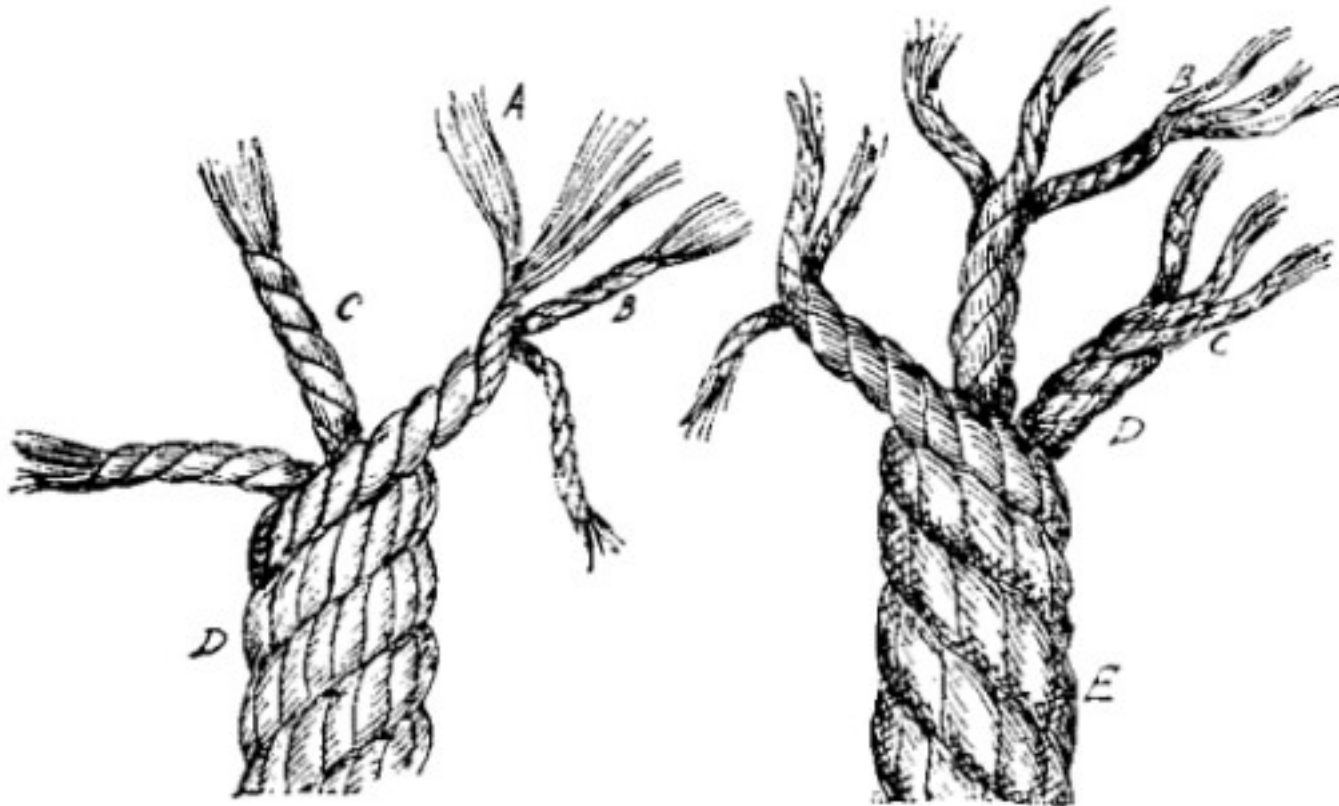


An Introduction of parallel computing programming



/SED/

Guillermo Andrade B.
Service d'Expérimentation et Développement

Guillermo.Andrade@inria.fr

Jan. 2021

inria
informatics mathematics

Overview

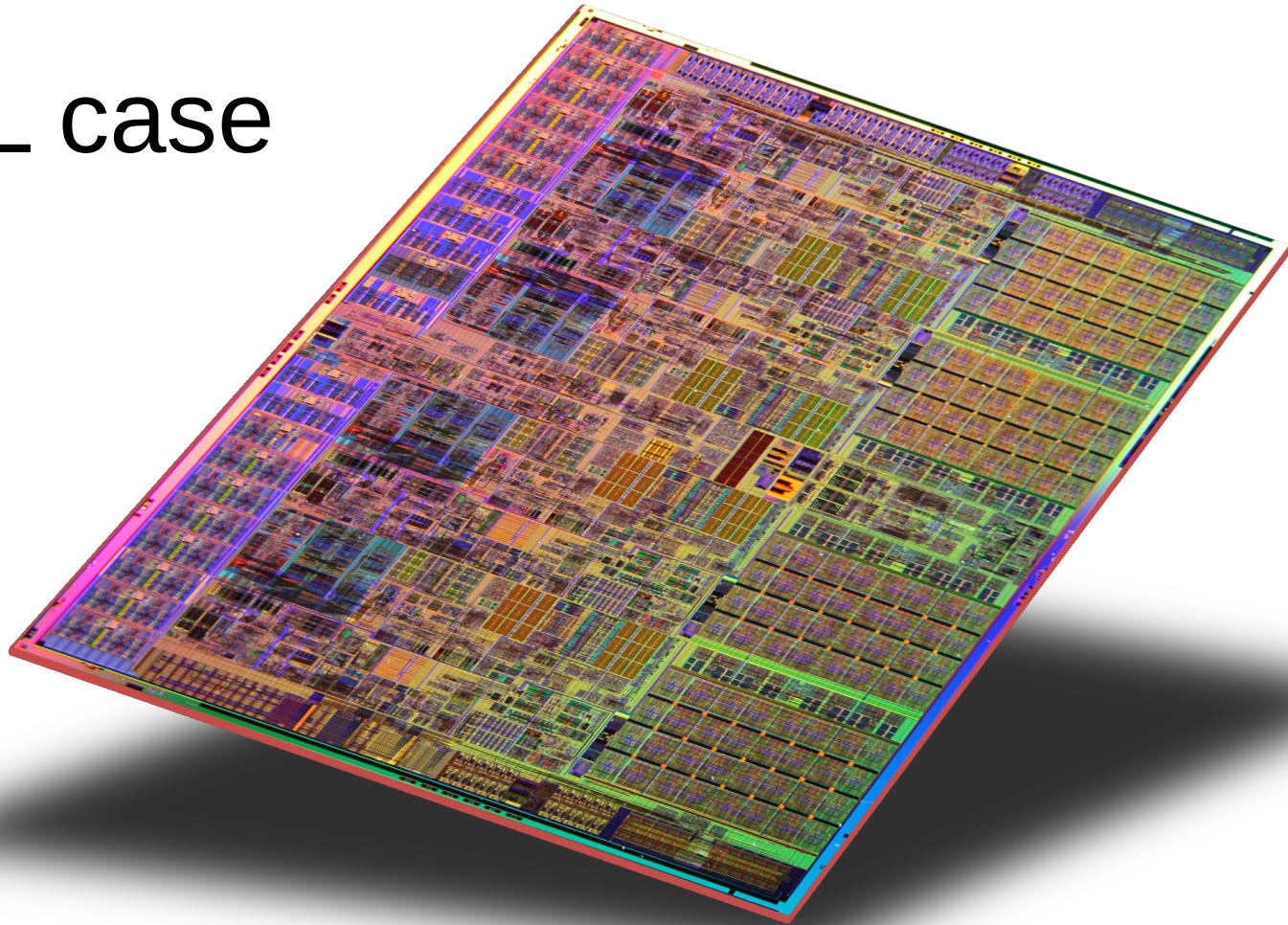
- **Part 1 : Basics on parallel machine**
- Part 2 : GPU and CUDA
- Part 3 : OpenCL & portable tools
- Part 4 : Performance

Part 1 : Basics on parallel machine

- CPU architectures
- Threads
- Intel SSE instructions set
- Multi-cores with OpenMP

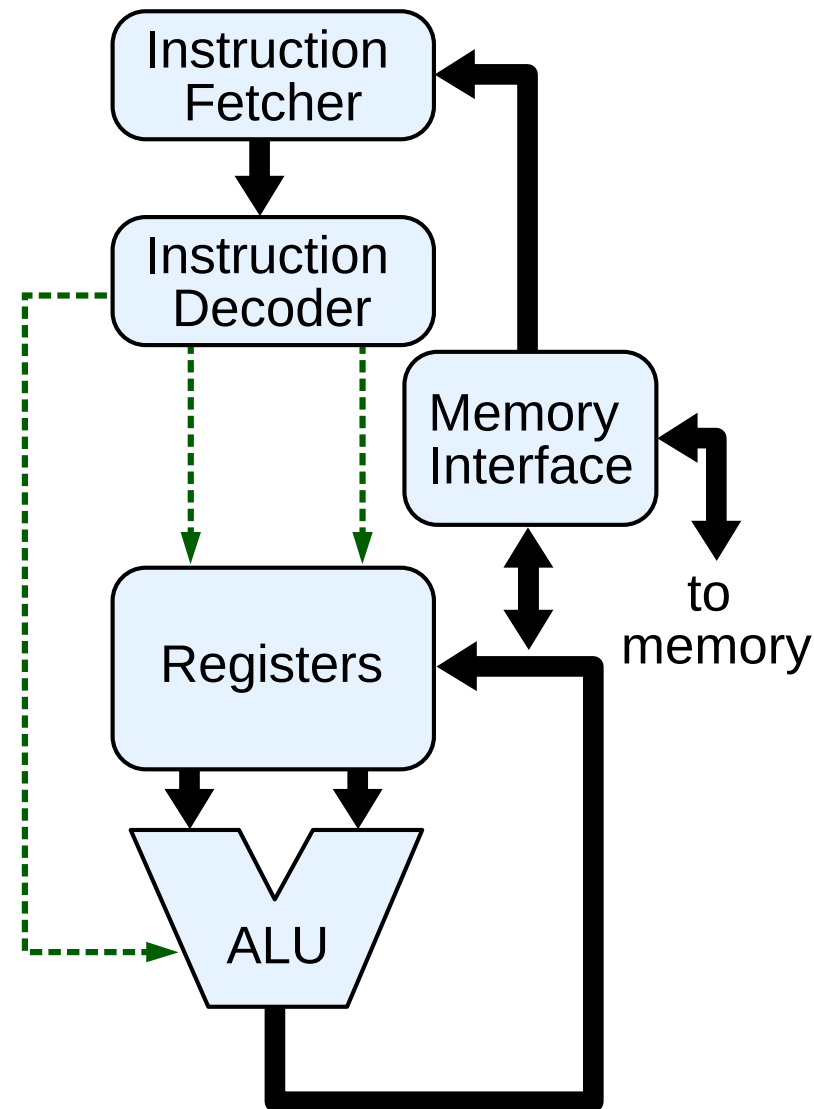
CPU architectures

- INTEL case



Intel Core i7

Sequential machine



https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/Les_composants_d%27un_processeur

Sequential machine

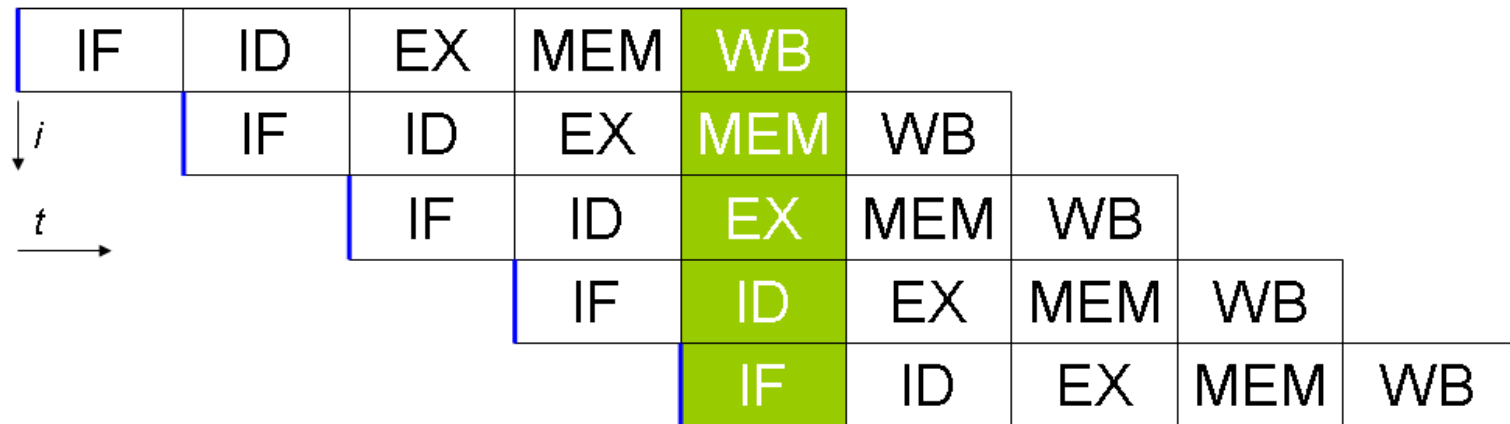


IF (Instruction Fetch)
ID (Instruction Decode)
EX (Execute)
MEM (Memory)
WB (Write Back)

Consequences:
Unroll loops => boost performance

Pipeline

- **i486**



Consequences:

Issues on Conditional instructions => empty pipeline => predictive branch analysis

