence** d'un programme exécuté par le **processeur central**.

*Le contrôleur communique avec le processeur central par l'intermédiaire d'une **interruption**, un signal matériel qui le déroute de l'exécution de la séquence d'instructions courante pour exécuter une séquence d'instructions différente.*

Problème ?

l'intégration de contrôleur de périphérique pose le problème que certaines parties d'un programme peuvent s'exécuter dans un **ordre imprévisible** !

Ainsi, si un programme est en train de **modifier la valeur d'une variable**, une **interruption** peut arriver et peut conduire à ce qu'une autre partie du programme essaie de changer la valeur de cette **même variable** (**notion de section critique**).

Les machines multi processeurs

Il a été très vite possible de construire des machines possédant **plusieurs processeurs**.

Ces machines permettent d'exécuter un seul programme plus rapidement à condition de le réécrire pour utiliser plusieurs processeurs à la fois....

Mais :
- ▷ comment **synchroniser l'activité** de ces différents processeurs ?
- ▷ comment **utiliser plusieurs processeurs** pour accélérer un programme ?



Alors ?



Plusieurs niveaux de parallèlisme

machines

⟹architectures distribuées

processeurs

⟹ machines multi-processeurs

unités de calcul

⟹processeurs superscalaires

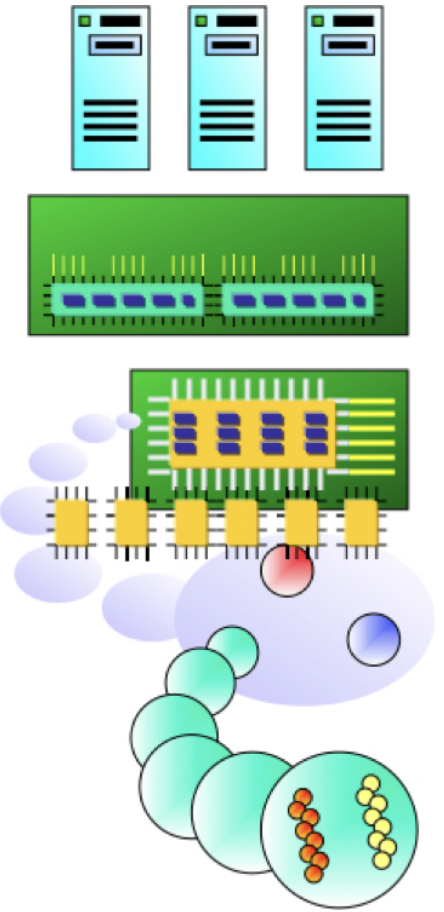
processus

⟹multi-programmation

⟹temps partagé

threads ou processus de poids léger

⟹ multi-programmation à **grain fin**



Comment Paralléliser un algorithme ?

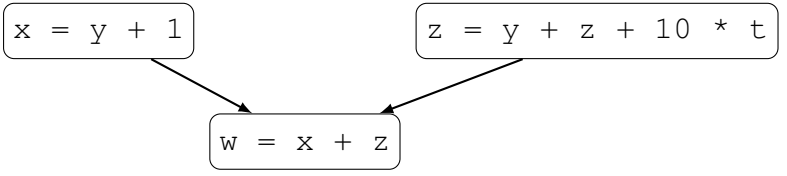


```
❶ x := y + 1
❷ z := y + z + 10 * t
❸ w := x + z
```

L'obtention du résultat à partir de x, y et z nécessite **plusieurs étapes** dont certaines peuvent **s'exécuter en parallèle**.

Graphe de précédence

- les nœuds sont les **opérations à réaliser** pour résoudre le problème ;
- les **arcs orientés** sont les **contraintes de précédence** entre les opérations.



⟹On obtient un **ordre partiel** et les **opérations non ordonnées** par cet ordre partiel sont :

▷ **indépendantes**

▷ capables de s'exécuter **en parallèle**.

Sur l'exemple :

▷ ❶ *et* ❷ *peuvent s'exécuter en parallèle ;*

▷ *par contre* ❸ *doit attendre la fin de* ❶ *et* ❷ *avant de débuter.*

Le **graphe de précédence** donne une **analyse statique** du **parallélisme fonctionnel** exploitable :

- la **longueur de son plus long chemin** donne le nombre d'opérations **nécessairement séquentielles** ;
- la **largeur du graphe** donne le **nombre maximum d'opérations exécutables en parallèle** (le degré moyen des nœuds du graphe est également important).

