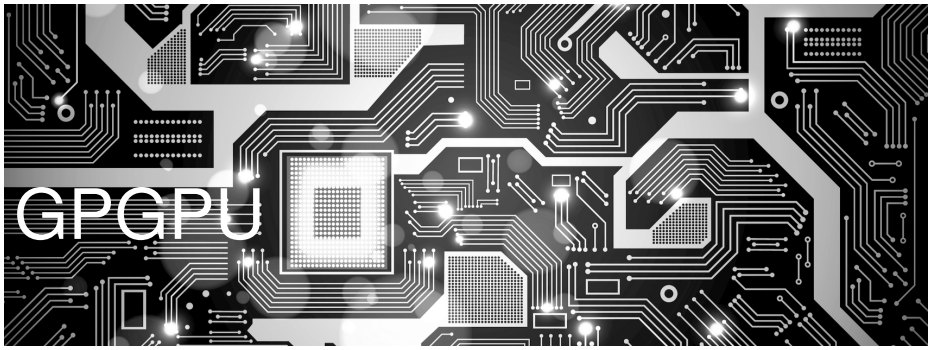
Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année



Développement GPGPU — P-F. Bonnefoi

Version du 11 décembre 2022

Table des matières

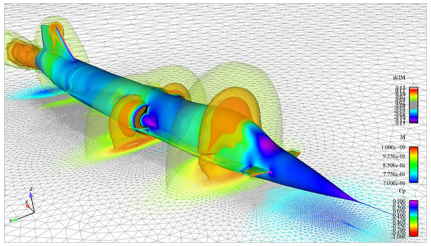
Quels sont les besoins ?

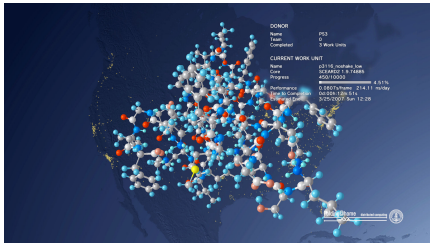
1 Pourquoi du parallélisme ?

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.



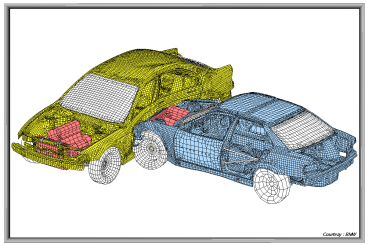


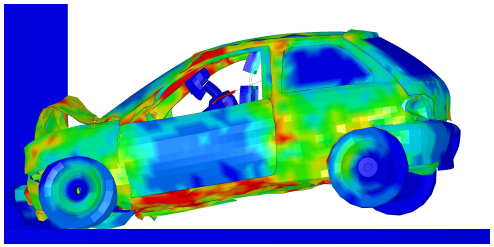
En **puissance de traitement** :
- base de données
- serveurs multimédia
- Internet

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

Besoin en parallélisme

Industrie automobile

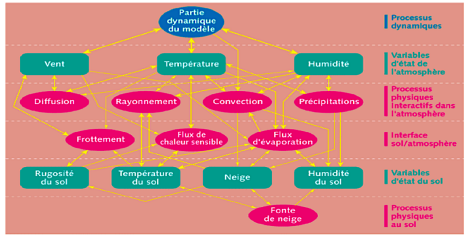




Industrie des effets spéciaux

Besoin de parallélisme

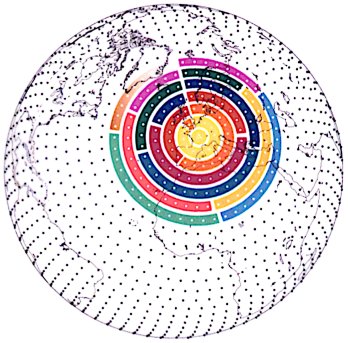
Météorologie : Modèle Arpège 1998



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).



Besoin de parallélisme

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 ;

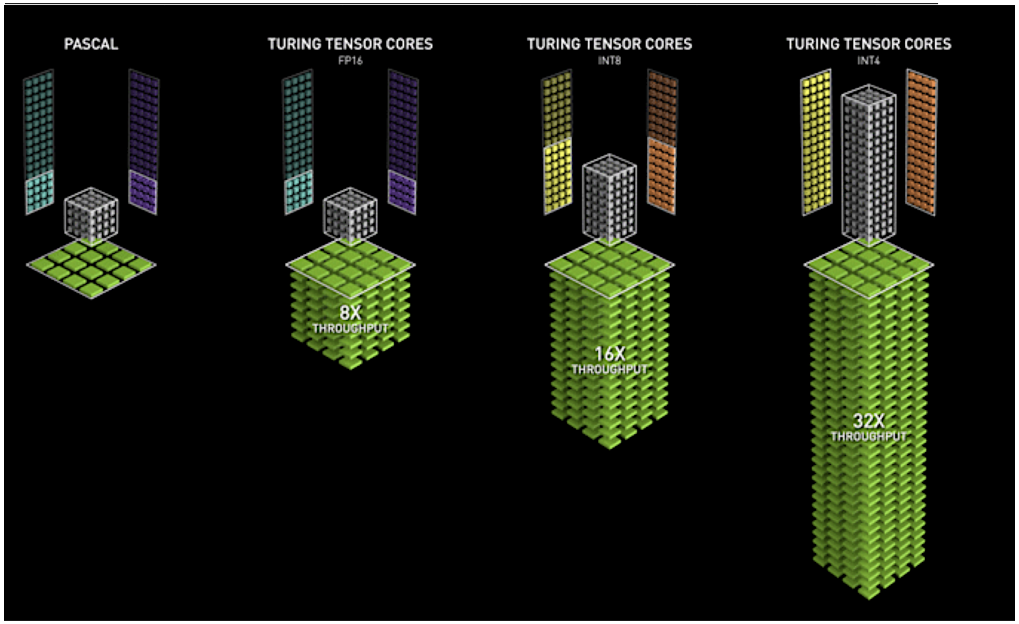
Les multiplicatifs : Kilo = 2^{10} ; Mega = 2^{20} ; Giga = 2^{30} ; Tera 2^{40} ; Peta 2^{50} ; Exa 2^{60}

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**.

Intelligence artificielle : calculs sur des tensors



Et le matériel ?
Quels sont les ordinateurs parallèles ?

2 Historique

1950 → 1970 : les pionniers

CDC 6600, (1964) :
- unités de calcul en parallèle,
- 10MHz,
- 2Mo,
- 3 MFlops

Utilisé par Niklaus Wirth pour définir Pascal

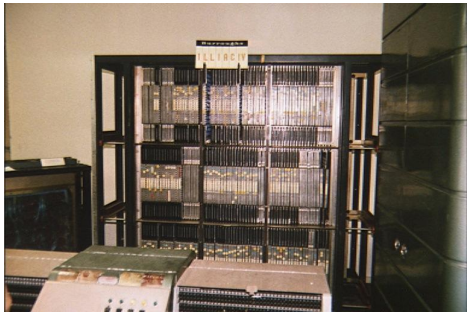
CDC7600 (1969) :
équivalent à 7 CDC6600 : 21 MFlops

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).

Historique

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 !

Historique

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.

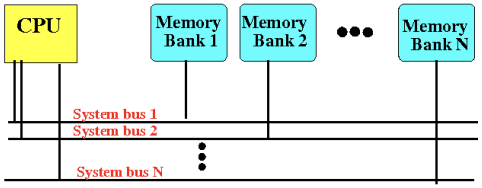
Viabilité des solutions pas toujours très étudiée
- difficultés de mise au point ;
- solutions dépassées dès leur disponibilité.

Une utilisation peu pratique
- systèmes d'exploitation propriétaires ;
- difficulté d'apprentissage.

Manque ou absence d'outils
- difficulté d'exploitation.

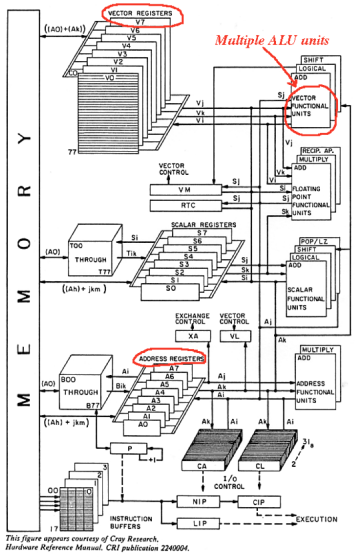
CRAY-1, «*vector computer*»

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.

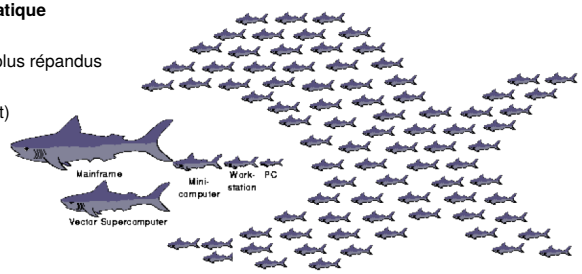
Figure 10.6 CRAY-1's central processor

Historique

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



NOW

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)

3 Les clusters

Par ordre de puissance

`https://www.top500.org/`
Qui va arriver au PétaFlop ?

Rank	Site	Computer	Processors	Year	R_{max}	R_{peak}
1	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution IBM	212992	2007	478200	596378
2	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution IBM	65536	2007	167300	222822
3	SGI/New Mexico Computing Applications Center (NMCAC) United States	SGI Altix ICE 8200, Xeon quad core 3.0 GHz SGI	14336	2007	126900	172032
4	Computational Research Laboratories, TATA SONS India	EKA - Cluster Platform 3000 BL460c, Xeon 53xx 3GHz, Infiniband Hewlett-Packard	14240	2007	117900	170880
5	Government Agency Sweden	Cluster Platform 3000 BL460c, Xeon 53xx 2.66GHz, Infiniband Hewlett-Packard	13728	2007	102800	146430
6	NNSA/Sandia National Laboratories United States	Red Storm - Sandia/ Cray Red Storm, Opteron 2.4 GHz dual core Cray Inc.	26569	2007	102200	127531
7	Oak Ridge National Laboratory United States	Jaguar - Cray XT4/XT3 Cray Inc.	23016	2006	101700	119350

...and the winner is :

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz , Voltaire Infiniband / 2008 IBM	129600	1105.00	1456.70	2483.47
2	Oak Ridge National Laboratory United States	Jaguar - Cray XT5 QC 2.3 GHz / 2008 Cray Inc.	150152	1059.00	1381.40	6950.60
3	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz / 2008 SGI	51200	487.01	608.83	2090.00
4	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution / 2007 IBM	212992	478.20	596.38	2329.60
5	Argonne National Laboratory United States	Blue Gene/P Solution / 2007 IBM	163840	450.30	557.06	1260.00

Power data in KW for entire system
Puissance exprimée en Tflop.

En début 2011

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
2	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
3	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	2580.00
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	1192.00	2287.63	1398.61
5	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz / 2010 Cray Inc.	153408	1054.00	1288.63	2910.00
6	Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bullx super-node S6010/S6030 / 2010 Bull SA	138368	1050.00	1254.55	4590.00
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	1042.00	1375.78	2345.50
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	831.70	1028.85	3090.00

En début 2012

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	548352	8162.00	8773.63	9898.56
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
3	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
4	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	2580.00
5	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	1192.00	2287.63	1398.61
6	DOE/NNSA/LANL/SNL United States	Cielo - Cray XE6 8-core 2.4 GHz / 2011 Cray Inc.	142272	1110.00	1365.81	3980.00
7	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon 5570/5670 2.93 Ghz, Infiniband / 2011 SGI	111104	1088.00	1315.33	4102.00
8	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz / 2010 Cray Inc.	153408	1054.00	1288.63	2910.00
9	Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bullx super-node S6010/S6030 / 2010 Bull SA	138368	1050.00	1254.55	4590.00
10	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	1042.00	1375.78	2345.50

2013

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
2	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	16324.8	20132.7	7890
3	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
4	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8162.4	10066.3	3945
5	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393216	4141.2	5033.2	1970
6	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147456	2897.0	3185.1	3423
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi Dell	204900	2660.3	3959.0	
8	National Supercomputing Center in Tianjin China	**Tianhe-1A** - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 NUDT	186368	2566.0	4701.0	4040

Novembre 2014

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
8	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301

Novembre 2015

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	DOE/NNSA/LANL/SNL United States	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	301,056	8,100.9	11,078.9	
7	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325

Novembre 2016

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
2	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P , NUDT National Super Computer Center in Guangzhou China	3,120,000	33,862.7	54,902.4	17,808
3	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209
4	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom , IBM DOE/NNSA/LLNL United States	1,572,864	17,173.2	20,132.7	7,890
5	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/SC/LBNL/NERSC United States	622,336	14,014.7	27,880.7	3,939

Novembre 2017

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
2	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P , NUDT National Super Computer Center in Guangzhou China	3,120,000	33,862.7	54,902.4	17,808
3	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
4	**Gyoukou** - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz , ExaScaler Japan Agency for Marine-Earth Science and Technology Japan	19,860,000	19,135.8	28,192.0	1,350
5	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209

Juin 2018

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,397,824	143,500.0	200,794.9	9,783
2	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384

Juin 2019

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
2	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR , Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	

Juin 2020

Processeurs ARM !

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482

Juin 2022

ExaFlop !

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438

Hum...

Comment s’y retrouver ?

4 Les machines parallèles : différentes architectures

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)

Différentes architectures

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** :

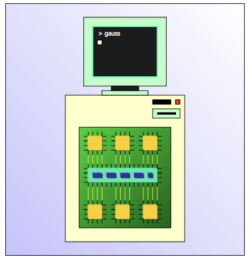
- grand nombre de processeurs ordinaires à mémoire locale
- communication par envoi de messages à travers des réseaux de communication
- chaque processeurs a son propre programme

Les multi processeurs à **mémoire partagée** :

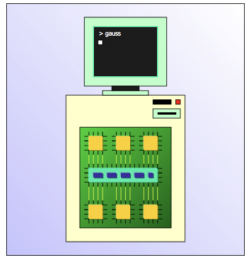
- Si le temps d'accès est égal pour chaque processeur à la mémoire, on parle de UMA, «*Uniform Memory Acess*», ou «*Symmetric Multiprocessors*» (SMP) Exemple : un Core 2 Duo ou multi-cores…
- Si le temps d'accès n'est pas le même on parle de NUMA : «*Non Uniform Memory Access*».

Différentes approches matérielles

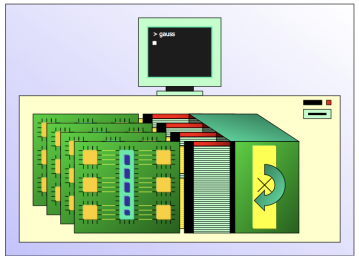
machine à **mémoire partagée**



machine à **mémoire distribuée**



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



Ferme, grappe ou cluster

Un ensemble de machines

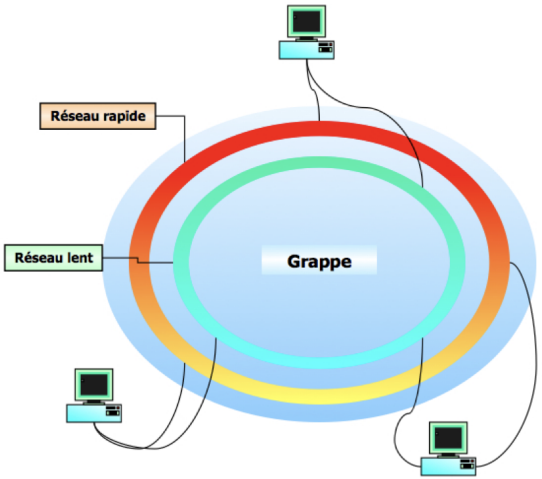
- des PC du commerce

Un réseau classique :

- lent
- réservé à l'administration

Un réseau rapide

- temps de transfert réduit
- débit élevé
- réservé aux applications



Réseau InfiniBand de la société Mellanox

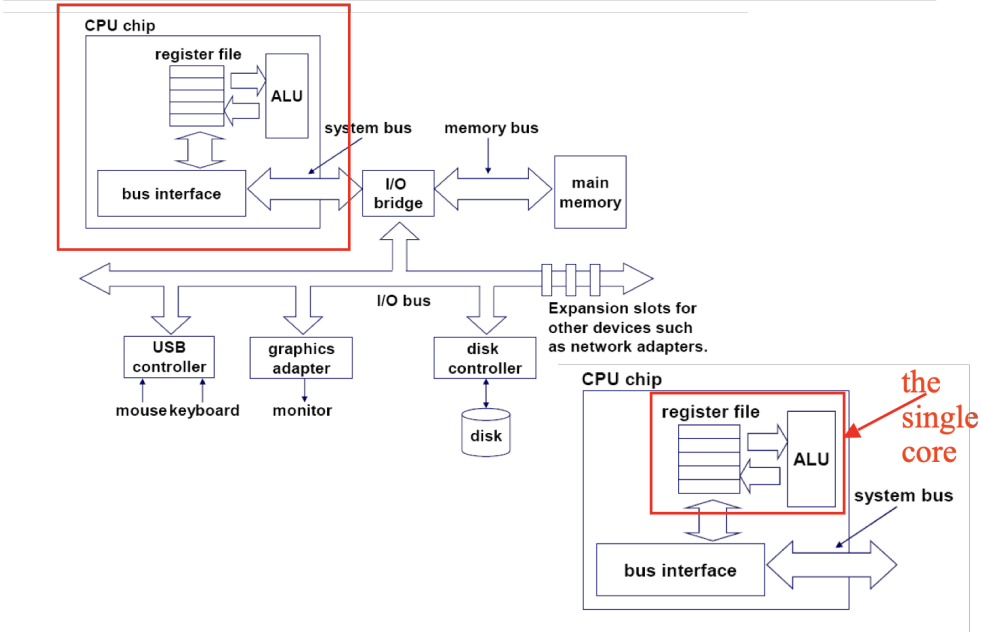
Wikipedia

InfiniBand (IB) is a computer networking communications standard used in high-performance computing that features **very high throughput** and **very low latency**.
As of 2014, it was the most commonly used interconnect in supercomputers. In 2016, Ethernet replaced InfiniBand as the most popular system interconnect of TOP500 supercomputers.

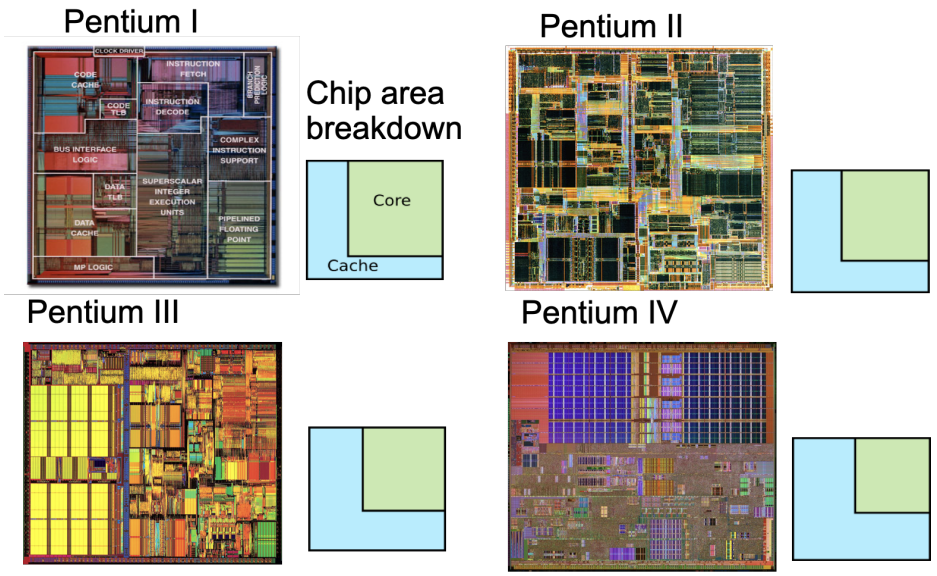
Characteristics										
		SDR	DDR	QDR	FDR10	FDR	EDR	HDR	NDR	XDR
Signaling rate (Gbit/s)		2.5	5	10	10.3125	14.0625	25.78125	50	100	250
Theoretical effective throughput (Gb/s)	for 1 link	2	4	8	10	13.64	25	50	100	250
	for 4 links	8	16	32	40	54.54	100	200	400	1000
	for 8 links	16	32	64	80	109.08	200	400	800	2000
	for 12 links	24	48	96	120	163.64	300	600	1200	3000
Encoding (bits)		8b/10b			64b/66b				PAM4	t.b.d.
Adapter latency (µs)		5	2.5	1.3	0.7	0.7	0.5	less?	t.b.d.	t.b.d.
Year		2001 2003	2005	2007	2011	2011	2014	2018	2021	after 2023?

⟹2019: Nvidia acquired Mellanox for $6.9B

Et les multi-cores ?

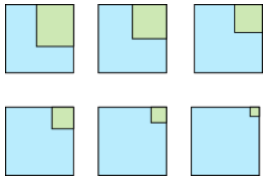


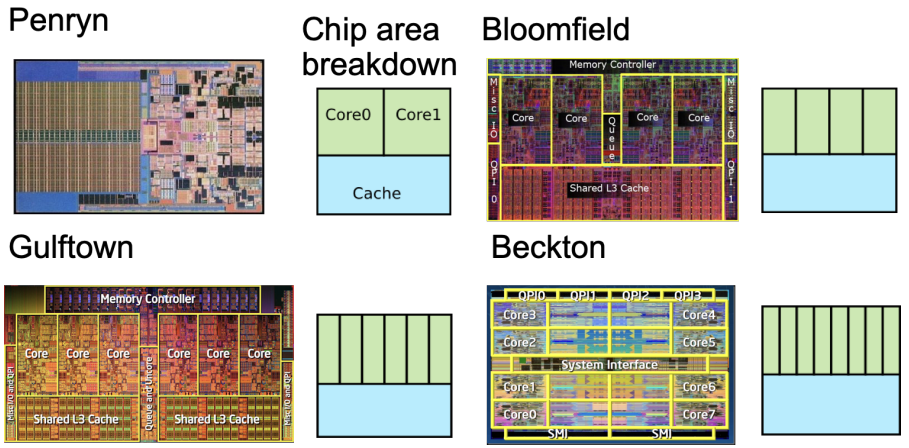
Et les multi-cores ?



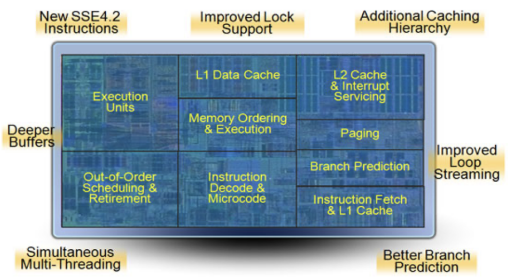
Et les multi-cores ?