Unroll loops => boost performance

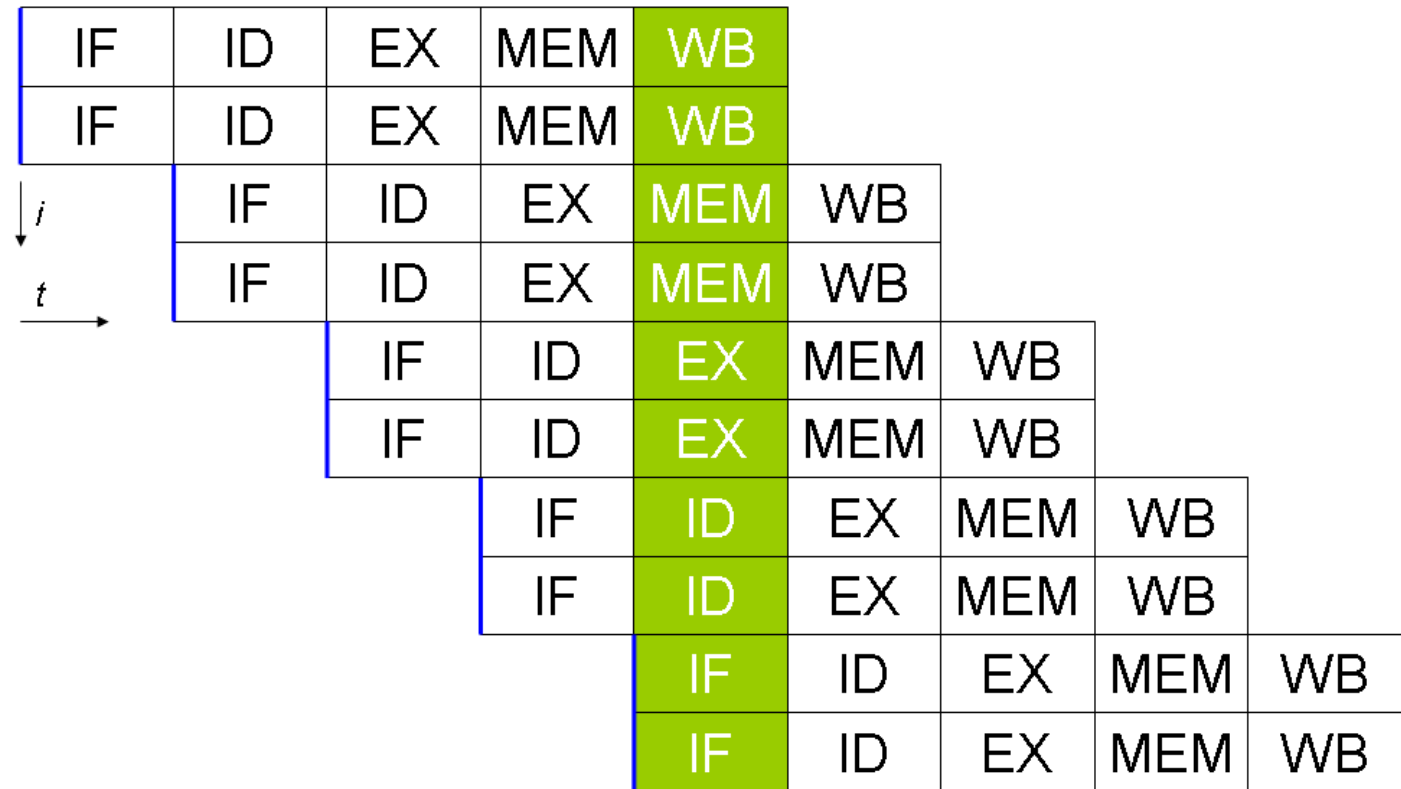
Super-scalar

- **Pentium**

- 1 port for int
- 1 port for float

- **Core 2**

- 4 ports



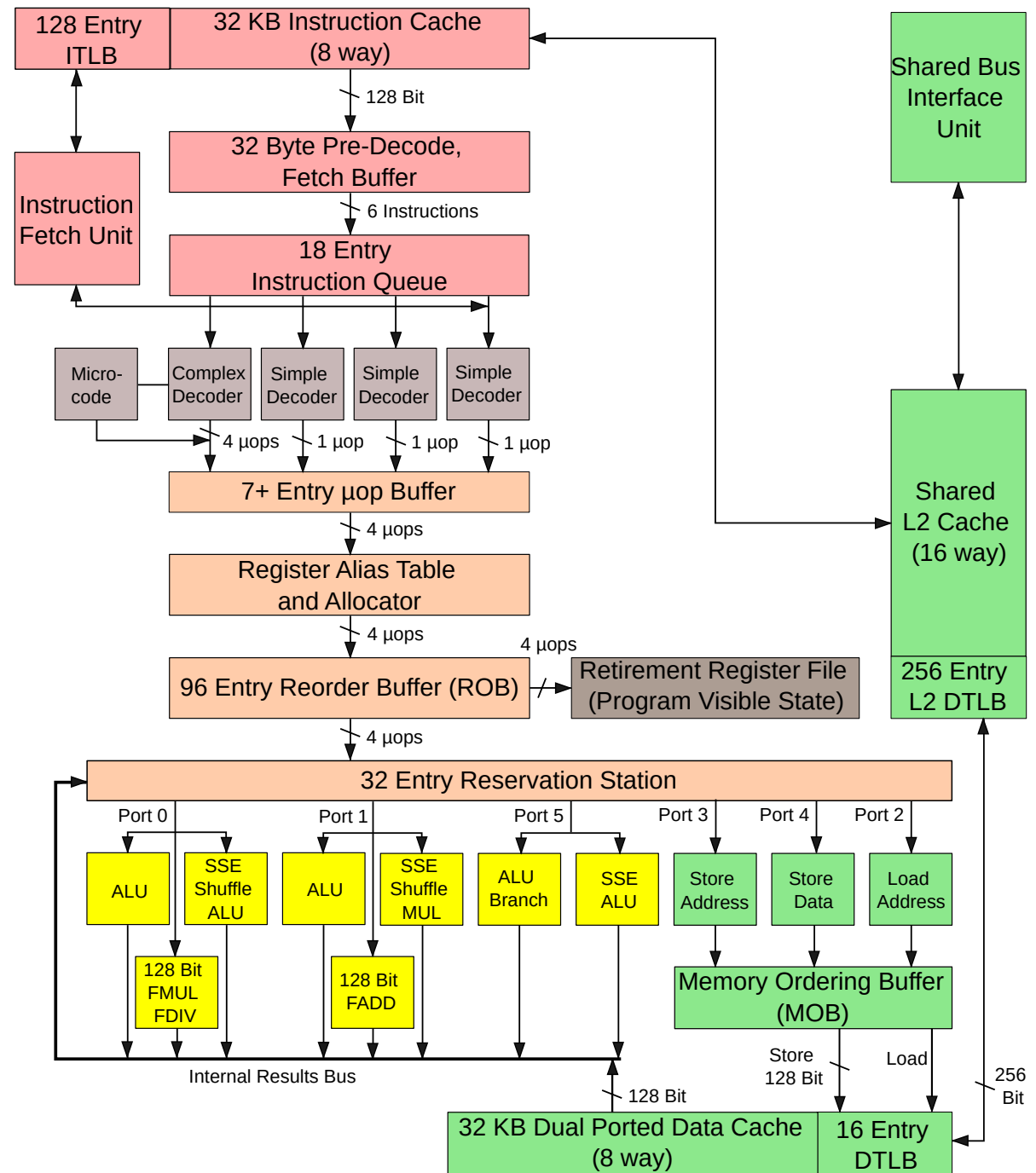
Consequences:

Issues on Conditional instructions => predictive branch analysis

Balance charge => re-order instructions boost performance => **out-of-order** architecture will be boost performance

Unroll loops => don't increase performance (not always)

Super-scalar Core 2 example



<https://en.wikipedia.org/wiki/Microarchitecture>

Intel Core 2 Architecture

SIMD instructions

- **Pentium MMX** (4 integers 16 bits)
- **Pentium III** (SSE 4 floats 32 bits)
- **Sandy Bridge** (AVX 8 floats 32 bits)
- **Skylake** (AVX-512 16 floats 32 bits)

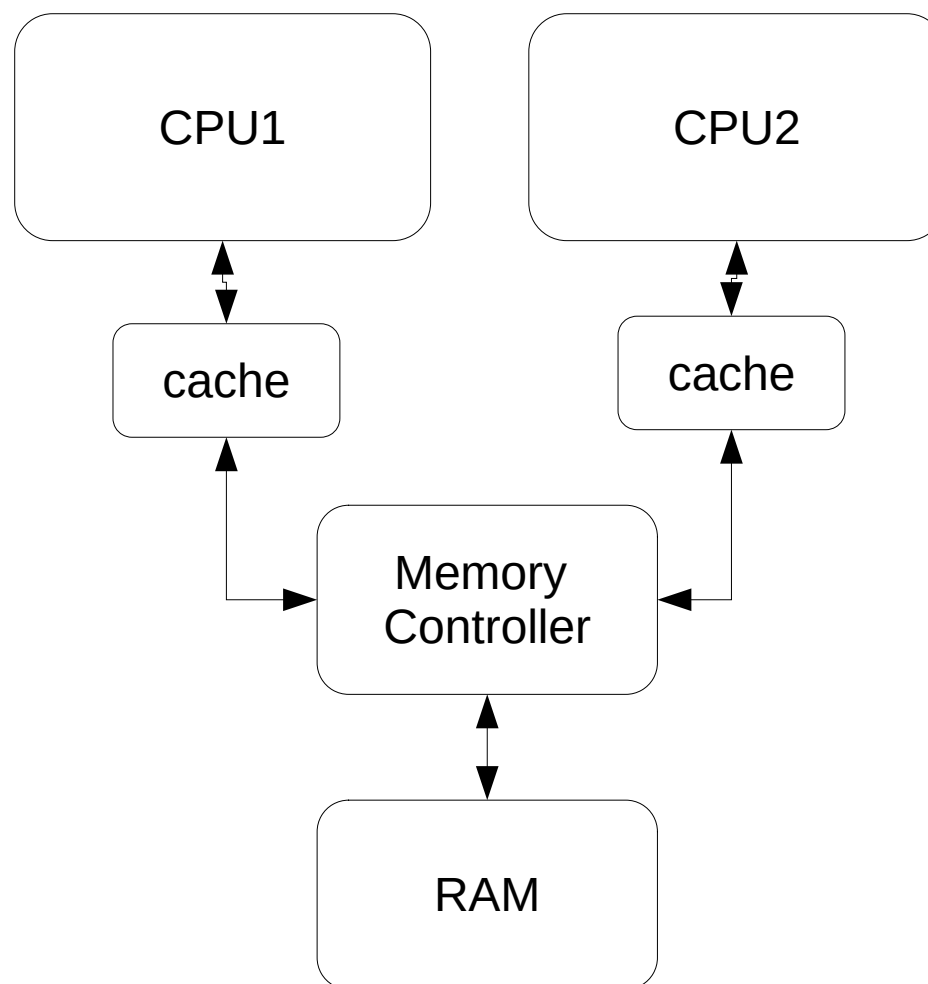
$$\begin{array}{c} \{ \\ X1, \\ Y1, \\ Z1, \\ W1 \\ \} \end{array} \quad \text{op} \quad \begin{array}{c} \{ \\ X2, \\ Y2, \\ Z2, \\ W2 \\ \} \end{array} = \begin{array}{c} \{ \\ X1 \text{ op } X2, \\ Y1 \text{ op } Y2, \\ Z1 \text{ op } Z2, \\ W1 \text{ op } W2 \\ \} \end{array}$$

Consequences: data parallel algorithms boosted

Symmetric Multi-processor (SMP)

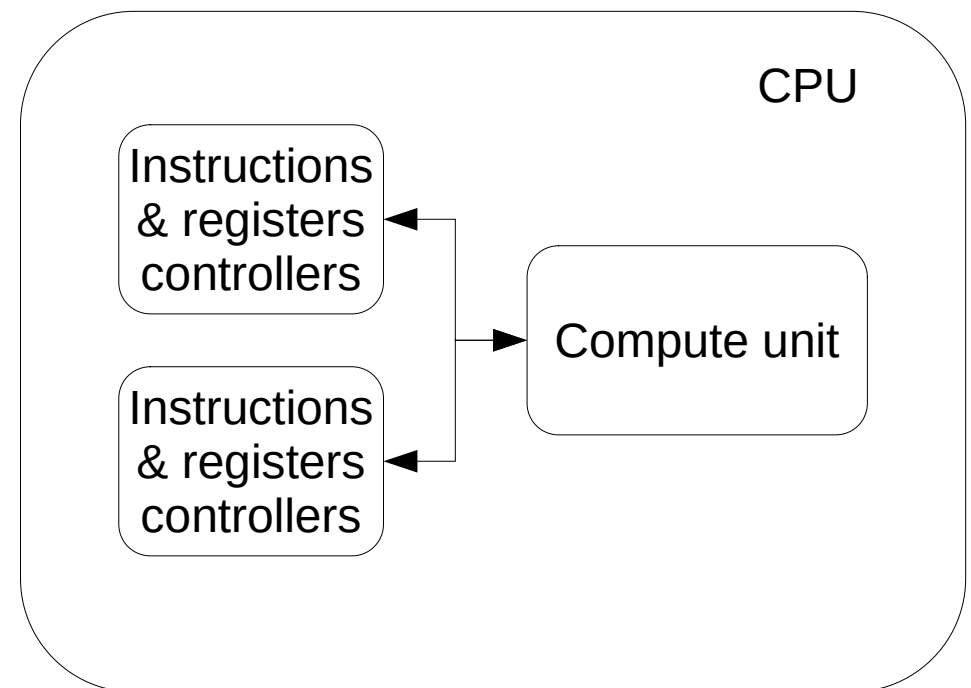
- **Pentium PRO**

Consequences:
write-memory caches need a memory controller => memory controller is often a bottleneck



