GPU Computing

E. Carlinet, J. Chazalon {firstname.lastname@lrde.epita.fr}

October'19

EPITA Research & Development Laboratory (LRDE)

Agenda

GPU and architectures

Scientific Computing

GPU vs CPU for parallelism

GPU vs CPU architectures

GPU memory model

Agenda

Agenda

1. *GPU and architectures* (2h)
2. *Programming GPUs with CUDA* (2h)
3. *TP CUDA* (4h)
4. *More about memory, unified virtual space & communication* (2h)
5. *Re-Thinking IP Algorithms for GPU* (2h)

GPU and architectures

Why using GPU ?

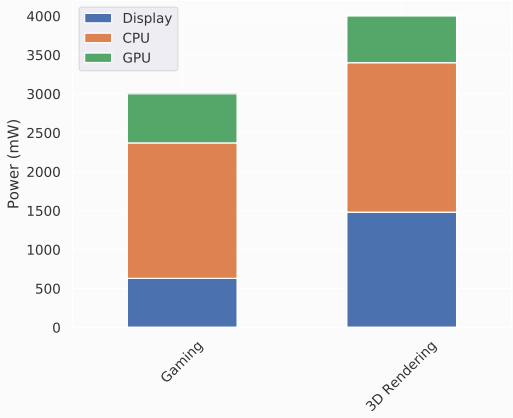
We want to have things *done* **quickly**.

Mobile dev. | Big data | Real time computing

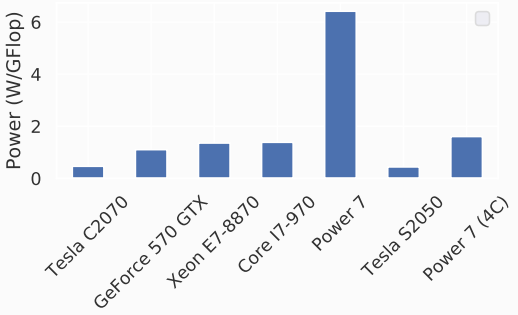
- Mobile development : limited battery
- Big data analysis : huge data volume
- Real time system : has to provide a response in a bounded time

Power Consumption on Smartphones

CPU is a major source of power consumption in smartphones (even with graphical-oriented app)



Power Consumption of Some Processors



Fabricant	Type	Modèle	Gflops	Prix	Watt
Nvidia	1x GPU (448 cœurs)	Tesla C2070	515	2500 $	238 W
Nvidia	1x GPU (448 cœurs)	GeForce 570 GTX	198	350 $	218 W
Intel	1x CPU (10 cœurs)	Xeon E7-8870	96	4616 $	130 W
Intel	1x CPU (6 cœurs)	Core I7-970	94	583 $	130 W
IBM	CPU (8 cœurs)	Power 7	265	34 152 $	1700 W
Nvidia	4xGPU (1792 cœurs)	Tesla S2050	2060	12 000 $	900 W
IBM	4xCPU (32 cœurs)	Power 7	1060	101 952 $	1700 W

Scientific Computing

A bit of history - The first GPU

- Back in 70's GPU were for Image Synthesis
- First GPU : Ikonas RDS-3000

- N. England & M. Whitton foundend Ikonas Graphics Systems
- Tim Van Hook wrote microcode for ray tracing (SIGRAPH'86)
- "All computation is taking place in the Adage 3000 Display" (1)

The GRAPHICS SYSTEM for the 80's

RDS-3000 Graphics Processor and Raster Display System

If your graphics and imaging applications are demanding, the IKONAS RDS-3000 series is the system that can meet your need. The RDS-3000 offers:

POWER
- High Speed Architecture designed for computer graphics and image processing
- Fast 32 bit processor for graphics data generation
- Hardware Matrix Multiplier for 3-D transformations vector products, and filtering operations
- Real Time Video Processing Module for image processing applications
- Video Input Module for real time "frame grabbing"

FLEXIBILITY
- Software selectable 512² or 1024² display format
- Variable frame and line rates: 200-2000 lines/frame
- Pan and scroll in pixel increments, zoom in integer ratios
- Full Window and Viewport Control

PROGRAMMABILITY
- Graphics Processor is completely user microprogrammable and executes the highly parallel code needed for real time and near real time applications
- IDL, the IKONAS DISPLAY LANGUAGE, is a high level command language which makes the IKONAS package of standard graphics routines easy to use.

EXPANDABILITY
- RDS-3000 components are modular allowing easy expansion of systems.
- Small frame buffer systems can be upgraded at a later time by adding processing modules and image memories up to 1024² x 32.

Photo credits: TerrainModel/R. Wilson, B. Marshall, Computer Graphics Research Group, Ohio State University. Mountain/L.oren Carpenter, Boeing Computer Services

IKONAS

IKONAS GRAPHICS SYSTEMS, INC.
403 Glenwood Ave
Raleigh, NC 27603 919/833-5401

IKONAS strives to meet the graphics requirements of advanced, high technology research groups with our standard products or custom design. Call IKONAS for high performance raster graphics equipment.

Reader Service Number 31

(1) http ://www.virhistory.com/ikonas/ikonas.html

A bit of history - The first GPGPU (’99-’01)

First programmable GPU :

- Vertex Shaders – programmable vertex transforms, 32-bit float
- Data-dependent, configurable texturing + register combiners

A bit of history - The first GPGPU (’99-’01)

First programmable GPU :

- Vertex Shaders – programmable vertex transforms, 32-bit float
- Data-dependent, configurable texturing + register combiners

Enabled early GPGPU results :

- Hoff (1999) – Voronoi diagrams on NVIDIA TNT2
- Larsen &McAllister (2001) : first GPU matrix multiplication (8-bit)
- Rumpf & Strzodka (2001) : first GPU PDEs (diffusion, image segmentation)
- NVIDIA SDK Game of Life, Shallow Water (Greg James, 2001)

GPGPU for physics simulation on Gefore 3

Approximate simulation of natural phenomena :

- Boiling liquid,
- fluid convection,
- chemical reaction-diffusion

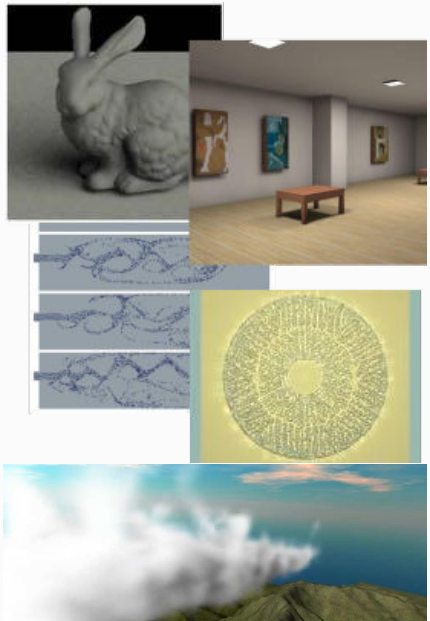
"Physically-Based Visual Simulation on Graphics Hardware". Harris, Coombe, Scheuermann, and Lastra. Graphics Hardware 2002

At that time, limited by computing precision (mostly integers).

A bit of history. GEFORCE FX (2003) : floating point

True programmability enabled broader simulation research :

- Ray Tracing (Purcell, 2002), Photon Maps (Purcell, 2003)
- Radiosity (Carr et al., 2003 & Coombe et al., 2004)
- PDE solvers
  - Red-black Gauss-Seidel (Harris et al., 2003)
  - Conjugate gradient (Bolz et al. 2003, Krueger et al. 2003)
  - Multigrid (Goodnight et al. 2003)
- Physically-based simulation
  - Fluid and cloud simulation [(Krueger et al. 2003, Harris et al. 2003)}
  - Cloth simulation (Green, 2003)
  - Ice crystal formation (Kim and Lin, 2003)
  - Thermodynamics (latent heat, diffusion)
  - Water condensation / evaporation
- FFT (Moreland and Angel, 2003)
- High-level language : Brook for GPUs (Buck et al. 2004)



A bit of history - GPGPU becomes a trend (2006)

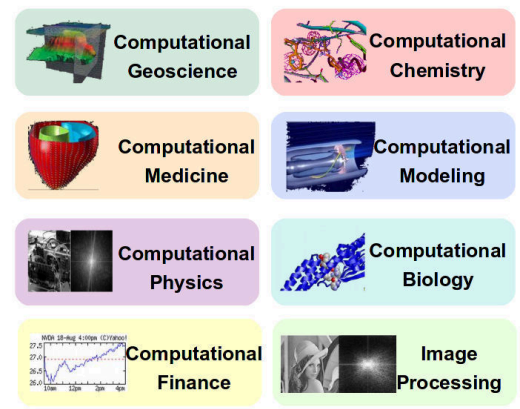
Two factors for the massive surge in GPGPU dev :

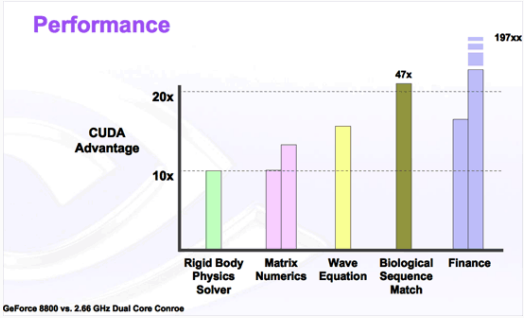
- **Architecture Nvidia G80** GPU arch. and software platform designed for computing
  - Dedicated computing mode – threads rather than pixels/vertices
  - General, byte-addressable memory architecture
- **Software support**. C and C++ languages and compilers for GPUs (spoiler... it's **CUDA**)

A bit of history - GPGPU becomes a trend (2006) ...