Deux types de source de parallélisme

- le **parallélisme fonctionnel** ou **parallélisme de contrôle** ;
- le **parallélisme de données**.

Parallélisme fonctionnel

Il correspond à :

- découper un problème en **tâches** (opérations élémentaires) ;
- indiquer **l'ordonnancement** de ces tâches (graphe de précédence).

Exemple : produit itératif de n éléments

```
P := a(0) ;
Pour i in 1 .. n - 1 faire
    P := P * a(i) ;
```

P est le produit du premier élément avec le produit des $n-1$ éléments.

Analyse :

- le temps d'exécution est linéaire en n, soit $O(n)$;
- son **graphe de précédence** est une **chaîne** : il ne peut être parallélisable.

Pourtant il est **facile** de concevoir un **algorithme parallèle** à l'aide de n processus en $O(log_2(n))$.

Mais ici, on exploite une **propriété de la multiplication** : son **associativité**.

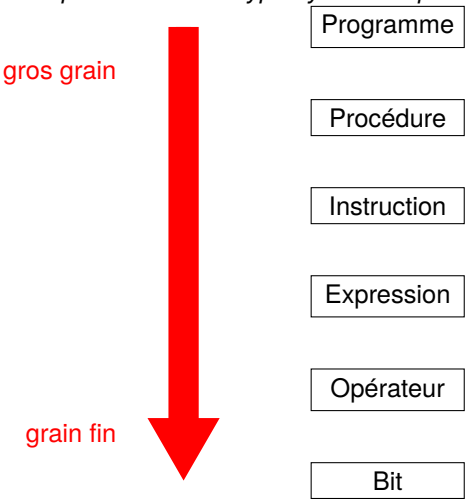


Exploitation (automatique) du Parallélisme



Définition

Le «*grain*» correspond à la taille moyenne des tâches élémentaires.
Le choix du grain lié à l'architecture de la machine (quel type de grain peut être exploité efficacement ?).
Si le grain = programme alors on exploite du parallélisme de type système réparti.



Le «*degré*» de parallélisme reflète le **nombre de processeurs** pouvant être utilisés (degré 1 pour les parties purement séquentielles).
Il peut varier dans les différentes parties des programmes : *degré maximal, minimal, moyen d'un programme*.
Le degré maximal constitue une **borne supérieure** au nombre de processeur qu'on utilisera : avec ce nombre exécution rapide mais efficacité médiocre
en effet si le degré max > degré moyen alors des processeurs seront inactifs pendant une partie de l'exécution du programme

Parallélisme de données

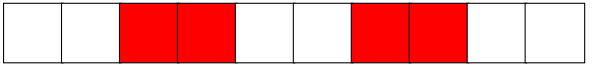
- ▷ les données sont souvent plus nombreuses que les processeurs.
- ▷ la régularité de structures de données permet de distribuer de façon régulière ces données sur les différents processeurs : par exemple pour une matrice $n * n$ sur p processeurs :
  - ◇ chaque processeur possède n/p lignes ;
  - ◇ le processeur est dit **propriétaire** de ces lignes ;
  - ◇ il est **responsable** de la réalisation de toutes les tâches les concernant.

Les **schémas de distribution** les plus classiques sont :

- ☐ répartition par **bloc** :

- ☐ répartition **cyclique** :

- ☐ répartition par **blocs cycliques** :



Exécution synchrone

Le parallélisme de données s'adapte facilement sur un modèle SIMD avec une grande importance donnée à la **synchronisation**.

Exemple :

```
pour i = 0 à n - 1 faire en parallèle
   a[i] = 2.i
pour i = 0 à n - 1 faire en parallèle
   b[i] = a [i] + a[n - 1 - i]
```

Répartition par bloc :

- ▷ `b[0]` est réalisé sur P_0 et nécessite : `a[0]` et `a[n - 1]`
- ▷ **a[0]** calculé sur P_0 mais `a[n-1]` sur P_{n-1}
- ▷ **attention** : pour `b[0]` certaines valeurs de a doivent être disponibles
- ▷ la **synchronisation** est **assurée** par l'utilisation du **modèle SIMD**.

Utilisation de machine du modèle MIMD

Il est possible d'exploiter le **parallélisme de données** sur des machines MIMD :

- ▷ il faut veiller à ce que les processeurs travaillent de **manière coordonnée**

Dans l'exemple, il suffit d'attendre que tous les processeurs aient fini de calculer les valeurs du vecteur `a` avant que l'un d'entre eux ne commence le calcul de `b`.