Hyper-threading

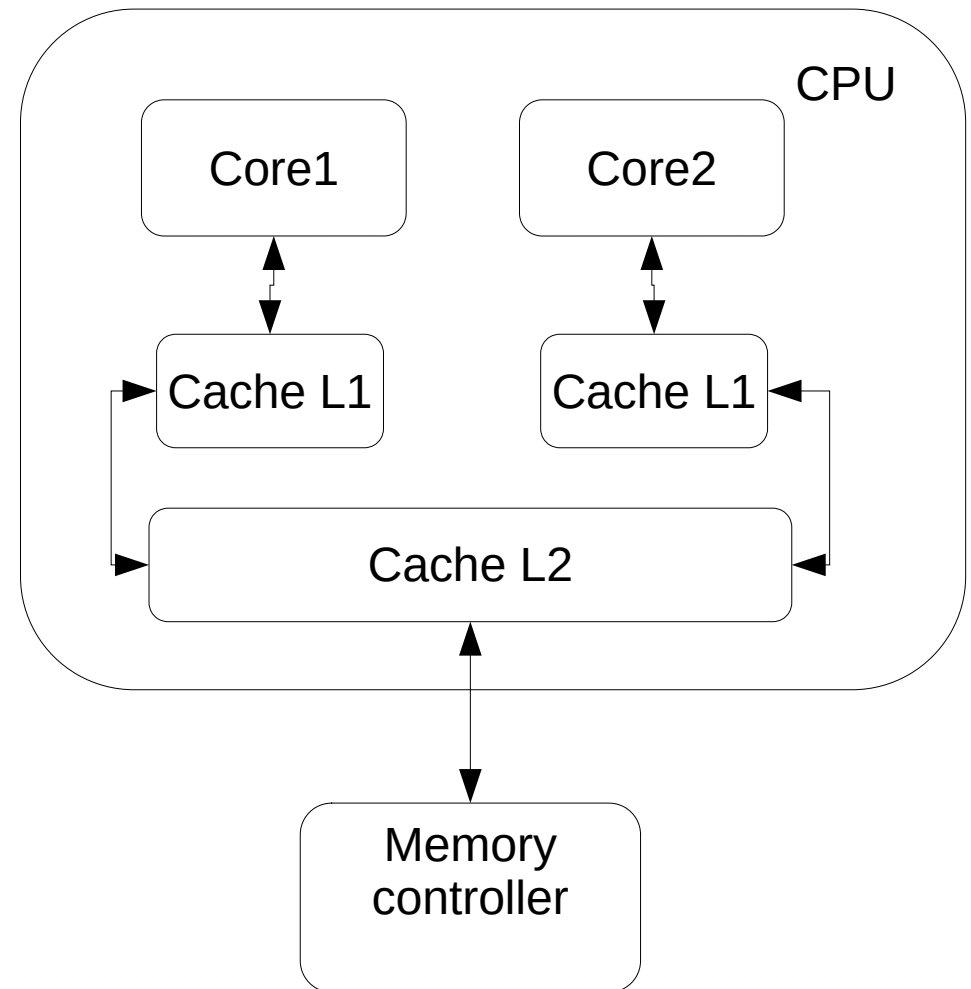
- **Pentium 4**
- **Core i7**



Consequences: Multi-task switching is improved => multi-thread programming with cooperative access is needed to preserve cache benefits

Multi-cores

- **Core 2**
 - Hierarchical coherent caches



Consequences: multi-process programming is improved => OS virtualisation is possible

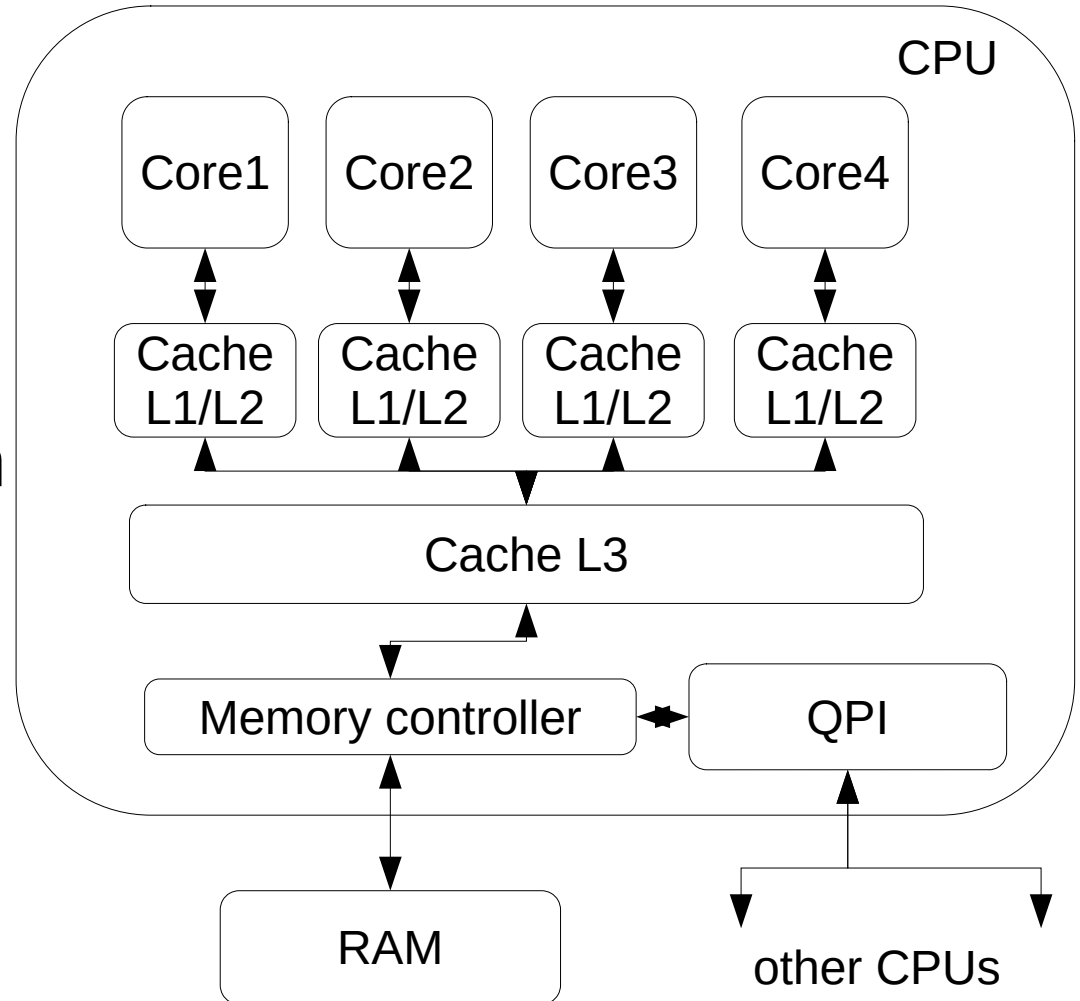
The true quad-core

- **Core i7**

- Hyper-threading
 - 8 threads on the air
- QPI : point-to-point CPU interconnection
 - Up to 4 CPU =>
 $4 \times 4 \times 2 = 32$ threads!

- **Sandy Bridge, ...**

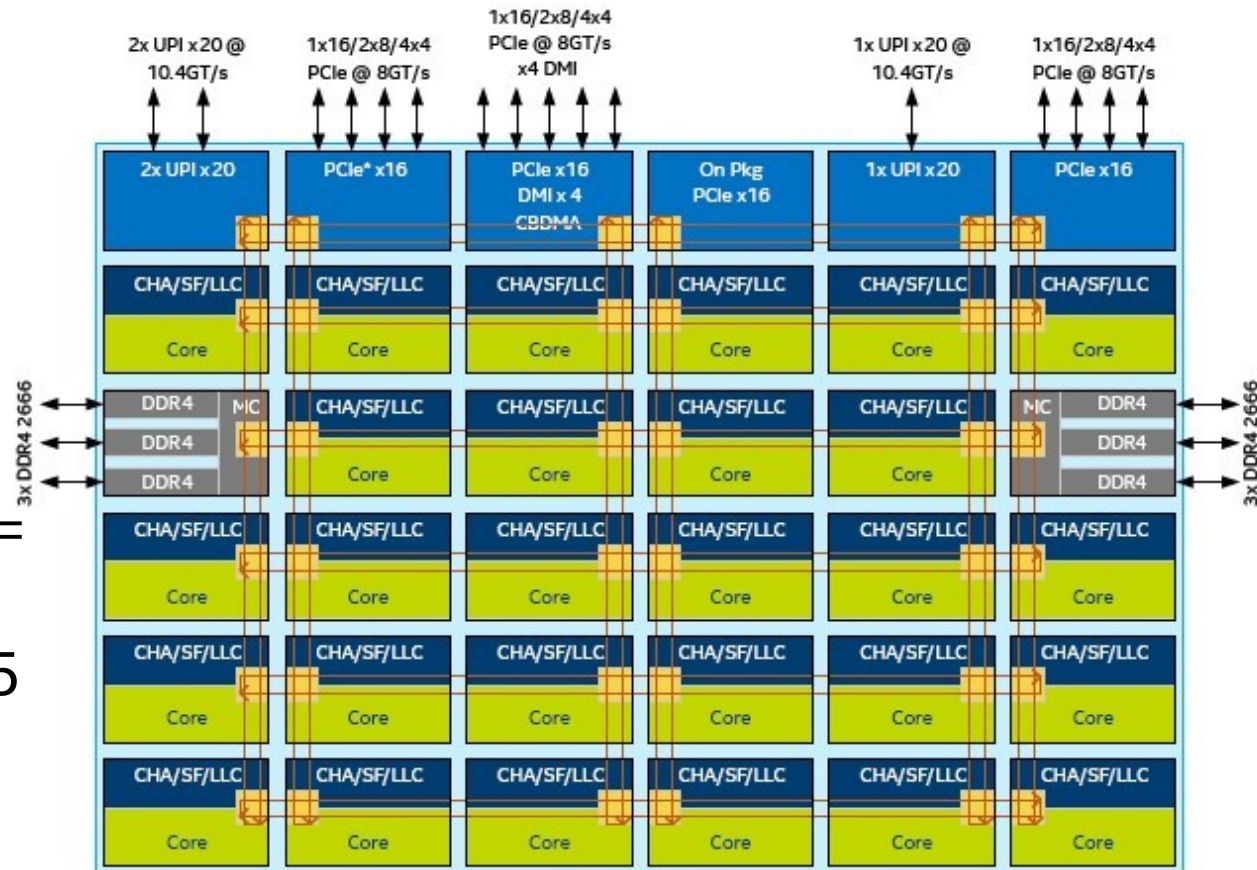
- Up to 8 cores by CPU



Cooper Lake (3rd Xeon scalable CPU)

- **Up to 28 cores**

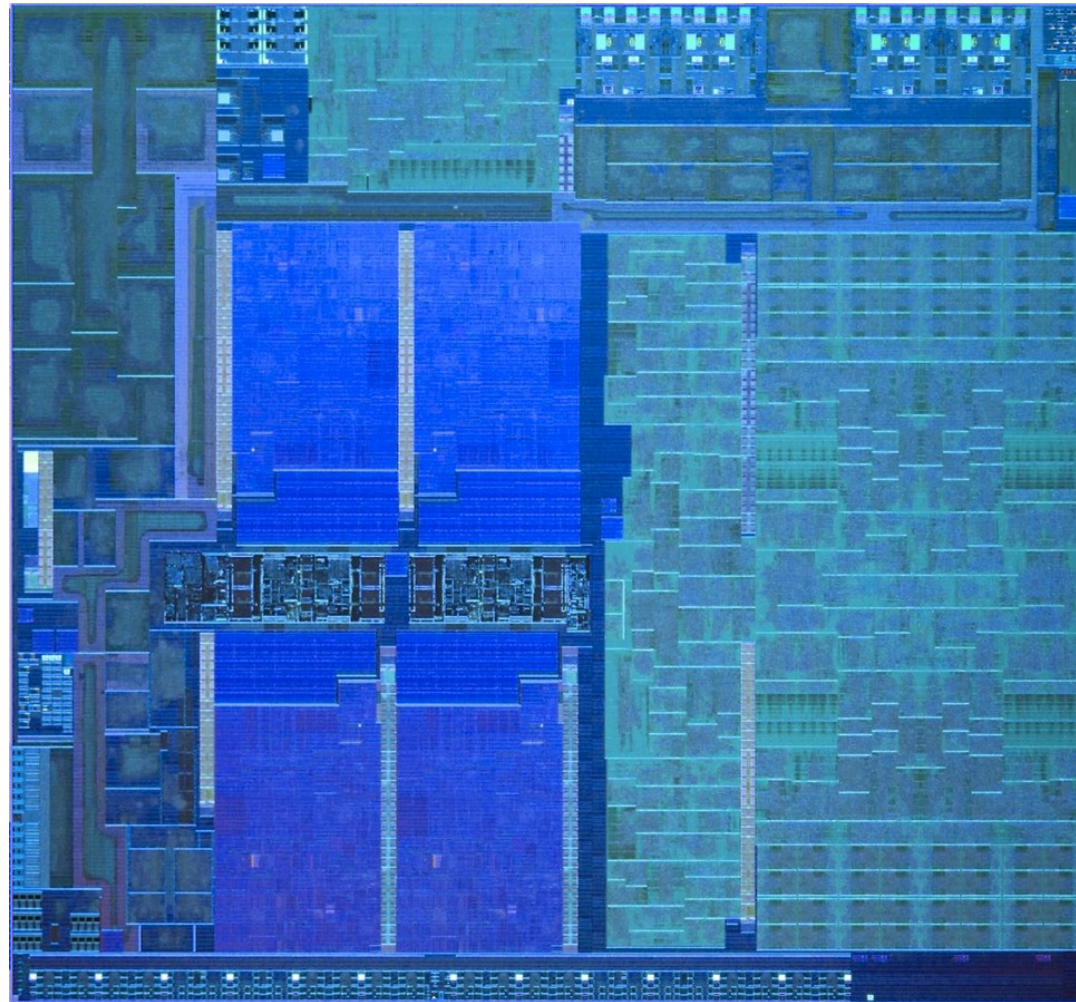
- Hyper-threading 2-way
 - 56 threads on the air
- Up to 3 UPI : point-to-point CPU interconnection
 - Up to 8 CPU => 8x28x2= 448 threads!
- (LLC) ~ L3 Cache 1.375 MiB/core
- AVX-512
 - VNNI, BF16 for neuron computation