Nvidia's G80 commercial :
A programmer will be able to treat G80 like a hugely parallel data processing engine. Applications that require massively parallel compute power will see huge speed up when running on G80 as compared to the CPU. This includes financial analysis, matrix manipulation, physics processing, and all manner of scientific computations.

... everywhere


Computational Geoscience
Computational Chemistry
Computational Medicine
Computational Modeling
Computational Physics
Computational Biology
Computational Finance
Image Processing


Performance
197xx
47x
20x
CUDA Advantage
10x
Rigid Body Physics Solver
Matrix Numerics
Wave Equation
Biological Sequence Match
Finance
GeForce 8800 vs. 2.66 GHz Dual Core Conroe

A bit of history - 2010's (1/3)

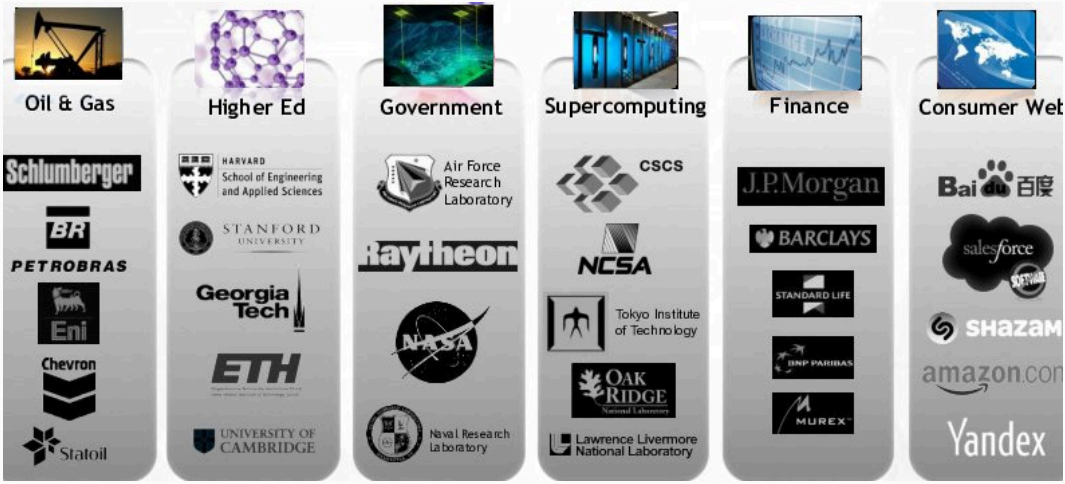
GPGPU provides the computing power...

Accelerating Discoveries

> *Using a supercomputer powered by 3,000 tesla processors, university of illinois scientists performed the first all-atom simulation of the hi virus and discovered the chemical structure of it capsid — "the perfect target for fighting the infection."*

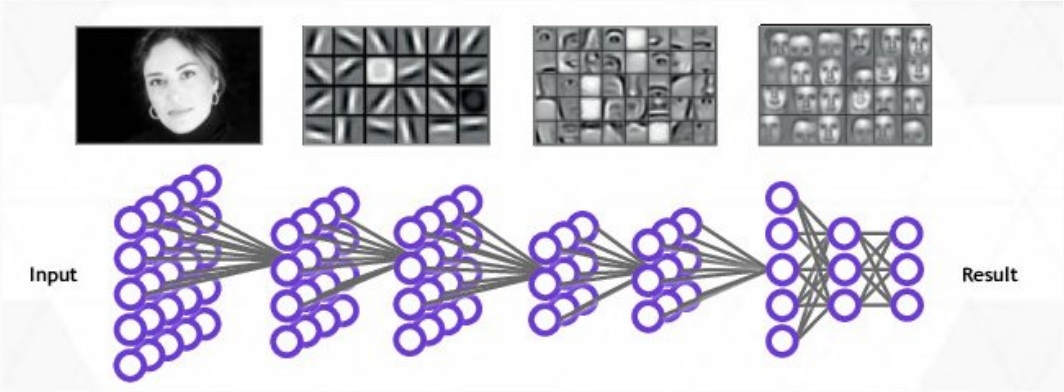
`Without gpus, the supercomputer would need to be 5x larger for similar performance.`

"High Performance Computing" (HPC) gives birth to Enterprise Datacenters



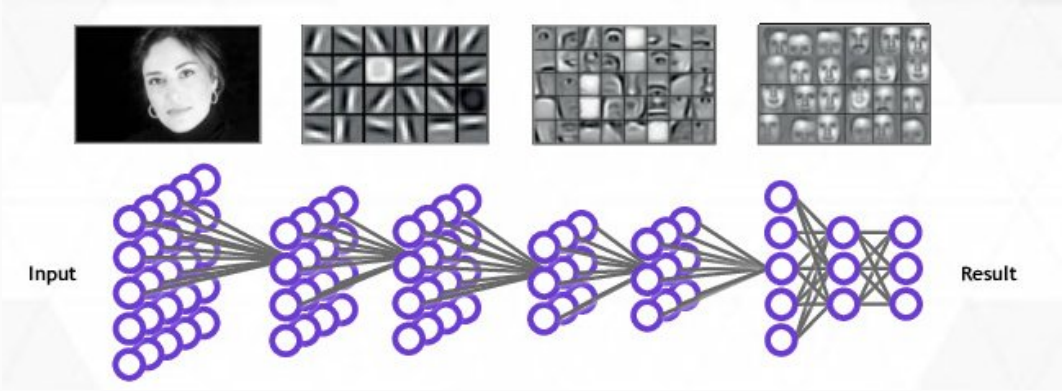
A bit of history - 2010's (3/3)

And data center gave birth to Deep-Learning (... *)



A bit of history - 2010's (3/3)

And data center gave birth to Deep-Learning (... *)



(*...) and made image processing experts useless :'(

Embedded systems - The *real*-time constraints

Clement Farabet, Camille Couprie, Laurent Najman and Yann LeCun : Learning Hierarchical Features for Scene Labeling, IEEE Transactions on Pattern Analysis and Machine Intelligence, August, 2013

Need both the two worlds :

- Need ultra-performance computing
- With limited resources

GPU vs CPU for parallelism

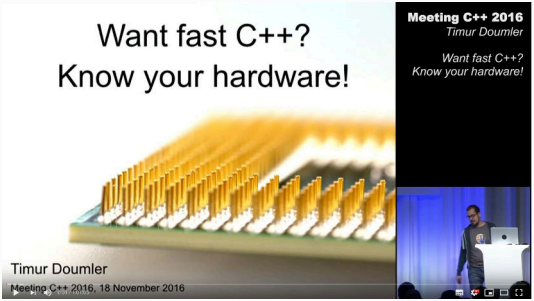
How to get things *done* quicker

1. Do less work
2. Do *some* work better (i.e. the one being the more time-consuming)
3. Do *some* work at the same time
4. Distribute work between different workers

How to get things *done* quicker

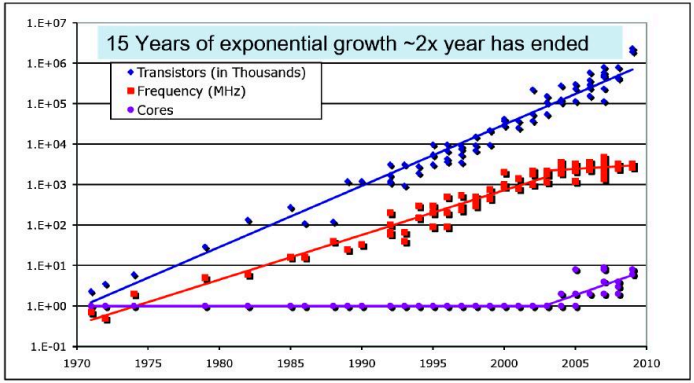
1. Do less work
2. Do *some* work better (i.e. the one being the more time-consuming)
3. Do *some* work at the same time
4. Distribute work between different workers

- (1) Choose the most adapted **algorithms**, and avoid re-computing things
- (2) Choose the most adapted **data structures**
- (3,4) Parallelism



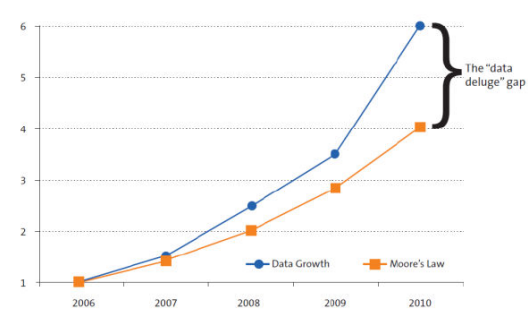
Why parallelism ?