On **synchronise** donc l'ensemble des processeurs **entre les deux boucles**.

Le parallélisme de données sur des machines MIMD implique un travail de synchronisation de la part du programmeur.

Dépendance des données

Reprenons l'exemple précédent dans une forme *condensée* :

```
a[0]  = 0
sp[0] = a[0]
pour i = 1 à n - 1 faire
     a[i]  = 2.i
     sp[i] = a[i] + sp [i - 1]
```

Le calcul des éléments du vecteur sp ne peut être fait en parallèle...mais pourquoi ?

Notions de variables lus et modifiées pour une instruction S

- ▷ $L(S)$ ensemble des variables utilisées en **lecture** ;
- ▷ $M(S)$ ensemble des variables qui subissent une **modification** (une affectation).

Condition de Bernstein pour assurer l'indépendance en deux instructions séquentielles S_1 & S_2

- □ $M(S_1) \cap L(S_2) = \emptyset$: dépendance vraie, ou RAW, «S_2*Read-After-*S_1*Write*»
- □ $L(S_1) \cap M(S_2) = \emptyset$: anti-dépendance, ou WAR, «S_2*Write-After-*S_1*Read*»
- □ $M(S_1) \cap M(S_2) = \emptyset$: dépendance de sortie, ou WAW, «S_2*Write-After-*S_1*Write*»

*Ces 3 équations doivent être **vérifiées** pour pouvoir réaliser S_1 et S_2 en **parallèle**.*

Il est possible de définir :

- ▷ S comme une **séquence d'instructions** au lieu d'une seule instruction ;
- ▷ les ensembles L et M en les calculant sur les instructions de chaque séquence.

La **condition de Bernstein** exprime la possibilité de calculer en **parallèle** les deux séquences d'instructions S_1 et S_2.

Dépendance des données

Exemple :

```
a[0]  = 0
sp[0] = a[0]
pour i = 1 à n - 1 faire
     a[i]  = 2.i
     sp[i] = a[i] + sp [i - 1]
```

Le calcul des éléments du vecteur sp ne peut être fait en parallèle...mais pourquoi ?

Condition de Bernstein pour assurer l'indépendance en deux instructions séquentielles S_1 & S_2

- $M(S_1) \cap L(S_2) = \emptyset$: dépendance vraie
- $L(S_1) \cap M(S_2) = \emptyset$: anti-dépendance
- $M(S_1) \cap M(S_2) = \emptyset$: dépendance de sortie

*Ces 3 équations doivent être **vérifiées** pour pouvoir réaliser S_1 et S_2 en **parallèle**.*

Exemple

On numérote S_1, et S_2 les instructions de la boucle :

xterm

```
a[0]  = 0
sp[0] = a[0]
pour i = 1 à n - 1 faire
   S1: a[i] = 2.i
   S2: sp[i] = a[i] + sp [i - 1]
```

- $M(S_1) = \{a[i]\}$
- $L(S_2) = \{a[i]\}$

} *alors* $M(S_1) \cap L(S_2) \neq \emptyset$, *on a une* «dépendance vraie» *ou RAW.*

Dépendance des données

Exemple :

```
a[0]  = 0
sp[0] = a[0]
pour i = 1 à n - 1 faire
     a[i]  = 2.i
     sp[i] = a[i] + sp [i - 1]
```

Le calcul des éléments du vecteur sp ne peut être fait en parallèle...mais pourquoi ?

Condition de Bernstein pour assurer l'indépendance en deux instructions séquentielles S_1 & S_2

- $M(S_1) \cap L(S_2) = \emptyset$: dépendance vraie
- $L(S_1) \cap M(S_2) = \emptyset$: anti-dépendance
- $M(S_1) \cap M(S_2) = \emptyset$: dépendance de sortie

*Ces 3 équations doivent être **vérifiées** pour pouvoir réaliser S_1 et S_2 en **parallèle**.*

Exemple

On note S_1 les instructions de la boucle et S_2 les instructions de l'occurrence suivante :

xterm

```
a[0] = 0
sp[0] = a[0]
pour i = 1 à n - 1 faire        Soit j une occurrence de la boucle
   S1: a[j] = 2.j
         sp[j] = a[j] + sp [j - 1]
   S2: a[j+1] = 2.(j+1)
         sp[j+1] = a[j+1] + sp [j]
```