CHA – Caching and Home Agent ; SF – Snoop Filter; LLC – Last Level Cache;
Core – Skylake-SP Core; UPI – Intel® UltraPath Interconnect

<https://wccfttech.com/intel-xeon-scalable-family-roadmap-revealed-points-out-cascade-lake-sp-in-q4-2018-cooper-lake-sp-in-q4-2019-ice-lake-sp-in-1h-2020/>

Hybrid CPUs

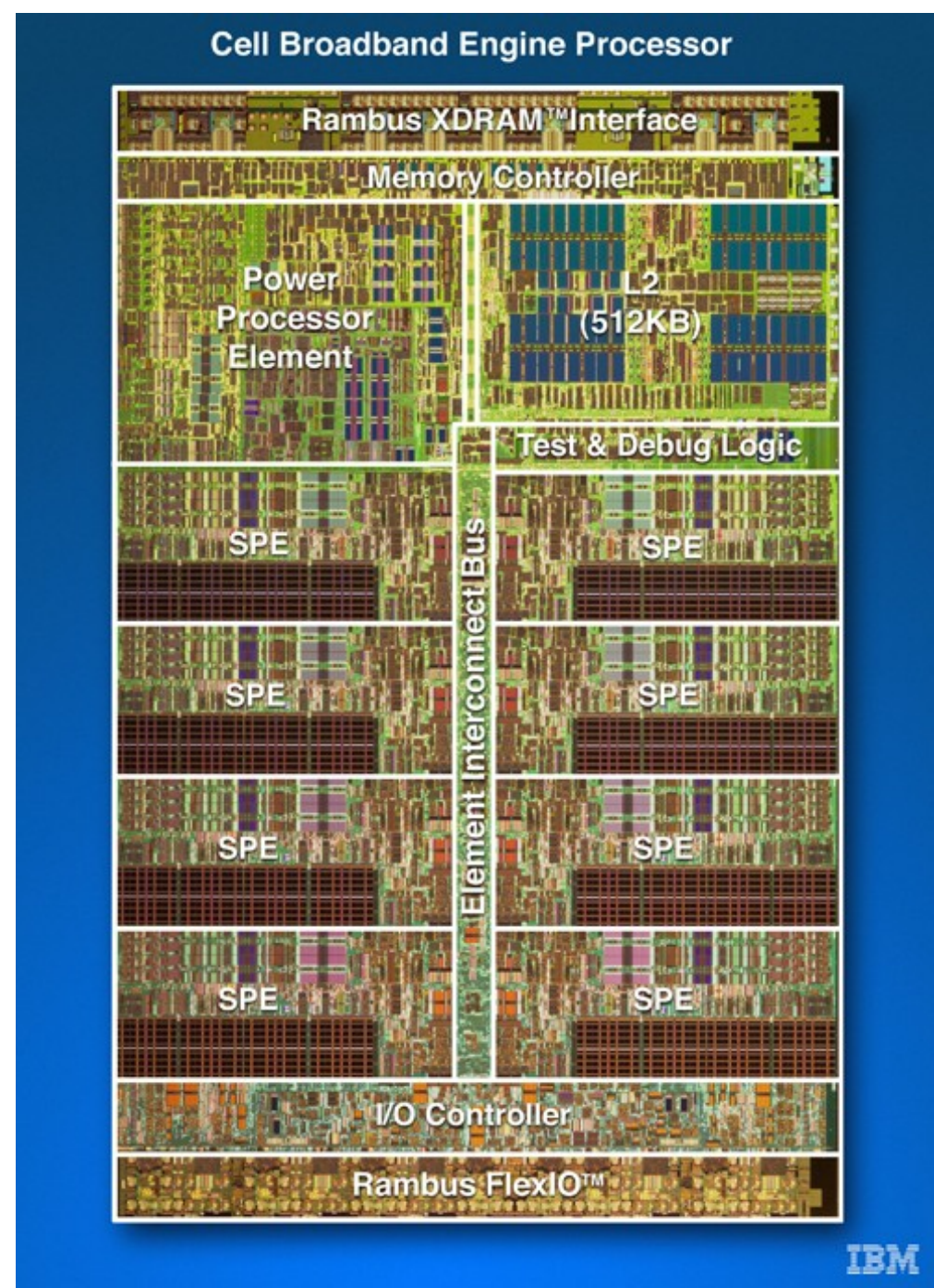


Intel IceLake

[https://en.wikichip.org/wiki/intel/microarchitectures/ice_lake_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/ice_lake_(client))

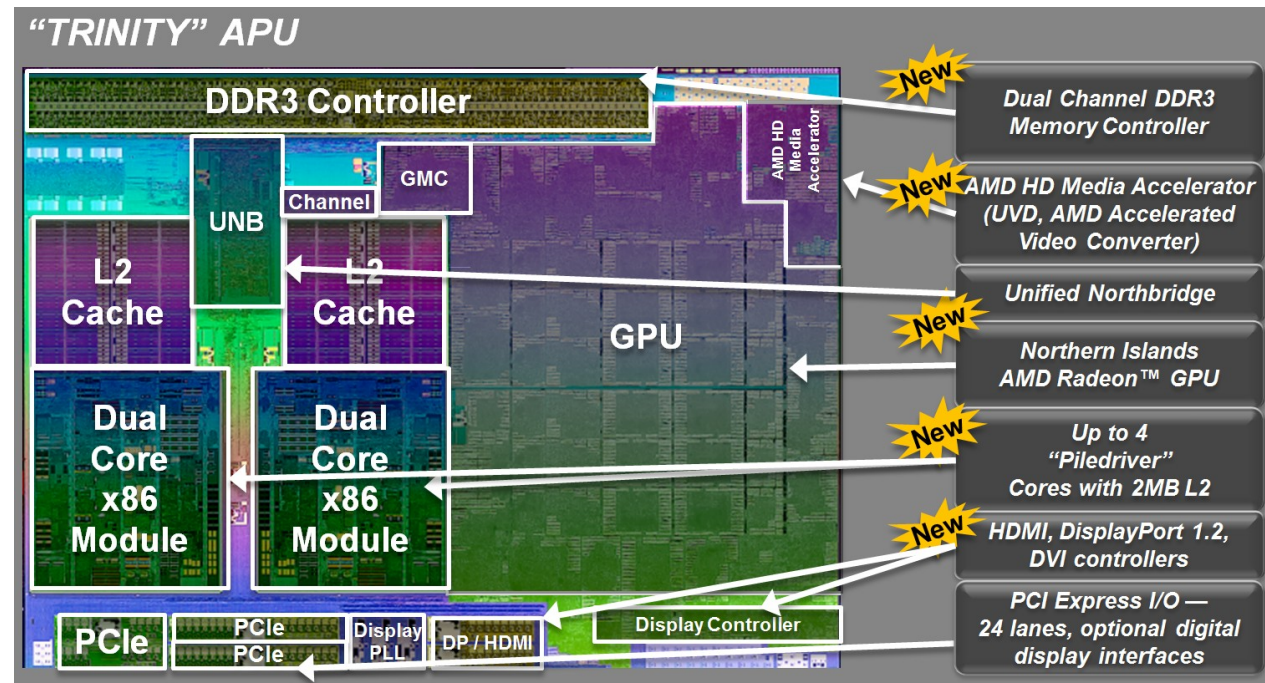
CellBE (IBM, TOCHIBA, SONY)

- “classic” core ~ PowerPC
- 8 cores SPE :
 - SIMD 4-ways
 - Equivalent to SSE or AltiVec instructions set
- 200 Gflops ($2 * 10^{11}$ ops / sec)



Trinity Fusion AMD

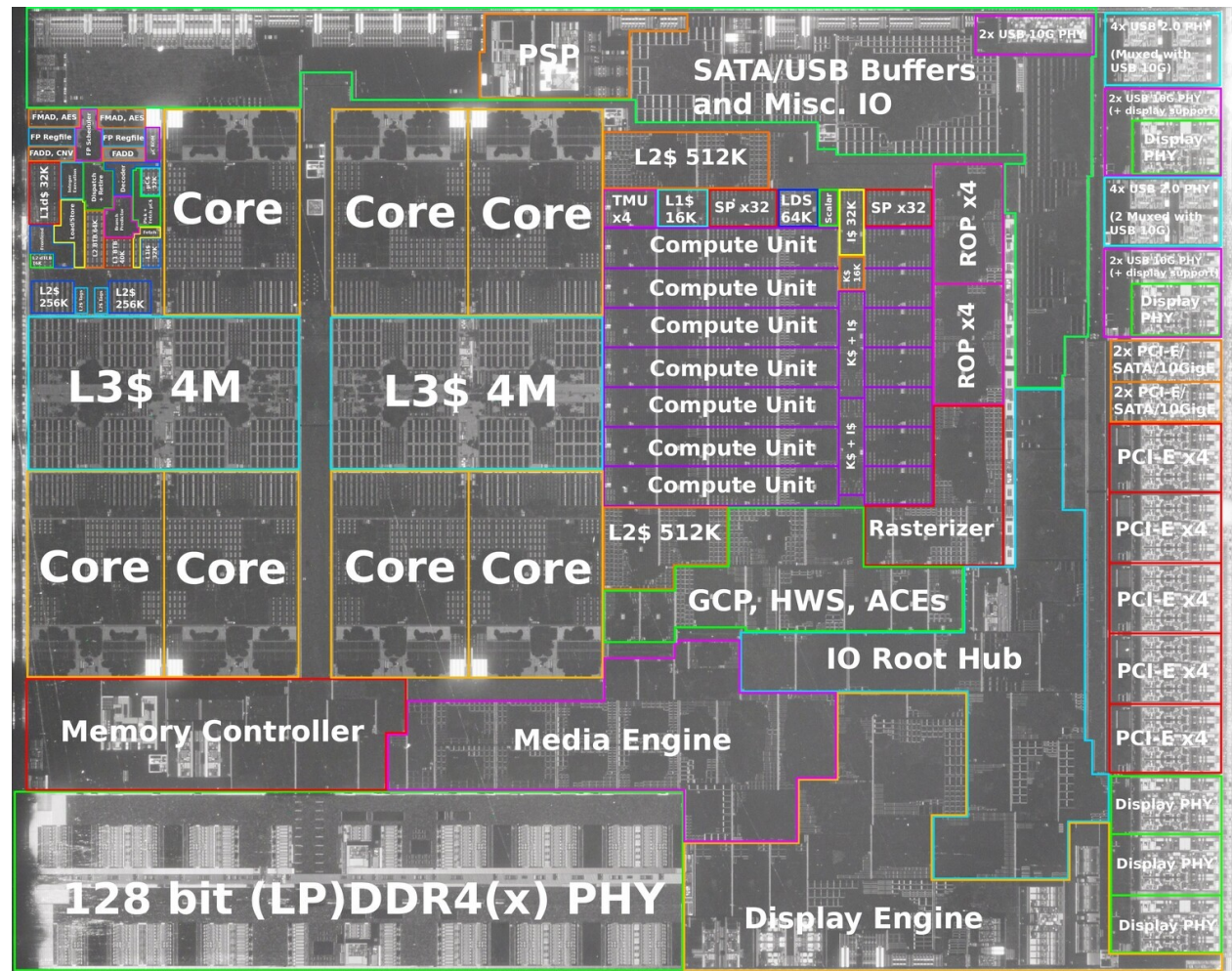
- 4 cores (*Piledriver*)
 - Instructions x86
 - AVX support
 - Hyper-threading x2
- GPU (Northern Islands)
 - 16 VLIW4 array =
384 ALUs



Source : <http://www.bjorn3d.com/2012/10/amd-virgo-platform-2nd-generation-apu/#.UIWhyRKyJAc>

AMD “Renoir” APU

- 8 cores (*Zen2*)
 - Instructions x86
 - AVX2 support
 - 2 hyper-thread
- GPU *Vega*
 - 8 Compute units
 - 512 ALUs
- OpenCL 2

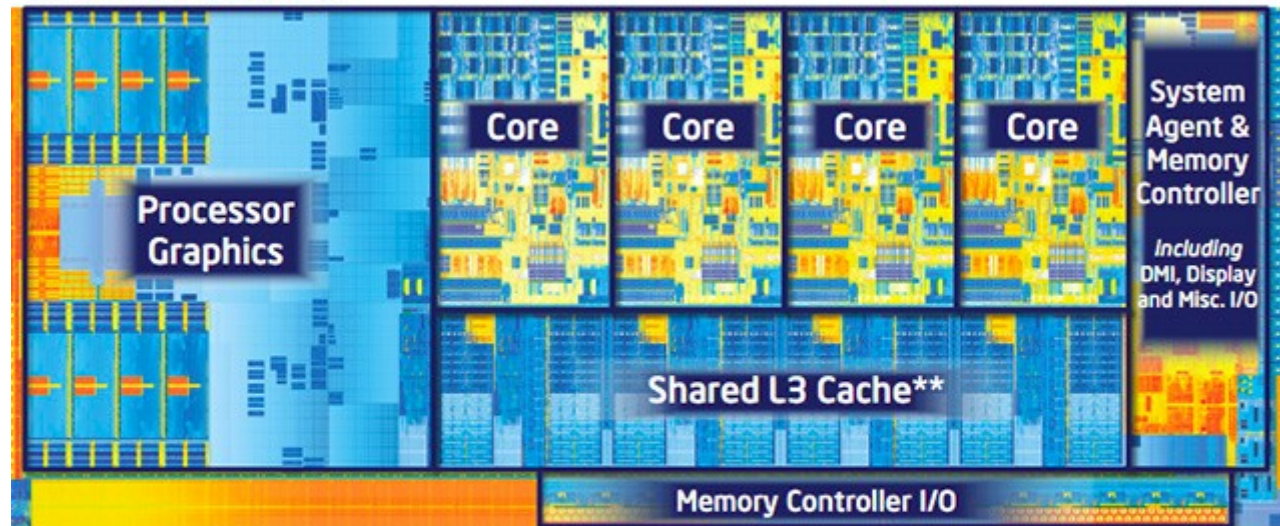


Source : <https://potatopc.net/amd-renoir-die-annotation-raises-hopes-of-desktop-chips-featuring-x16-peg/>
<https://en.wikichip.org/wiki/amd/cores/renoir>