- Moore's law : processors are **not** getting twice as powerful every 2 years anymore



- So the processor is getting smarter :
  - Out-of-order execution / dynamic register renaming
  - Speculative execution with branch prediction
- And the processor is getting super-scalar :
  - ISA gets vectorized instructions

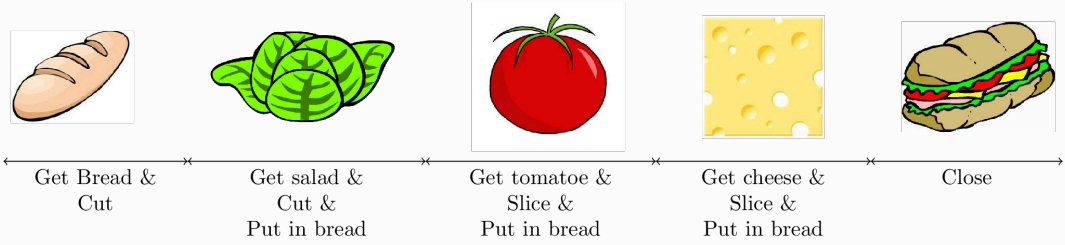
Toward data-oriented programming



- while the CPU clock rate got bounded…
- … the quantity data to process has shot up !

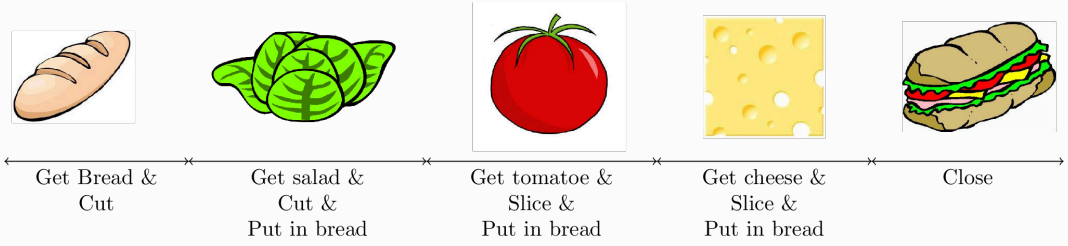
We need another way of thinking “speed”

The burger factory assembly line



How to make several sandwiches as fast as possible ?

The burger factory assembly line



How to make several sandwiches as fast as possible ?

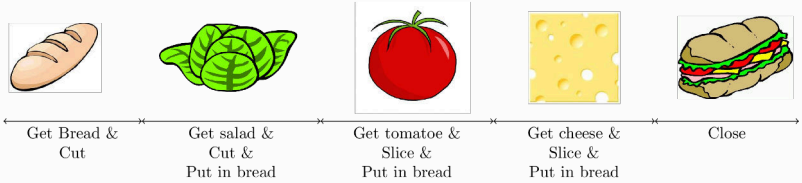
- Optimize for **latency** : time to get 1 sandwich done.
- Optimize for **throughput** : number of sandwiches done during a given *duration*

Flynn's Taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

- SISD : no parallelism
- SIMD : same instruction on data group (vector)
- MISD : rare, mostly used for fault tolerant code
- MIMD : usual parallel mode

Optimize for latency (MIMD)



4 **super-workers** (4 CPU cores) collaborate to make 1 sandwich.

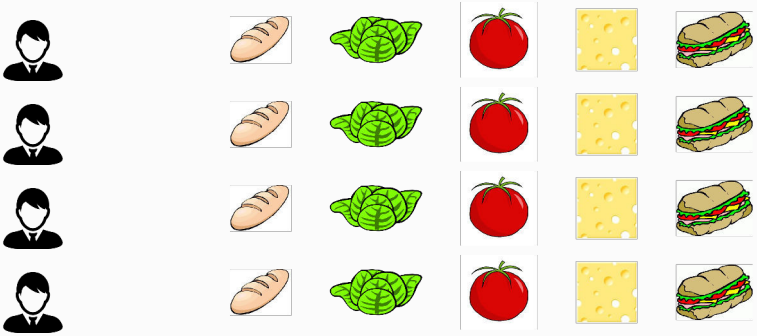
- A gets the bread and cuts
- B slices the salad
- C slices the the tomatoes
- D slices the cheeses and wait the others to put it in the bread

Time to make 1 sandwich : $\frac{s}{4}$ (400% speed-up)

*This is optimized for **latency** (CPU are good for that).*

Optimize for throughput (SIMD)

An army of small-workers making sandwiches with **all** same operations



Time to make 4 sandwiches : $4 \times s$ (400% speed-up)

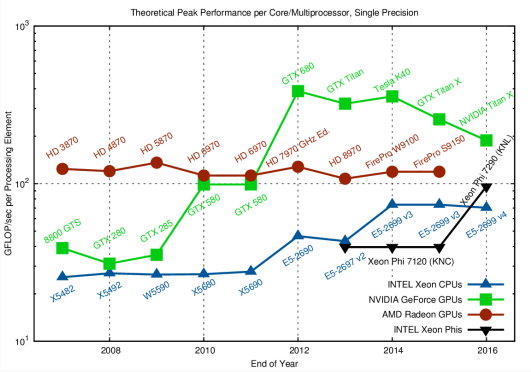
More cores is trendy

Data-oriented design have changed the way we make processors (even CPUs) :

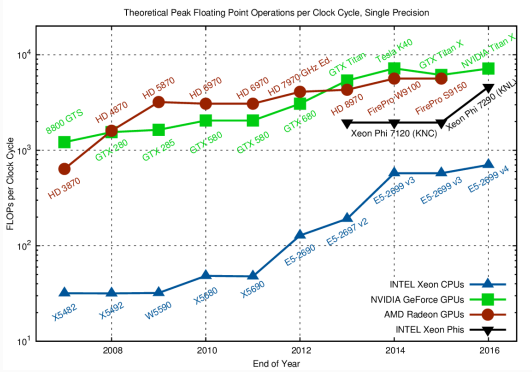
- Lower clock-rate
- Larger vector-size, more vector-oriented ISA
- More cores (processing units)

	64bits Intel Xeon	Xeon 5100 series	Xeon 5500 series	Xeon 5600 series	Xeon E5 2600 series	Xeon Phi 7120P
Freq	3.6 Ghz	3.0 Ghz	3.2 Ghz	3.3 Ghz	2.7 Ghz	1.24 Ghz
Cores	1	2	4	6	12	61
Threads	2	2	8	12	24	244
SIMD Width	128 bits (2 clocks)	128 bits (1 clock)	128 bits (1 clock)	128 bits (1 clock)	256 bits (1 clock)	512 bits (1 clock)