«*Dépendance vraie*» entre les instances de l'instruction S_1 (pour j) et S_2 (pour $j+1$) dans la boucle $M(S_1) \cap L(S_2) = \{sp[j]\} \neq \emptyset \Longrightarrow$ Condition de Bernstein **non vérifiée**

Modifications pour tirer parti du parallélisme

xterm
```
a[0] = 0
sp[0] = a[0]
pour i = 0 à n - 1 faire
   a[i] = 2.i
   sp[i] = a[i] + sp [i - 1]
```

Peut être réécrit en :

xterm
```
a[0] = 0
sp[0] = a[0]
pour i = 0 à n - 1 faire
   a[i] = 2.i
pour i = 0 à n - 1 faire
   sp[i] = a[i] + sp [i - 1]
```

Et enfin :

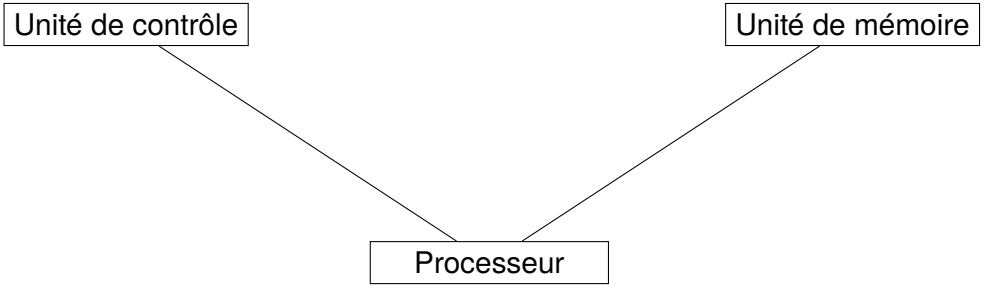
xterm
```
a[0] = 0
sp[0] = a[0]
pour i = 1 à n-1 faire_en_parallèle
   a[i] = 2.i

pour i = 1 à n - 1 faire
   sp[i] = a[i] + sp [i - 1]
```

Au final :

- ▷ Le **première** boucle peut être **exécutée en parallèle !**
- ▷ La **seconde** boucle établie une **dépendance temporelle** entre une occurrence de la boucle et sa suivante :
  $\Longrightarrow$**pas d'exécution parallèle possible !**


Et par rapport aux architectures parallèles ?

- un flot d'instructions ;
- un flot de données ;



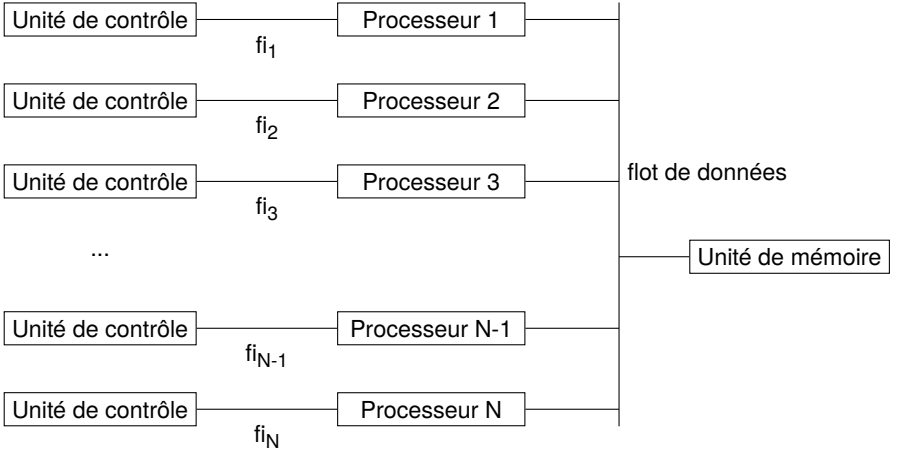
- ▷ À chaque étape de calcul, l'unité de contrôle produit une instruction qui opère sur une donnée obtenue à partir de l'unité mémoire.
- ▷ Il n'y a qu'un seul processeur⟹**modèle séquentiel**.

A chaque étape, une donnée est :

- fournie par l'unité mémoire ;
- traitée parallèlement par les N processeurs qui exécutent chacun une instruction spécifique.

On a plusieurs **flots d'instructions**, fi, opérant sur un seul **flot de données**.

On a N processeurs, $N > 1$, chacun ayant son **unité de contrôle**, partageant une unique **unité mémoire**.