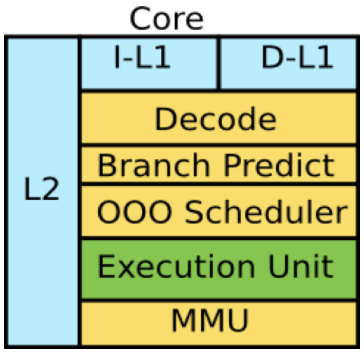
L'avenir ?
Non ! le multi-cœurs :





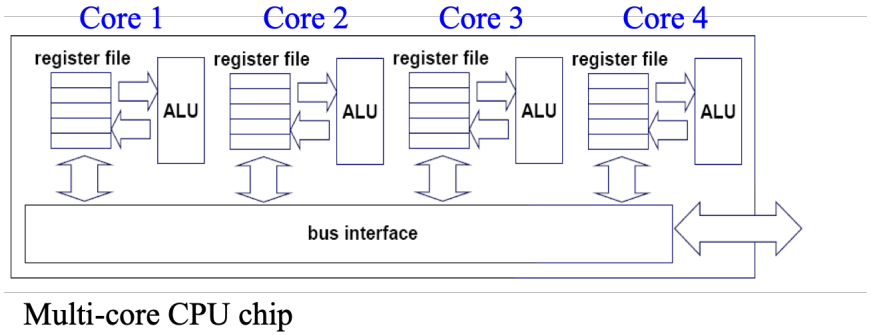
De plus près





Moins de 10%de la surface sert à l'exécution rélle
OOO : «Out Of Order»

Et les «*multi-cores*» ?



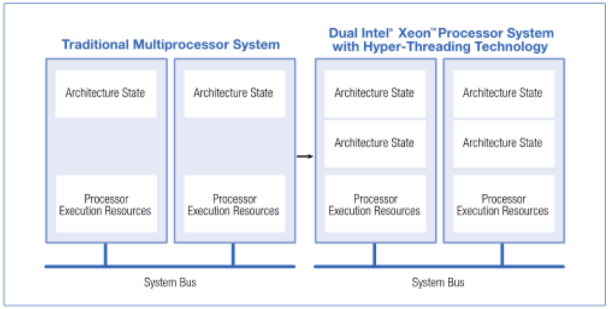
Multi-core CPU chip

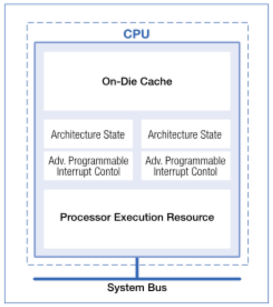
- ▷ On «*grave*» plusieurs processeurs sur le même support.
- ▷ Chaque cœur est vu par le système d'exploitation comme un **processeur séparé**.

Avantages

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

«*Hyperthreading*» ? Qu'est-ce que c'est ?





Hyperthreading : la notion de «*processeur logique*»

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**.

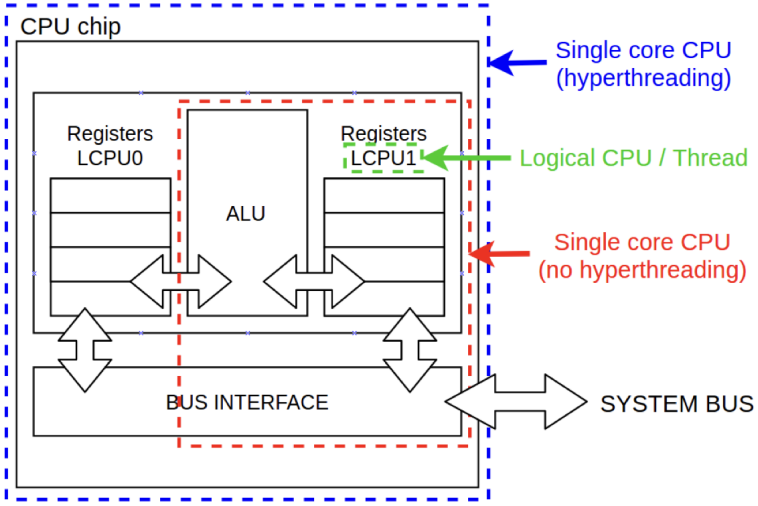
Tous les **processeurs logiques** partagent la partie «*exécution*» :

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

D'après la documentation d'Intel

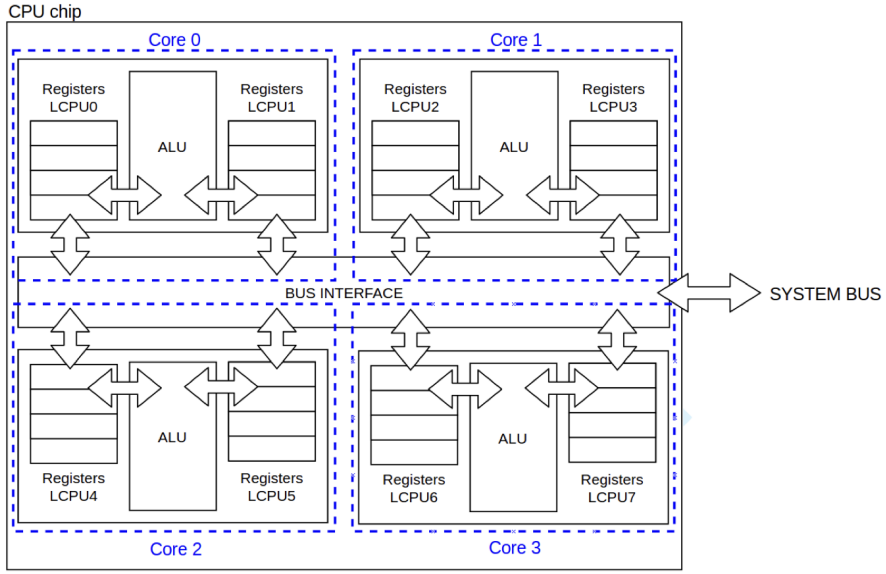
Each logical processor maintains a complete set of the architecture state. The **architecture state** consists of registers including the general-purpose registers, the control registers, the advanced programmable interrupt controller (APIC) registers and some machine-state registers.
From a software 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.

Hyperthreading



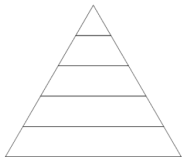
Single core hyperthreading CPU
(2 logical CPU's / threads)

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 $45nm$:

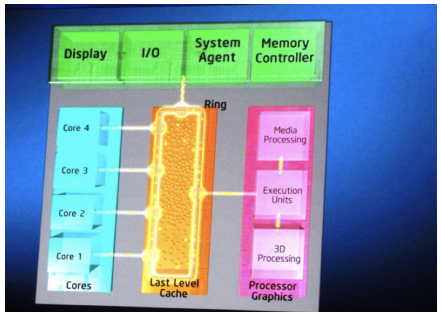
- 64bits Int ADD consomme $1pJ$;
- 64bits FP FMA consomme $200pJ$;

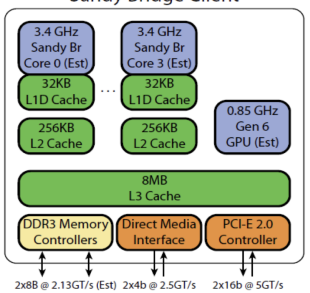
Difficile d'augementer la densité des processeurs : $7nm$ en 2023

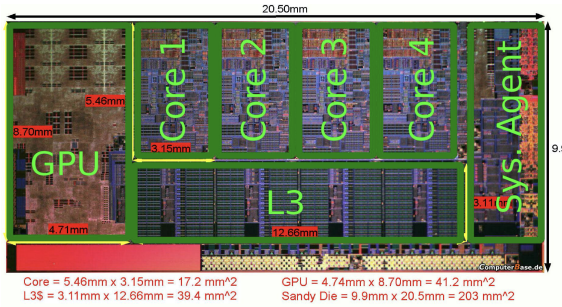
Unreleased Intel Mainstream Desktop CPU Series Specs

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







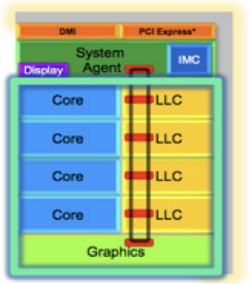
Highlight

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

Sandy Bridge's New CPU-GPU interface

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*)



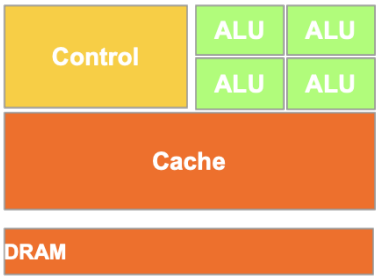
Much higher Graphics performance, DRAM power savings, more DRAM BW available for Cores

IDF2010

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

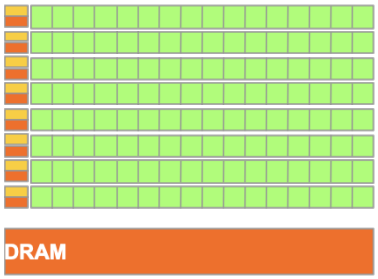
Et les GPUs ?

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.



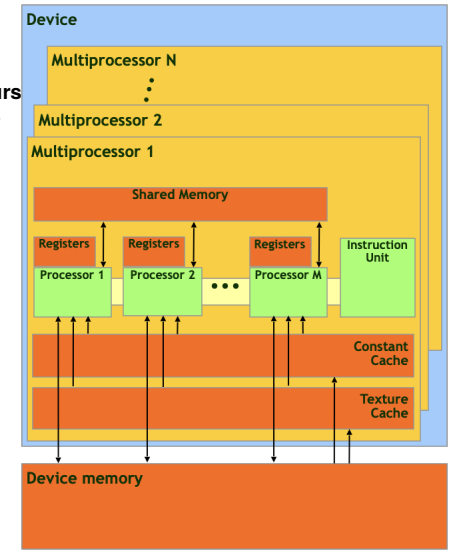
GPU

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

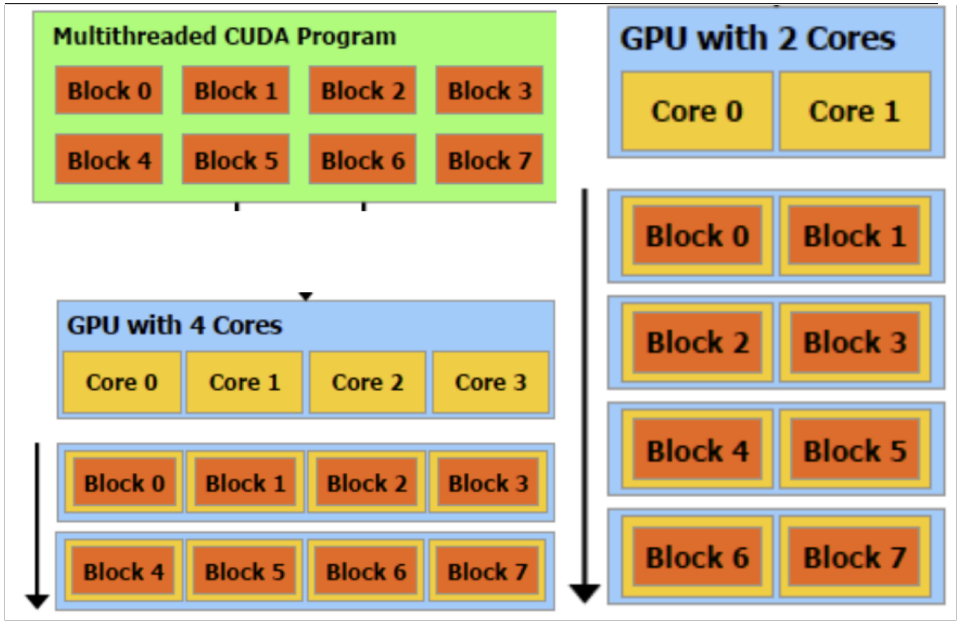
5 Et les GPUs ?

Une architecture complexe

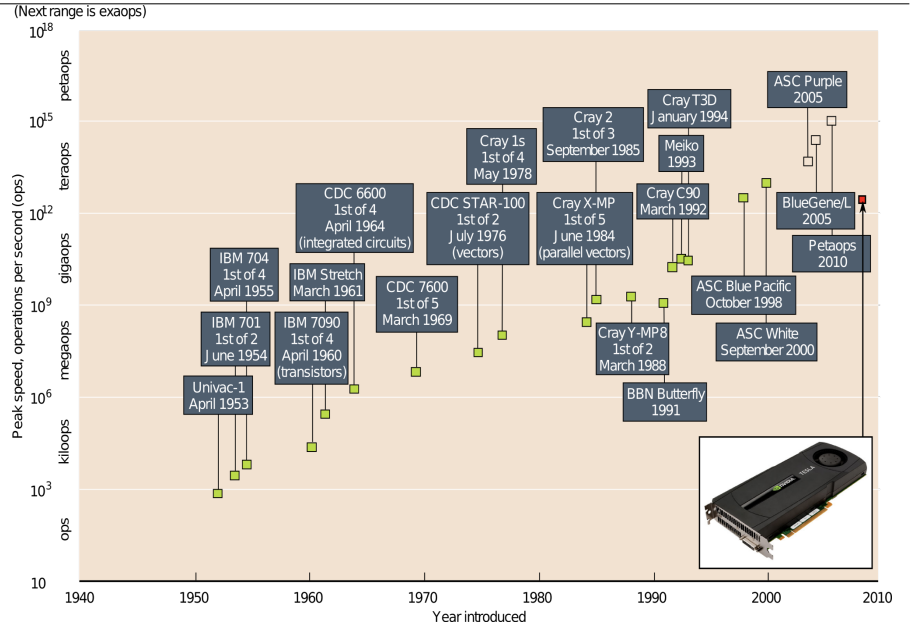
- ☐ CUDA, «*Compute Unified Device Architecture*» ;
- ☐ **Architecture hiérarchique** ;
  - ◇ Une carte contient **plusieurs multiprocesseurs**
  - ◇ Plusieurs «*cuda cores*» par multiprocesseur (32 en général)
  - ◇ Une unité de contrôle unique.
- ☐ **Différents espaces mémoires**
  - ◇ Mémoire de la 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.



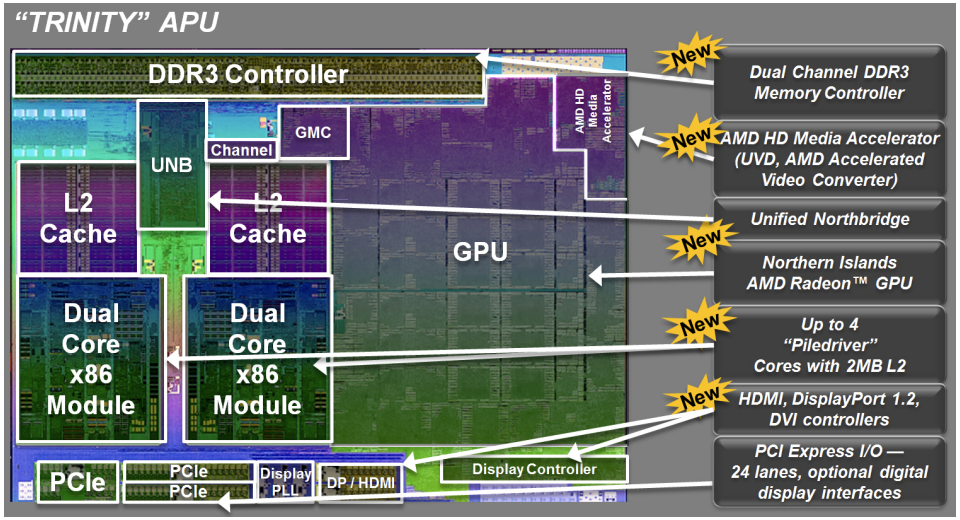
Et les GPUs ? Un modèle extensible adaptable à l'architecture disponible



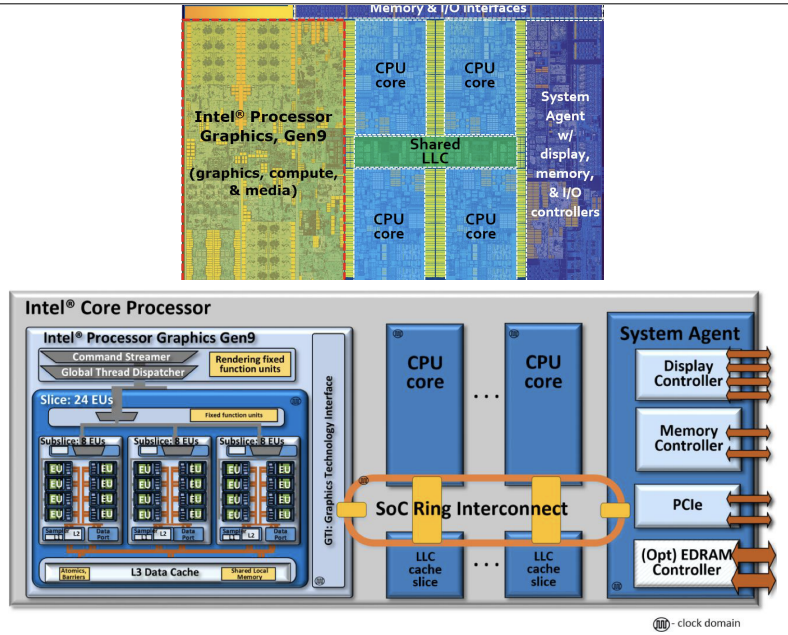
Et les GPUs ? des Résultats !



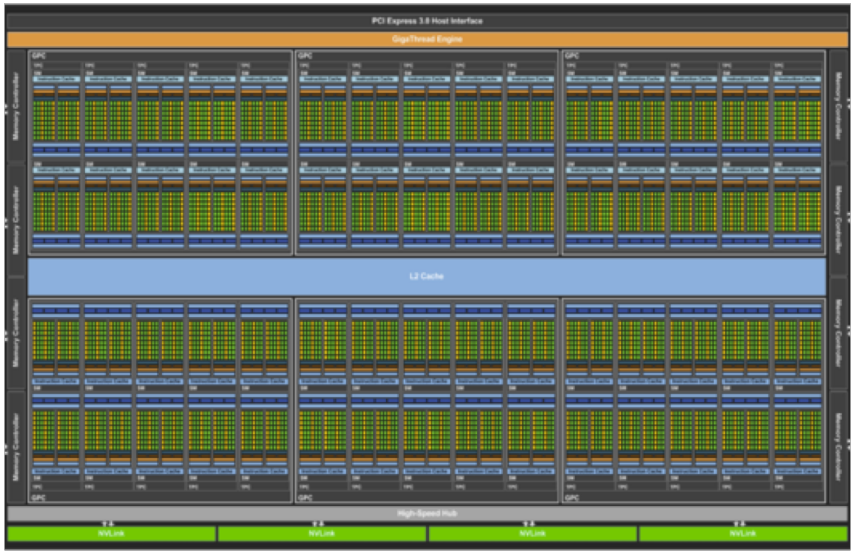
Les APUS, «*Accelerated Processor Unit*»



Skylake



Nvidia Pascal



Et du point de vue du logiciel ?

6 Qu'est-ce que le parallélisme ?

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**.

Le parallélisme

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.

Programmation concurrente

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.

Exclusion mutuelle et synchronisation conditionnelle

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.

Un peu d'histoire sur les ordinateurs séquentiels

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 ?

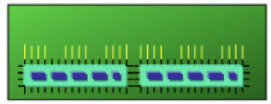
Plusieurs niveaux de parallèlisme

machines

⟹architectures distribuées

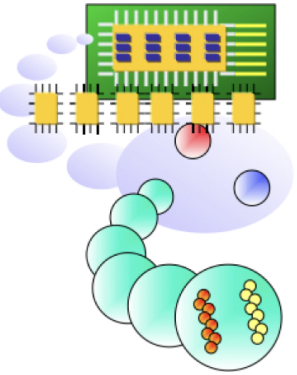
processeurs

⟹ machines multi-processeurs



unités de calcul

⟹processeurs superscalaires



processus

⟹multi-programmation
⟹temps partagé

threads ou processus de poids léger

⟹ multi-programmation à **grain fin**

Et l'exploitation du parallélisme ?

7 Recherche de la performance

Accélération ou speedup

Accélération = gain de temps obtenu lors de la parallélisation du programme séquentiel.

Définition :

▷ Soit T_1 le temps nécessaire à un programme pour résoudre le problème A sur un ordinateur séquentiel ;
▷ Soit T_p le temps nécessaire à un programme pour résoudre le même problème A sur un ordinateur parallèle contenant p processeurs ;
▷ Alors l'accélération «*Speed-Up*» est le rapport : $S(p) = T_1/T_p$
Cette définition n'est pas très précise

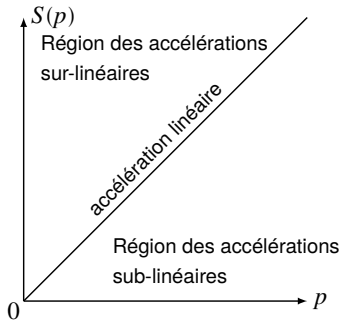
Pour obtenir des résultats comparables il faut utiliser les mêmes définitions d'**Ordinateur Séquentiel** et de **Programme Séquentiel**.

Ordinateur Séquentiel :
□ Ordinateur // configuré avec un seul processeur ;
□ Ordinateur séquentiel d 'une puissance similaire à l 'ordinateur //.

Programme Séquentiel :
□ Programme // configuré pour s'exécuter sur un seul processeur ;
□ Programme séquentiel utilisant le même algo que le programme // ;
□ Programme séquentiel le plus rapide connu utilisant le même algo que le programme // ;
□ Programme séquentiel (ou le plus rapide) résolvant le même pb.
Beaucoup de combinaisons, il faut préciser dans chaque cas.

Accélération et efficacité

Accélération



Efficacité

– Soit $T_1(n)$ le temps nécessaire à l'algorithme pour résoudre une instance de problème de taille n avec un seul processeur,
– Soit $T_p(n)$ celui que la résolution prend avec p processeurs
– Soit $S(n,p) = T_1(n)/T_p(n)$ le facteur d'accélération.