More cores is trendy



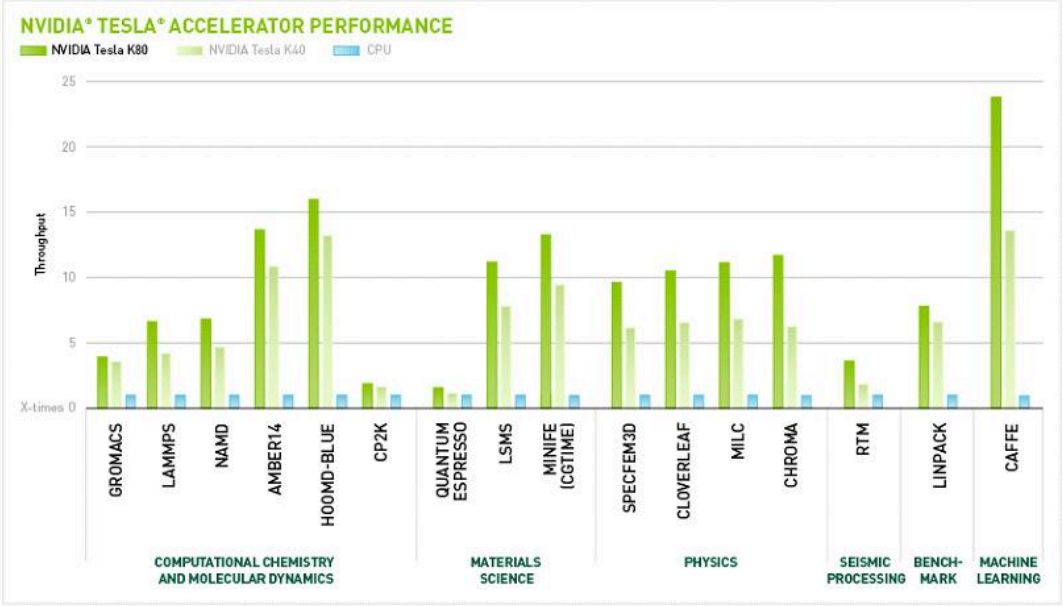
Peak performance / core is getting lower



Global peak performance is getting higher (with more cores !)

CPU vs GPU performance

And you see it with HPC apps :



But don't forget, you may need to optimize both **latency** and **throughput**.

What is the bounds speedup attainable on a parallel machine with a program which is parallelizable at **P** % (i.e. must run sequentially for **(1 - P)**).

But don't forget, you may need to optimize both **latency** and **throughput**.

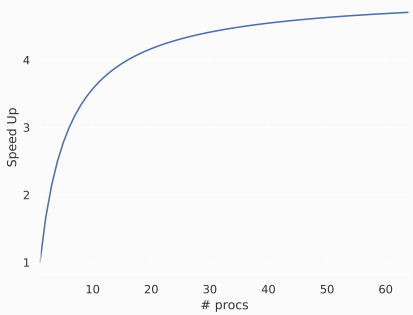
What is the bounds speedup attainable on a parallel machine with a program which is parallelizable at **P** % (i.e. must run sequentially for **(1 - P)**).

If you have *N* processors, the speed-up is :

$$S = \frac{t\_\text{old}}{t\_\text{new}} = \frac{1}{(1-P) + P/N}$$

- Time to run the sequential part
- Time to run the parallel part

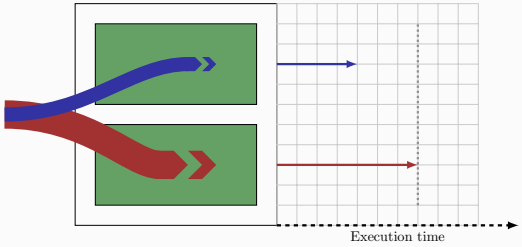
- P = 80%, max speed-up = 5



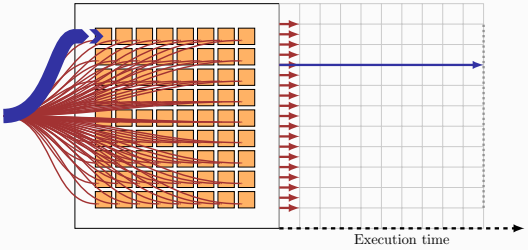
$$S = \frac{t\_\text{old}}{t\_\text{new}} = \frac{1}{(1-P) + P/N}$$

- Time to run the sequential part
- Time to run the parallel part

Latency-optimized (multi-core CPU)
☞ Poor perfs on parallel portions



Throughput-optimized (GPU)
☞ Poor perfs on sequential portions



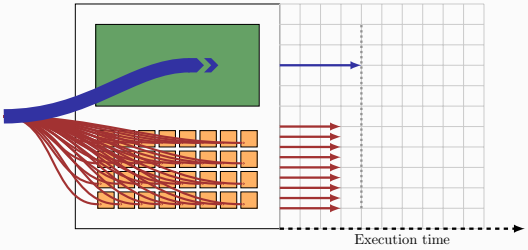
$$S = \frac{t\_\text{old}}{t\_\text{new}} = \frac{1}{(1-P) + P/N}$$

- Time to run the sequential part
- Time to run the parallel part

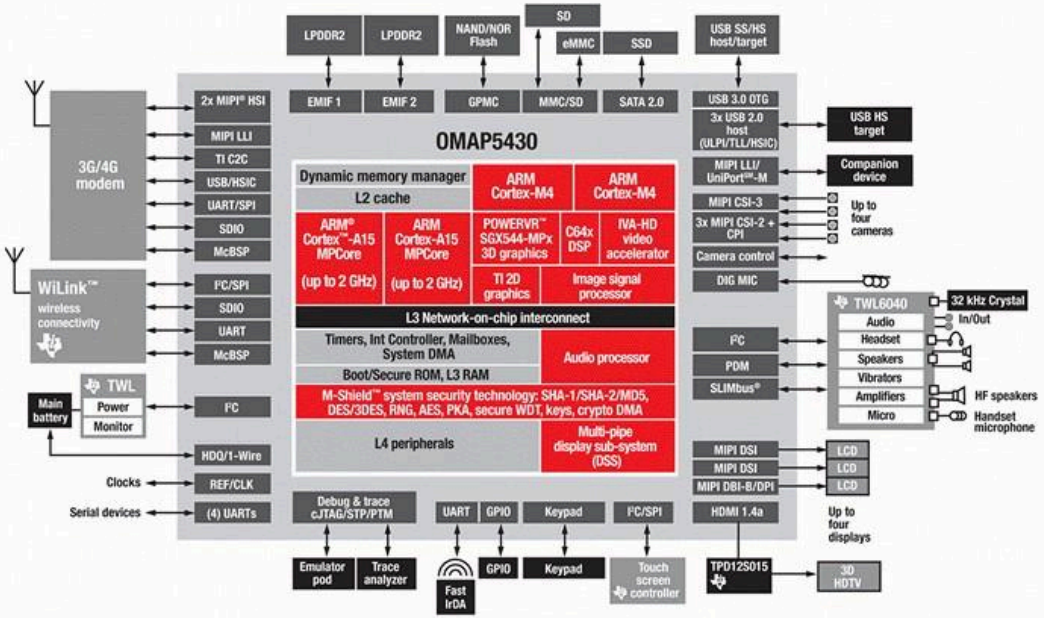
Heterogeneous (CPU+GPU)

👍 Use the right tool for the right job

👍 Allows aggressive optimization for latency or for throughput



Toward Heterogeneous Architectures



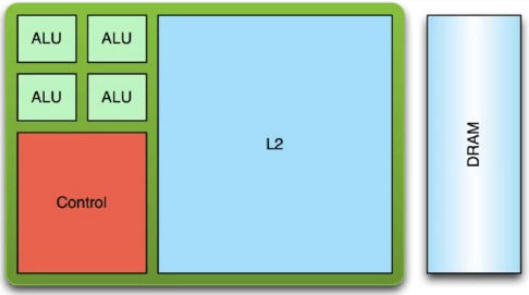
GPU vs CPU architectures

GPU vs CPU architectures

How to explain that :

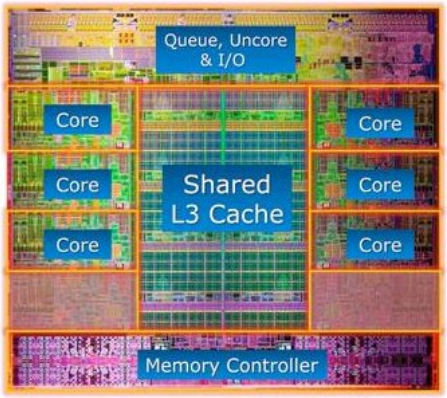
- CPU are low-latency
- GPU are high-throughput

It’s all about data… the CPU :

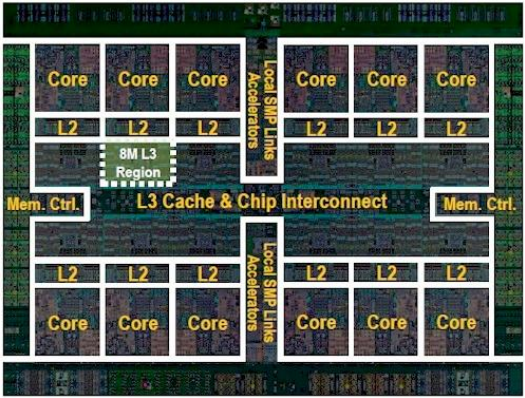


- Optimized for low-latency access (many memory caches)
- Control logic for out-of-order and speculative execution