Ivy Bridge INTEL

- 4 cores x86
- SIMD 8-way AVX
- Hyper-threading x2
- Support OpenCL

3rd Generation Intel® Core™ Processor:
22nm Process

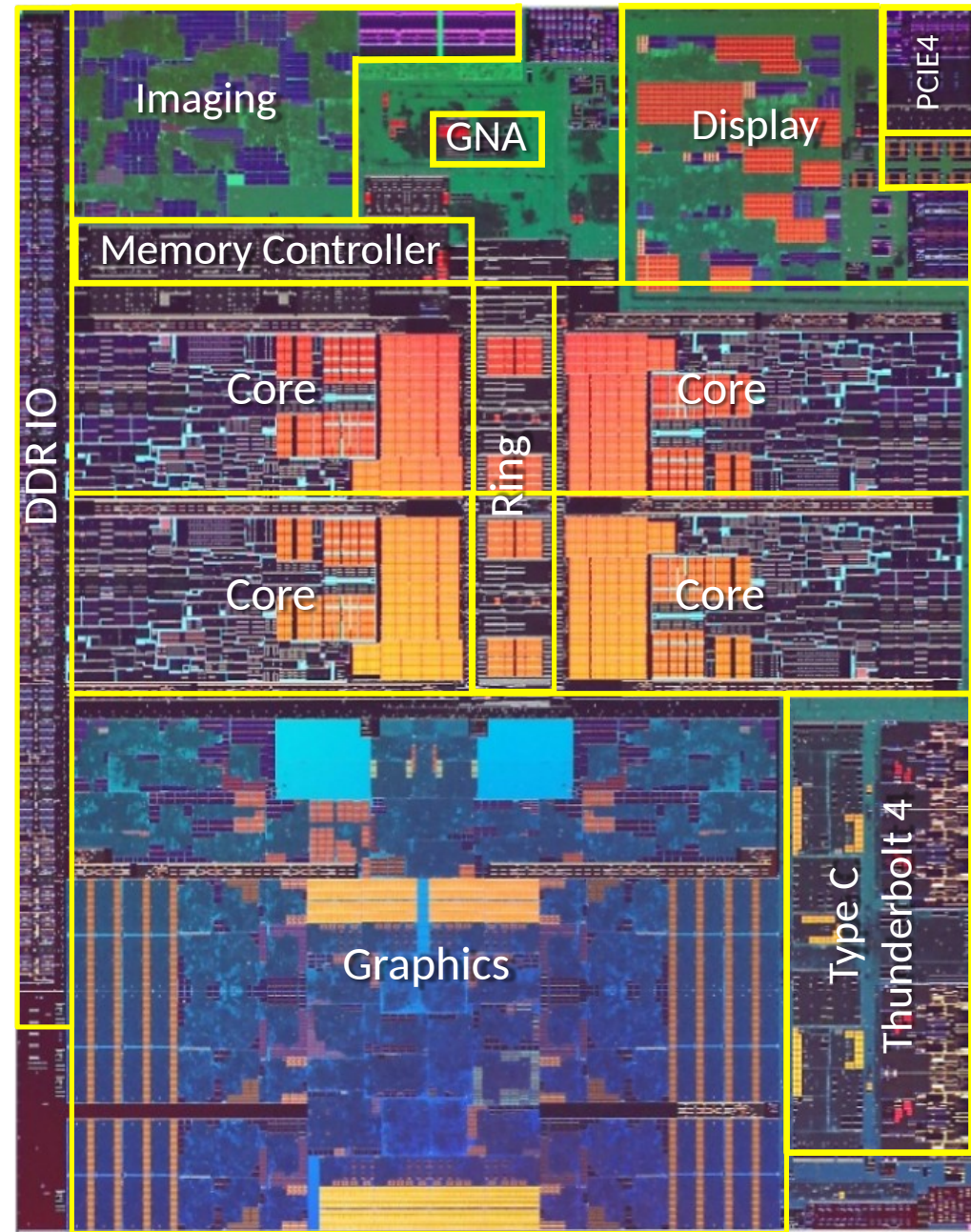


New architecture with shared cache delivering more performance and energy efficiency

Quad Core die with Intel® HD Graphics 4000 shown above
Transistor count: 1.4Billion Die size: 160mm²
** Cache is shared across all 4 cores and processor graphics

Tiger Lake INTEL

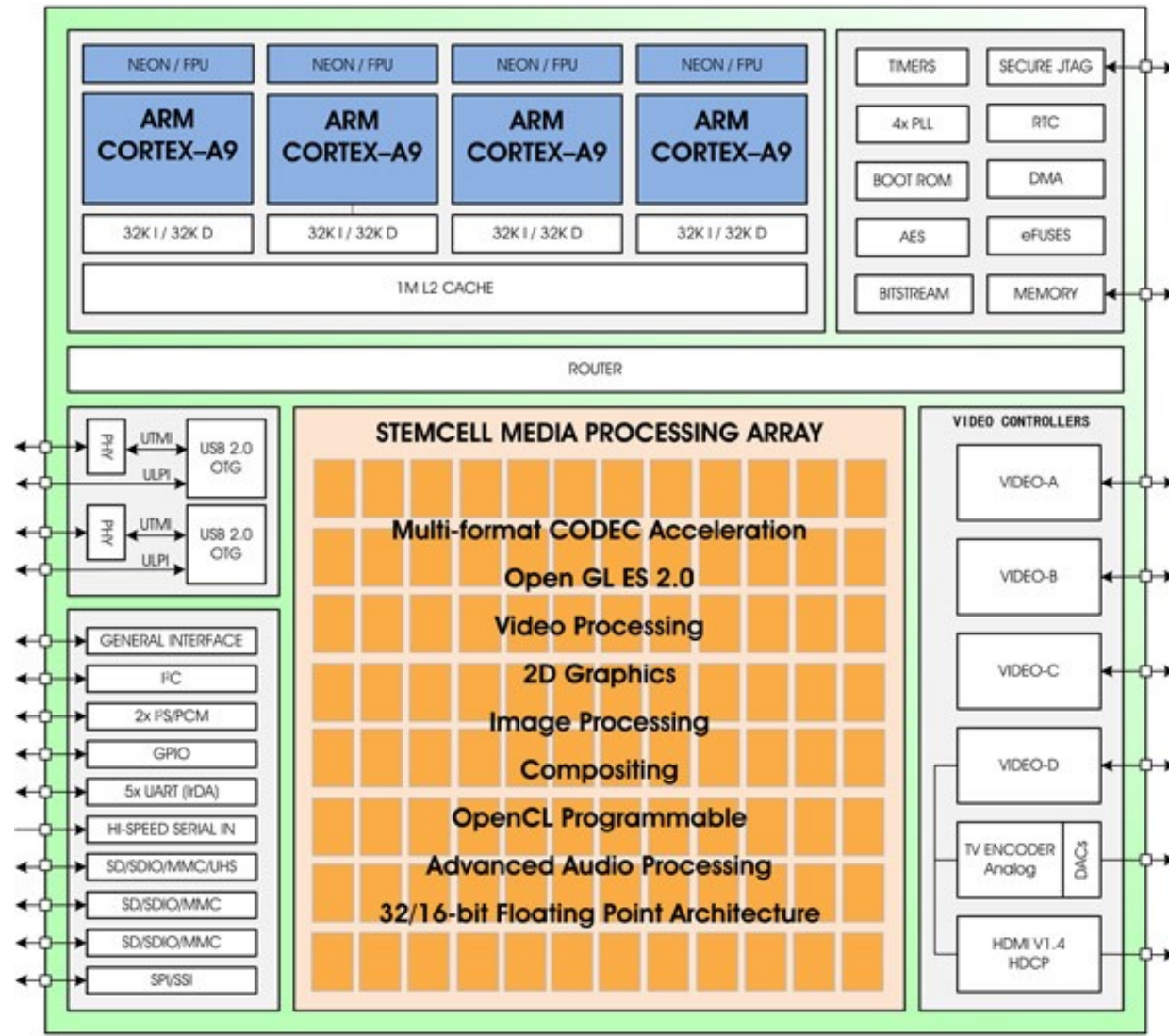
- 4 cores x86 (*Willow Cove*)
 - SIMD 8-way AVX2
 - SIMD 16-way AVX-512
 - Hyper-threading x2
- GPU (*Xe Gen 12*)
 - 96 Execution Units (EU)
= 1536 FP ALU
- GNA (Gaussian Neural Accelerator)
- IPU (Image Process Unit)



https://newsroom.intel.com/wp-content/uploads/sites/11/2020/09/Intel-Blueprint-Series_11th-Gen-Intel-Core-Processors.pdf

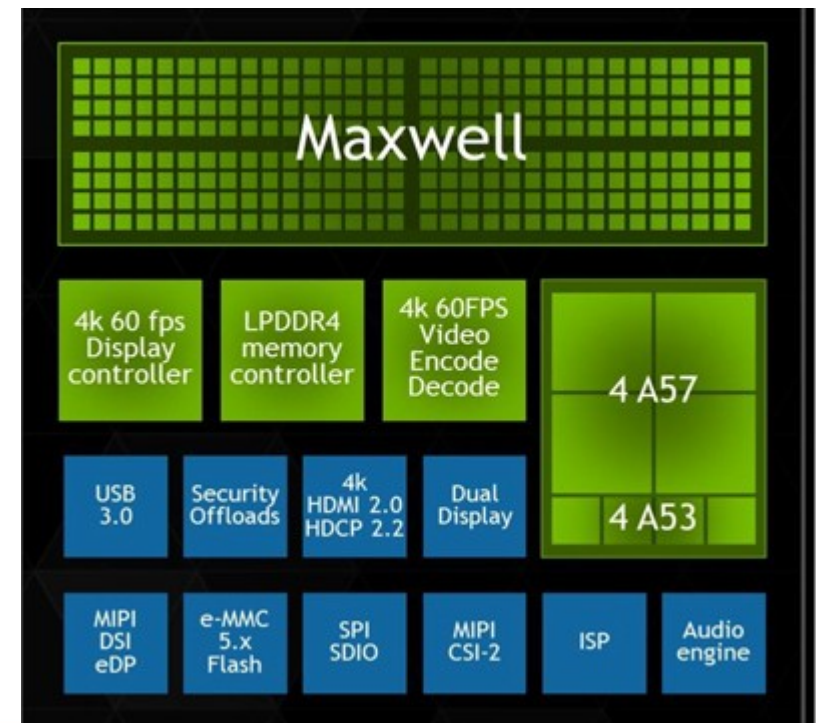
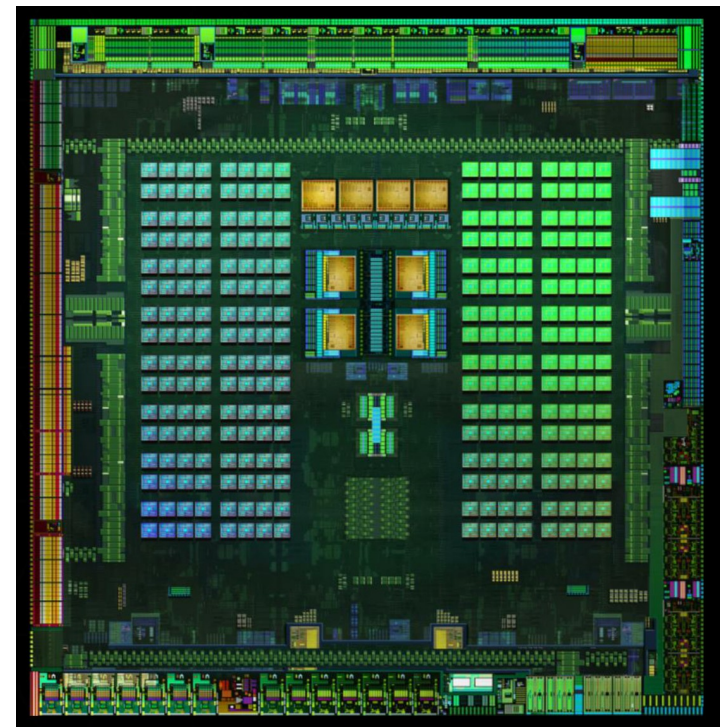
Tablet CPU: ZiiLABS ZMS-40