On appelle **efficacité** de l 'algorithme le nombre
$$E(n,p) = S(n,p)/p$$

Efficacité = normalisation du facteur d 'accélération

Exemple

Multiplication de matrices : algorithme A moins bon que algorithme B

Algorithme A
□ Temps en séquentiel : 10 minutes
□ Nombre de processeurs : 10
□ Temps en // : 2 minutes
□ **Accélération** : $10/2 = 5$ (l'application va 5 fois plus vite)
□ **Efficacité** : $5/10 = 1/2 = 0,5$

Algorithme B
□ Temps en séquentiel : 10 minutes
□ Nombre de processeurs : 3
□ Temps en // : 4 minutes
□ **Accélération** : $10/4 = 5/2 = 2,5 < 5$
□ **Efficacité** : $(5/2)/3 = 0,8 > 0,5$

Loi d'Amdahl

Le temps d 'exécution T_1 d 'un programme séquentiel peut être décomposé en deux temps :
- T_s consacré à l 'exécution de la partie intrinsèquement séquentielle
- $T_{//}$ consacré à l 'exécution de la partie parallélisable

$$T_1 = T_s + T_{//}$$

Seul $T_{//}$ peut être diminué par la parallélisation.

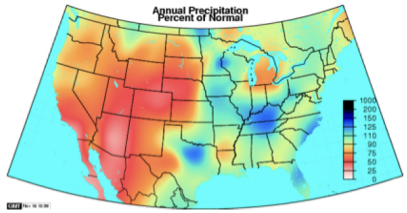
Dans le **cas idéal**, on obtiendra **au mieux** un temps $T_{//}/p$ pour la partie parallélisée.

$$T_p \geq T_s + T_{//}/p$$

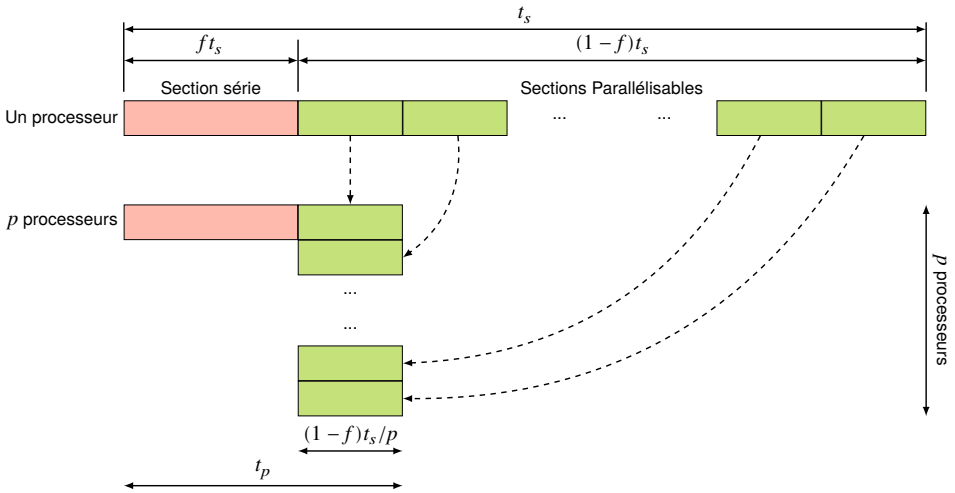
$\Longrightarrow$**L'accélération** d'un programme est **limitée par le pourcentage de code intrinsèquement séquentiel** qu'il contient.

Exemple : filtre graphique parallèle

- partie intrinséquement séquentielle
  - ◇ capture
  - ◇ chargement sur le serveur
- partie parallélisable
  - ◇ découpage
  - ◇ calculs pour le traitement de l'image ...



Loi d'Amdahl



La fraction f exprime le rapport entre la partie séquentielle et parallèle par rapport au temps complet t_s :

▷ ft_s pour le temps de la partie séquentielle ;

▷ $(1-f)t_s$ pour le temps de la partie parallèle.

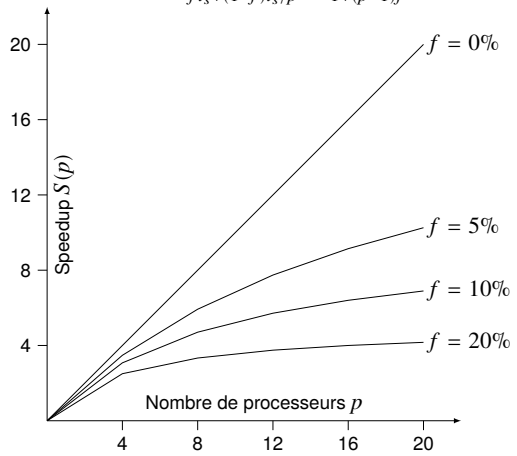
Speedup

Même avec un nombre infini de processeurs l'accélération maximale est de $1/f$

Exemple : avec seulement 5%de calcul séquentiel, le speedup maximal est de 20.

Calcul du speedup avec f:

$$S(p) = \frac{t_s}{ft_s+(1-f)t_s/p} = \frac{p}{1+(p-1)f}$$



f exprime le pourcentage de la partie séquentielle.

En conclusion

Une **accélération linéaire** correspond à un gain de temps égal au nombre de processeurs (100%activité)

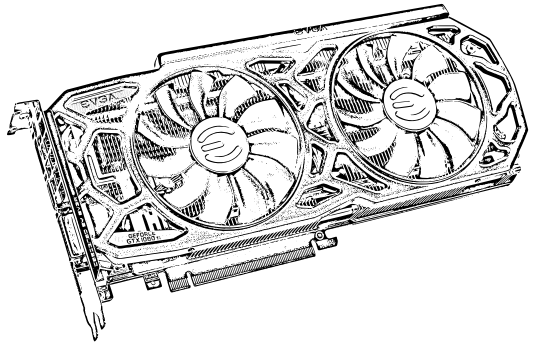
Une **accélération sub-linéaire** implique un taux d'activité des processeurs $<$ 100 %(communication, coût du parallélisme...)

Une **accélération sur-linéaire** implique un taux d'utilisation des processeurs $>$ à 100 %ce qui paraît impossible (en accord avec la loi d 'Amdhal).

Cela se produit parfois (architecture, mémoire cache mieux adaptée que les machines mono-processeurs, utilisation de pipeline…)

Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année



NVIDIA CUDA

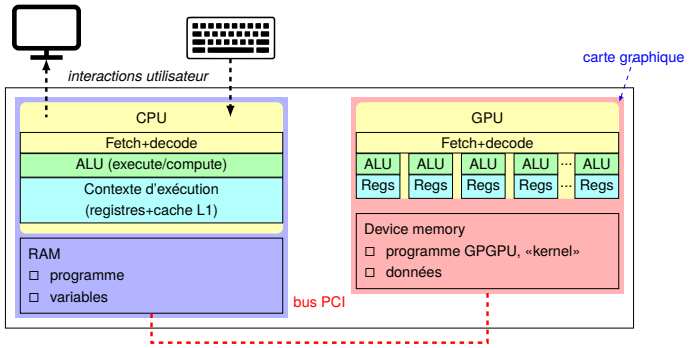
Développement GPGPU — P-F. Bonnefoi

Version du 11 décembre 2022

Table des matières

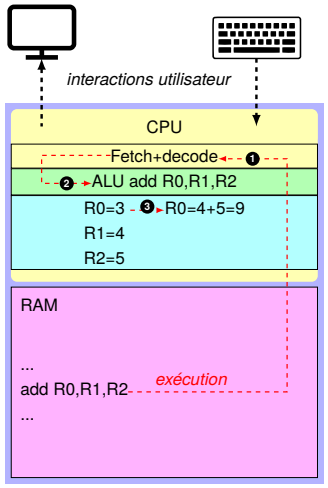
1 CUDA est un modèle SIMD

Le CPU est composé de :
- une unité de contrôle chargée de :
  - chercher en mémoire les instructions à exécuter, «*fetch*» ;
  - décoder ces instructions en termes d'opération à faire sur les registres et la mémoire ;
- une unité ALU, «*Arithmétique et Logique*» ;
- des registres et de la mémoire cache pour limiter les accès à la RAM ;



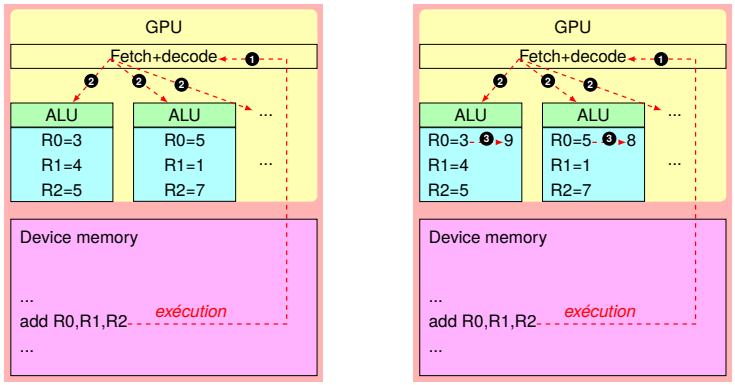
Le GPU est composé de :
- une unité de contrôle ;
- nombreuses unités ALU+Registres combinées (plusieurs milliers).

L'exécution sur le CPU



Déroulement de l'exécution d'une instruction sur le CPU :
1. ⇒une instruction est récupérée depuis la RAM et décodée ;
2. ⇒elle déclenche des opérations de l'ALU sur les registres et/ou le contenu de la mémoire ;
3. ⇒le résultat est rangé dans un registre ;
4. ⇒on recommence sur l'instruction suivante.

L'exécution sur le GPU



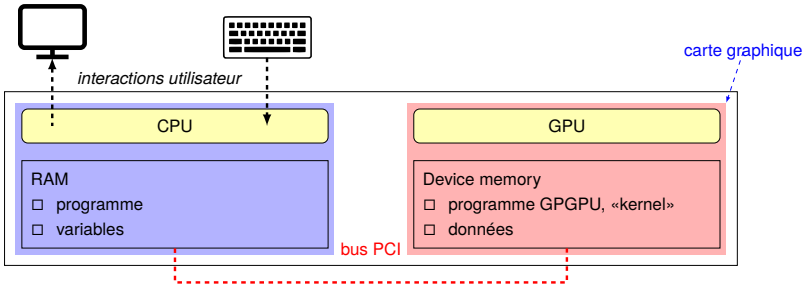
Déroulement de l'exécution d'une instruction sur le GPU :

❶⇒une instruction est récupérée depuis la RAM et décodée ;

❷⇒elle déclenche des opérations identiques sur les différentes ALU entre les registres associés et/ou le contenu de la mémoire ;

❸⇒le résultat est rangé dans un registre local.

L'architecture CUDA, «*Compute Unified Device Architecture*»



Architecture CUDA : le système «*host*», CPU, et le système GPU, la carte graphique sont **séparés** :

▷ la RAM ou la mémoire de l'hôte est accessible uniquement par le CPU ;

▷ la «*Device Memory*» est accessible uniquement par le GPU ;

⟹les **données** conservées dans la RAM ou la «*memory device*» doivent être **échangées** entre les deux à l'aide du bus PCI en mode DMA ;

⟹un **programme** qui utilise le GPU est constitué de deux parties :

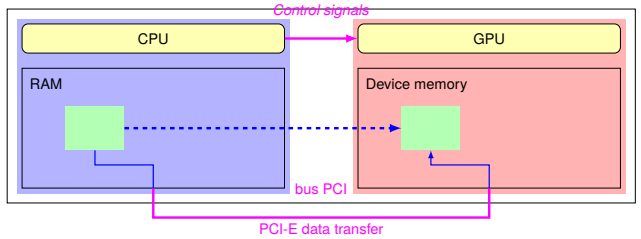
- ◇ une partie tournant sur le **CPU**, c-à-d sur l'hôte ;
- ◇ une partie tournant sur le **GPU**, c-à-d sur le «*device*».

Comment faire travailler le GPU sur les données ?

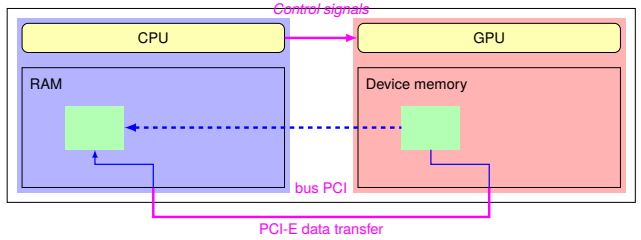
CUDA mets à disposition des fonctions de transfert de données entre la mémoire RAM et la mémoire du «*device*» :

De la RAM vers la mémoire du «*device*» :

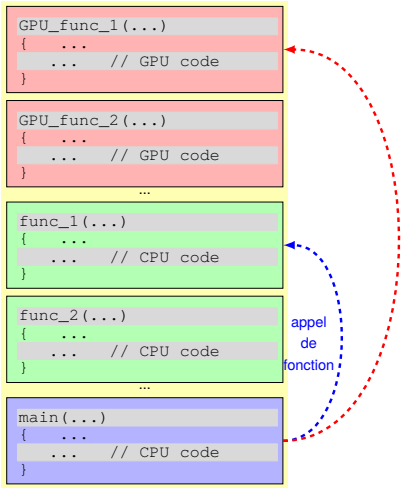
Des signaux de contrôle sont utilisés par le CPU pour déclencher l'opération sur le GPU.



De la mémoire du «*device*» vers la RAM :



Comment programmer ?



Le programmeur écrit **un seul programme source** constitué de deux types de fonctions :

▷ les fonctions `func_x` sont exécutées par le CPU comme des fonctions ordinaires ;

▷ les fonctions `GPU_func_x` sont exécutées par le GPU sur la carte graphique ;

▷ la fonction principale, `main`, exécutée par le CPU appelle les différents types de fonctions : c'est elle qui est appelée en premier au lancement du programme.

Il existe **deux types de fonction** GPU en CUDA :

▷ précédées par `__global__` : peuvent être appelées par le «*host*» ou par une autre fonction du «*device*» ;

```
__global__ void GPU_function1 ( parameters) {...}
```

▷ précédées par `__device__` : ne peuvent être appelées que par une autre fonction du «*device*»

```
__device__ void GPU_function2 ( parameters) {...}
```

Les deux types de fonctions `__global__` et `__device__` ne doivent rien renvoyer (`void`).

⟹ **une fonction GPU ne retourne pas de valeur !**

Comment est gérer la mémoire ? 9

Le «*device*» ne dispose par d'OS, «*Operating System*» : il ne sait pas gérer sa mémoire !

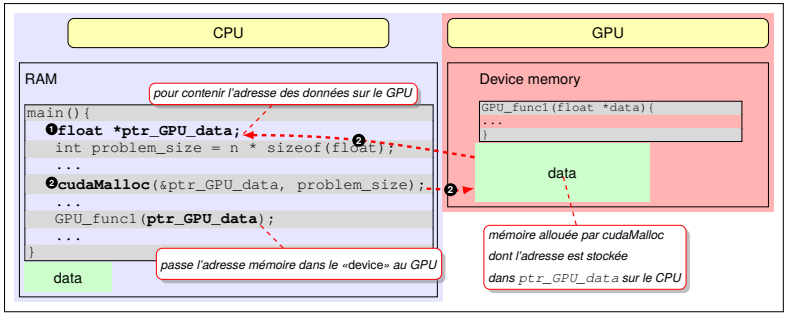
⟹C'est le «*host*» qui gère la mémoire pour le GPU :

▷ il **déclare une variable du type pointeur** sur le type de données à manipuler `type *ptr` ❶ ;

▷ il **alloue de la mémoire sur le GPU** grace à la fonction `cudaMalloc(&ptr, nombre_octets)` ❷ :

◇ de la mémoire est «*réservée*» sur le GPU (c'est le CPU qui contrôle les espaces mémoires du device) ;

◇ l'**adresse de cette zone mémoire** est stockée dans le pointeur `ptr` ;



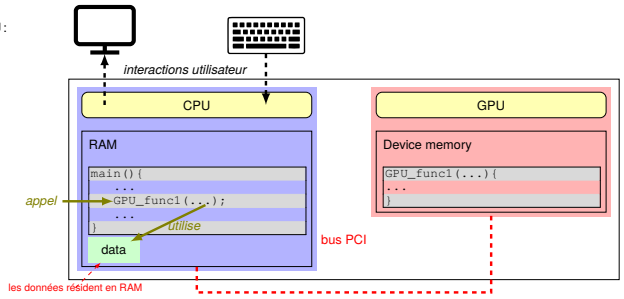
▷ lors de l'appel de la fonction `GPU_func1`, le CPU transmettra en paramètre l'adresse de la zone mémoire allouée précédemment stockée dans `ptr` à la fonction.

⟹la fonction GPU `GPU_func1` peut travailler sur la zone mémoire du device réservée à cet usage.

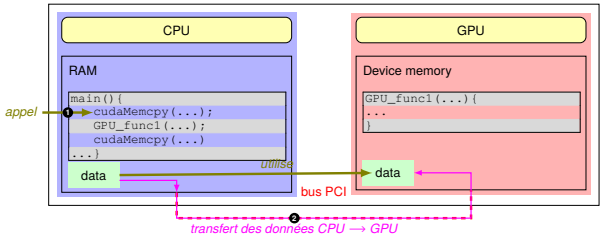
Comment déclencher le travail entre le CPU et le GPU ? 10

L'utilisateur n'interagit uniquement avec le programme CPU :
⟹les données sont **uniquement** dans la RAM accessible par le CPU.

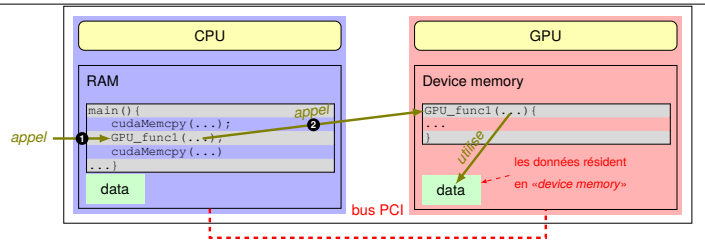
⟹l'appel de la fonction GPU `GPU_func1()` ne peut être fait directement.

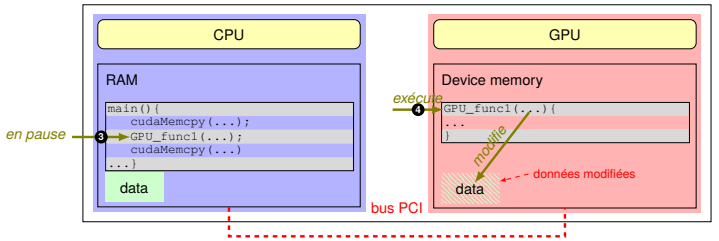


Les données sont transférées du CPU vers la mémoire «*device*» grâce à une opération `cudaMemcpy()` : les données sont transférées au travers du bus PCI.

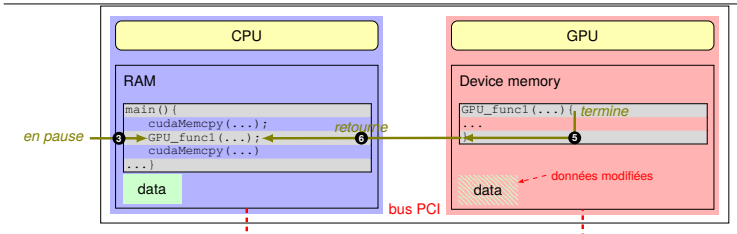


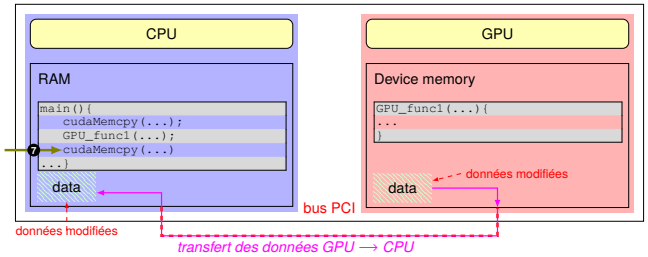
Comment déclencher le travail sur le GPU ? 11





Comment déclencher le travail sur le GPU ? 12





C'est un **environnement logiciel** qui permet d'utiliser le GPU, «*Graphics Processing Unit*» au travers de programme de haut niveau comme le C ou le C++ :

- le programmeur écrit un programme C avec des extensions CUDA, de la même manière qu'un programme OpenMP ;
- CUDA nécessite une carte graphique équipée d'un processeur NVIDIA de type Fermi, GeForce 8XXX/Tesla/Quadro, *etc.*
- les fichiers source doivent être compilés avec le compilateur C CUDA, `NVCC`.

Un **programme CUDA** utilise des «*kernels*» pour traiter des «*data streams*», ou «flux de données».

Ces flux de données peuvent être par exemple, des vecteurs de nombres flottants, ou des ensembles de frames pour du traitement vidéo.

Un «*kernel*» est exécuté dans le GPU en utilisant des threads exécutées en parallèles.

CUDA fournit **3 mécanismes** pour paralléliser un programme :

- un **regroupement hiérarchique** des threads ;
- des **mémoires partagées** ;
- des **barrières de synchronisation**.

Ces mécanismes fournissent du parallélisme à **grain fin** *imbriqué dans du parallélisme à* **gros grain**.

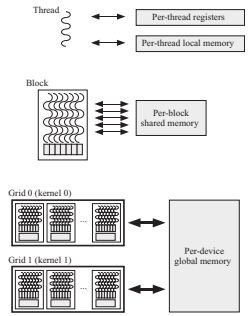
Des définitions

Terme	Définition
Host ou CPU	c'est l'ordinateur qui sert d'interface avec l'utilisateur et qui contrôle le «device» utilisé pour exécuter les parties de calcul intensif basé sur un parallélisme de données. *L'hôte est responsable de l'exécution des parties séquentielles de l'application.*
GPU	est le processeur graphique, «*General-Purpose Graphics Processor Unit*», pouvant réaliser du travail générique qui peut être utilisé pour implémenter des algorithmes parallèles.
Device	est le GPU connecté à «l'hôte» et qui va exécuter les parties de calcul intensif basé sur un parallélisme de données. *Le device, ou périphérique, est responsable de l'exécution de la partie parallèle de l'application.*
kernel	est une fonction qui peut être appelée depuis «l'hôte» et qui est exécutée en parallèle sur le «*device*» CUDA par de nombreuses threads.

⋆ Le «kernel» est exécuté simultanément par des milliers de threads.

⋆ Une application ou une fonction de bibliothèque consiste en un ou plusieurs kernels.
Fermi peut exécuter différents kernels à la fois, s'ils appartiennent tous à la même application.

⋆ Un kernel peut être écrit en C avec des annotations pour exprimer le parallélisme :
- localisation des variables ;
- utilisation d'opération de synchronisation fournie par l'environnement CUDA.