Intel i7

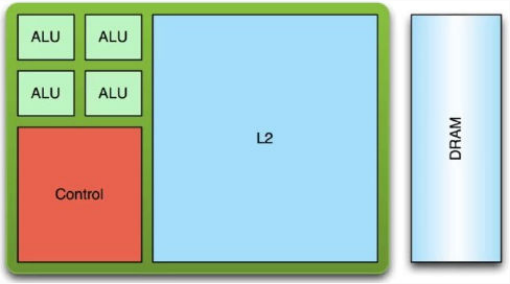


IBM Power 8 (2014)



It’s all about data… the GPU :

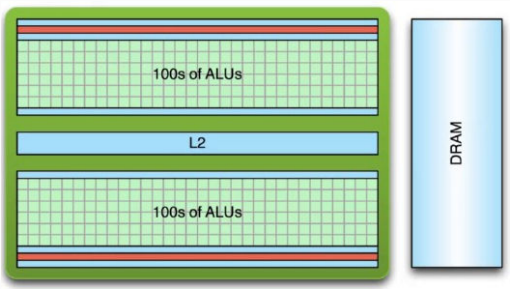
CPU



- Low-latency access
- Many control logic

But how… ?

GPU



- Throughput computation (ALUs)
- Tolerant to memory latency

It’s all about data… Latency hiding

So… you want to hide the latency of getting data from global memory… how ?

It’s all about data... Latency hiding

So... you want to hide the latency of getting data from global memory... how ?

Do other operations when waiting for data :

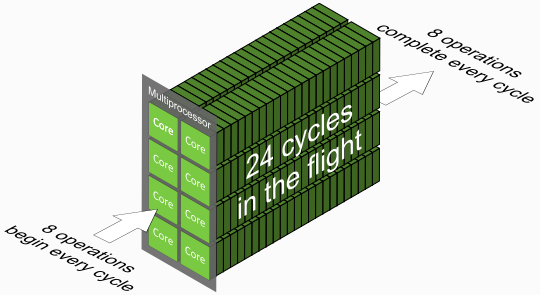
- will run faster
- but not faster than the peak
- what is the peak by the way ?

It's all about data... Little's law

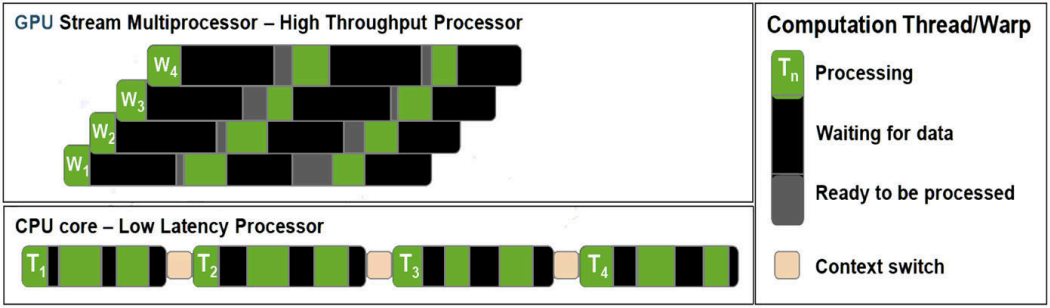
- Customer arrival rate : Throughput
- Customer time spent : Latency
- Avergage customer count : Concurrency (Data in the pipe)" = throughput * Latency

Concurrency is the number of items processed at the same time.

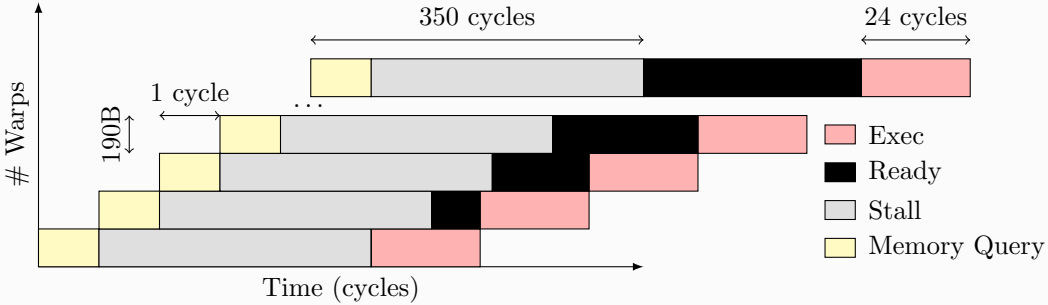
	Latency	Peak Throughput	Needed Concurrency
GPU-arithmetic	24 cycles	8 IPC	192 inst
GPU-memory	350 ns	190 GB/s	65K



Hiding latency With thread parallelism & pipelining



- In CPU :
  - low-latency memory to get data ready
  - each thread context switch has a cost
- In GPU :
  - memory latency hidden by pipelining
  - context switch is free

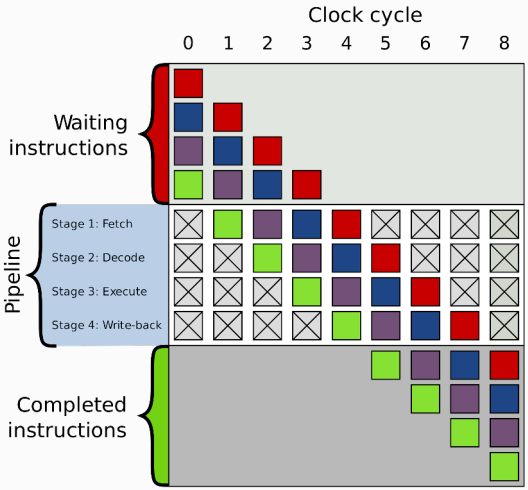


- Memory throughput : 190 GB/s
- Memory latency : 350 ns
- Data in flight = 65KB

At 1 Ghz : * 190 Byte/cycle * 350 cycles to wait

Hiding latency

With thread parallelism & pipelining
Note that pipeling exists on CPUs (cycle de Von Neumann) :

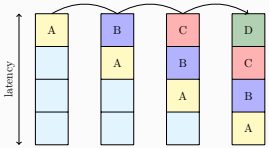


- IF instruction fetch
- ID instruction decode
- OPF Operand fetch
- EX execute
- WB result write back

Pipeline at **Instruction Level** vs pipeline at **Thread (Warp) Level**

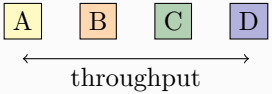
Vertical parallelism for latency hiding

Pipelining keeps units busy when waiting for dependencies, memory



Horizontal parallelism for throughput

More units working in parallel



More about forms of parallelism (the how!)

Instruction-Level Parallelism (ILP)

Between independent instructions.

```
1. add r3 ← r1, r2
2. mul r0 ← r0, r1
3. sub r1 ← r3, r0
```

'1 and '2 run concurrently

More about forms of parallelism (the how!)

Instruction-Level Parallelism (ILP)

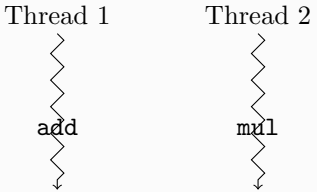
Between independent instructions.

```
1. add r3 ← r1, r2
2. mul r0 ← r0, r1
3. sub r1 ← r3, r0
```

Thread-Level Parallelism (TLP)

Between independent execution contexts : threads

'1 and '2 run concurrently



More about forms of parallelism (the how!)