- Quad-core 1.5GHz
ARM Cortex-A9
- 96 x 32-bit ALUs ~
50GFlops !
- OpenCL 1.1



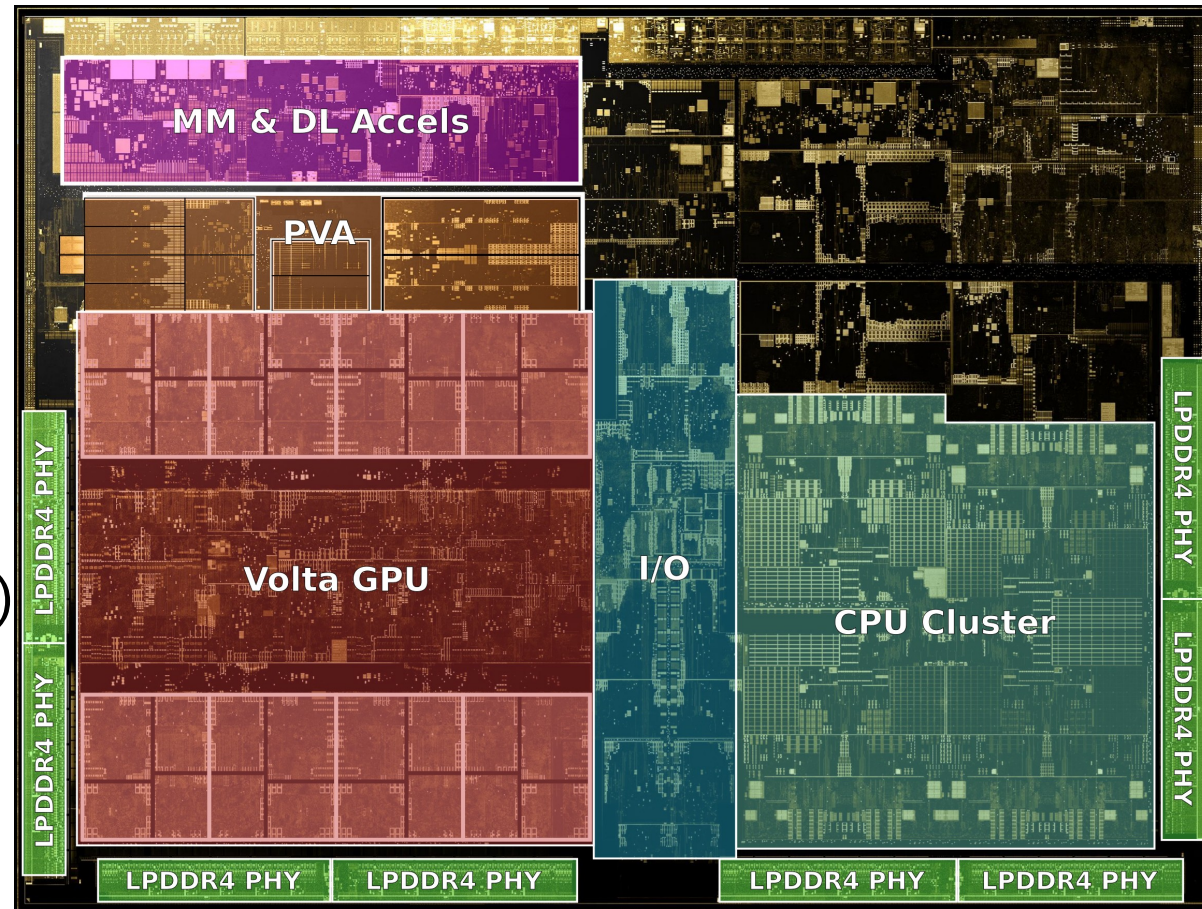
NVIDIA TEGRA X1

- CPU cores:
 - 4 ARM's Cortex-A57
 - (high performance)
 - 4 ARM's Cortex-A53 (high efficiency)
- GPU
 - Maxwell 256 CUDA cores



NVIDIA TEGRA Xavier

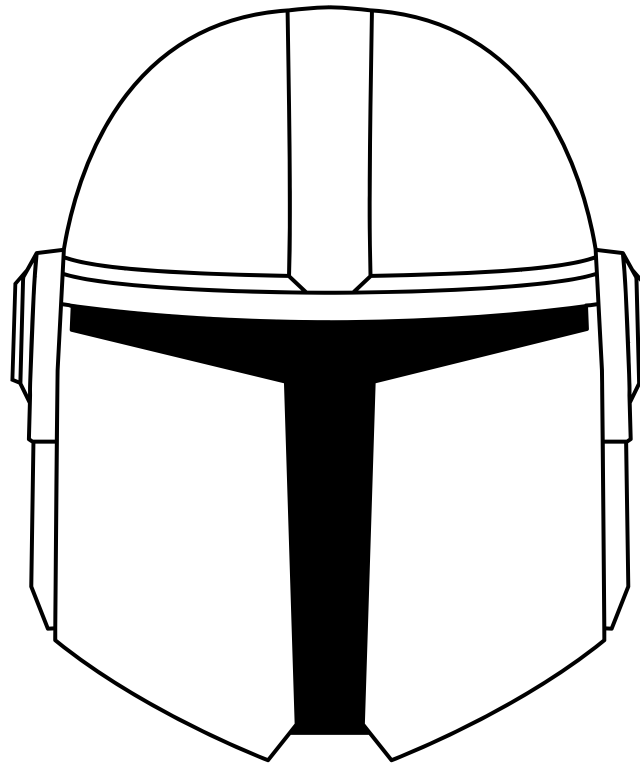
- CPU cores:
 - 4 ARM's Cortex-A57 (high performance)
 - 4 ARM's Cortex-A53 (high efficiency)
- GPU
 - Volta 512 CUDA cores
- DLA (Deep Learning Accelerator)
- PVA (Programmable Vision Accelerator)
- Stereo & Optical Flow Engine



<https://en.wikichip.org/wiki/nvidia/tegra/xavier>

First Conclusion:

- To get access to full power of current and next generation of CPU, parallel programming is the way!



<https://dribbble.com/shots/13435275>

Multi-task (1/4)

- Why ?
 - Many Task to do
 - Longs tasks
 - But autonomous
 - Launch and wait
 - Interruptible ?
 - Reacting needs
 - User Interaction
 - Client / server
 - Many execution units

Multi-task (2/4)

- Why ?

- Many Task to do
 - Longs tasks
 - But autonomous
 - Launch and wait
 - Interruptible ?
- Reacting needs
 - User Interaction
 - Client / server
- Many execution units

Familiar examples:

- While oven warm-up:
 - dress up the table,
 - mix cake ingredients , ...
- Phone rings, ...
- Darling please pick up the phone, I busy with the oven.

Multi-task (3/4)

- How?
 - Save context
 - Manage shared resources
 - Scheduler
 - CPU
 - Critical resources

Multi-task (4/4)

- How?

- Save context
- Manage shared resources
 - Scheduler
 - CPU
 - Critical resources

Familiar examples:

- What was I doing?
- I have only two arms!
 - What can I do first?
- I'm trying to remove the cake from the oven, I can't answer the phone.

Execution context of a task

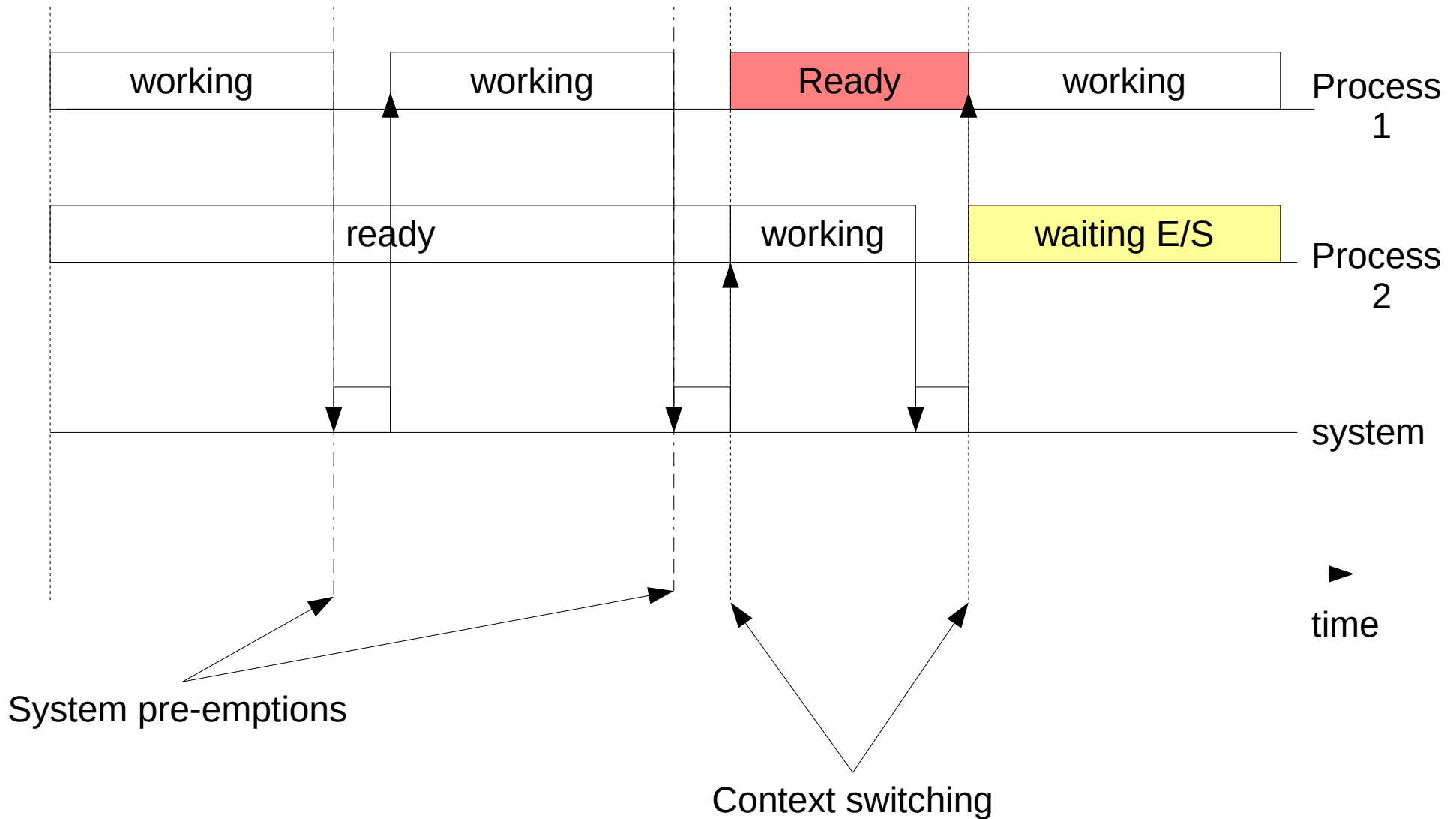
```
int f(int c) {  
    if (c>1)  
        return c*f(c-1);  
    else  
        return 1;  
}
```

- Instructions stack
- Registers
- Instruction counter
- Allocated memory
- Opened files

Share resources: the CPU

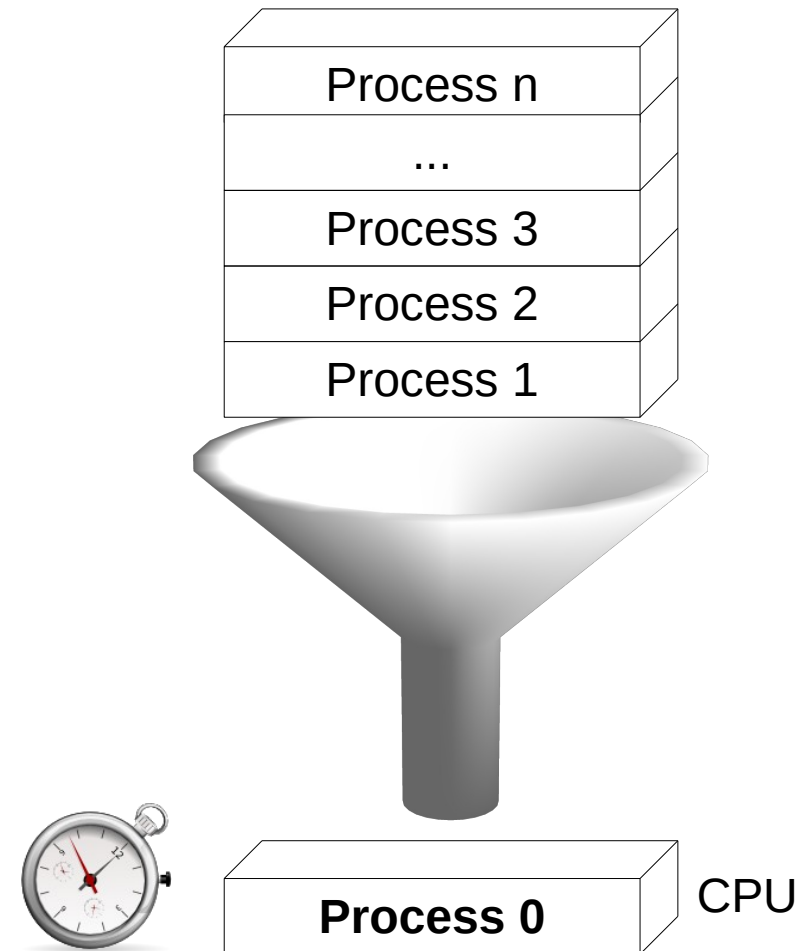
- N process and M CPU
 - $N > M \Rightarrow$ using a scheduler
- Scheduler
 - Part of OS kernel
 - Access I/O
 - Interruptions
 - pre-emptive
 - Share CPU time between all process
 - Scheduling policies

sharing CPU



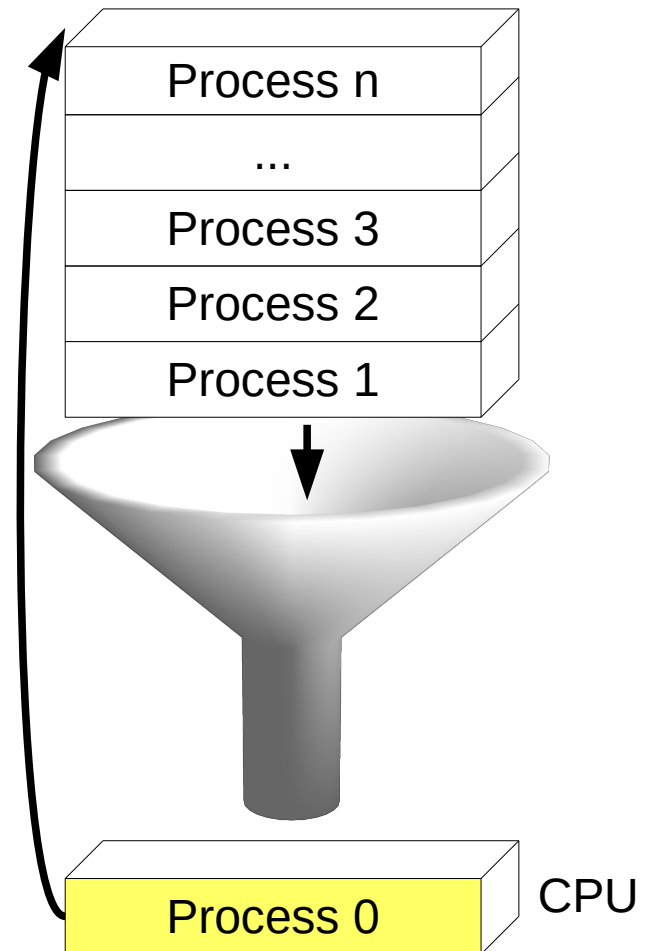
The Round-Robin scheduler (1/4)