La hiérarchie de mémoire et de threads est la suivante :

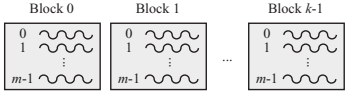
1. la **thread** au niveau le plus bas de la hiérarchie ;
2. le **bloc** composé de plusieurs threads exécutées de manière concurrente ;
3. la **grille** composée de plusieurs blocs de threads exécutés de manière concurrente ;
4. de la **mémoire locale** dédiée à chaque thread, *per-thread local memory*, visible uniquement depuis la thread (cela concerne également des registres) ;
   Les registres sont sur le processeur et disposent de temps d'accès très rapide.
   La mémoire locale, indiquée en gris sur le schéma, dispose d'un temps d'accès plus lent que celui des registres.
5. de la **mémoire partagée** associée à chaque bloc visible, *per-block shared memory*, uniquement par toutes les threads du bloc ;
   Le bloc dispose de sa propre mémoire partagée et privée pour permettre des communications inter-thread rapides et de taille réglable.
6. de la **mémoire globale**, «per-device global memory», utilisable par le «device».
   Une grille, «grid», utilise la mémoire globale. Cette mémoire globale permet de communiquer avec la mémoire de l'hôte et sert de lien de communication entre l'hôte et le GPGPU.

Grille & Bloc : organisation et localisation d'une thread

Le programmeur doit spécifier le nombre de threads dans un bloc et le nombre de blocs dans une grille.
Le nombre de blocs dans la grille est spécifié par la variable `gridDim`.

Exemple : un tableau à une seule dimension

- on peut organiser les blocs en un tableau à une seule dimension, et le nombre de blocs sera :
  `gridDim.x = k` ,
  ainsi si $k = 10$, alors on aura 10 blocs dans la grille.
- on peut organiser les threads en un tableau à une seule dimension de m threads par bloc :
  `blockDim.x = m`
- chaque bloc dispose d'un identifiant unique, «ID», appelé `blockIdx` qui est compris dans l'intervalle :
  `0≤ blockId ≤ gridDim`



Pour associer une thread à la i[ème] case d'un vecteur, on doit trouver à quel bloc appartient la thread et ensuite la localisation de la thread dans le bloc : `i = blockIdx.x * blockDim.x + threadIdx.x`

Généralisation du concept de Grille et de blocs

- Les variables `gridDim` et `blockIdx` sont définies automatiquement et sont de type `dim3`.
- Les blocs dans la grille peuvent être organisés suivant une, deux ou trois dimensions.
- Chaque dimension est accédée par la notation `blockIdx.x`, `blockIdx.y` et `blockIdx.z`.

La commande CUDA suivante définit le nombre de blocs dans les dimensions x, y et z :

```
dim3 dimGrid(4, 8, 1) ;
```

Cette commande définit 32 blocs organisés en un tableau à deux dimensions avec 4 lignes de 8 colonnes.
Le nombre de threads dans un bloc est défini par la variable `blockDim`.

Répartition du travail entre threads regroupées en bloc

Chaque thread dispose d'un identifiant unique, «ID», appelé `threadIdx` qui est compris dans l'intervalle :

$$0 \leq \texttt{threadId} < \texttt{blocDim}$$

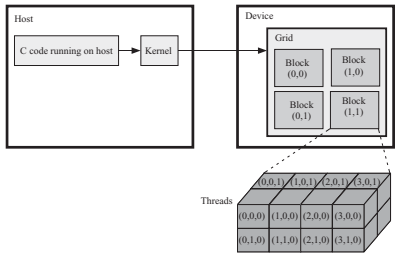
- Les variables `blockDim` et `threadIdx` sont définies automatiquement et sont de type `dim3`.
- Les threads dont un bloc peuvent être organisées suivant une, deux ou trois dimensions.
- Chaque dimension est accédée par la notation `threadIdx.x`, `threadIdx.y` et `threadIdx.z`.

La commande CUDA suivante définit le nombre de threads dans les dimensions x, y et z :

```
dim3 dimBlock(100, 1, 1);
```

Cette commande définit 100 threads organisées en un tableau de 100 cases.

Rapport entre «kernel» et grille



- Chaque «*kernel*» est associé avec une grille dans le «*device*».
- le choix du nombre de threads et de blocs est conditionné par la nature de l'application et la nature des données à traiter.

Le but est d'offrir au programmeur des moyens d'organiser les threads de manière adaptée à l'organisation des données, afin de simplifier l'accès à ces données :
«les données sont organisées en grille ? alors les threads aussi».

Un seul programme source mixte CPU/GPU

Le code source «*mon_prog.cu*» est compilé en deux parties :

- un code pour le CPU en instructions x86/amd64 ;
- un code pour le GPU en instructions PTX, «*Parallel Thread eXecution*» ;

mon_prog.cu

```
int main() {
  float* arr;
  cudaMalloc(&arr, 128*sizeof(float));
  Write42<<<1, 128>>>(arr);
}
```

```
__global__ void Write42(float *out)
{
  out[threadIdx.x] = 42.0f;
}
```

Le compilateur `nvcc` fourni par NVidia :

- ▷ réalise la répartition des codes à partir d'un fichier source unique ;
- ▷ compile chaque partie indépendamment ;
- ▷ construit un exécutable contenant les deux parties et capable de charger le code GPU sur le «*device*».

Assigner une fonction pour être exécutée par un «kernel» dans CUDA

Pour définir une fonction qui va être exécutée en tant que «kernel», le programmeur modifie le code C du prototype de la fonction en plaçant le mot clé «`__global__`» devant ce prototype :

```
__global__ void kernel_function_name(function_argument_list);
```

La fonction doit renvoyer `void`.

Le programmeur doit ensuite indiquer au compilateur C NVIDIA, `nvcc`, de lancer le «kernel» pour être exécuté sur le «device» :

```
int main()
{
/*
   Une partie séquentielle du code
*/
/* Le début de la partie parallèle du code */

kernel_function_name<<< gridDim, blockDim >>> (function_argument_list);

/* La fin de la partie parallèle */

/*
   Une partie séquentielle du code
*/
}
```

Le programmeur modifie le code C en spécifiant la structure des blocs dans la grille et la structure des threads dans un bloc en ajoutant la déclaration `<<<gridDim, blockDim>>>` *entre le nom de la fonction et la liste des arguments de la fonction.*

Communication entre «l'hôte» et le «*CUDA device*»

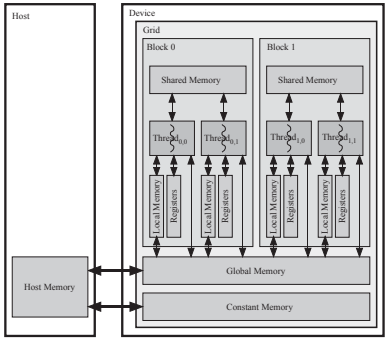
- ▷ L'ordinateur hôte dispose de sa propre hiérarchie de mémoire de même que le «*device*».
- ▷ L'échange de données entre l'hôte et le «device» est réalisé en copiant des données entre la DRAM, «dynamic ram», et la mémoire DRAM globale du «*device*».
- ▷ De la même façon qu'en C, le programmeur doit allouer de la mémoire dans la mémoire globale du «*device*» pour les données et libérer cette mémoire une fois l'application terminée.

Les **appels systèmes CUDA** suivants permettent de réaliser ces opérations :

Fonction	Description
`cudaDeviceSynchronize()`	bloque jusqu'à ce que le «device» ait terminé les tâches demandées précédemment
`cudaThreadSynchronize()`	*version précédente de* `cudaDeviceSynchronize()`
`cudaChooseDevice()`	retourne le «device» qui correspond aux propriétés spécifiées
`cudaGetDevice()`	retourne le «device» utilisé actuellement
`cudaGetDeviceCount()`	retourne le nombre de «device» capable de faire du GPGPU
`cudaGetDeviceProperties()`	retourne les informations concernant le «device»
`cudaMalloc()`	alloue un objet dans la mémoire globale du «device». Nécessite deux arguments : l'adresse d'un pointeur qui recevra l'adresse de l'objet, et la taille de l'objet
`cudaFree()`	libère l'objet de la mémoire globale du «device»
`cudaMemcpy()`	copie des données de l'hôte vers le «device». Nécessite quatre arguments : le pointeur destination, le pointeur source, le nombre d'octets et le mode de transfert.

La communication hôte - «device»

L'interface mémoire entre l'hôte et le «device» :



La mémoire globale en bas du schéma est le moyen de communiquer des données entre l'hôte et le «device».

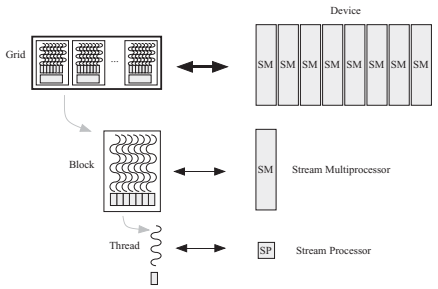
Le contenu de la mémoire globale est visible depuis toutes les threads et est accessible en lecture/écriture, celle indiquée «constant memory», n'est accessible qu'en lecture seulement.

La mémoire partagée par bloc est visible depuis toutes les threads de ce bloc.

La mémoire locale, comme les registres, n'est visible que de la thread.

Exécution d'une application parallèle sur le «device»

L'hôte déclenche une fonction «kernel» :



- Le «*kernel*» est exécuté sur une grille de blocs de threads.
- Différents «*kernels*» peuvent être exécutés par le «*device*» à différents moments de la vie du programme.
- Chaque bloc de threads est exécuté sur un multiprocesseur à flux, «*streaming multiprocessor*», «SM».
- Le SM exécute plusieurs blocs de threads à la fois.
- Des copies du «*kernel*» sont exécutées sur le «streaming processor», «SP», ou «*thread processors*», ou «*Cuda core*», qui exécute une thread qui évalue la fonction.
- Chaque thread est allouée à un SP.

Actuellement, dans les salles de TP :

- au plus 1024 threads par dimension du bloc qui communiquent par mémoire partagée ;
- chaque dimension d'une grille doit être inférieure à 65536 ;
- la mémoire partagée dans un block $\simeq 16ko$;
- la mémoire constante $\simeq 64Ko$;
- nombre de registres disponibles par block 8192 à 16384.

Comment est-ce que cela se passe dans le GPU ?

Utilisation du profiler `nvprof`

Exemple sur l'exercice du TD n°1 :

```
xterm
$ nvprof ./TD1
==21398== NVPROF is profiling process 21398, command: ./TD1
Result : 25723564731392.000000
==21398== Profiling application: ./TD1
==21398== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   84.51%  45.057us         2  22.528us  22.272us  22.785us  [CUDA memcpy HtoD]
                   14.05%  7.4880us         1  7.4880us  7.4880us  7.4880us  dot(double*,
double*, double*)❶
                    1.44%     768ns         1     768ns     768ns     768ns  [CUDA memcpy
DtoH]
      API calls:   99.23%  125.79ms         3  41.930ms  6.2840us  125.66ms  cudaMalloc
                    0.34%  429.54us        94  4.5690us     524ns  173.12us  cuDeviceGetAttribute
                    0.20%  249.19us         3  83.064us  10.325us  121.90us  cudaFree
                    0.09%  115.94us         3  38.646us  14.710us  56.079us  cudaMemcpy
                    0.08%  103.67us         1  103.67us  103.67us  103.67us  cuDeviceTotalMem
                    0.03%  44.132us         1  44.132us  44.132us  44.132us  cuDeviceGetName
                    0.02%  24.364us         1  24.364us  24.364us  24.364us  cudaLaunch
                    0.00%  2.5540us         3     851ns     532ns  1.3180us  cuDeviceGetCount
                    0.00%  1.4740us         2     737ns     590ns     884ns  cuDeviceGet
                    0.00%     943ns         1     943ns     943ns     943ns  cudaConfigureCall
                    0.00%     930ns         3     310ns     142ns     529ns  cudaSetupArgument
```

❸ ❷

Le **profiler** fournit les informations suivantes :

- le **nombre d'appels** des différentes fonctions (sous la rubrique «*Calls*») ;
- le temps d'exécution des **différentes fonctions CUDA** ❷ ;
- le temps d'exécution du **kernel** (❶, le kernel qui ici s'appelle `dot`) ;
- le **rapport** entre le temps d'exécution du **kernel** et des **transferts de mémoire** ❸.

Comment est-ce que cela se passe dans le GPU ?

Utilisation du profiler `nvprof`

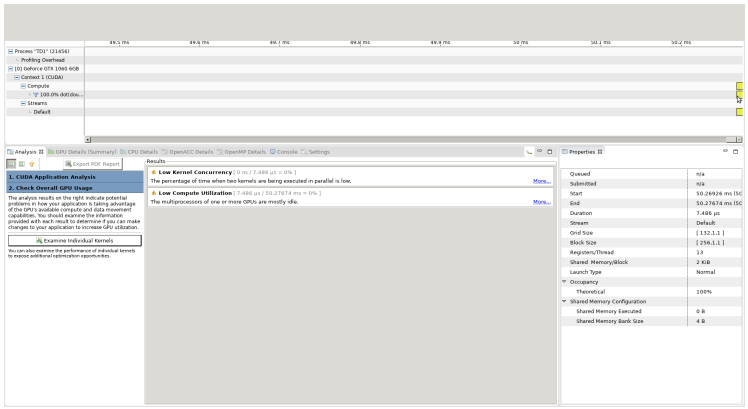
Ici, on va demander à récupérer des informations concernant les activités du GPU avec l'option `--print-gpu-trace` :

```
xterm
$ nvprof --print-gpu-trace ./TD1
==21538== NVPROF is profiling process 21538, command: ./TD1
Result : 25723564731392.000000
==21538== Profiling application: ./TD1
==21538== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput
SrcMemType  DstMemType           Device   Context    Stream  Name

228.80ms  22.913us                    -               -         -         -         -  264.00KB  10.988GB/s
Pageable      Device  GeForce GTX 106         1         7  [CUDA memcpy HtoD]

228.84ms  22.208us                    -               -         -         -         -  264.00KB  11.337GB/s
Pageable      Device  GeForce GTX 106         1         7  [CUDA memcpy HtoD]

228.87ms  7.4560us           (132 1 1)❶       (256 1 1)❷        13❸  2.0000KB        0B         -
-         -           -  GeForce GTX 106         1         7  dot(double*, double*, double*) [111]

228.88ms     768ns                    -               -         -         -         -  1.0313KB  1.2806GB/s
Device    Pageable  GeForce GTX 106         1         7  [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA
driver and/or tools and can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
```

On obtient des informations sur le code PTX, «*Parallel Thread eXecution*» produit :

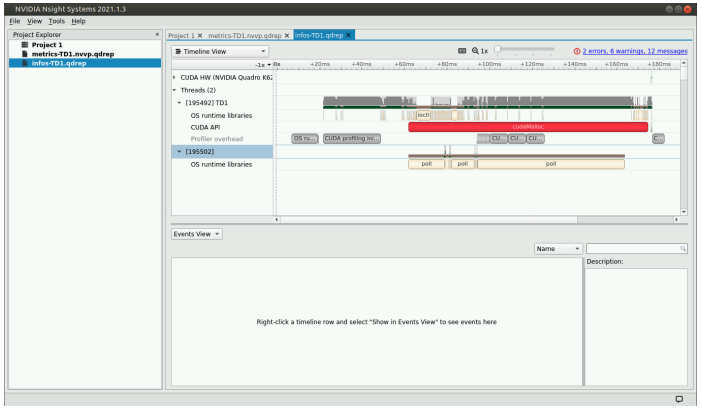
❶⇒la géométrie de la grille ;
❷⇒celle du bloc ;
❸⇒le nombre de registres utilisés par thread, c-à-d le nombre de variables locales utilisées par le kernel (si on dépasse, on est obligé d'utiliser de la mémoire locale à la thread moins performante).

Comment est-ce que cela se passe dans le GPU ? 25



Une copie d'écran de l'outil «nvvp».

Comment est-ce que cela se passe dans le GPU ? 26



Une copie d'écran de l'outil «nsight».

La commande utilisée :

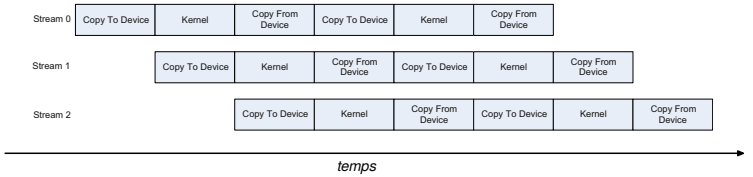
xterm

```
$ nsys profile -w true -t cuda,nvtx,osrt,cudnn,cublas -s cpu --cudabacktrace=true -x true -o
infos-TD1 ./TD1
```


Exécution d'une application parallèle sur le «device» 27

La notion de «stream»

- sur une carte pro : plusieurs streams possibles ;
- sur une carte grand public : un seul stream.



Ici, la carte GPGPU est capable de supporter plusieurs streams, mais offre un accès séquentielle pour les transferts des données de l'hôte vers la carte, «Copy To device».

Attention

La possibilité de faire des transferts *asynchrones*, c-à-d de «*recouvrir*» des communications par du calcul est obtenu à l'aide de la fonction `cudaMemcpyAsync()` :

- ▷ la copie de mémoire **vers le GPGPU** depuis le CPU peut être réalisé en **même temps** que du travail sur le CPU (le GPU récupère ses données simultanément) ;
- ▷ la copie **vers** et **depuis** le GPU peut être faire pendant que le GPU réalise du travail.

Cette capacité est disponible suivant la capacité CUDA de la carte nvidia utilisée.

La notion de «*warp*» 28

Soit le programme suivant et sa parallélisation :

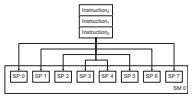
```
void some_func(void)
{
    int i;
    for (i=0;i<128;i++)
        { a[i] = b[i] * c[i]; }
}
```

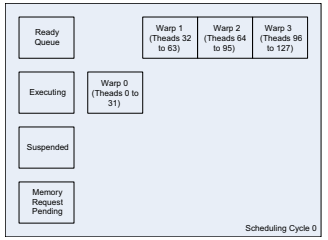
On parallélise la boucle en créant une thread par occurence de la boucle :

```
__global__ void some_kernel_func(int *a, int *b, int *c)
{   unsigned int thread_idx = threadIdx.x;
    a[thread_idx] = b[thread_idx] * c[thread_idx]; }
```

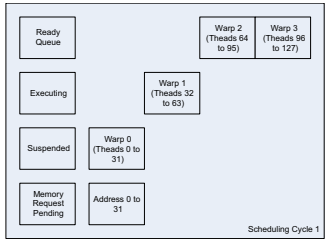
Un «*warp*»

- ▷ correspond à 32 threads ;
- ▷ exécute du code SPMD, ou SPMT, «*Simple Program Multiple Thread*», ⇒
- ▷ est ordonnancé, *scheduled*, dans le SP, suivant son état :





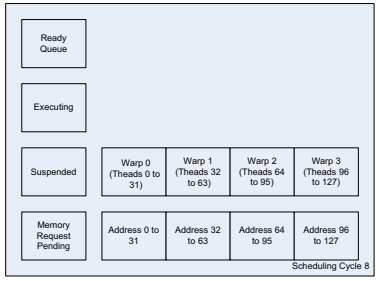
Le «Warp 0» progresse de «Ready Queue» à «Executing».



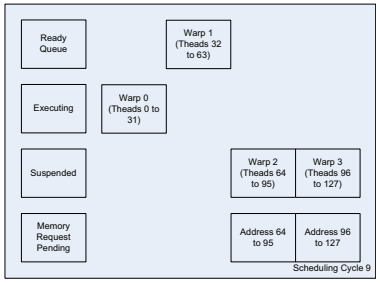
Le «Warp 0» réalise des demandes d'accès mémoire et se suspend : c'est le «Warp 1» qui s'exécute.

La notion de «*warp*» 29

Le «*scheduler*» fait progresser le warp de l'état prêt, «*Ready Queue*», à l'état exécuté, «*Executing*».
Dans le cas où le warp réclame du contenu dans la mémoire : il passe en «*Suspended*» et des accès mémoires attendent d'être résolus : «*Memory Request Pending*».



Toutes les threads sont bloquées en attente du retour des données depuis la mémoire...



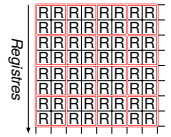
Les données 0 à 63 sont obtenues : les threads 0 à 31 sont exécutées et les threads 32 à 63 sont prêtes.

Mais comment ça marche concrétement ?

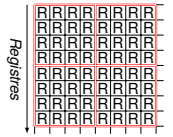
Compilation du «*kernel*», création des threads et répartition des registres 31

La **répartition** des registres entre les threads est **régulière** : **chaque thread** reçoit le **même nombre** de registres.

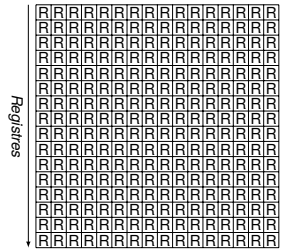
Allocation de 4 registres par thread :



Allocation de 16 registres par thread :



Mais le nombre de registres est **limité**...



⟹ On peut **épuiser** le nombre de **registres** à **partager** entre toutes les threads que l'on veut exécuter simultanément !

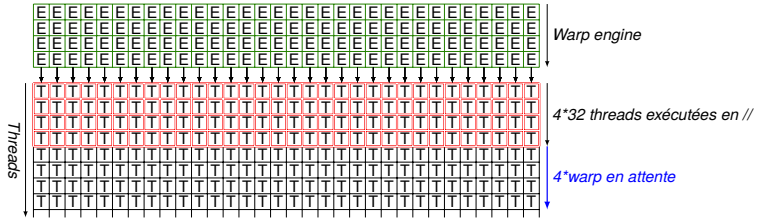
⟹ les threads devront utiliser leur **mémoire locale**, **limitée** et **moins rapide** ;

⟹ On parle de «*register spilling*» : les registres débordent sur la mémoire locale.

Exécution des threads créés par le SM, «*symetrical multiprocessor*» 32

Les threads définies sont exécutées par groupe de 32 threads, un «*warp*» :

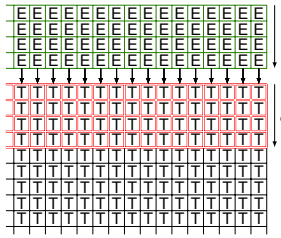
Un processeur, SM, d'un GPU exécute en **parallèle**, «//», plusieurs warps (on peut parler de «*warp engine*») :



Ici, le SM peut exécuter 4 warps simultanément en //.