Instruction-Level Parallelism (ILP)

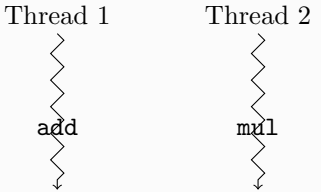
Between independent instructions.

```
1. add r3 ← r1, r2
2. mul r0 ← r0, r1
3. sub r1 ← r3, r0
```

Thread-Level Parallelism (TLP)

Between independent execution contexts : threads

'1 and '2 run concurrently



Data-Level Parallelism (DLP)

Between elements of a vector : same operation on several elements

```
vadd r ← a, b
  a₁ a₂ a₃
    +
  b₁ b₂ b₃
  ————————
  r₁ r₂ r₃
```

Extracting parallelism

	Horizontal	Vertical
ILP	Superscalar	Pipeline
TLP	Multi-cores / SMT	Interleaved / Switch-on-event multi-threading
DLP	SIMD / SIMT	

CPU (Intel Haswell)

	Hor.	Vert.
ILP	8	✓
TLP	4	2
DLP	8	

- 8 ALUs for executing non-dependent instructions
- 4 cores. Physical cores
- 4*2 logical hyper-cores
- Lane of 8x32bits SIMD registers supporting AVX 256

General-purpose multi-cores : balance ILP, TLP and DLP

CPU (Intel Haswell)

	Hor.	Vert.
ILP	8	✓
TLP	4	2
DLP	8	

- 8 ALUs for executing non-dependent instructions
- 4 cores. Physical cores
- 4*2 logical hyper-cores
- Lane of 8x32bits SIMD registers supporting AVX 256

General-purpose multi-cores : balance ILP, TLP and DLP

CPU (NVidia Kepler)

	Hor.	Vert.
ILP	2	
TLP	16x4	64
DLP	32	

- Dual instruction issue for executing non-dependent instructions
- 16 Multiprocessors (physical cores) Can execute 4 simultaneous warps
- Multithreading (64 warps / SM)
- 128 (4 x 32) CUDA cores. SIMT of width 32

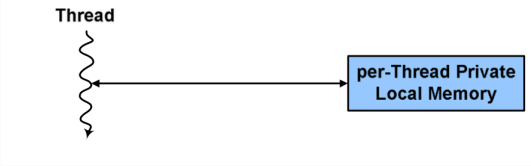
GPU : focus on DLP, TLP horizontal and vertical.

- All processors use hardware to turn parallelism into performance
- GPUs focus on Thread-level and Data-level parallelism # From programming model to hardware parallelism

Programming Model (CUDA terminology)

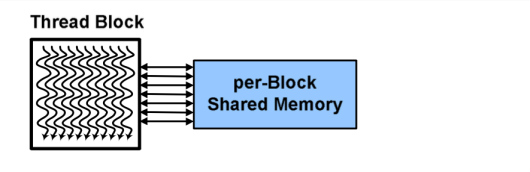
Thread

- Instance of one *kernel* (list of instructions)
- If *active*, it has a *Program Counter*, *registers*, *private memory*, IO
- Thread ID = ID a *block*



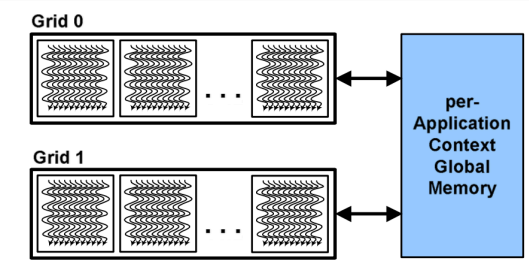
Block
A set of *threads* that cooperate :

- Synchronisation
- Shared memory
- Block ID = ID in a grid



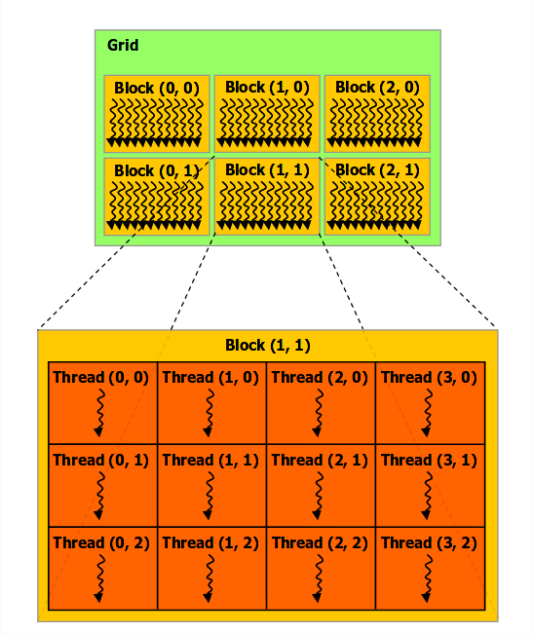
Grid
Array of blocks executing same *kernel* :

- Access to global GPU memory
- Sync. by stop and start a new kernel

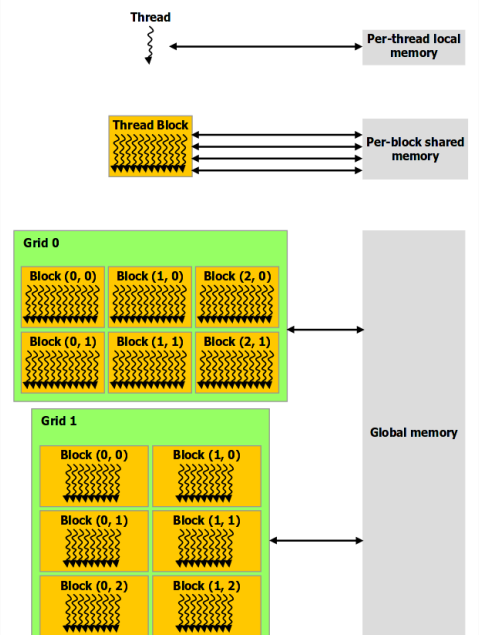


Programming Model - Summary

Hierarchy

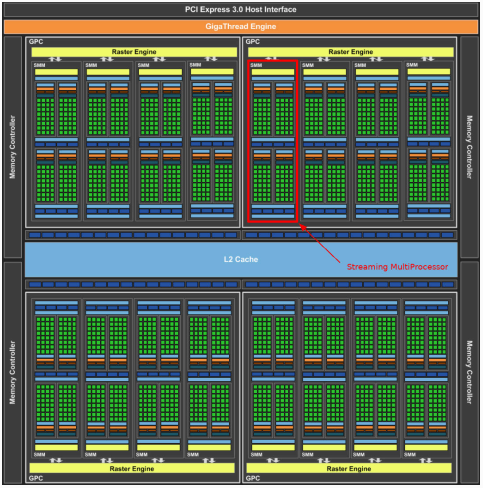


Memory



Mapping Programming model to hardware - the SMs

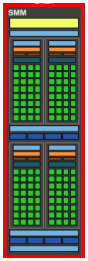
- CUDA's Thread/block/grid is mapped to GPU (N°Threads ≠ N°Cores)

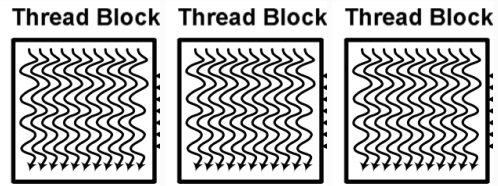


- GTX 980 - 16 Streaming Multi-processors (SM)
- SMs are rather independant

Mapping Programming model to hardware - the SMs

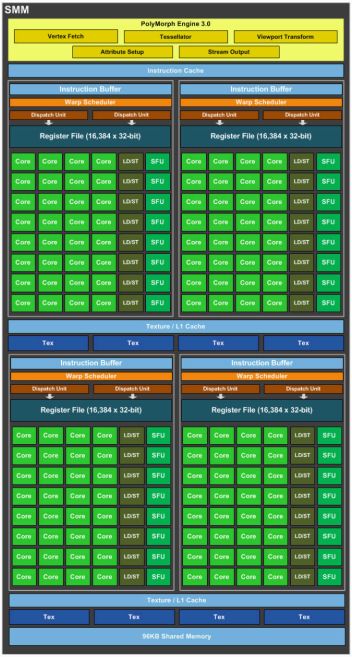
- *Any block* of *any grid* can be mapped to *any SM*





- Once *thread block* is affected to a SM, it won't move to another one.