Exemple

On veut savoir si un nombre z est premier.

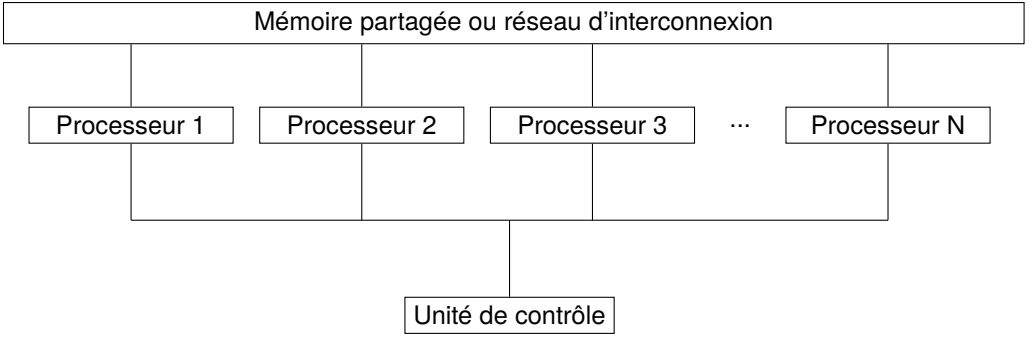
On peut associer à chaque processeur un diviseur potentiel de E (nombres entre z et z-1).

Un processeur :

- reçoit z ;
- fait la division par son le diviseur potentiel ;
- répond par « oui » ou « non » dans une zone mémoire fixe, connue par tous les processeurs.

Il n'existe pas de calculateur fonctionnant avec ce modèle !


On dispose de N processeurs, tous identiques ayant chacun une mémoire locale où on peut stocker des données et des instructions.



On suppose la mémoire locale intégrée au processeur.

Tous les processeurs opèrent sous le contrôle d'un unique flot d'instruction (ceci est équivalent à avoir dans chaque mémoire locale une copie d'un unique programme).

Les processeurs opèrent de manière **synchrone** (horloge globale) sur les N flots de données (à chaque pas de temps, ils exécutent tous la même instruction).

À certaines étapes, certains processeurs peuvent rester inactifs, en fait ils exécutent « une instruction d'attente » ayant la durée voulue (nécessité de conserver une totale synchronisation entre tous les processeurs.

Les processeurs communiquent (échangent des données) pour coordonner leurs résultats. Ceci se fait de deux manières différentes :

- **mémoire partagée** SM SIMD (shared memory) ;
- **réseau d'interconnexion** (interconnection network).

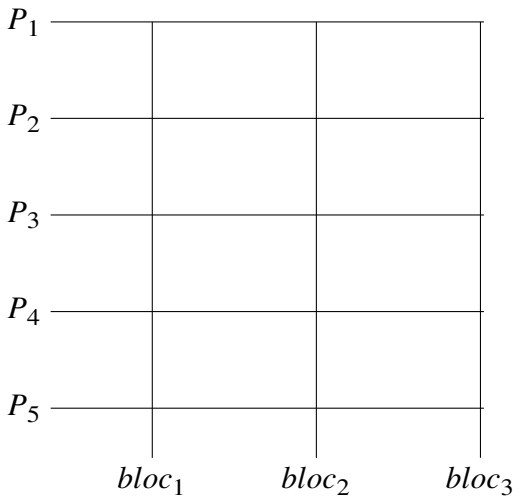
Ce modèle est très puissant mais il est peut réaliste à cause de la complexité des circuits d'accès à la mémoire.

Si la mémoire possède M zones élémentaires, le coût du circuit permettant un calcul d'adresse pour un processeur varie comme $f(M)$ ou f est croissante.

Dans le cas de N processeurs partageant la mémoire, le coût du circuit global varie comme $N * f(M)$.

$\Longrightarrow$Ceci est **irréaliste** pour N et M grands.

Une solution pour diminuer ce coût est de diviser la mémoire en R blocs de taille M/R chacun.



La **concurrence** se fait au niveau du bloc.

Un processeur quelconque peut accéder à un bloc quelconque.

Il y a $N * R$ «*switches*» : un à chaque intersection bus mémoire/bus processeur.

L'accès se fait par une logique « ligne-colonne » à travers les switches.

Chaque bloc de mémoire dispose d'un circuit d'adressage interne (M/R possibilités)

Le coût est de : $R * f(M/R) + (N * R) *$ « coût d'un switch »
Le coût d'un switch est négligeable.