Exécution parallèle des Warps : effet pipeline

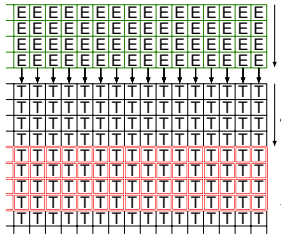
Étape 1 :



Code du kernel :

```
...
x = tab[tid];❶
y = x +2;❷
...
```

Étape 2 :

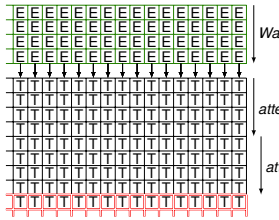


Le pipeline permet de réduire la **latence**, c-à-d l'attente due à l'accès mémoire ❶ :

- ▷ le warp fait une requête mémoire qui se déroule en plusieurs cycles ;
- ▷ pendant ces cycles d'attente :
  - ◇ les threads courantes exécutées passent en attente ;
  - ◇ le «*warp engine*» exécute les threads suivantes qui elles aussi vont faire l'accès mémoire ❶ ;

Exécution parallèle des Warps : effet pipeline

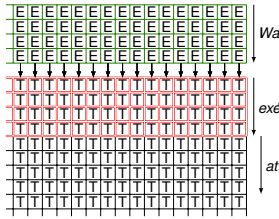
Étape n :



Code du kernel :

```
...
x = tab[tid];❶
y = x +2;❷
...
```

Étape $n+1$:



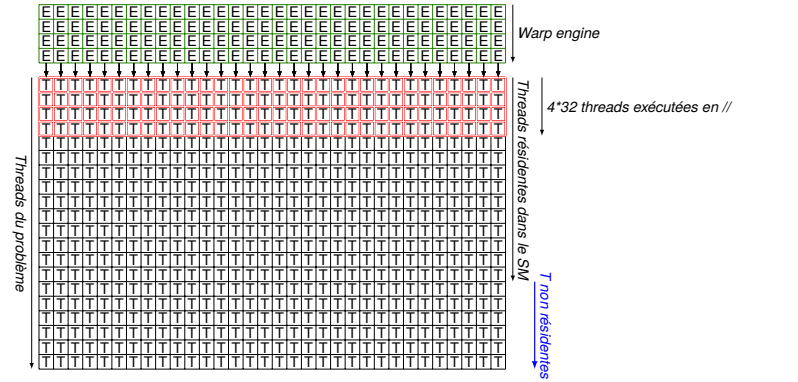
Lorsque les données demandées en ❶ sont disponibles :

- ▷ le **warp engine** preprend l'exécution des threads en attente qui sont maintenant **débloquées** ;
- ▷ les threads exécutent l'instruction ❷ ;

Exécution parallèle des threads créés par le SM, «*symetrical multiprocessor*»

Toutes les threads créées **ne peuvent pas être en même temps** dans le processeur SM :

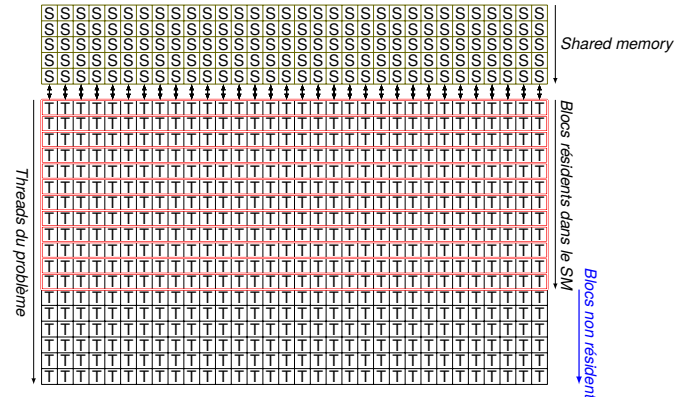
- – certaines sont **résidentes**, c-à-d elles sont prêtes à être exécutées :
  - ◇ chaque thread dispose de ses registres ;
  - ◇ le **passage** d'une thread à une autre est **rapide** : brancher/débrancher en **hardware** un jeu de registres à un autre ;
- – certaines sont **non résidentes** : le **passage** est **lent** : il faut installer ses registres.



Threads, Blocs et mémoire partagée

Les différentes threads sont regroupées en **blocs** :

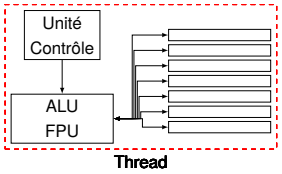
- ▷ chaque bloc appartient à un **unique SM** ;
- ▷ les threads d'un bloc partagent l'accès à de la **mémoire partagée** ;
- ▷ tous les blocs créés peuvent ne pas être **résidents** dans le SM...

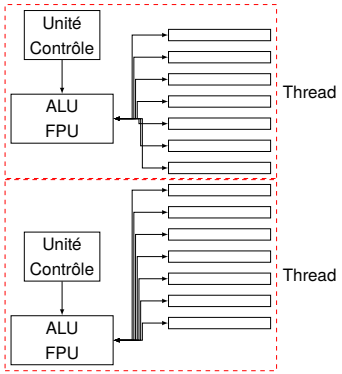


Mais qu'est-ce qu'une thread au final ? 37

Qu'est-ce qu'une thread en **exécution** ?

- une **unité de contrôle** exécutant le «*kernel*» qui contrôle l'ALU/FPU ;
- une **liste variable** de registres ;





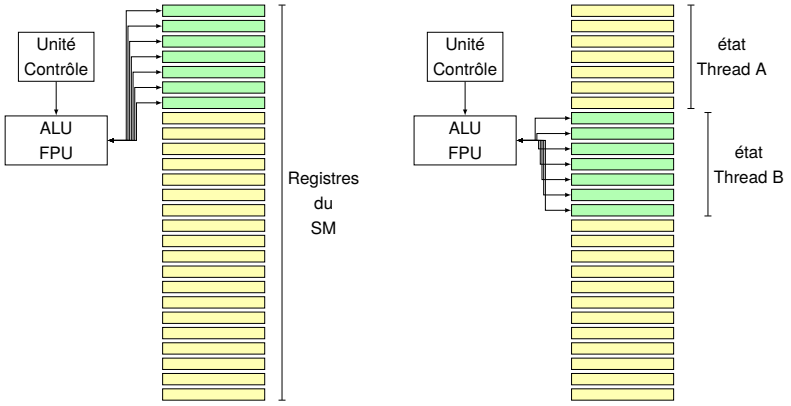
Deux threads en **exécution** :

- ▷ deux unités de contrôles et deux FPU/ALU ;
- ▷ deux jeux de registres.

Comment passe-t-on de l'exécution d'une thread résidente à une autre ? 38

Une thread est **résidente** si **ses registres**, son «*état*», sont **disponibles** dans le SM.

Exécuter une nouvelle thread consiste à **échanger son état**, ses registres, avec l'état d'une autre thread.



Passage de l'exécution de la Thread A à la Thread B : la thread B était en attente d'exécution.

Exécution d'une application parallèle sur le «device» 39

La notion de divergence

```
__global__ some_func(void)
{
   if (some_condition)
   {
      action_a(); // +
   }
   else {
      action_b(); // -
   }
}
```

Imaginons que les threads paires réalisent le travail positif et les threads impaires le travail négatif par rapport à la condtion :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+

Le Warp exécute du code SPMT : les threads «+» s'exécutent pendant que les autres sont bloquées.

Une solution : l'association par demi warp, soient 16 threads :

```
if ((thread_idx % 32) < 16)
{
   action_a();
}
else {
   action_b();
}
```

On bénéficie

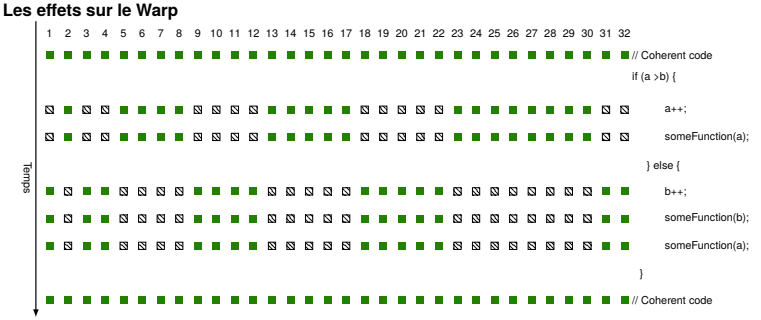
- ⋆ *d'une exécution paralléle de la partie «+» et «−» sur chacun des demi-warps en SPMT ;*
- ⋆ *éventuellement d'accès mémoire sur* $4 * 16 = 64$ *octets pour les données sur 32bits utilisées par le demi-warp ;*

3 La notion de divergence 40

Soit le code suivant :

```
// Coherent code
if (a >b) {
   a++;
   someFunction(a);
} else {
   b++;
   someFunction(b);
   someFunction(a);
}
// Coherent Code
```

Les effets sur le Warp



⟹Le **travail des threads est le même** : elles font toutes le travail des deux branches mais on annule le **résultat du travail inutile**.

Comparaison de performance divergence/pas de divergence 41

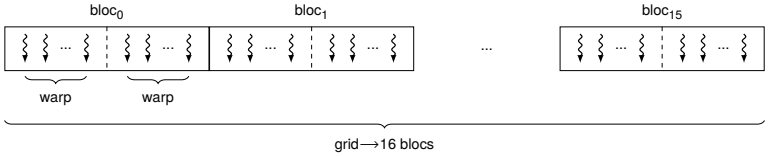
Exemple de code

```
#define WARP_SIZE 32
#define BLOCK_SIZE 2*WARP_SIZE
#define GRID_SIZE 16

  ...
  dim3 dimBlock(BLOCK_SIZE);
  dim3 dimGrid(GRID_SIZE);
  ...
  divergence<<<dimGrid, dimBlock>>>(ref_a, ref_b);
```

Dans le kernel, la numérotation de la thread est similaire à l'index du tableau de données :

```
   int a = blockIdx.x*blockDim.x + threadIdx.x;
```

Répartition du code entre les blocs et les threads



- ▷ chaque bloc dispose de 64 threads, soient 2 Warps ;
- ▷ il y a 16 blocs dans la grille.

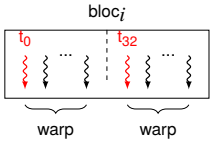
Comparaison de performance divergence/pas de divergence 42

Exemples de Kernels : «*divergence*» et «*noDivergence*»

```
__global__ void divergence(float *A, float *B){
  int a = blockIdx.x*blockDim.x + threadIdx.x;
  if((threadIdx.x % WARP_SIZE) == 0)
  {
    for(int i=0;i<ITERATIONS;i++){
      B[a]=A[a]+1;
    }
  } else
    for(int i=0;i<ITERATIONS;i++){
      B[a]=A[a]-1;
    }
}
```

th0, th32

autres threads

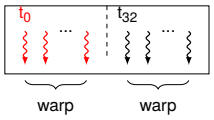


Une seule thread effectue un travail différent (la première thread du warp, en rouge sur le schéma).

```
__global__ void noDivergence(float *A, float *B){
  int a = blockIdx.x*blockDim.x + threadIdx.x;
  if(threadIdx.x >= WARP_SIZE)
  {
    for(int i=0;i<ITERATIONS;i++){
      B[a]=A[a]+1;
    }
  } else
    for(int i=0;i<ITERATIONS;i++){
      B[a]=A[a]-1;
    }
}
```

th32, th33 à th63

th0, th1 à th31



Un warp complet réalise chaque branche de la condition (en rouge sur le schéma).

Comparaison de performance divergence/pas de divergence 43

Au niveau de l'exécution

Sans divergence : un warp complet exécute une des branches de la condition.

```
xterm
pef@fpga:~/CUDA/MESURES_PERFS$ ./divergence
kernel invocation
Sans divergence
kernel execution time (msecs): 154.822556 ms
32 -> 1.000000.0 33 -> 1.000000.0 34 -> 1.000000.0 35 -> 1.000000.0 36 ->
1.000000.0 37 -> 1.000000.0 38 -> 1.000000.0 39 -> 1.000000.0 40 -> 1.000000.0
41 -> 1.000000.0 42 -> 1.000000.0 43 -> 1.000000.0 44 -> 1.000000.0 45 ->
1.000000.0 46 -> 1.000000.0 47 -> 1.000000.0 48 -> 1.000000.0 49 -> 1.000000.0
50 -> 1.000000.0 51 -> 1.000000.0 52 -> 1.000000.0 53 -> 1.000000.0 54 ->
1.000000.0 55 -> 1.000000.0 56 -> 1.000000.0 57 -> 1.000000.0 58 -> 1.000000.0
59 -> 1.000000.0 60 -> 1.000000.0 61 -> 1.000000.0 62 -> 1.000000.0 63 ->
1.000000.0 96 -> 1.000000.0 97 -> 1.000000.0 98 -> 1.000000.0 99 -> 1.000000.0
100 -> 1.000000.0 101 -> 1.000000.0 102 -> 1.000000.0 103 -> 1.000000.0 104 ->
1.000000.0 105 -> 1.000000.0 106 ->
...
```

Sans divergence : un thread par warp introduit une divergence :

```
xterm
pef@fpga:~/CUDA/MESURES_PERFS$ ./divergence yes
kernel invocation
Avec divergence
kernel execution time (msecs): 290.697205 ms
0 -> 1.000000.0 32 -> 1.000000.0 64 -> 1.000000.0 96 -> 1.000000.0 128 ->
1.000000.0 160 -> 1.000000.0 192 -> 1.000000.0 224 -> 1.000000.0 256 ->
1.000000.0 288 -> 1.000000.0 320 -> 1.000000.0 352 -> 1.000000.0 384 ->
1.000000.0 416 -> 1.000000.0 448 -> 1.000000.0 480 -> 1.000000.0 512 ->
1.000000.0 544 -> 1.000000.0 576 -> 1.000000.0 608 -> 1.000000.0 640 ->
1.000000.0 672 -> 1.000000.0 704 -> 1.000000.0 736 -> 1.000000.0 768 ->
1.000000.0 800 -> 1.000000.0 832 -> 1.000000.0 864 -> 1.000000.0 896 ->
1.000000.0 928 -> 1.000000.0 960 -> 1.000000.0 992 -> 1.000000.0
```

⟹ **Les performances vont du simple au double !**

Synchronisation & Communication 44

Lorsqu'un programme parallèle est exécuté sur le «device», la synchronisation et les communications parmi les threads doivent être réalisées à différents niveaux :

1. «Kernels» et «grids» ;
2. Blocs ;
3. Threads.

Grille & Kernels

Différents «kernels» peuvent être exécutés sur le «device».

```
void main() {
…
kernel_1<<<nblocks_1, blocksize_1>>>(fonction_argument_liste_1)
kernel_2<<<nblocks_2, blocksize_2>>>(fonction_argument_liste_2);
…
```

- ⋆ `kernel_1` va être exécuté en premier sur le «device» :
  - ◇ il va définir une grille qui contiendra `dimGrid` blocs, chacun de ces blocs contiendra `dimBlock` threads.
  - ◇ toutes les threads vont exécuter le même code spécifié par le «kernel».
- ⋆ lorsque `kernel_1` aura terminé, alors `kernel_2` sera transmis vers le «device» pour son exécution.

Attention

Le communication entre les différentes grilles est **indirecte** : elle consiste à laisse en place les données dans l'hôte ou la mémoire globale du «device» pour être utilisées par le prochain «kernel».

Les blocs

- Tous les blocs d'une grille s'exécutent indépendamment les uns des autres : il n'y a **pas de mécanisme de synchronisation** entre les blocs.
- lorsqu'une grille est lancée, les blocs sont assignés à un SM, dans un **ordre arbitraire** qui n'est pas **prédictible**.

Les communications entre les threads à l'intérieur d'un bloc sont réalisées au travers de la **mémoire partagée du bloc** :

- une variable est déclarée comme étant partagée par les threads du même bloc en préfixant sa déclaration à l'aide du mot-clé «`__shared__`».
  Cette variable est alors stockée dans la mémoire partagée du bloc.
- lors de l'exécution d'un «kernel», une version privée de cette variable est créée dans la mémoire locale de la thread.

Des temps d'accès mémoire différents

- La **mémoire partagée** associée au bloc est sur la même puce que les cœurs, «cores», exécutant les threads ;

  La communication est relativement rapide, car la SRAM, «*Static RAM*», est plus rapide que la mémoire située en dehors de la puce, «off-chip» de type DRAM ;
- chaque thread dispose d'un **accès direct à ses registres** inclus dans la puce et d'un accès direct à sa mémoire locale qui est en «off-chip». *Les registres sont beaucoup plus rapides que la mémoire locale* ;
- chaque thread peut également avoir **accès à la mémoire globale** du «device» ;
- l'accès d'une thread à la mémoire locale et globale souffre des problèmes inhérents aux communications entre puces, «interchip» : *retard, consommation de puissance, débit.*

Les threads et le SIMD : la notion de «*warp*»

Un grand nombre de threads sont exécutées sur le «device».
Un bloc qui est assigné à un SM est divisé en groupe de 32 threads *warps*. Chaque warp représente le «SIMD» du GPU.
Chaque SM peut gérer plusieurs «warps» simultanément, et lorsque certains «warps» sont bloqués à cause d'accès à la mémoire, le SM peut ordonnancer l'exécution d'un autre «warp» (suivant un scheduler).

- Les threads d'un **même bloc** peuvent se synchroniser à l'aide de la fonction `__syncthreads()`, réalisant une barrière de synchronisation entre les différents warps exécutés.
- une thread peut utiliser une opération **atomique** pour obtenir l'accès exclusif à une variable pour un réaliser une opération donnée : `atomicAdd(result, input[threadIdx.x])` *coûteux en synchronisation.*
- Chaque thread utilise ses registres et sa mémoire locale, qui utilise tous les deux de la SRAM, ce qui induit une petite quantité de mémoire disponible, mais des communications rapides et peut coûteuse en énergie.
- Chaque thread peut également utiliser la mémoire globale qui est plus lente car elle utilise de la DRAM.

La répartition des variables entre les différentes zones mémoire

Pour définir la localisation, on utilise des *préfixe* pour la déclaration ou des règles automatiques.

Déclaration	Lieu de stockage de la variable	Pénalité	Portée	Lifetime
int var;	un registre de la thread		thread	thread
int tableau[10];	la mémoire locale de la thread	*plus lent*	thread	thread
`__shared__ int var;`	la mémoire partagée du bloc		bloc	bloc
`__device__ int var;`	la mémoire globale du «device»	*plus lent*	grille	application
`__constant__ int const;`	la mémoire constante du bloc.		grille	application

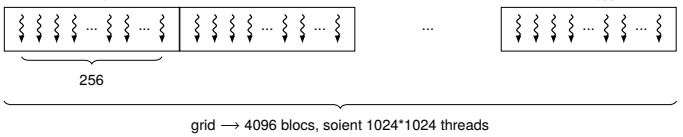
Les variables `__constant__` et `__device__` sont à déclarer en dehors de toute fonction.

Des accès décalés ou répartis

permet de choisir float ou double

```
template <typename T> __global__ void offset(T* a, int s)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x + s;
  a[i] = a[i] + 1;
}
template <typename T> __global__ void stride(T* a, int s)
{
  int i = (blockDim.x * blockIdx.x + threadIdx.x) * s;
  a[i] = a[i] + 1;
}
```

```
int blockSize = 256; // taille du bloc
...
int n = nMB*1024*1024/sizeof(T); // si nMB = 4, n=1048576
...
checkCuda( cudaMalloc(&d_a, n * 33 * sizeof(T)) ); // 33*1024*1024 var. type T
...
offset<<<n/blockSize, blockSize>>>(d_a, i); // la grille est de 4096 blocs
...
  if (bFp64) runTest<double>(deviceId, nMB); // on utilise des doubles
  else       runTest<float>(deviceId, nMB); // ou des floats
```



Décalage des accès ou «*offset*»

On décale 32 fois :

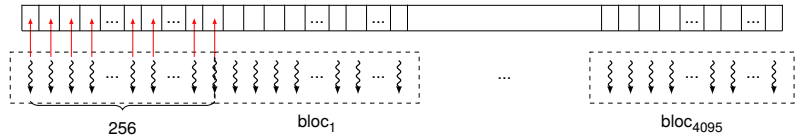
```
for (int i = 0; i <= 32; i++) {
  offset<<<n/blockSize, blockSize>>>(d_a, i);
  ...
  printf("%d, %f\n", i, 2*nMB/ms); // on mesure le débit
}
```

Le travail du Kernel :

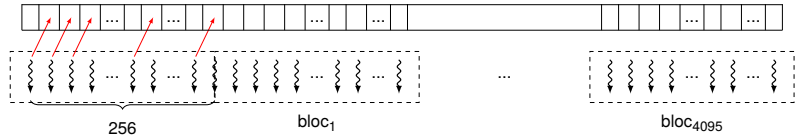
```
__global__ void offset(T* a, int s)
{
  int i = blockDim.x * blockIdx.x
          + threadIdx.x + s;
  a[i] = a[i] + 1;
}
```

addition

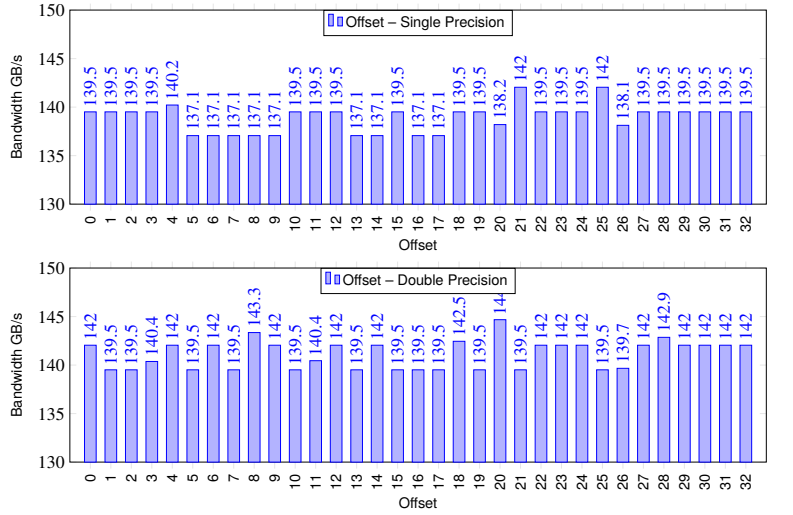
Offset de zéro



Offset de un



Aggrégation, «*coalescence*», des accès mémoire : décalage



Aggrégation, «*coalescence*», des accès mémoire : saut

Saut des accès ou «*stride*»

On accède à 2 fois, puis 3 fois, *etc* :

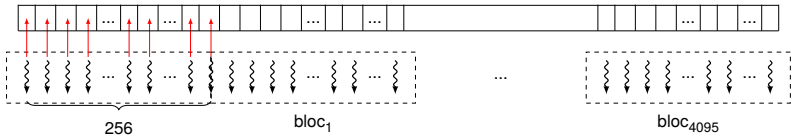
```
for (int i = 0; i <= 32; i++) {
  stride<<<n/blockSize, blockSize>>>(d_a, i);
  ...
  printf("%d, %f\n", i, 2*nMB/ms); // on mesure le débit
}
```

Le travail du Kernel :

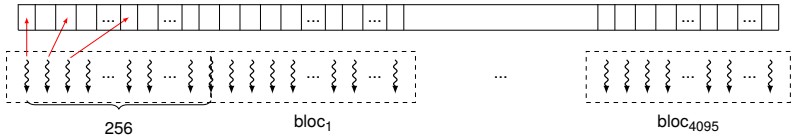
```
__global__ void offset(T* a, int s)
{
  int i = blockDim.x * blockIdx.x
          + threadIdx.x * s;
  a[i] = a[i] + 1;
}
```

multiplication

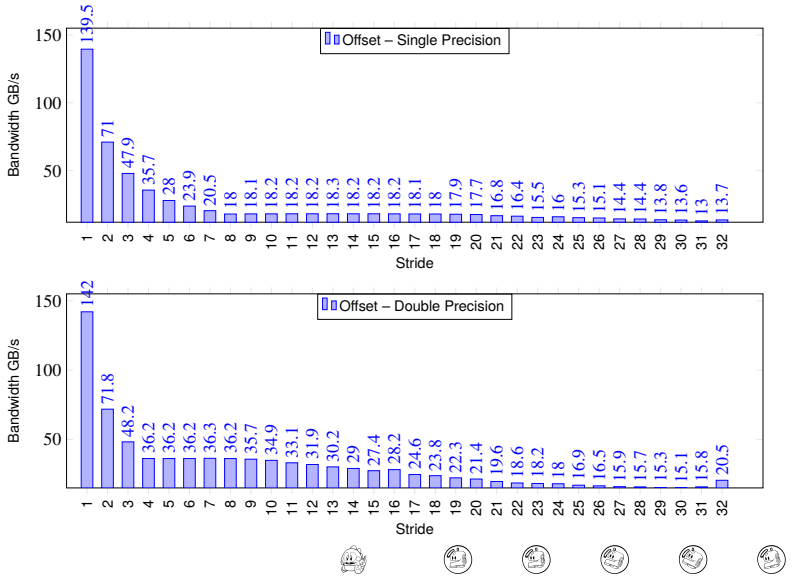
Saut de zéro



Saut de un



Aggrégation, «*coalescence*», des accès mémoire : saut

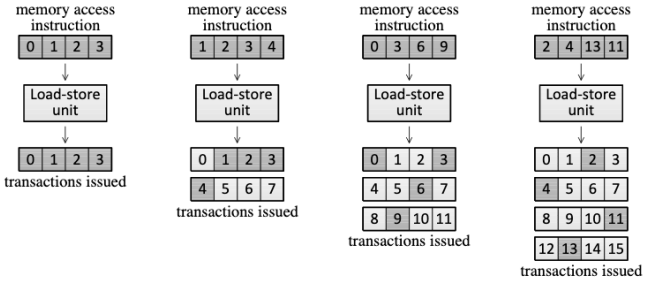


Aggrégation, «*coalescence*», des accès mémoire

Que se passe-t-il au niveau du contrôleur de mémoire

Des accès **non alignés** et non contigüs à la mémoire nécessitent plusieurs «*transactions*» mémoire.