Zoom on the SM



- SM organizes **blocks** into **warps**
- 1 warp = group of 32 threads

GTX 920 :

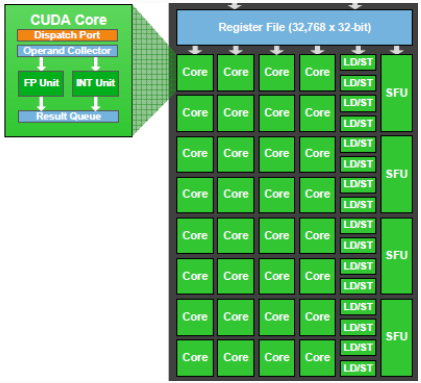
- 128 cores = 4 x 32 cores
- Quad warp scheduler selects **4** warps (**TLP**)
- And **2** independent instructions per warp can be dispatched each cycle (**ILP**)

Example :

- 1 (logical) *block* of 96 threads maps to : 3 (physical) *warps* of 32 threads

Zoom on the CUDA cores

1 core = 1 thread



- A *warp* executes 32 threads on the 32 CUDA cores
- The threads executes **the same** instruction (**DLP**)
- All instructions are SIMD (width = 32) instructions

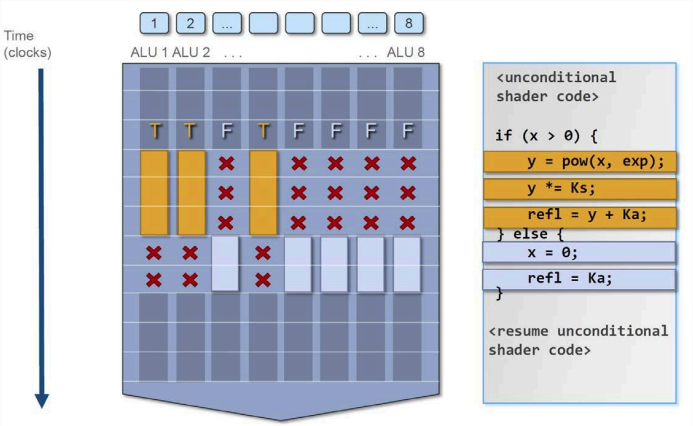
Each core :

- Floating point & Integer unit
- Fused multiply-add (FMA) instruction
- Logic unit
- Move, compare unit
- Branch unit
- The first IF/ID of the pipeline is done by the SM

Warning : SIMT allows to specify the execution and branching behavior of a single thread but maps to SIMD processors !

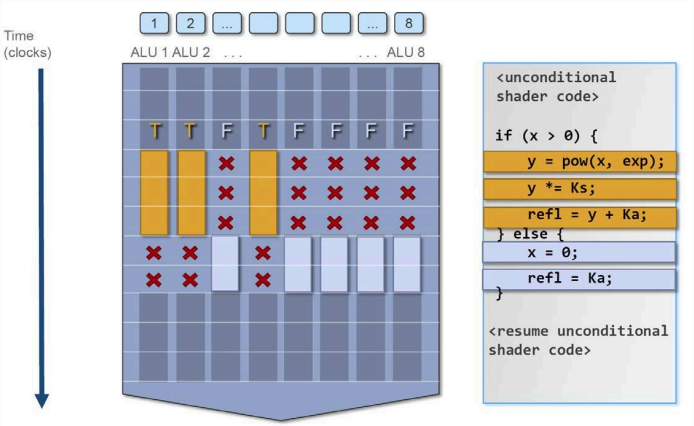
The SIMT Execution Model on CUDA Cores

- Divergent code paths (branching) pile up!



The SIMT Execution Model on CUDA Cores

- Divergent code paths (branching) pile up !



A mask allow to dis/activate threads :

If branch	1	1	0	1	0	0	0	0
Else branch	0	0	1	0	1	1	1	1

What is the latency (in term of inst) of this code in the better and worst case ?

```
if a > 0:
    inst-a
    if b > 0:
        inst-b;
    else
        inst-c
else:
    inst-d
```

What is the latency (in term of inst) of this code in the better and worst case ?

```
if a > 0:
    inst-a
    if b > 0:
        inst-b;
    else
        inst-c
else:
    inst-d
```

- Best case : `a > 0` is false for every thread. For all threads : `inst-d`
- Worst case : `a > 0` and `b > 0` is true for some but not all threads. For all threads :

```
inst-a
inst-b
inst-c
inst-d
```

Loops

```
i = 0
while i < thread_id:
    i += 1
```

Loops

```
i = 0
while i < thread_id:
    i += 1
```

Unrollable loops cost = max iterations, ie :

- Keep looping until all threads exit
- Mask out threads that have exited the loop

Execution trace	T0	T1	T2	T3
`i = 0`	0	0	0	0
`i < tid`	0	1	1	1
`i++`	0	1	1	1
`i < tid`	0	0	1	1
`i++`	0	1	2	2
`i < tid`	0	0	0	1
`i++`	0	1	2	3
`i < tid`	0	0	0	0

Occupancy : ILP vs TLP (1/2)

Occupancy

number of warps executed at the same time divided by the maximum number of warps that can be executed at the same time.

Generation	Warps per SM	Warps per scheduler	Issue rate	Issue width
G80	24	24	2	1
GT 200	32	32	2	1
Fermi	48	24	2	2
Kepler/Maxwell	64	16	1	2

Occuppancy (warps/SM)	G80	GT200	Fermi	Kepler	Maxwell
100 %	24	32	48	64	64
50 % ...	12	16	24	32	32

Occupancy : ILP vs TLP (1/2)

Occupancy

number of warps executed at the same time divided by the maximum number of warps that can be executed at the same time.

Generation	Warps per SM	Warps per scheduler	Issue rate	Issue width
G80	24	24	2	1
GT 200	32	32	2	1
Fermi	48	24	2	2
Kepler/Maxwell	64	16	1	2

Occuppancy (warps/SM)	G80	GT200	Fermi	Kepler	Maxwell
100 %	24	32	48	64	64
50 % ...	12	16	24	32	32

CUDA Best Practice

It is a common and recommended practice to launch many more thread blocks than can be executed at the same time.

`No, increasing ILP is another way`

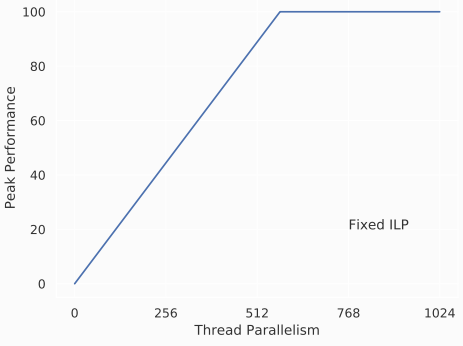
Occupancy : ILP vs TLP (2/2)

Check it on the GTX 480

ILP = 1

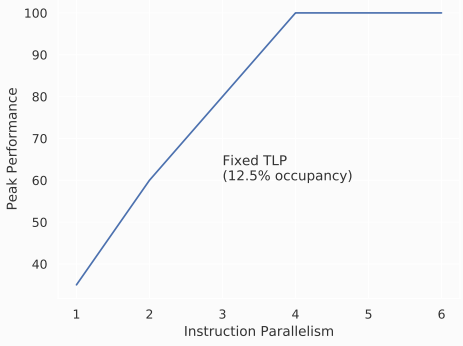
```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ )
{
    u = a * u + b;


}
```



ILP = 2 and more

```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ )
{
    u = a * u + b;
    v = a * v + b;
    ...
}
```



Final note about terminology

	NVidia/CUDA	AMD/OpenCL	“CPU”
Thread	Cuda Processor	Processing Element	Lane
	Cuda Core	SIMD unit	Vector
GPU Core	Streaming Multiprocessor	Compute Unit	Core
	GPU Device	GPU Device	Device

GPU memory model

Computation cost vs. memory cost

- Power measurements on NVIDIA GT200

	Energy/op (nJ)	Total power (W)
Instruction Control	1.8	18
Mult-add 32-wide warp	3.6	36
Load 128B from DRAM	80	90

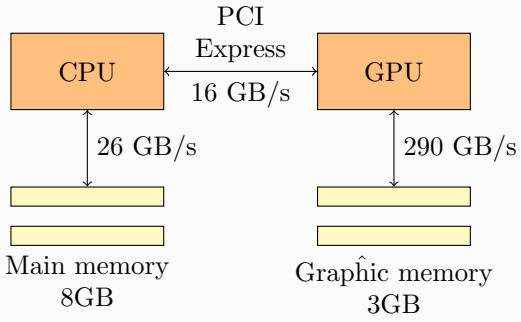
With the same amount of energy :

- Load 1 word from external memory (DRAM)
- Compute 44 flops

→ Must optimize memory first

Classical CPU-GPU model

- Split memory space
- Highest bandwidth from GPU memory
- Transfers to main memory are slower

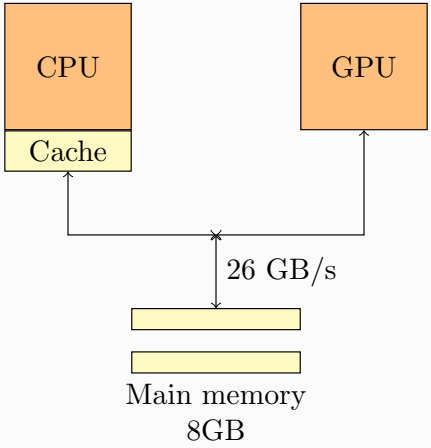


Intel i7 4770 / GTX 780

External memory : embedded GPU

Most GPUs today :