- CPU Time sharing policies
 - Several waiting process in a queue
 - One active process consume a *quanta* of time



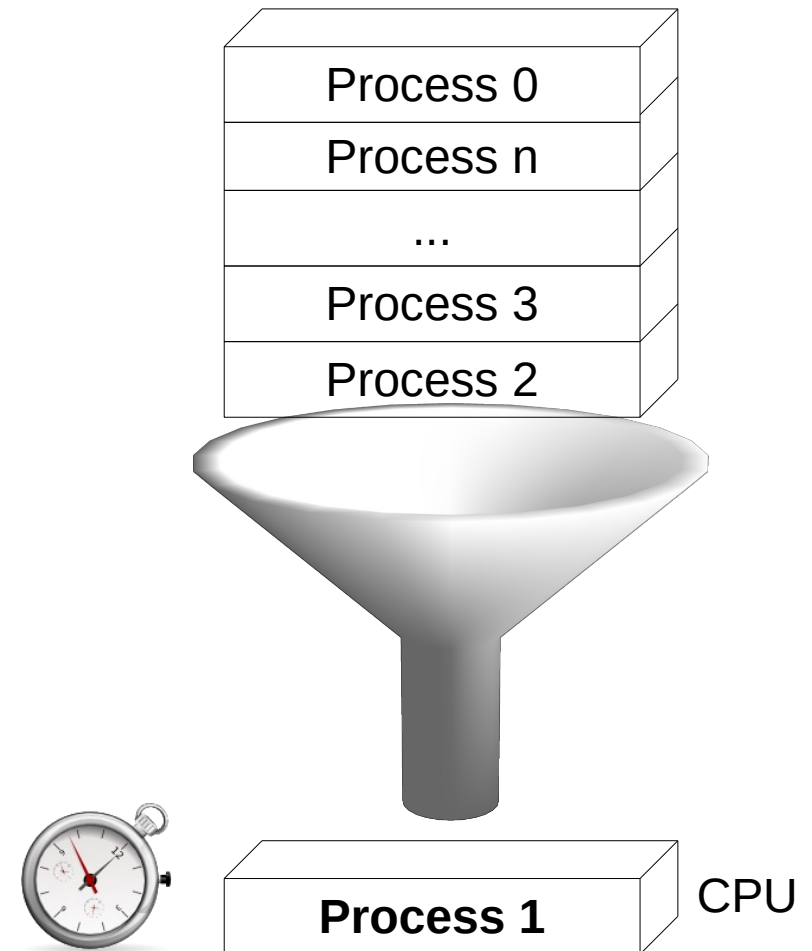
The Round-Robin scheduler (2/4)

- CPU Time sharing policies
 - The current process releases the CPU (ex. I/o wait) or lost it (quanta consumed)
 - The scheduler switches active context



The Round-Robin scheduler (3/4)

- CPU Time sharing policies
 - The first process into the queue becomes active
 - The precedent process is placed at last position on waiting queue



The Round-Robin scheduler (1/4)

- Trade off Reactivity versus Efficiency
 - Quanta time size = Granularity, but
 - short quanta => high context switch overhead
 - Long quanta => poor reactivity

Round-Robin with priorities

- CPU time sharing policies
 - Many queues with different probabilities to get access to CPU
 - Every queue has a different size of quanta of time
 - High priority => short quanta of time
 - Low priority => long quanta of time

Round-Robin with **dynamic** Priorities

- CPU time sharing policies
 - If a process consumes the whole quanta of time assigned in CPU, next time it will decrease his priority
 - If a process releases the CPU for I/O wait before his quanta of time is finished, next time it will increase his priority
 - Implemented on Windows

Completely Fair Scheduler CFS

- CPU time sharing policies
 - Work with a binary tree called “red/black” that allow rapid process sort
 - Sort criteria : difference between ideal cpu time of a process (one cpu per process) and real cpu time consumed by process.
 - Implemented at Linux 2.6.23 version and newer

Heavy and light process (1/3)

- **Heavy** process has independent resources :
 - Instructions stack
 - Registers
 - Instructions counter
 - Allocated memory
 - Opened files
 - Process state

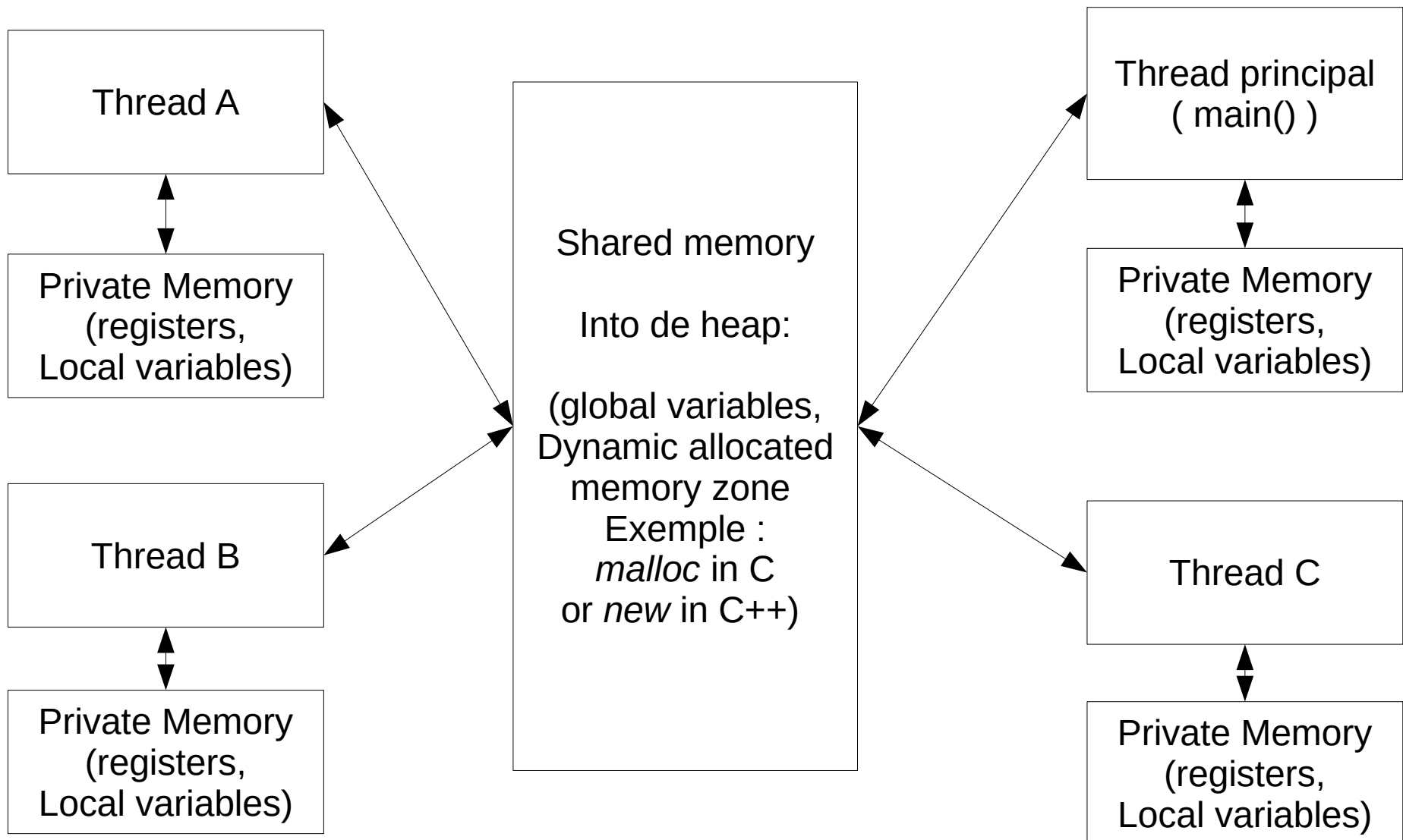
Heavy and light process (2/3)

- **Heavy** process has private resources :
 - Instructions stack
 - Registers
 - Instructions counter
 - Allocated memory
 - Opened files
 - Process state
- Light process has private resources:
 - Instructions stack
 - Registers
 - Instructions counter
- And **shared resources** :
 - Allocated memory
 - Opened files
 - Process state

Heavy and light process (3/3)

- Heavy process
 - Independent execution
 - Slow context switching
 - **extern** inter-process communication
- Light process (*Thread*)
 - fast context switching
 - Attached to father heavy process
 - Inter-process communication throw shared memory
 - possible incoherent read/write in memory

Shared memory



Multi-task: API Pthread

- Light process(threads) management
- Simple
- Norm IEEE POSIX 1003.1c
- Portable
 - Provide as standard on several OS (Mac OS X, Linux, SunOS, etc.)
 - On MS Windows, 2 options:
 - Services for UNIX (SFU) 3.5
 - Open Source POSIX Threads for Win32

Thread Definition

access to API

```
#include<pthread.h>
```

```
void* myThread(  
                void *arg) {  
  
    // example:  
  
    float* a= (float  
*)arg;  
  
    ...  
  
    pthread_exit((void*)code);  
  
    return code;  
}
```

Thread Definition

Function with thread
code

```
#include<pthread.h>
```

```
void* myThread(  
                void *arg){  
  
    // example:  
    float* a= (float  
*)arg;  
  
    ...  
  
    pthread_exit((void*)code);  
    return code;  
}
```

Thread Definition

Generic pointer
passed at thread
creation

```
#include<pthread.h>

void* myThread(
    void *arg) {
    // example:
    float* a= (float
    *)arg;

    ...

    pthread_exit((void*)code);

    return code;
}
```

Thread Definition

It could be converted
to another type (here
to float pointer)

```
#include<pthread.h>

void* myThread(
    void *arg) {

    // example:
    float* a= (float
    *) arg;

    ...

    pthread_exit((void*) code);

    return code;

}
```

Thread Definition

```
#include<pthread.h>

void* myThread(
    void *arg) {

    // example:

    float* a= (float
*)arg;

    ...

    pthread_exit ( (void*) co
de) ;

    return code;

}
```

Thread exit function
(mandatory)



Thread defination

```
#include<pthread.h>
void* myThread(
    void *arg){
    // example:
    float* a= (float
*)arg;
    ...
    pthread_exit( (void*) co
de) ;
    return code;
}
```

It will be return a
value (generic pointer
type)

Thread Creation

Declare a thread id

```
float A[100];
```

```
int main() {
```

```
pthread_t thread;
```

```
int
```

```
rc=pthread_create(
```

```
    & thread,
```

```
    NULL,
```

```
    myThread,
```

```
    (void *)A );
```

```
...
```

```
}
```

Thread Creation

Thread creation and launching

```
float A[100];  
int main() {  
    pthread_t thread;  
  
    int  
    rc=pthread_create(  
        & thread,  
        NULL,  
        myThread,  
        (void *)A );  
  
    ...  
}
```

Thread Creation

Write back thread id

```
float A[100];  
int main() {  
    pthread_t thread;  
    int  
    rc=pthread_create(  
        & thread,  
        NULL,  
        myThread,  
        (void *)A );  
  
    ...  
}
```

Thread Creation

Passing optional
attributes of thread
(here default values)

```
float A[100];  
  
int main() {  
    pthread_t thread;  
  
    int  
    rc=pthread_create(  
        & thread,  
        NULL,  
        myThread,  
        (void *)A );  
  
    ...  
}
```

Thread Creation

Passing thread
function

```
float A[100];  
int main() {  
    pthread_t thread;  
    int  
    rc=pthread_create(  
        & thread,  
        NULL,  
        myThread,  
        (void *)A );  
    ...  
}
```

Thread Creation

```
float A[100];
```

```
int main() {  
    pthread_t thread;  
    int  
    rc=pthread_create(  
        & thread,  
        NULL,  
        myThread,  
        (void *)A );  
    ...  
}
```

Passing a parameter
to thread (here an
array of floats in
shared memory)

Thread end and junction

```
int main() {  
    pthread_t thread;  
    Pthread_create(  
        & thread,  
        ... );
```

Call this function for
waiting the execution
end of a already
lunched thread

```
...  
void *res;  
pthread_join(thread,  
&res);  
...  
}
```

Thread end and junction

Get the thread
returned value (can
be NULL if not useful)

```
int main() {  
    pthread_t thread;  
    Pthread_create(  
        & thread,  
        ... );  
  
    ...  
    void *res;  
    pthread_join(thread,  
        &res);  
    ...  
}
```