Une **transaction** correspond à un accès mémoire suivant la **taille du bus de données**.
Exemple : sur une carte GTX 1060, le bus de données est de 192bits, soient une transaction de 6 données sur 32bits simultanément.



Sur le schéma la taille du bus est de $4 * 32bits$ *soient 128bits.*

Optimisation de la vitesse en localisant mieux les données

Exemple de code utilisant la «*shared memory*»

```
// Calcul de différence de données adjacentes
// calculer result[i] = input[i] - input[i-1]
__device__ int result[N];

__global__ void adj_diff_naive(int *result, int *input)
{
  // calculer l'identifiant de la thread en fonction
  // de sa position dans la grille
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // Utiliser des variables locales à la thread
    int x_i = input[i];
    int x_i_minus_one = input[i-1];
    // Deux accès sont nécessaires pour i et i-1
    result[i] = x_i - x_i_minus_one;
  }
}
```

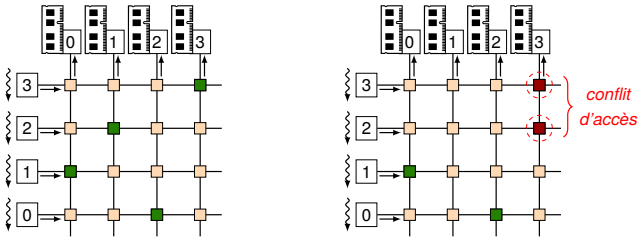
```
// version optimisée
__device__ int result[N];
__global__ void adj_diff(int *result, int *in
put)
{
  // raccourci pour threadIdx.x
  int tx = threadIdx.x;
  // allouer un __shared__ tableau,
  // un élément par thread
  __shared__ int s_data[BLOCK_SIZE];
  // chaque thread lit un élément dans s_data
  unsigned int i = blockDim.x * blockIdx.x
                 + tx;
  s_data[tx] = input[i];

  // éviter une race condition: s'assurer
  // des chargements avant de continuer
  __syncthreads();
  if(tx > 0)
    result[i] = s_data[tx] - s_data[tx-1];
  else if(i > 0)
  { // traiter les accès aux bords du bloc
    result[i] = s_data[tx] - input[i-1];
  }
}
```

Cette optimisation se traduit par une nette amélioration des performances : $36,8GB/s$ *vs* $57.5GB/s$.

Accès concurrent à la mémoire partagée, «*Shared Memory*»

La mémoire partagée est **répartie** en module de mémoire ou «*bank*» :

- chaque module de mémoire ou «*bank*» est de **taille identique** ;
- chaque module est **connecté** à chaque cœur par un réseau de type «*crossbar*» ;
- chacune de ces «*banks*» peut être **accédée simultanément** s'il n'y a pas de conflit d'accès :

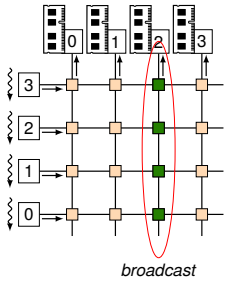


Accès simultanés *Conflit : les accès sont **sérialisés**.*

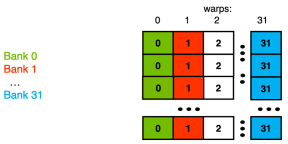
- la mémoire partagée est constituée de **32 banks** pour toutes les architectures CUDA : de 4.5 à 8.6 ;
- 32 banks pour un warp de 32 threads ⟹ des **conflits** peuvent arriver !
- chaque module à un **débit** de 32bits par cycle d'horloge ;
- l'accès en écriture ou en lecture de n addresses quelconques dans des **banks distinctes** peut être fait **simultanément** ⟹ le débit peut atteindre n fois le débit d'une seule bank !

Accès concurrent à la mémoire partagée, «*Shared Memory*»

- Si **toutes les threads** accèdent en lecture à la **même donnée** sur 32bits comprise dans la **même bank**, alors cette valeur est accédée en un seul cycle et «*broadcastée*» à chaque thread :



Comment les données sont organisées entre les différentes banks ?



les données sont réparties de manières régulières entre les différentes banks : les données successives sont réparties dans des banques successives ;

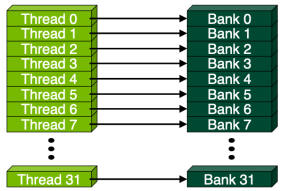
Accès concurrent à la mémoire partagée, «*Shared Memory*»

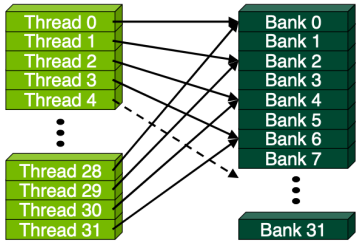
Accès sans conflit : exemple pour 16 threads

```
__shared__ float partage[32];

float data = partage[threadIdx.x];
```

Les données sont réparties entre les différentes banks :

- ▷ `partage[0]` *est dans la bank 0 ;*
- ▷ *...*
- ▷ `partage[31]` *est dans la bank 31 ;*





Conflit :

```
__shared__ double partage[32];

double data = partage[threadIdx.x]
```

Ici, les données sont des `double`*, soient 64 bits ou 2*32bits :*

- ▷ `partage[0]` *est dans la bank 0 pour ses premiers 32bits et dans la bank 1 pour ses derniers 32bits*
- ▷ `partage[16]` *est dans la bank 0 pour ses premiers 32bits et dans la bank 1 pour ses derniers 32bits*

⟹**Conflits d'accès double** !

Cela peut également arriver avec : `__shared__ short partage[32];`

Un `short` *est sur 16bits soient un conflit d'accès double sur 32bits*

Accès concurrent à la mémoire partagée, «*Shared Memory*» 57

Encore des conflits ?

Les données sont groupées par 32bits :

```
__shared__ char partage[32];

char valeur = partage[threadIdx.c];
```

Solution :

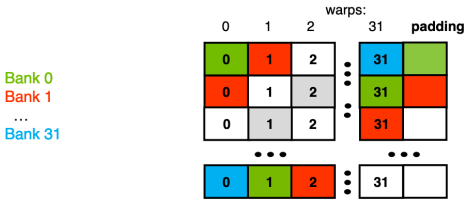
```
__shared__ char partage[32];

char data = shared[4 * threadIdx.x];
```

Comment résoudre les conflits d'accès ?

En décomposant un double en deux :
⟹*pas bien !*

```
__shared__ intshared_lo[32];
__shared__ intshared_hi[32];
double dataIn;
shared_lo[BaseIndex+tid]= __double2loint(dataIn);
shared_hi[BaseIndex+tid]= __double2hiint(dataIn);
double dataOut = __hiloint2double(shared_hi[BaseIndex+tid],
                                  shared_lo[BaseIndex+tid]);
```

En utilisant du «*padding*» :

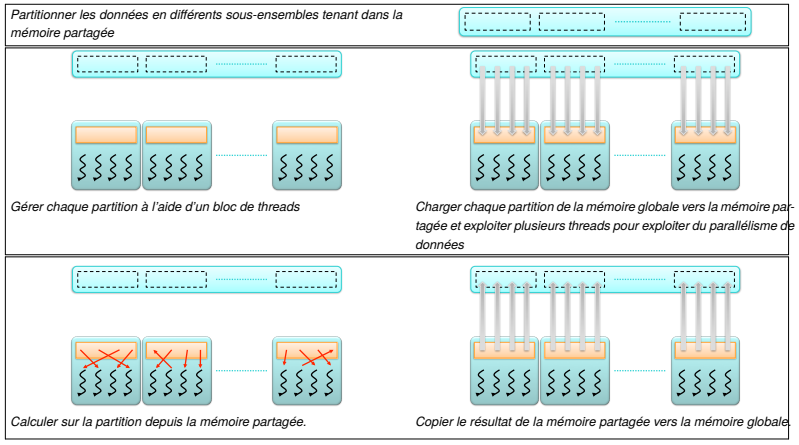


On ajoute une colonne qui ne sert qu'à réaliser un décalage et éviter le conflit.

Stratégie de développement 58

- la **mémoire globale** réside dans la mémoire globale du «device» en DRAM (plus lente) ;
- il faut **partitionner les données** pour tirer parti de la **mémoire partagée** plus rapide :
  - utiliser la technique vu sur le transparent précédent ;
  - utiliser une méthode «*divide and conquer*»

Partitionner les données en différents sous-ensembles tenant dans la mémoire partagée



Comment aller plus loin ?
Exécuter plusieurs instructions à la fois dans la **même thread**

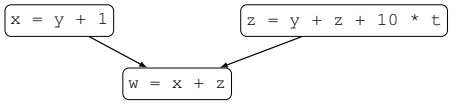
Analyser le potentiel parallèle : le graphe de précédence 60

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

Analyser le potentiel parallèle

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 $O(log_2(n))$.

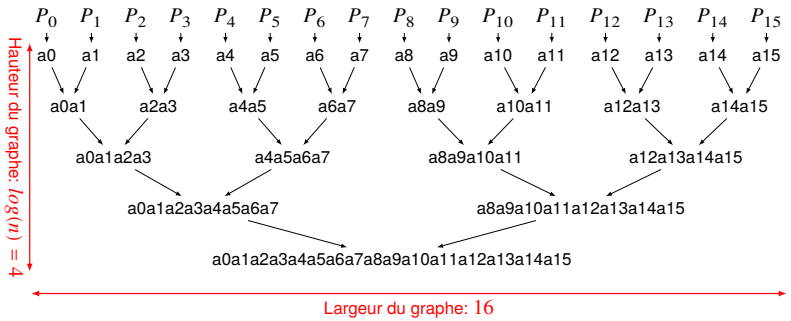
Mais ici, on exploite une **propriété de la multiplication** : son **associativité**.

Exploiter le potentiel parallèle

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.



Ici, l'**algorithme parallèle** à l'aide de n processus est en $O(log_2(n))$

Exploiter le potentiel parallèle

Parallélisme de données

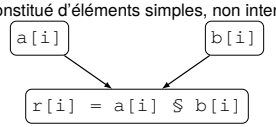
Dans ce cas là, la même opération (SIMD) ou le même programme (SPMD) est effectué sur des données différentes.

Ce **parallélisme** est souvent exploitable pour des programmes travaillant sur des **vecteurs**.

Exemple : soient les vecteurs `a[ ]`, `b[ ]` et `r[ ]` Calculer `r[i] = a[i] § b[i]` pour `i = 0, .. , n - 1` avec § étant un opérateur quelconque

Il existe deux façons de l'exploiter :

– le mode **parallèle** sur les données : Les opérations `r[i] = a[i] § b[i]` sont indépendantes, le graphe de précédence est constitué d'éléments simples, non interconnectés :



Cette forme de parallélisme est dite «*parfaite*».

Il nécessite parfois une **fusion des résultats** obtenus pour obtenir le **résultat final** (exemple : la somme de tous les `r[i]`).

Certain langages de programmation parallèle comme HPF, «High Performance Fortran»*, ou OpenMP disposent d'instructions et d'outils automatiques pour l'exploiter (**opération de réduction**).*

Exploiter le potentiel parallèle

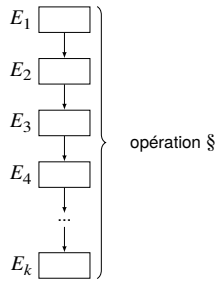
Le mode pipeline

On suppose qu'une opération § :

– est **complexe à effectuer**

– peut être découpée en k sous-calculs successifs E_1, ... , E_k **plus simples**.

$$a\S b = E_k(E_{k-1}((((...(E_1(a,b))...))))$$



Ce qui donne :

- à l'étape 1 : on calcule $r[0]_1 = E_1(a[0], b[0])$;
- à l'étape 2 : on calcule $r[0]_2 = E_2(r[0]_1)$;
- *etc.*

Au bout de k étapes, le résultat $r[0] = r[0]_k = a[0]\S b[0]$ est obtenu en sortie de E_k.

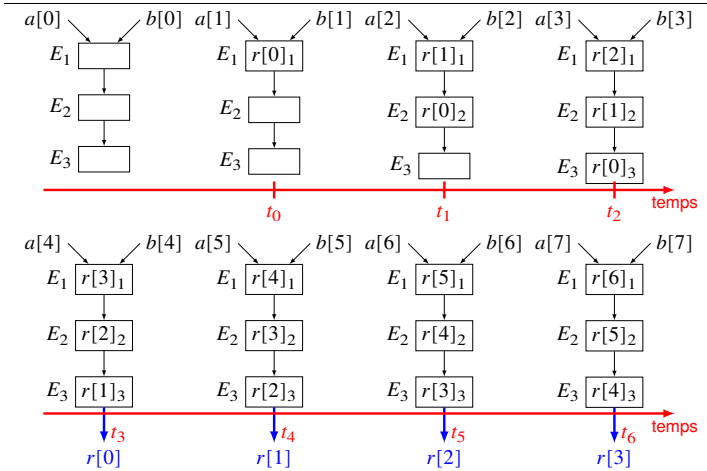
On alimente la série d'opérateurs E_i de **manière continue** pour obtenir l'effet pipeline.

Chaque E_i est appelé «*étage*» du pipeline $\Longrightarrow$ il est choisi pour durer «*un cycle d'horloge*».

Le **temps de calcul global** est alors $k + n$–1 au lieu de $k * n$.

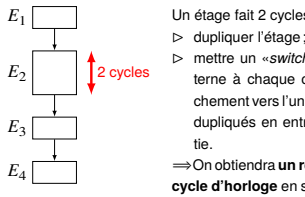
L'accéleration est de $\frac{k*n}{k+n-1} \asymp k$ pour des grandes valeurs de n.

Exploiter le potentiel parallèle : Effet pipeline



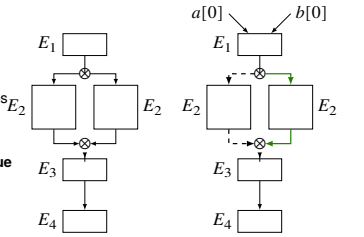
Si chaque E_i prend **un cycle d'horloge** alors une valeur sort du pipeline **à chaque cycle d'horloge**.
⟹ On est passé d'une opération § sur **3 cycles d'horloge** à une opération § sur un **cycle d'horloge** !

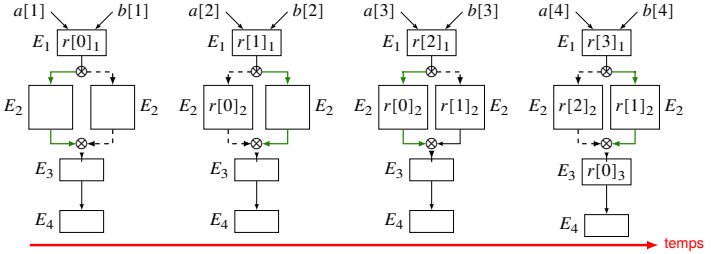
Exploiter le potentiel parallèle : Effet pipeline



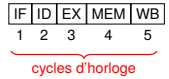
Un étage fait 2 cycles ?
- ▷ dupliquer l'étage ;
- ▷ mettre un «*switch*», ⊗, qui alterne à chaque cycle le branchement vers l'un des deux étages dupliqués en entrée et en sortie.

⟹On obtiendra **un résultat à chaque cycle d'horloge** en sortie du pipeline !

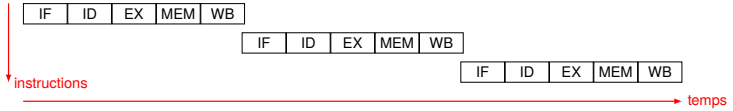




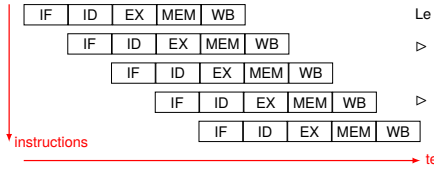
Pipeline et Processeur de type RISC, «*Reduced Instruction Set Computer*»



1. IF «*Instruction Fetch*» :charge l'instruction à exécuter dans le pipeline ;
2. ID «*Instruction Decode*» : décode l'instruction et adresse les registres ;
3. EX «*Execute*» : exécute l'instruction (par la ou les unités arithmétiques et logiques).
4. MEM «*Memory*» :
   - ◇ STORE : registre vers mémoire (accès en écriture) ;
   - ◇ LOAD : mémoire vers registre (accès en lecture) ;
5. WB «*Write Back*» : stocke le résultat dans un registre.
   La source de ce résultat peut être :
   - ◇ la mémoire (à la suite d'une instruction LOAD),
   - ◇ l'unité de calcul (à la suite d'un calcul à l'étape EX)
   - ◇ un registre (cas d'une instruction MOV).



Pipeline

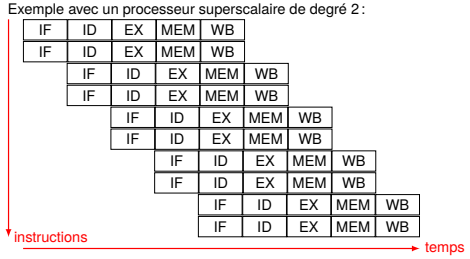


Le **pipeline** permet **d'accélérer** le travail du processeur :
- ▷ **sans pipeline** : **15 cycles d'horloges** pour exécuter **3 instructions** ;
- ▷ **avec pipeline** : **9 cycles** pour exécuter **5 instructions** puis on a **une instruction par cycle** après !

Architecture superscalaire et pipeline

Une architecture «*superscalaire*» dispose de plusieurs pipelines en parallèle.

Exemple avec un processeur superscalaire de degré 2 :



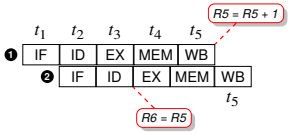
2 instructions sont chargées simultanément depuis la mémoire.

Chaque pipeline peut être **spécialisé** dans le traitement **d'un certain type d'instruction** : seules des instructions de types **compatibles** peuvent être exécutées simultanément dans le pipeline associé.

Problèmes avec l'utilisation de pipeline ?

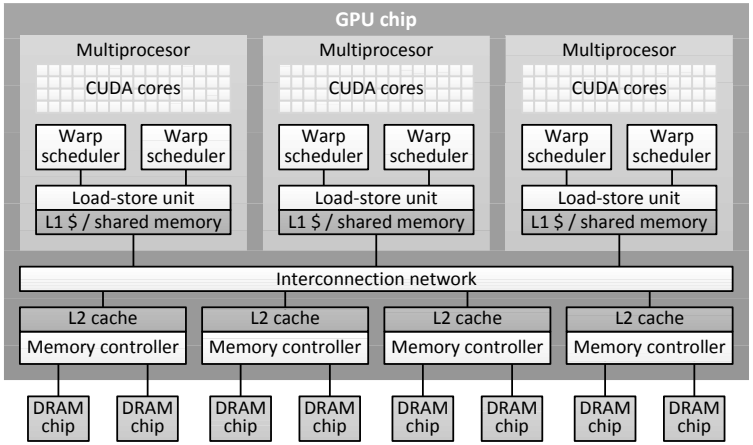
```
❶ADD 1 to R5
❷COPY R5 to R6
```

- ▷ ❶ commence à t_1 et termine à t_5
- ▷ ❷ commence à t_2 et finit à t_6



⟹ La valeur utilisée par l'instruction ❷ n'est **pas la bonne** !
⟹ C'est au **compilateur** de tenir compte de cela !

Source d'ILP sur le GPU ? Le scheduler



Comparaison GPU/CPU

Les **CPUs** sont optimisés pour la «*latence*» :
- **pipeline**, «***out-of-order***», **superscalaire** ;
- **mémoire cache**, contrôleur mémoire intégré ;
- exécution **spéculative**, «*branch prediction*» prédiction de condition ;
- les **cœurs** occupent une petite portion du circuit ;

Les **GPUs** sont optimisés pour le débit :
- des centaines de **cœurs** de calcul ;
- coût minimal d'ordonnancer l'exécution de milliers de threads ;
- les cœurs de calcul occupent la majeur partie du circuit.