$\Longrightarrow$Le modèle est **moins puissant** à cause de l'accès au niveau des blocs.

On veut étendre l'idée précédente :
Les M éléments de mémoire sont distribués sur les N processeurs, chacun ayant localement M/N mémoire.

Chaque **paire de processeurs** est connectée par un lien bidirectionnel et à chaque étape un processeur quelconque P_i peut :

- ▷ recevoir une donnée d'un processeur P_j
- ▷ envoyer un autre message à P_k (k peut être égal à j).

Chaque processeur doit avoir un circuit d'adressage de coût $f(N-1)$ lui permettant de sélectionner n'importe lequel des autres processeurs.

Il doit avoir un circuit d'adressage de coût $f(M/N)$ pour permettre un accès à sa propre mémoire.

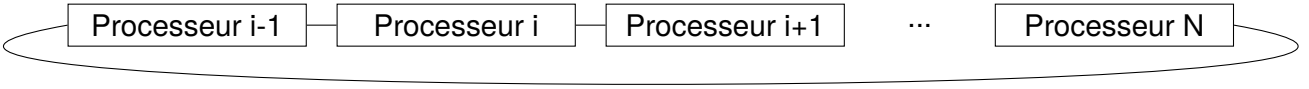
On aura en tout :

$$N * (f(N-1) + f(M/N))$$

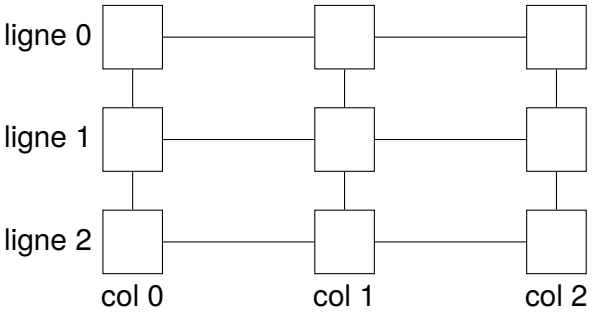
Le coût est **trop important** mais le modèle est **plus puissant** que celui avec découpage en blocs de la mémoire.

$\Longrightarrow$On a besoin de réseaux « **faiblement couplés** » par opposition au **graphe complet**.

La chaîne «*1D*»



La grille «*2D*»

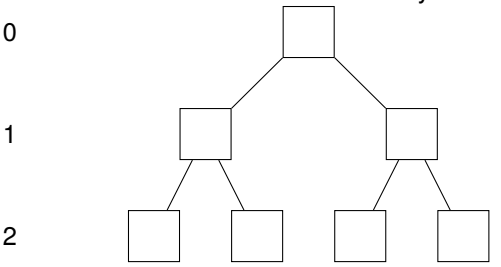


Processeur $P_{i,j}$:

- ligne i ;
- colonne j ;

L'arbre binaire

On a un arbre binaire complet avec d niveaux numérotés de 0 à $d-1$ et il y a $N = 2d-1$ processeurs.



Le «*Perfect Shuffle*» ou mélange parfait

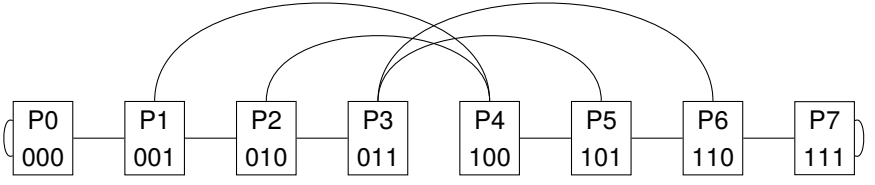
On a N processeurs numérotés P_0, ... , P_{N-1}, avec $N = 2^P$

P_i est connecté à P_j si et seulement si :

▷ $j = 2 * i$ pour $0 \leq i \leq N/2-1$;

▷ $j = 2 * i + 1-N$ pour $N/2 \leq i \leq N-1$;

$\Longrightarrow$L'écriture binaire de j se déduit de celle de i par un décalage circulaire à gauche : liens « shuffle ».



On rajoute des liens d'échanges entre tout processeur ayant un numéro pair et son suivant.

On a le « shuffle exchange ».

Ce type de réseau est bien adapté pour le calcul de la **transformée de Fourier rapide** ou FFT en parallèle.