- Same memory
- May support memory coherence (GPU can read directly from CPU caches)
- More contention on external memory



GPU : on-chip memory

Cache area in CPU vs GPU :

Figure 1-2. The GPU Devotes More Transistors to Data Processing

GPU : on-chip memory

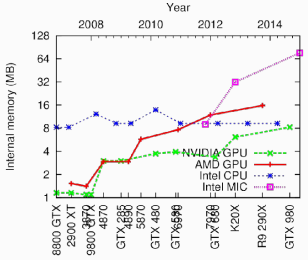
Cache area in CPU vs GPU :

Figure 1-2. The GPU Devotes More Transistors to Data Processing

But if we include registers :

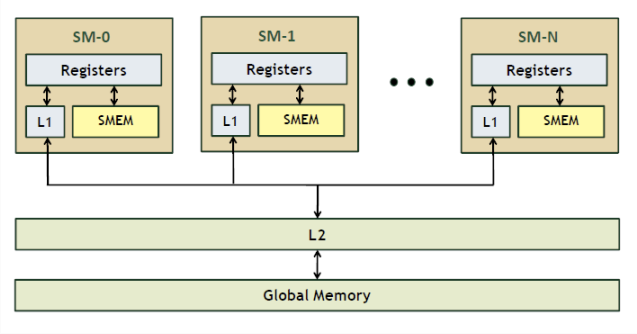
GPU	Registers files + caches
NVidia Maxwell	8.3 MB
AMD Hawaii GPU	15.8 MB
Core i7 CPU	9.3 MB

	CPU	GPU
Register / Core	256	65K



→ GPU has many more registers but made of simpler memory

Memory model hierarchy (hardware)

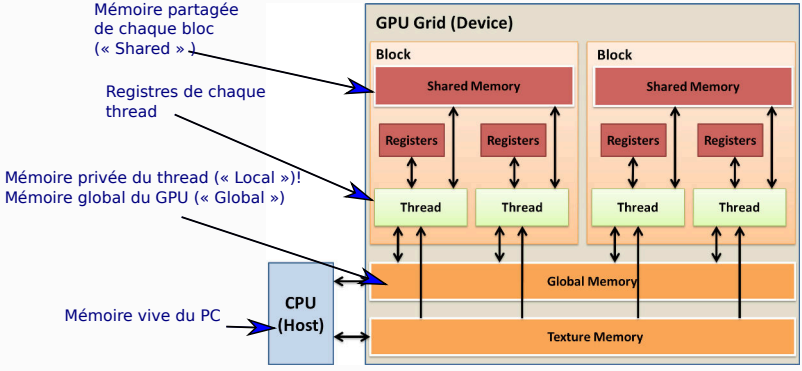


Cache hierarchy :

- Keep frequently-accessed data Core
- Reduce throughput demand on main memory L1
- Managed by hardware (L1, L2) or software (shared memory)

- On CPU, caches are designed to avoid memory latency
- On GPU, multi-threading deals with memory latency

Memory model hierarchy (software)



Memory	On chip	Cached	Access	Scope	Lifetime
Register	✓	n/a	RW	1 thread	Thread
Local	✗	✓	RW	1 thread	Thread
Shared	✓	n/a	RW	All threads block	Block
Global	✗	✓	RW	All threads + host	Host
Constant	✗	✓	R	All threads + host	Host
Texture	✗	✓	R	All threads + host	Host

Memory alignment

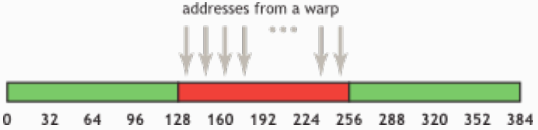
- Memory addresses must be type-aligned (ie `sizeof(T)`)
- Otherwise : poor perf (unaligned load)
- cudaMalloc = alignment on 256 bits (at least)

Coaleseced Access

- Minimize memory accesses caused by wrap threads
- Remind : all threads of a warp executes the same instruction
  → if a load, may be 32 different adresses

We need a load stategy :

- 32 threads of warp acces a 32-bit word = 128 bytes
- 128 bytes = L1 bus width (single load)

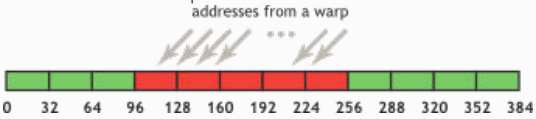


- Data for *inactive* threads are loaded too !
- Access permutation has no (or very low) overhead

Non-aligned and strided load in L1 & L2

- If data are not 128-bits alignes, two load are required

Adresses 96-224 required...

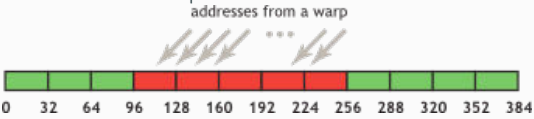


... but 0-256 loaded

Non-aligned and strided load in L1 & L2

- If data are not 128-bits alignes, two load are required

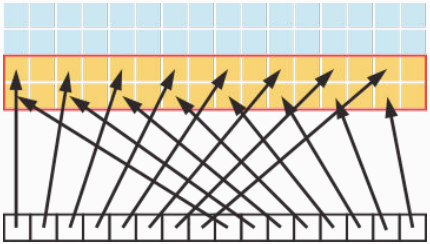
Adresses 96-224 required...



... but 0-256 loaded

- If data is accessed strided

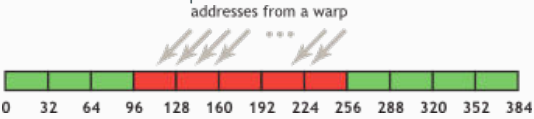
e.g. `u[2*k]`,...



Non-aligned and strided load in L1 & L2

- If data are not 128-bits alignes, two load are required

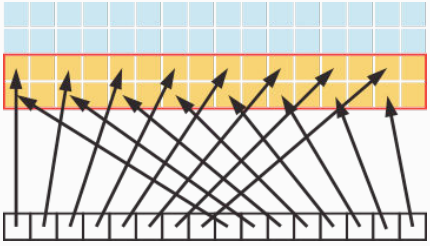
Adresses 96-224 required...



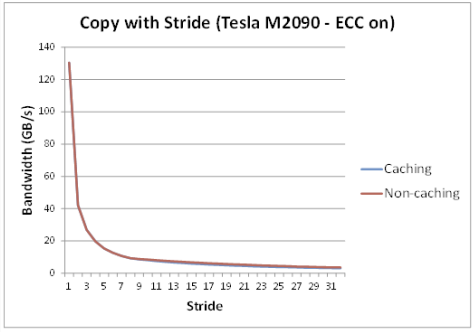
... but 0-256 loaded

- If data is accessed strided

e.g. `u[2*k]`,...



...cry !



Non-aligned and strided load in L1 & L2

Q : how to make aligned-load with 2D-arrays ?

```
ima(x,y) = y * width + x
```

Non-aligned and strided load in L1 & L2

Q : how to make aligned-load with 2D-arrays ?

```
ima(x,y) = y * width + x
```

To have coalesced access, you need to have a multiple of 32 (warp size) for :

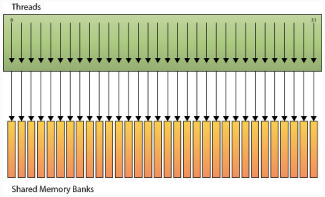
- the array width (it means *padding*)
- the thread block size

```
ima(x,y) = y * stride + x
```

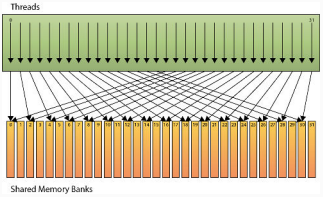
Shared Memory & conflicts

Used a lot for local copy and fast-access

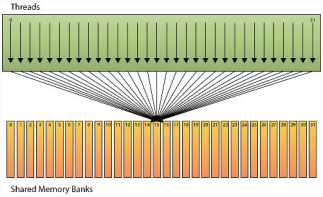
- Organized in memory bank (accessed concurrently)
- Every bank can provide 64 bits every cycle
- Only two modes :
  - Change after 32 bits
  - Change after 64 bits



No conflict



2-way conflict



No conflict (broadcast)

Conflict = Serialized access (bad !)

Next time...

CUDA crash course