Le cas de **blocage** d'une *thread* sont :
- ▷ CPU/GPU : **l'échec** de l'accès à une mémoire au travers du **cache** ⟹ attente de données depuis la mémoire ;
- ▷ CPU : **échec** de la **prédiction** de condition : la «*branche*» de la condition choisie n'est pas la bonne ;
- ▷ CPU/GPU :une **dépendance** de donnée : une instruction est en attente du résultat d'une autre instruction.
- ◦ **SIMD**, «*Single Instruction Multiple Data*» :
  - ◇ on calcule les différents éléments d'un **vecteur** en parallèle ;
  - ◇ on découpe le problème en **petits vecteurs**, calculés les uns après les autres.
  - ◇ le hardware supporte de **l'arithmétique** sur de grandes valeurs ;
- ◦ **SMT**, «*Simultaneous MultiThreading*» :
  - ◇ les instructions de différentes threads sont exécutées en parallèle : lorsqu'une thread est bloquée, une autre thread peut s'exécuter ;
  - ◇ décomposer un problème en différentes tâches et les assigner à différentes threads ;
  - ◇ le hadware supporte le multi-trheading : l' «*Hyper-Threading*» d'Intel ;
- ◦ **SIMT**, «*Simple Instruction Multiple Threads*»:
  - ◇ traitement de vecteur et multi-threading **léger** ;
  - ◇ le **warp** est l'unité d'exécution : à chaque cycle il exécute la même instruction ;
  - ◇ ordonnancement de threads et changement de contexte rapide entre différents Warps permet de minimiser les blocages dus à des accès mémoire non résolus.

Occupancy

Au lancement d'un «*kernel*», on choisit :
- ☐ **nombre de bloc** de threads ;
- ☐ **nombre de warps** par bloc de threads ;
- ☐ la **taille de la mémoire partagée** par bloc de threads.

Il est recommandé de déclencher plus de bloc de threads que l'on ne peut en exécuter en même temps.

Lorsque **tous les warps** d'un bloc ont terminé, le **prochain bloc** de threads est déclenché.

Le **nombre de blocs** de threads exécutés en même temps dépend :
- ▷ de la **taille mémoire partagée** allouée par bloc de threads ;
- ▷ du **nombre de registres** utilisés par chaque warp ;

L'**occupancy** est :

$$\frac{\text{nombre de warps exécutés en même temps}}{\text{nombre maximal de warps exécutables en même temps}} \text{ exprimé en pourcentage.}$$

Le nombre maximal de warps exécutables en même temps dépends de la «Compute capability» *de l'architecture de la carte GPU, par exemple une GTX 1060 a une* «compute capability» *de 6.1.*

Les différentes «*compute capability*»

CUDA Major.Minor

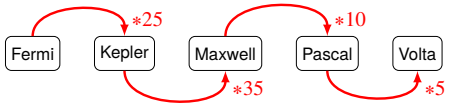
Certaines générations amènent des nouveautés : Tensors, Ray Tracing.

		CUDA					Pro			Pro
architecture	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing	Ampere	Lovelace	Hopper
année	2006	2010	2012	2014	2016	2017	2018	2020	2022	2024
major	1	2	3	5	6	7	7.5	8	9	10
		sm_20	sm_30	sm_50	sm_60	sm_70	sm_75	sm_80	sm_90?	sm_100c?
			sm_35	sm_52	sm_61	sm_72		sm_86		
			sm_37	sm_53	sm_62					

spécial HPC TensorCores

RT Cores, «ray-tracing»

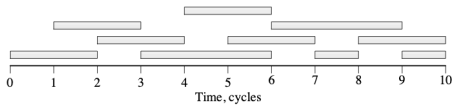
Accélération obtenue à chaque changement d'architecture



Comment mesurer le temps d'exécution d'un programme ?

- ▷ en exécution **séquentielle** : le temps total d'exécution est la somme du temps d'exécution de chaque instruction ;
- ▷ en exécution **concurrente** : le temps d'exécution d'une instruction peut se superposer à celui d'une autre et leur somme ne correspond au temps total d'exécution.

Schéma d'activité d'une exécution concurrente :



chaque rectangle :

- ▷ *correspond à une instruction, un warp, une transaction mémoire etc.*
- ▷ *temps de démarrage ;*
- ▷ *temps de fin ;*

L'activité globale se déroule entre 0 et le cycle 10.

Les métriques :

- ☐ «*Latence*» : la **valeur moyenne** des différentes latences de chaque élément c-à-d la différence entre leur fin et leur début.
  Ici, les latences individuelles sont 1, 2 ou 3, soit une latence moyenne de 2 ;
- ☐ «*Débit*», «*throughput*» ou rythme d'exécution : le nombre d'éléments qui se retrouve dans l'intervalle de temps choisi divisé par la durée de cet intervalle.
  Ici, 10 éléments sur une période de 10 cycles, soit un élément par cycle ;
- ☐ «*Concurrence*» : le nombre d'éléments exécutés en même temps. Il peut être mesuré à chaque instant ou donner une valeur moyenne sur la durée de l'intervalle.
  Ici, la concurrence varie de 1 à 3 pour une valeur moyenne de 2.

Little's law

$$\textbf{concurrence moyenne} = \textbf{latence moyenne} * \textbf{débit}$$

*Cette loi permet d'évaluer la «*concurrence*» nécessaire pour atteindre un certain débit d'exécution d'instructions comme celles arithmétiques et celles pour l'accès mémoire.*

En utilisant la **loi de Little** sur les instructions :

- ☐ **latence d'instruction** : latence de «*dépendance de registre*» : c'est la partie du temps d'exécution total
  - ◇ qui débute quand l'instruction est «*issued*», c-à-d émise par l'unité de contrôle ;
  - ◇ qui finit quand l'instruction suivante dépendant de la valeur d'un registre peut être émise.

  Une instruction **dépendante de la valeur d'un registre** est une instruction qui utilise en **entrée**, la **sortie** d'une instruction donnée. Cette «*sortie*» est faite dans un registre.
- ☐ **débit** d'exécution d'instruction : nombre d'instructions exécutées divisé par un interval de temps ;
- ☐ **concurrence** : nombre d'instructions exécutées en même temps.

$$\text{débit d'instructions} = \frac{\text{instructions en cours d'exécution}}{\text{latence d'instruction}}$$

Exemple : sur un GPU d'**architecture Maxwell**, la latence d'une **instruction arithmétique** de base est de **6 cycles**.

- ▷ L'exécution d'**une instruction arithmétique à la fois** correspond à un débit de $1/6$ instructions par cycle ou IPC, «*Instruction Per Cycle*».
- ▷ L'exécution de **deux instructions arithmétiques à la fois** donne un débit de $1/3$ IPC.
- ▷ L'exécution de **trois instructions à la fois** donne un débit de $1/2$ IPC,
- ▷ *etc.*

Dans l'architecture Maxwell, on dispose de 128 cœurs CUDA par SM, et on ne peut pas exécuter d'instructions arithmétique plus rapidement que 4 IPC par SM, car 1 instruction réalise 32 opérations arithmétiques (un warp).
Cette valeur représente le **débit de crête**, ou «*peak throughput*».
*La valeur dépend du type d'instruction mais aussi des conflits d'accès aux «*banks*», à la coalescence/divergence* etc.

Comment atteindre le débit de crête ou le «*peak throughput*» ?

En utilisant la **loi de Little**, on calcule : latence instruction $*$ débit de crête $= 6 * 4 = 24$,
$\Longrightarrow$il faut **24 instructions arithmétiques** par SM pour l'atteindre.

En utilisant la **loi de Little** sur l'accès à la mémoire :

- ▷ latence d'accès mémoire : dépend du fait que la mémoire demandée ne soit pas dans le cache, qu'elle soit de 32bits, qu'elle soit aggrégée, «*coalescence*» et que seulement un ou un peu plus de ces accès soient demandés.
  exemple : sur l'architecture Maxwell, la latence d'un accès mémoire est de 368 cycles.
  Le débit de crête est de 224GB/s.
  Ce débit correspond à 0,086 IPC par SM.
  Pour le déterminer, on calcul : $\frac{\text{débit}}{\text{clock rate}*\text{nombre de SM}*\text{nombre d'octets demandés par instruction}}$, ce qui donne : $\frac{224}{1{,}266*16*128} = 0{,}086$
  En utilisant la loi de Little, on obtient : concurrence $= 368 * 0{,}086 = 32$, ce qui veut dire qu'il faut 32 accès mémoire pour atteindre ce débit de crête.

Occupancy vs Instruction concurrency

La **concurrence** peut être définie de deux manière différentes :

- ☐ le **nombre de warps** exécutés en même temps ;
- ☐ le **nombre d'instructions** exécutées en même temps ;

Est-ce la même chose ? Non !

Exploiter de l'ILP, «*Instruction Level Parallelism*» dans un code signifie que ce code permet d'exécuter **plus d'une instruction en même temps** et pour le **même warp**.

Mais seulement si ces deux instructions sont **indépendantes**, c-à-d que le registre de sortie de l'une n'est pas utilisé comme entrée de l'autre$\Longrightarrow$dépendance de registre.

Comment faire ?

En utilisant **moins de warps** !

Par exemple : si en moyenne deux accès mémoires sont exécutés dans chaque warp en même temps, la concurrence d'instruction recherchée de 32 instructions d'accès mémoire par SM peut être atteinte en utilisant 16 warps par SM.

Illustration : masquer la latence «*latency hiding*»

```
x = a + b;// prends ≈20 cycles pour s'exécuter
y = a + c;// indépendante, peut démarrer n'importe quand
 bloqué
z = x + d;// dépendante, doit attendre x
```

Latence : temps requis pour réaliser une opération :

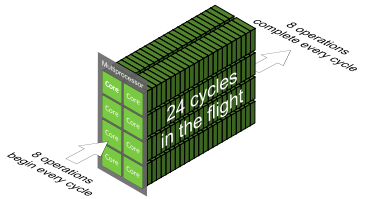
- – 20 cycles pour une opération arithmétique, 400 cycles pour une opération mémoire ;
- – on ne peut pas démarrer une instruction **avec une dépendance** pour masquer ce temps d'attente ;
- – on ne peut pas **masquer** cette latence en la **recouvrant** avec une autre opération ;

Latence $\neq$ débit :
les opérations arithmétiques sont 20x plus rapides que les opérations mémoires...
mais l'un est un temps d'attente et l'autre un débit

Le **débit** est le nombre d'opérations que l'on peut finir par cycle

- ☐ **arithmétique** : 1,3Tflop/s = 480 ops/cycle avec une opération MAD, «*multiply-add*» ;
- ☐ **mémoire** : 177GB/s= 32 ops/cycle avec une opération d'echange sur 32bits.

Comment cacher la latence ? Comment atteindre le débit de crête ?



Needed parallelism = **Latency** x **Throughput**

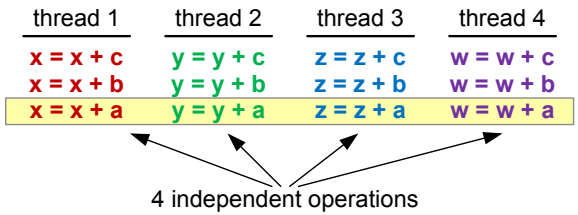
Augmenter le **nombre d'opérations programmables** par le **scheduler** du GPU :

GPU model	Latency (cycles)	Throughput (cores/SM)	Parallelism (operations/SM)
G80-GT200	≈24	8	≈192
GF100	≈18	32	≈576
GF104	≈18	48	≈864

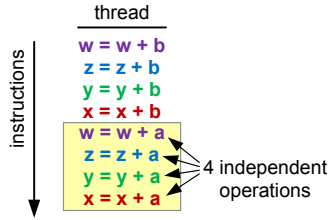
En utilisant le TLP, «*Thread Level Parallelism*» et l'ILP, «*Instruction Level Parallelism*» .

ILP vs TLP

Thread Level Parallelism : augmenter le nombre de threads



Instruction Level Parallelism : augmenter le nombre d'instructions par thread



ILP vs TLP

ILP=1

```
#pragma unroll UNROLL
for(int i=0; i<N; i++ )
{
    a = a * b + c;
}
```

ILP=2

```
#pragma unroll UNROLL
for(int i=0; i<N; i++ )
{
    a = a * b + c;
    d = d * b + c;
}
```

ILP=3

```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ )
{
    a = a * b + c;
    d = d * b + c;
    e = e * b + c;
}
```

Observations

- Si une opérande n'est pas prête, alors le warp est bloqué.
- Quand un warp est **bloqué**, on effectue un **changement de contexte** vers un warp capable de s'exécuter.
- Les **registres** et la **mémoire partagée** sont alloués par bloc aussi longtemps que le bloc est **actif**.
- Quant un **bloc** est **actif**, il reste **actif** tant que toutes les threads du bloc ne sont pas **terminées**.
- Le changement de contexte est **très rapide** parce que les registres et la mémoire partagée ne doivent être **ni sauvegardés ni restaurés**.

But : avoir suffisamment de transaction en vol pour saturer le bus de mémoire :

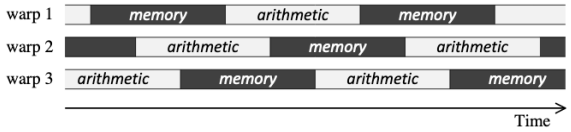
▷ la latence peut être **masquée** en ayant plus de transaction en vol ;
▷ **augmenter** le nombre de **threads actives** ou le parallélisme niveau instruction, ILP, «*Instruction Level Parallelism*»

Règles

▷ Plus le nombre d'ILP ↗ moins il faut de threads pour atteindre la puissance de crête, «*peak*» ;
▷ Plus le nombre de threads ↘ :
  ◇ plus on dispose de registres par threads ;
  ◇ plus l'**occupancy** ↘ ;
▷ le registre est la mémoire la plus rapide disponible, plus rapide que la mémoire partagée ;
les variables locales à un kernel sont allouées sous forme de registres
▷ plus l'**occupancy**, liée au TLP, ↘ plus les performances dépendent de l'ILP ;

Occupancy vs Instruction concurrency

Exemple :



▷ deux types d'instructions : arithmétique et accès mémoire ;
▷ chaque warp alterne entre accès mémoire et opération arithmétique : moitié du temps pour l'une et moitié du temps pour l'autre ;
▷ avec 3 warps : le nombre d'opérations arithmétique est de 1,5 en moyenne, et le nombre d'acc`es mémoire est également de 1,5 ;

Si on recherche également la concurrence de 32 accès mémoire pour atteindre le **débit de crête**, on a besoin de 64 warps exécutés en même temps.

Sur le transparent précédent, on avait juste besoin de 16 warps avec l'**ILP**.
On **améliore** encore en utilisant plus de type d'instruction : SFU, FP, accès mémoire global, accès mémoire partagée.

«*Warp latency*» et «*throughput*»

On considère les warps comme élément dans le **loi de Little** :

▷ **latence** : différence entre les temps de début et de fin d'un warp ;
▷ **concurrence** : nombre de warps exécutés en même temps ⟹ **Occupancy** !
▷ **débit** : nombre de warps exécutés divisé par le temps total d'exécution.

occupancy moyenne = warp latency moyenne ∗ débit de warp

Exemple d'ILP : on «*déroule le kernel*»


```
#define N_ITERATIONS 8192
#define BLOCKSIZE   512
```

```
__global__ void kernel0(float *d_a, float *d_b, float *d_c, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) {
        float a = d_a[tid];
        float b = d_b[tid];
        float c = d_c[tid];
        for (unsigned int i = 0; i < N_ITERATIONS; i++) {
            a = a * b + c;
        }
        d_a[tid] = a;
    }
}
kernel0 <<<iDivUp(N, BLOCKSIZE), BLOCKSIZE >>>(d_a, d_b, d_c, N);
```

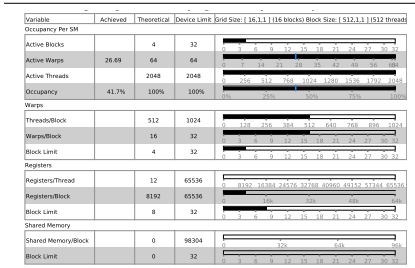
```
__global__ void kernel1(float *d_a, float *d_b, float *d_c, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N / 2) {
        float a1 = d_a[tid];
        float b1 = d_b[tid];
        float c1 = d_c[tid];

        float a2 = d_a[tid + N / 2];
        float b2 = d_b[tid + N / 2];
        float c2 = d_c[tid + N / 2];

        for (unsigned int i = 0; i < N_ITERATIONS; i++) {
            a1 = a1 * b1 + c1;
            a2 = a2 * b2 + c2;
        }
        d_a[tid] = a1;
        d_a[tid + N / 2] = a2;
    }
}
kernel1 <<<iDivUp(N / 2, BLOCKSIZE), BLOCKSIZE >>>(d_a, d_b, d_c, N);
```

Exemple d'ILP : on «*déroule le kernel*» 81

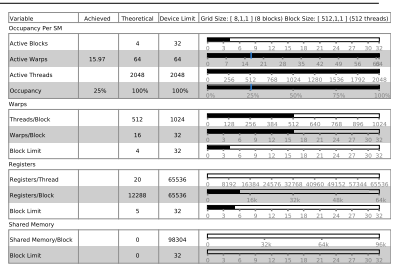
```
__global__ void kernel2(float *d_a, float *d_b, float *d_c, int N) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < N / 4) {
        float a1 = d_a[tid];
        float b1 = d_b[tid];
        float c1 = d_c[tid];

        float a2 = d_a[tid + N / 4];
        float b2 = d_b[tid + N / 4];
        float c2 = d_c[tid + N / 4];

        float a3 = d_a[tid + N / 2];
        float b3 = d_b[tid + N / 2];
        float c3 = d_c[tid + N / 2];

        float a4 = d_a[tid + 3 * N / 4];
        float b4 = d_b[tid + 3 * N / 4];
        float c4 = d_c[tid + 3 * N / 4];

        for (unsigned int i = 0; i < N_ITERATIONS; i++) {
            a1 = a1 * b1 + c1;
            a2 = a2 * b2 + c2;
            a3 = a3 * b3 + c3;
            a4 = a4 * b4 + c4;
        }
        d_a[tid] = a1;
        d_a[tid + N / 4] = a2;
        d_a[tid + N / 2] = a3;
        d_a[tid + 3 * N / 4] = a4;
    }
}
kernel2 <<<iDivUp(N / 4, BLOCKSIZE), BLOCKSIZE >>>(d_a, d_b, d_c, N);
```



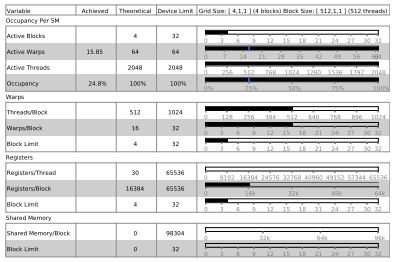

Exemple d'ILP : on «*déroule le kernel*» 82



Variable	Achieved	Theoretical	Device Limit	Grid Size: [16,1,1] (16 blocks) Block Size: [512,1,1] (512 threads)
Occupancy Per SM				
Active Blocks		4	32	
Active Warps	26.69	64	64	
Active Threads		2048	2048	
Occupancy	41.7%	100%	100%	
Warps				
Threads/Block		512	1024	
Warps/Block		16	32	
Block Limit		4	32	
Registers				
Registers/Thread		12	65536	
Registers/Block		8192	65536	
Block Limit		8	32	
Shared Memory				
Shared Memory/Block		0	98304	
Block Limit		0	32	



Variable	Achieved	Theoretical	Device Limit	Grid Size: [8,1,1] (8 blocks) Block Size: [512,1,1] (512 threads)
Occupancy Per SM				
Active Blocks		4	32	
Active Warps	15.97	64	64	
Active Threads		2048	2048	
Occupancy	25%	100%	100%	
Warps				
Threads/Block		512	1024	
Warps/Block		16	32	
Block Limit		4	32	
Registers				
Registers/Thread		20	65536	
Registers/Block		12288	65536	
Block Limit		5	32	
Shared Memory				
Shared Memory/Block		0	98304	
Block Limit		0	32	



Variable	Achieved	Theoretical	Device Limit	Grid Size: [4,1,1] (4 blocks) Block Size: [512,1,1] (512 threads)
Occupancy Per SM				
Active Blocks		4	32	
Active Warps	15.85	64	64	
Active Threads		2048	2048	
Occupancy	24.8%	100%	100%	
Warps				
Threads/Block		512	1024	
Warps/Block		16	32	
Block Limit		4	32	
Registers				
Registers/Thread		30	65536	
Registers/Block		16384	65536	
Block Limit		4	32	
Shared Memory				
Shared Memory/Block		0	98304	
Block Limit		0	32	



Exemple d'ILP : l'architecture 83

[0] NVIDIA GeForce GTX 1060 6GB

GPU UUID	GPU-1cb95576-2e2c-d5c0-57dc-4c74d3e326e5
Compute Capability	6.1
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	96 KiB
Max. Registers per Block	65536
Max. Registers per Multiprocessor	65536
Max. Grid Dimensions	[2147483647, 65535, 65535]
Max. Block Dimensions	[1024, 1024, 64]
Max. Warps per Multiprocessor	64
Max. Blocks per Multiprocessor	32
Half Precision FLOP/s	36.7 GigaFLOP/s
Single Precision FLOP/s	4.698 TeraFLOP/s
Double Precision FLOP/s	146.8 GigaFLOP/s
Number of Multiprocessors	10
Multiprocessor Clock Rate	1.835 GHz
Concurrent Kernel	true
Max IPC	6
Threads per Warp	32
Global Memory Bandwidth	192.192 GB/s
Global Memory Size	5.934 GiB
Constant Memory Size	64 KiB
L2 Cache Size	1.5 MiB
Memcpy Engines	2
PCIe Generation	3
PCIe Link Rate	8 Gbit/s
PCIe Link Width	16



Exemple d'ILP : les résultats 84

kernel0(float*, float const *, float const *, int)

Duration	45.738 μs
Grid Size	[16,1,1]
Block Size	[512,1,1]
Registers/Thread	12
Shared Memory/Block	0 B
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

kernel1(float*, float const *, float const *, int)

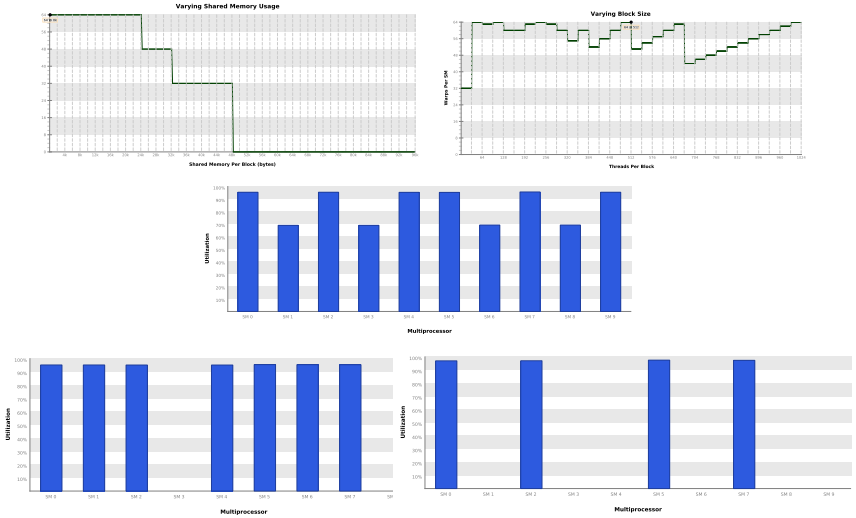
Duration	44.311 μs
Grid Size	[8,1,1]
Block Size	[512,1,1]
Registers/Thread	20
Shared Memory/Block	0 B
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

kernel2(float*, float const *, float const *, int)

Duration	85.825 μs
Grid Size	[4,1,1]
Block Size	[512,1,1]
Registers/Thread	30
Shared Memory/Block	0 B
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B



Exemple d'ILP : les résultats



ILP : est-ce automatique ?


```
__global__ void testKernel1(float* input, float* output, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < N) {
        float accum = 0.f;

        for (int i = 0; i < 8; i++) {
            accum = accum + input[n_loop*tid+i];
        }
        output[tid] = accum;
    }
}
```

```
mov.u32 %r3, %ntid.x;
mov.u32 %r4, %ctaid.x;
mov.u32 %r5, %tid.x;
mad.lo.s32 %r1, %r3, %r4, %r5;
setp.ge.s32     %p1, %r1, %r2;
@%p1 bra BB0_2;

cvta.to.global.u64 %rd3, %rd1;
cvta.to.global.u64 %rd4, %rd2;
shl.b32 %r6, %r1, 3;
mul.wide.s32 %rd5, %r6, 4;
add.s64 %rd6, %rd3, %rd5;
ld.global.f32 %f1, [%rd6];
add.f32 %f2, %f1, 0f00000000;
ld.global.f32 %f3, [%rd6+4];
add.f32 %f4, %f2, %f3;
ld.global.f32 %f5, [%rd6+8];
add.f32 %f6, %f4, %f5;
ld.global.f32 %f7, [%rd6+12];
add.f32 %f8, %f6, %f7;
ld.global.f32 %f9, [%rd6+16];
add.f32 %f10, %f8, %f9;
ld.global.f32 %f11, [%rd6+20];
add.f32 %f12, %f10, %f11;
ld.global.f32 %f13, [%rd6+24];
add.f32 %f14, %f12, %f13;
ld.global.f32 %f15, [%rd6+28];
add.f32 %f16, %f14, %f15;
mul.wide.s32 %rd7, %r1, 4;
add.s64 %rd8, %rd4, %rd7;
st.global.f32 [%rd8], %f16;

BB0_2:
ret;
}
```

On peut également utiliser la directive de compilateur :

```
#pragma unroll N
```

Cette directive du compilateur «déroule» la boucle en écrivant le code des différentes instances de la boucle en série ⟹ créer des instructions capables de s'exécuter en parallèle.