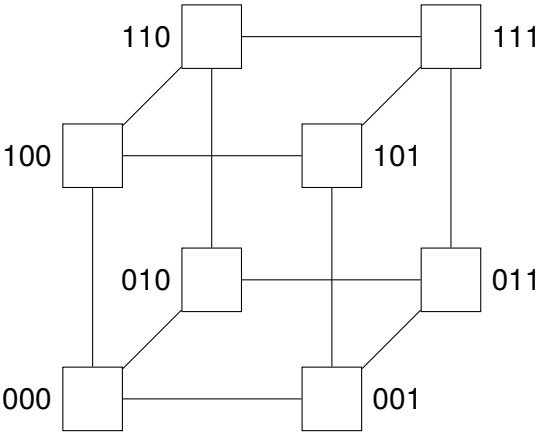
Le cube

On considère $N = 2^q$ processeurs numérotés $P_0, P_1, ..., P_{N-1}$ $(q \geq 1)$

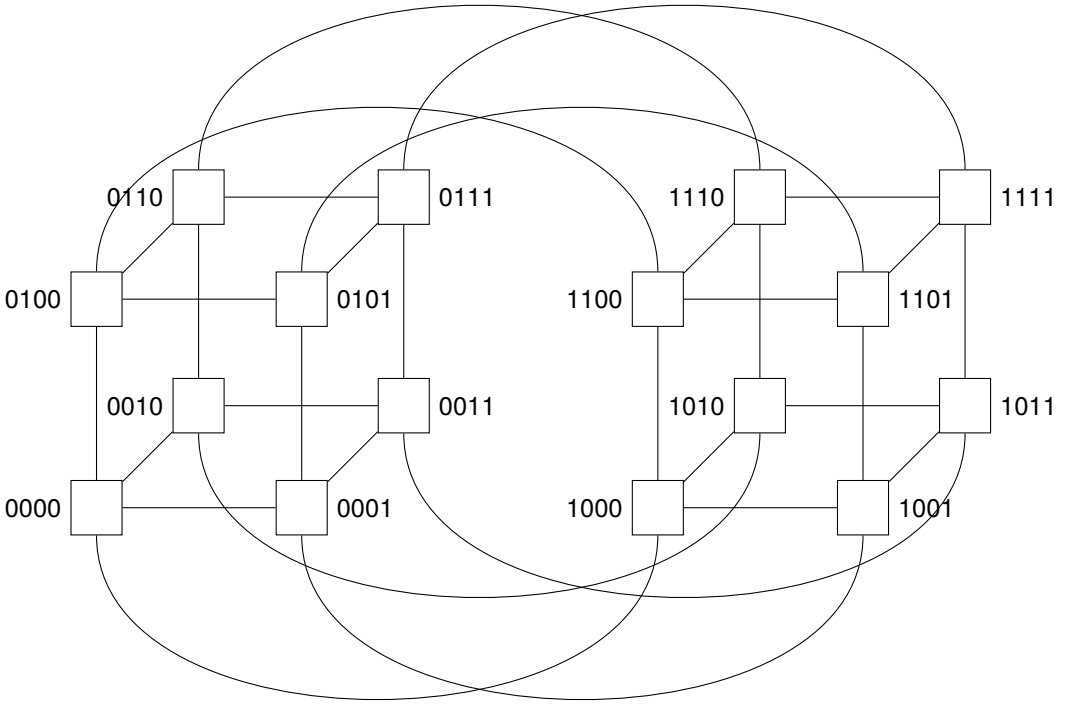
Un cube de dimension q (hypercube) est obtenu en connectant un sommet à exactement q voisins.

Ce qui donne un degré logarithmique.

Pi est connecté à Pj si et seulement si les écritures binaires de i et de j ne différents que d'un chiffre binaire.



Le cube de degré 4



Exemple sur l'arbre binaire

On veut faire la somme de n nombres $x_1, x_2, ... ,x_n$.

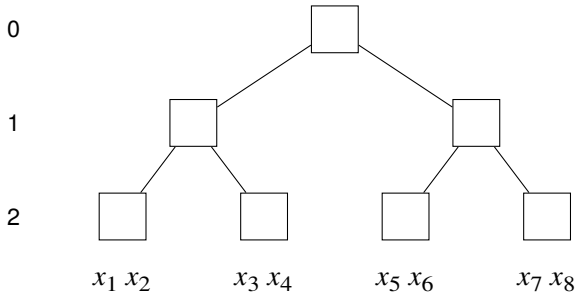
Dans un modèle SISD, il faut n étapes ($n-1$ additions et 1 affectation).

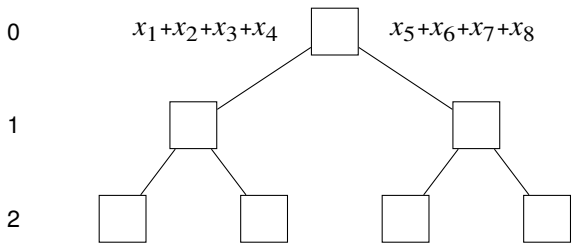
En utilisant un arbre ayant $log_2(n)$ niveaux et $n/2$ feuilles, le calcul peut se faire en $log_2(n)$ étapes.

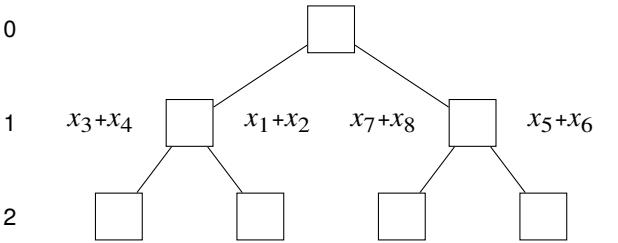
Chaque feuille a au départ 2 nombres, elle fait la somme et l'envoie au père.

Pour les processeurs des autres niveaux, on fait la chose suivante :

- ▷ recevoir une donnée de chacun des deux fils ;
- ▷ faire la comme ;
- ▷ envoyer le résultat au père.







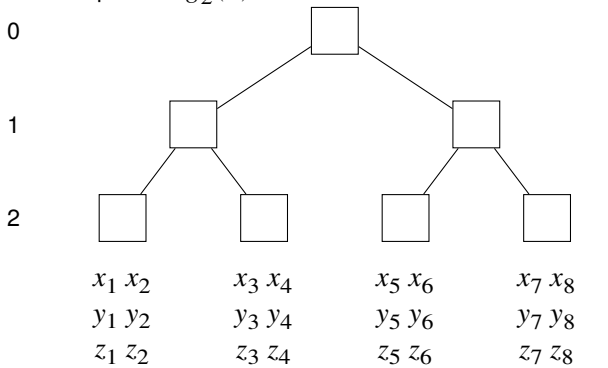
La racine n'a plus qu'à faire la **somme finale** :

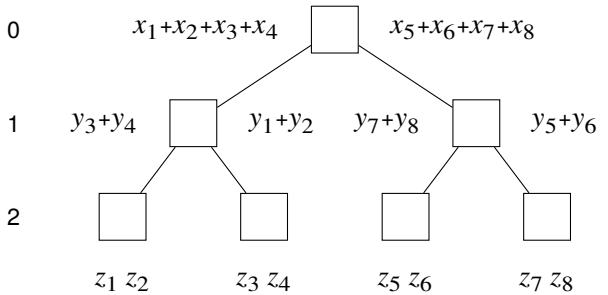
$$x_1+x_2+x_3+x_4 + x_5+x_6+x_7+x_8$$

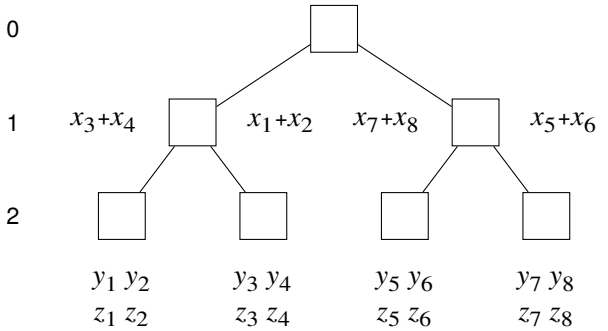
Effet pipeline

Si on veut faire m sommations de n nombres, on « pipeline » les différents calculs.

Le calcul prend $log_2(n) + m - 1$.







La racine fait la première **somme** :

$$x_1+x_2+x_3+x_4 + x_5+x_6+x_7+x_8$$

Puis à l'étape suivante, la seconde somme :

$$y_1+y_2+y_3+y_4 + y_5+y_6+y_7+y_8$$

Puis à l'étape suivante, la troisième somme :

$$z_1+z_2+z_3+z_4 + z_5+z_6+z_7+z_8$$ *etc.*