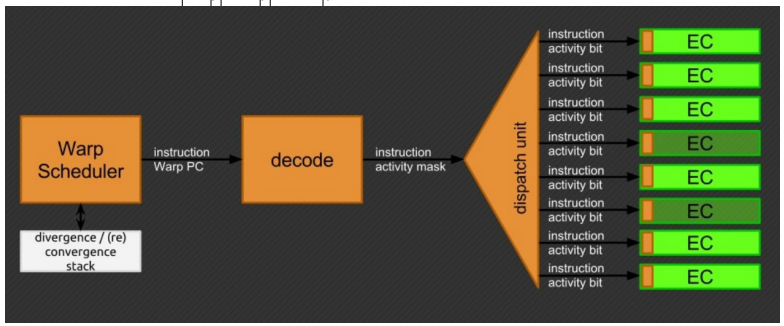
⟹ la boucle for n'existe plus... 8 Load... ⟹ le compilateur a déroulé le code !

ILP : est-ce automatique ?


```
sudo nvprof --metrics unique_warps_launched,sm_efficiency,achieved_occupancy,ipc,issued_ipc,is
sue_slot_utilization ./demo2_ilp
==124261== NVPROF is profiling process 124261, command: ./demo2_ilp
==124261== Profiling application: ./demo2_ilp
==124261== Profiling result:
==124261== Metric result:
    Metric Name                 Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1060 6GB (0)"
    Kernel: testKernel1(float*, float*, int)
     unique_warps_launched       Number of warps launched      262144      262144      262144
               sm_efficiency         Multiprocessor Activity      99.86%      99.86%      99.86%
          achieved_occupancy           Achieved Occupancy    0.755754    0.755754    0.755754
                         ipc                  Executed IPC    0.203224    0.203224    0.203224
                  issued_ipc                    Issued IPC    0.203273    0.203273    0.203273
     issue_slot_utilization       Issue Slot Utilization       4.81%       4.81%       4.81%
    Kernel: testKernel2(float*, float*, int)
     unique_warps_launched       Number of warps launched      262144      262144      262144
               sm_efficiency         Multiprocessor Activity      99.92%      99.92%      99.92%
          achieved_occupancy           Achieved Occupancy    0.750792    0.750792    0.750792
                         ipc                  Executed IPC    0.252322    0.252322    0.252322
                  issued_ipc                    Issued IPC    0.252369    0.252369    0.252369
     issue_slot_utilization       Issue Slot Utilization       6.31%       6.31%       6.31%
    Kernel: testKernel3(float*, float*, int, int)
     unique_warps_launched       Number of warps launched      262144      262144      262144
               sm_efficiency         Multiprocessor Activity      99.90%      99.90%      99.90%
          achieved_occupancy           Achieved Occupancy    0.760726    0.760726    0.760726
                         ipc                  Executed IPC    0.361742    0.361742    0.361742
                  issued_ipc                    Issued IPC    0.361795    0.361795    0.361795
     issue_slot_utilization       Issue Slot Utilization       8.36%       8.36%       8.36%
    Kernel: testKernel4(float*, float*, int, int)
     unique_warps_launched       Number of warps launched      262144      262144      262144
               sm_efficiency         Multiprocessor Activity      99.90%      99.90%      99.90%
          achieved_occupancy           Achieved Occupancy    0.796014    0.796014    0.796014
                         ipc                  Executed IPC    0.587706    0.587706    0.587706
                  issued_ipc                    Issued IPC    0.587767    0.587767    0.587767
     issue_slot_utilization       Issue Slot Utilization      13.08%      13.08%      13.08%
```


Aller plus loin en ILP ?

- ▷ «*Warp Scheduler*» : gère les warps, sélectionne celles prêtes à être exécutées ;
- ▷ «*Fetch/Decode Unit*» : est associé à un «*warp scheduler*» ;
- ▷ «*Execution Units*» : SC, SFU, LD/ST ;



Faire de l'ILP entre EU de **nature différentes** :

- ☐ SFU : «*Special Function Unit*» : fonctions transcendentales cos, sin, $expf$, *etc.*
- ☐ LD/ST : «*Load*» et «*Store*» : lit et écrit dans la mémoire ;
- ☐ SC : les instructions arithmétique et logiques.

Chaque EU peut travailler en même temps qu'une autre EU de ***nature différente****.*

Une exemple complet : calcul d'une fractale, «l'ensemble de Julia»

Dessiner un ensemble de Julia

On parcours une surface 2D :

- pour chaque point de coordonnées (x, y), on traduit ses coordonnées dans l'espace complexe :
  - Soit $DIM \times DIM$ la dimension de la surface en pixels ;
  - on centre cet espace complexe au centre de l'image, en décalant de $DIM/2$;
  - ce qui donne pour un point (x, y), les coordonnées
    $$((DIM/2 - x)/(DIM/2), (DIM/2 - y)/(DIM/2))$$
- on utilise également un «zoom» avec la variable *scale* pour zoomer ou dézoomer sur l'ensemble de Julia ;
- la constante $C = -0.8 + 1.5i$ donne une image intéressante.
- on considère le complexe $Z_0 = ((DIM/2 - x)/(DIM/2)) + ((DIM/2 - y)/(DIM/2))i$
- on calcule alors pour ce complexe Z_0, la limite de la suite $Z_{n+1} = Z_n^2 + C$:
  - si la limite de la suite diverge ou tend vers l'infini, alors le point n'appartient pas à l'ensemble et il sera coloré en noir ;
  - si la limite de la suite ne diverge pas, alors le point fait partie de l'ensemble et sera coloré en rouge.
- pour la limite, on utilise une simplification : si après 200 itérations de calcul de la suite, la valeur au carré de la suite est toujours inférieure à 1000 alors on considère qu'elle ne diverge pas.

Un exemple complet

Version CPU

```
int main( void ) {
   CPUBitmap bitmap( DIM, DIM ); // DIM = 1000
   unsigned char *ptr = bitmap.get_ptr();
   kernel( ptr );
   bitmap.display_and_exit();
}
```

```
int julia( int x, int y ) {
   const float scale = 1.5;
   float jx = scale * (float)(DIM/2 - x)/(DIM/2);
   float jy = scale * (float)(DIM/2 - y)/(DIM/2);

   cuComplex c(-0.8, 0.156);
   cuComplex a(jx, jy);
   int i = 0;
   for (i=0; i<200; i++) { // Calcul de la limite
      a = a * a + c;
   if (a.magnitude2() > 1000) // Infini ?
      return 0; // Pas dans l'ensemble
   }
   return 1; // Dans l'ensemble
}
```

```
void kernel( unsigned char *ptr ){
for (int y=0; y<DIM; y++) {
   for (int x=0; x<DIM; x++) {
      int offset = x + y * DIM; // 2D -> 1D

      int juliaValue = julia( x, y );
      ptr[offset*4 + 0] = 255 * juliaValue; // Red
      ptr[offset*4 + 1] = 0; // Green
      ptr[offset*4 + 2] = 0; // Blue
      ptr[offset*4 + 3] = 255; } // Alpha
   }
}
```

```
struct cuComplex { // Opérations sur les complexes
   float   r, i;

   cuComplex( float a, float b ) : r(a), i(b) {}

   float magnitude2(void) { return r * r + i * i;}

   cuComplex operator*(const cuComplex& a) {
      return cuComplex(r*a.r-i*a.i, i*a.r+r*a.i);
   }
   cuComplex operator+(const cuComplex& a) {
      return cuComplex(r+a.r, i+a.i);
   }
};
```


Un exemple complet

Version GPU

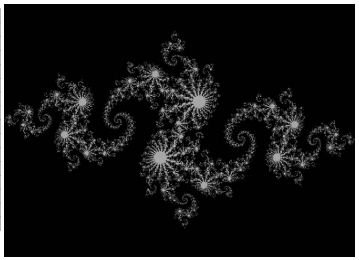
```
int main( void ) {
   CPUBitmap bitmap( DIM, DIM );
   unsigned char  *dev_bitmap;
   HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) ); // On alloue la zone de
l'image
   dim3   grid(DIM,DIM); // On utilise dim3 même si la dernière coordonnée vaudra seulement 1
   kernel<<<grid,1>>>( dev_bitmap ); // Un bloc par pixel et une thread par bloc
   HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap, bitmap.image_size(),
                             cudaMemcpyDeviceToHost ) );
   bitmap.display_and_exit();
   cudaFree( dev_bitmap );
}
```

```
__global__ void kernel( unsigned char *ptr ) {
   // map from threadIdx/BlockIdx to pixel position
   int x = blockIdx.x; // Calculé automatiquement
   int y = blockIdx.y; // Calculé automatiquement
   int offset = x + y * gridDim.x;
   // now calculate the value at that position
   int juliaValue = julia( x, y );
   ptr[offset*4 + 0] = 255 * juliaValue;
   ptr[offset*4 + 1] = 0;
   ptr[offset*4 + 2] = 0;
   ptr[offset*4 + 3] = 255;
}
```

```
__device__ int julia( int x, int y ) {
   const float scale = 1.5;
   float jx = scale * (float)(DIM/2 - x)/(DIM/2);
   float jy = scale * (float)(DIM/2 - y)/(DIM/2);
   cuComplex c(-0.8, 0.156);
   cuComplex a(jx, jy);
   int i = 0;
   for (i=0; i<200; i++) {
      a = a * a + c;
      if (a.magnitude2() > 1000)
         return 0;
   }
   return 1;
}
```


Un exemple complet


```
struct cuComplex {
   float   r, i;
   cuComplex( float a, float b ) : r(a), i(b) {}
   __device__ float magnitude2( void )
   {
      return r * r + i * i;
   }
   __device__ cuComplex operator*(const cuComplex& a)
   {
      return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
   }
   __device__ cuComplex operator+(const cuComplex& a)
   {
       return cuComplex(r+a.r, i+a.i);
   }
};
```



Les fonctions préfixées par `__device__` *:*

- *tournent sur le GPU ;*
- *ne peuvent être appelées que depuis une fonction préfixée par* `__device__` *ou* `__global__`*.*

Le code complet est accessible sur `http://developer.nvidia.com/cuda-cc-sdk-code-samples`

Extraction du parallèlisme de données et la définition de «grille»

Parcours imbriqué d'un tableau en version CPU

```
void add_matrix ( float* a, float* b, float* c, int N )
{
   int index;
   for ( int i = 0; i < N; ++i )
      for ( int j = 0; j < N; ++j ) {
         index = i + j*N;
         c[index] = a[index] + b[index];
         }
}
int main() {
   add_matrix( a, b, c, N );
}
```

Ici, la matrice est définie suivant un tableau à une seule dimension.

Parcours imbriqué d'un tableau en version GPU

```
__global__ add_matrix ( float* a, float* b, float* c,
int N )
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   int index = i + j*N;
   if ( i < N && j < N )
      c[index] = a[index] + b[index];
}
int main() {
      dim3 dimBlock( blocksize, blocksize );
      dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
      add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

On «déplie» les deux boucles imbriquées en une grille, où chaque élément de la grille définie une combinaison de valeurs pour i et j.


Exemple de programme type


```
   const int N = 1024;
   const int blocksize = 16;
   __global__ void add_matrix( float* a, float *b, float *c, int N )
   {   int i = blockIdx.x * blockDim.x + threadIdx.x;
       int j = blockIdx.y * blockDim.y + threadIdx.y;
       int index = i + j*N;
       if ( (i < N) && (j < N) ) c[index] = a[index] + b[index];
   }

   int main() {
       float *a = new float[N*N]; float *b = new float[N*N]; float *c = new float[N*N];
       for ( int i = 0; i < N*N; ++i ) { a[i] = 1.0f; b[i] = 3.5f; }
       float *ad, *bd, *cd;
       const int size = N*N*sizeof(float);
       cudaMalloc( (void**)&ad, size );
       cudaMalloc( (void**)&bd, size );
       cudaMalloc( (void**)&cd, size );

       cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
       cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

       dim3 dimBlock( blocksize, blocksize );
       dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
       add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

       cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

       cudaFree( ad );   cudaFree( bd ); cudaFree( cd );
       delete[] a; delete[] b; delete[] c;

       return EXIT_SUCCESS;
   }
```


Optimisation

Les contraintes d'architecture

- ▷ un GPU possède N multiprocesseur, *MP*, en général 2 ou 4 ;
- ▷ chaque MP possède M processeur scalaire, *SP* ;
- ▷ chaque MP traite des blocks par lot :
  - ◇ un bloc est traité par un MP uniquement ;
- ▷ chaque bloc est découpé en groupe de threads SIMD appelé warp :
  - ◇ un warp est exécuté physiquement en parallèle ;
- ▷ le scheduler « switche» entre les warps ;
- ▷ un warp contient des threads d'identifiant consécutifs, «thread ID» ;
- ▷ la taille d'un warp est actuellement de 32 ;

Les optimisations possibles

- ▷ $\frac{nombre\ de\ blocks}{nombre\ de\ MP} > 1 \Longrightarrow$ tous les MPs ont toujours quelque chose à faire ;
- ▷ $\frac{nombre\ de\ blocks}{nombre\ de\ MP} > 2 \Longrightarrow$ plusieurs blocs peuvent être exécutés en paralléle sur un MP ;
- ▷ les ressources par block $\leq$ au total des ressources disponibles :
  - ◇ mémoire partagée et registres ;
  - ◇ plusieurs blocs peuvent être exécutés en paralléle sur un MP ;
  - ◇ éviter les **branchements divergents** dans les conditions à l'intérieur d'un bloc :
    - ⋆ les différents chemins d'exécution doivent être **sérialisés** !


Les différents accès à la grille

Une grille 1D avec des blocs en 2D

```
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x;
```

Une grille 1D avec des blocs en 3D

```
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y * blockDim.z
      + threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```

Une grille 2D avec des blocs 1D

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.x + threadIdx.x;
```

Un grille 2D avec des blocs 2D

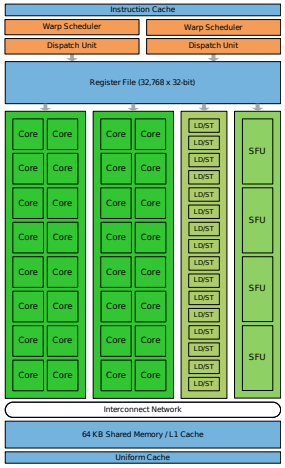
```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex =UniqueBlockIndex * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```

Une grille 2D avec des blocs 3D

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.z * blockDim.y * blockDim.x
      + threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```


Architecture matérielle Fermi

- «*GigaThread global scheduler*» :
  - distribue les blocs de threads aux SMs ;
  - gère le changement de contexte entre les threads pendant l'exécution ;
- DRAM : jusqu'à 6GB de mémoire grâce à un adressage sur 64bits, débit de 192GB/s ;
- Fréquence processeur : 1,5GHz, Peak performance : 1,5TFlops, Fréquence mémoire : 2GHz ;



- un «*core*» CUDA :
  - une ALU, «*Integer Arithmetic Logic Unit*» : supporte des opérations sur 32bits, peut également travailler sur 64bits ;
  - FPU, «*Floating Point Unit*» : réalise des «*Fused Multiply-Add*», $A * B + C$, jusqu'à 16 opérations en double précision par SM et par cycle ;
- LoaD/STore : calcule les adresses sources et destinations pour 16 threads par cycle d'horloge depuis/vers la DRAM ou le cache ;
- SM, «*Streaming Multi-processors*» contient :
  - 32 cœurs «*single precision*» ;
  - 4 unités SFUs, «*Special Function Units*» :
    - fonctions transcendentales sinus/cosinus, inverse et racine carrée ;
    - une instruction par thread et par cycle d'horloge : un «*warp*» exécute l'opération en 8 cycles ;
  - un bloc de 64KB de mémoire rapide : mémoire cache L1
  - 32k of 32 bits registers :
    - chaque thread possède ses propres registres entre 21 et 63 (l'augmentation du nombre de threads diminue le nombre de registres par threads) ;
    - la vitesse d'échange de ces registres est de 8GB/s ;
  - Warp scheduler : ordonnancement des warps de 32 threads ;

Architecture Fermi

512 «*cuda cores*» orgnisés en 16 SM de 32 cores chacun.

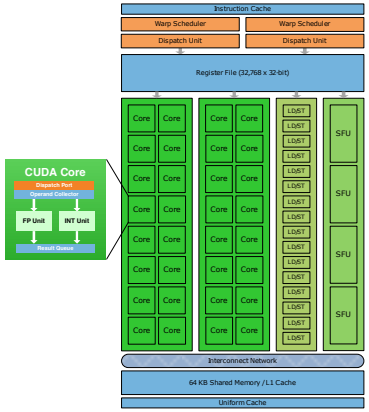
32 «*cuda cores*» par SM, «*Streaming Multiprocessor*».

Un «cuda core» correspond à un circuit capable de réaliser un calcul en simple précision, FPU, ou sur un entier, ALU (opérations binaire comprises), en 1 cycle d'horloge.

Une unité «*Special Function Unit*» exécute les opérations transcendentales comme le sin, cos, $sqrt$, $arccos$, *etc.*
Il en existe 4 : seuls 4 opérations de ce type peuvent avoir lieu simultanément.

Le GPU dispose de 6 partition mémoire de 64 bits, soient 16 unités sur 32bits.

16 unités «*Load/Store*» permettent d'accéder en écriture ou en lecture à des données pour 16 threads par cycle d'horloge.



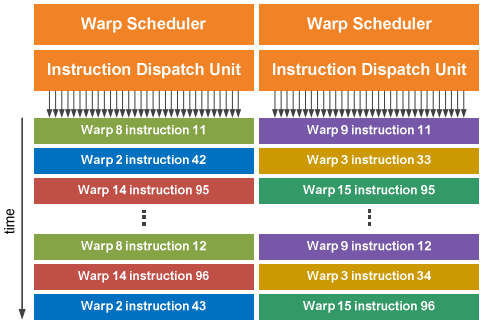
Fermi Streaming Multiprocessor (SM)

Architecture Fermi

Deux «*Warp Schedulers*» permettent de programmer l'exécution de deux Warps indépendants simultanément :

- seuls les warps capables de s'exécuter peuvent être exécutés simultanément ;
- un scheduler programme l'exécution d'une instruction d'un Warp vers un groupe de 16 cores/16 LD/ST or 4 SFUs.
- la plupart des instructions peuvent être émises par deux : deux instructions entières, deux instructions flottantes, un mix d'entier de flottant, lecture, écriture et opération transcendentale.

  Mais les instructions en **double précision ne peuvent pas être émises par deux**.



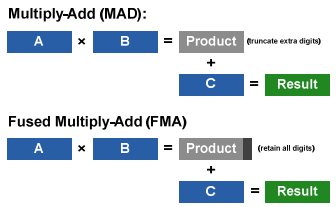
Architecture Fermi

Des opérations spéciales

MAD, «*multiply-add*» : une opération de multiplication et d'addition combinée, mais avec perte des bits dépassant la capacité.

FMA, «*fused multiply-add*» : réalise les mêmes opérations mais sans perte, en simple ou double précision.

L'utilisation de calcul en double précision permet 16 calcul de type MAD par SM et par cycle d'horloge.



La mémoire

- cache L1/mémoire partagée, utilisable au choix :
  - en mémoire partagée entre les threads pour leur permettre de coopérer en échangeant des données ;
  - pour conserver les valeur des registres nécessaires à une thread qui ne peuvent être affectées à des registres disponibles dans le SM. *On parle de* «spilled» *registers*, c-à-d du débordement de registres ;
  - débit : 1600Go/s ;
  - organisable en 16Ko de cache L1 et 48Ko de mémoire partagée ou 48Ko de cache L1 ;
  - organisable en 48Ko de cache L1 et 16Ko de mémoire partagée ou 48Ko de cache L1 ;
- cache L2 : 768Ko partagé entre les 16 SMs qui sert aux :
  - accès vers/depuis la mémoire globale ;
  - copies depuis/vers le Host ;
  - accès aux textures.