Life and death of a thread

Main Thread

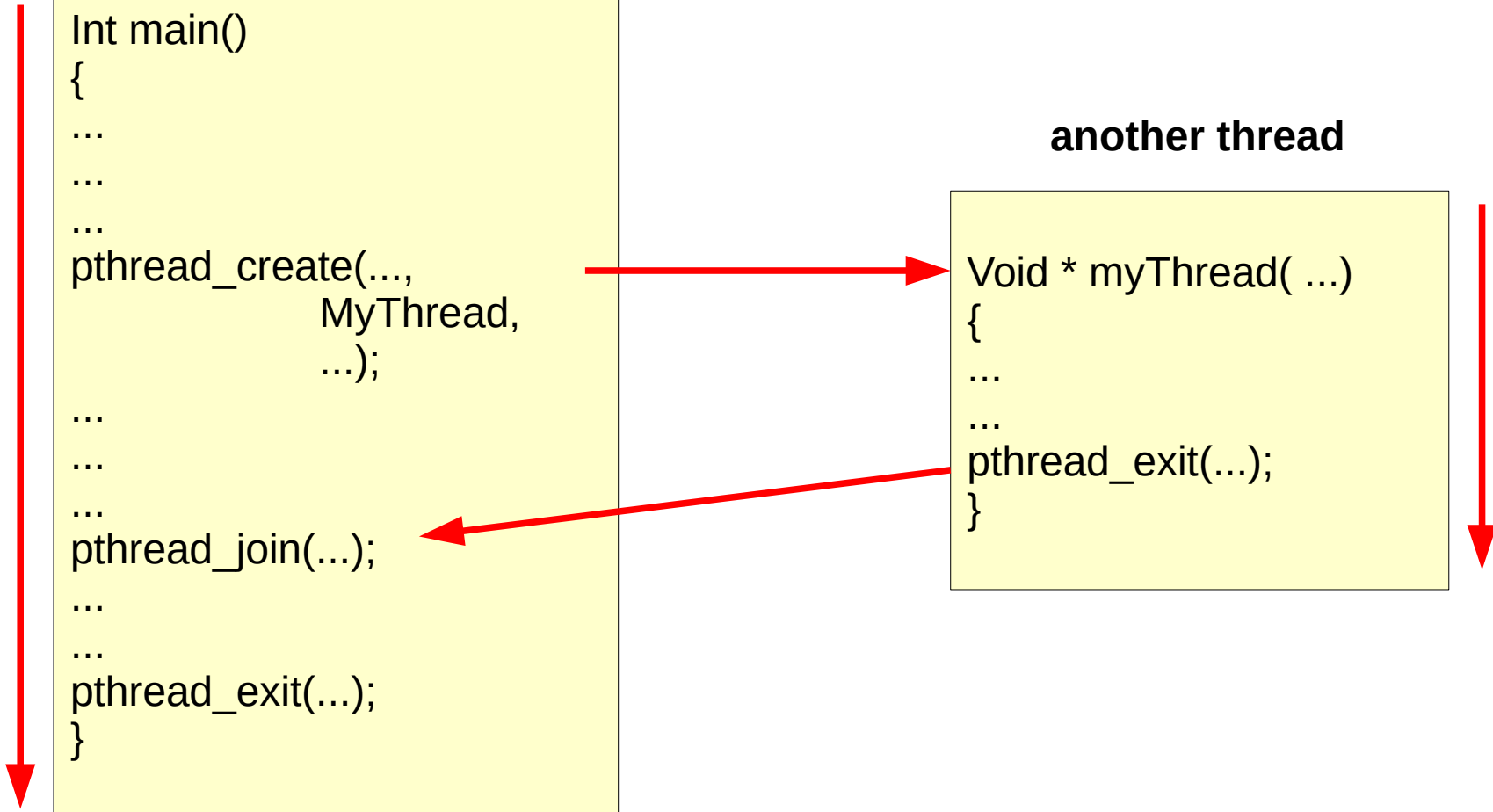
```
Int main()
{
...
...
...
pthread_create(...,
               MyThread,
               ...);

...
...
...
pthread_join(...);

...
...
pthread_exit(...);
}
```


another thread

```
Void * myThread( ...)
{
...
...
pthread_exit(...);
}
```




Mutex

Thread A



```
int X=10;
...
Void * ThreadA( ...)
{
    ...
    int Y=0;
    Y= X * 2;
    X=Y+1;
    ..
    ...
    pthread_exit(...);
}
```

Thread B




```
Void * ThreadB( ...)
{
    ...
    int Z=0;
    Z= X * 2;
    X=Z-1;
    ...
    pthread_exit(...);
}
```

X=10 Z=0 Y=0


Mutex

Thread A



```
int X=10;
...
Void * ThreadA( ...)
{
    ...
    int Y=0;
    Y= X * 2;
    X=Y+1;
    ..
    ...
    pthread_exit(...);
}
```

Thread B




```
Void * ThreadB( ...)
{
    ...
    int Z=0;
    Z= X * 2;
    X=Z-1;
    ...
    pthread_exit(...);
}
```

X=10 Z=0 Y=20


Mutex

Thread A



```
int X=10;
...
Void * ThreadA( ...)
{
    ...
    int Y=0;
    Y= X * 2;
    X=Y+1;
    ..
    ...
    pthread_exit(...);
}
```

Thread B




```
Void * ThreadB( ...)
{
    ...
    int Z=0;
    Z= X * 2;
    X=Z-1;
    ...
    pthread_exit(...);
}
```

X=10 Z=20 Y=20


Mutex

Thread A



```
int X=10;
...
Void * ThreadA( ...)
{
    ...
    int Y=0;
    Y= X * 2;
    X=Y+1;
    ..
    ...
    pthread_exit(...);
}
```

Thread B




```
Void * ThreadB( ...)
{
    ...
    int Z=0;
    Z= X * 2;
    X=Z-1;
    ...
    pthread_exit(...);
}
```

X=19 Z=20 Y=20


Mutex

Thread A



```
int X=10;
...
Void * ThreadA( ...)
{
    ...
    int Y=0;
    Y= X * 2;
    X=Y+1;
    ..
    ...
    pthread_exit(...);
}
```

Thread B



```
Void * ThreadB( ...)
{
    ...
    int Z=0;
    Z= X * 2;
    X=Z-1;
    ...
    pthread_exit(...);
}
```

X=21

Z=20

Y=20

Mutex

- Mutex = Mutual Exclusion
 - Avoid resources conflicts
 - Set a lock
 - If another thread try to set the lock, it will be locked
 - Release the lock
 - another thread can now set the lock

Mutex

- Mutex = Mutual Exclusion
 - Avoid resources conflicts
 - Set a lock
 - If another thread try to set the lock, it will be locked
 - Release the lock
 - another thread can now set the lock
- current example:
 - Toilets!
 - Only one person inside
 - You only lock the door if you are inside
 - If door is locked, other people wait (outside)
 - Release de lock and leave allows someone to lock the toilets


Mutex

- Declaration `pthread_mutex_t a_mutex;`
- Initialisation `pthread_mutex_init(&a_mutex, NULL);`
- Set a lock `pthread_mutex_lock(&a_mutex);`
- Release `pthread_mutex_unlock(&a_mutex);`
- Non-blocking test

```
if( pthread_mutex_trylock(&a_mutex) !=
    EBUSY )
    /*OK*/
else
    /* do something else */
```


Mutex

Thread A



```
int X=10;
...
Void * ThreadB( ...)
{
...
int Y=0;
pthread_mutex_lock(&a_mutex);
Y= X * 2;
X=Y+1;
pthread_mutex_unlock(&a_mutex);
..
...
pthread_exit(...);
}
```


Thread B



```
Void * ThreadA( ...)
{
...
int Z=0;
pthread_mutex_lock(&a_mutex);
Z= X * 2;
X=Z-1;
pthread_mutex_unlock(&a_mutex);
...
pthread_exit(...);
}
```


Mutex and condition

Worker Thread



```
int workingToDo=0;
...
Void * WorkerThread( ...)
{
...
While(1)
{
pthread_mutex_lock(&a_mutex);
If(workingToDo>0)
do(workingToDo);
pthread_mutex_unlock(&a_mutex);
...
}
..
...
pthread_exit(...);
}
```

Boss Thread



```
Void * BossThread( ...)
{
...
pthread_mutex_lock(&a_mutex);
workingToDo+= someWork;
pthread_mutex_unlock(&a_mutex);
...
...
pthread_mutex_lock(&a_mutex);
workingToDo+= someOtherWork;
pthread_mutex_unlock(&a_mutex);
...
...
pthread_exit(...);
}
```

Mutex and condition

Worker Thread

```
int workingToDo=0;
...
Void * WorkerThread( ...)
{
...
While(1)
{
pthread_mutex_lock(&a_mutex);
If(workingToDo>0)
do(workingToDo);
pthread_mutex_unlock(&a_mutex);
...
}
..
...
pthread_exit(...);
}
```

100% CPU !!!

Boss Thread

```
Void * BossThread( ...)
{
...
pthread_mutex_lock(&a_mutex);
workingToDo+= someWork;
pthread_mutex_unlock(&a_mutex);
...
...
pthread_mutex_lock(&a_mutex);
workingToDo+= someOtherWork;
pthread_mutex_unlock(&a_mutex);
...
...
pthread_exit(...);
}
```

Mutex and condition

- Declaration

```
pthread_cond_t a_cond;
```

- Initialisation

```
pthread_cond_init (&a_cond, NULL);
```

- Destruction

```
pthread_cond_destroy (&a_cond);
```

- Usage

- Start waiting

```
&a_mutex);
```

```
pthread_cond_wait (&a_cond,
```

- Release the mutex

- Send the signal

```
pthread_cond_signal (&a_cond);
```

- Waiting thread lock the mutex


- Broadcast the signal

```
(&a_cond);
```

```
pthread_cond_broadcast
```

Mutex and condition

Worker Thread



```
int workingToDo=0;
...
Void * WorkerThread( ...)
{
    ...
    While(1)
    {
        pthread_mutex_lock(&a_mutex);
        If(0==workingToDo)
            pthread_cond_wait (&a_cond,
                               &a_mutex);

        do(workingToDo);
        pthread_mutex_unlock(&a_mutex);
        ...
    }
    ...
    pthread_exit(...);
}
```

Boss Thread

```
Void * BossThread( ...)
{
    ...
    pthread_mutex_lock(&a_mutex);
    workingToDo+= someWork;
    pthread_cond_signal (&a_cond);
    pthread_mutex_unlock(&a_mutex);
    ...
    ...
    pthread_mutex_lock(&a_mutex);
    workingToDo+= someOtherWork;
    pthread_cond_signal (&a_cond);
    pthread_mutex_unlock(&a_mutex);
    ...
    ...
    pthread_exit(...);
}
```

Mutex and condition

Worker Thread

```
int workingToDo=0;
...
Void * WorkerThread( ...)
{
    ...
    While(1)
    {
        pthread_mutex_lock(&a_mutex);
        If(0==workingToDo)
            pthread_cond_wait(&a_cond,
                             &a_mutex);

        do(workingToDo);
        pthread_mutex_unlock(&a_mutex);
        ...
    }
    ...
    pthread_exit(...);
}
```

Boss Thread

```
Void * BossThread( ...)
{
    ...
    pthread_mutex_lock(&a_mutex);
    workingToDo+= someWork;
    pthread_cond_signal(&a_cond);
    pthread_mutex_unlock(&a_mutex);
    ...
    ...
    pthread_mutex_lock(&a_mutex);
    workingToDo+= someOtherWork;
    pthread_cond_signal(&a_cond);
    pthread_mutex_unlock(&a_mutex);
    ...
    ...
    pthread_exit(...);
}
```

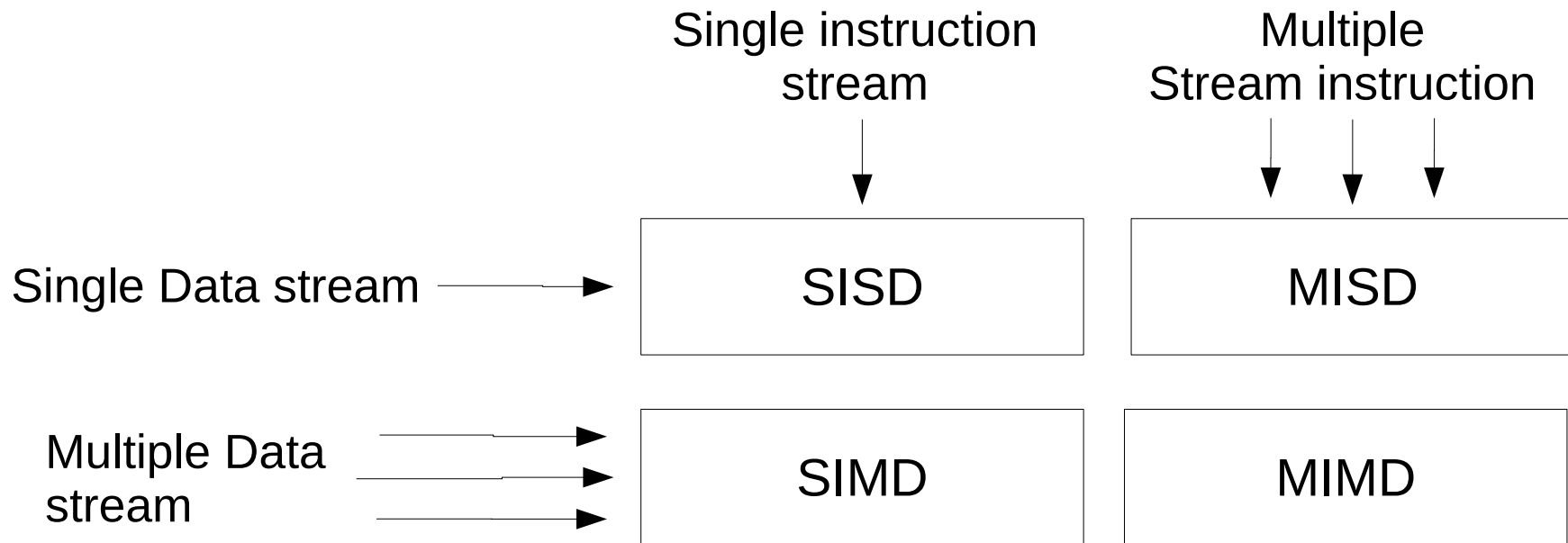
References

- Architecture CPUs [http://fr.wikipedia.org/wiki/Pipeline_\(informatique\)](http://fr.wikipedia.org/wiki/Pipeline_(informatique))
- Revue of some Intel CPU
<http://www.clubic.com/article-175286-2-intel-core-i7-nehalem-core-2-duo.html>
- Completely Fair Scheduler http://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- Introduction au pthreads <https://computing.llnl.gov/tutorials/pthreads/>
- Open Source POSIX Threads for Win32 <http://sourceware.org/pthreads-win32/>
- Download Microsoft Windows SFU 3.5
<http://www.microsoft.com/downloads/details.aspx?FamilyID=896C9688-601B-44F1-81A4-02878FF11778&displaylang=en>

Parallel Machine

- Definition: many process units work simultaneously

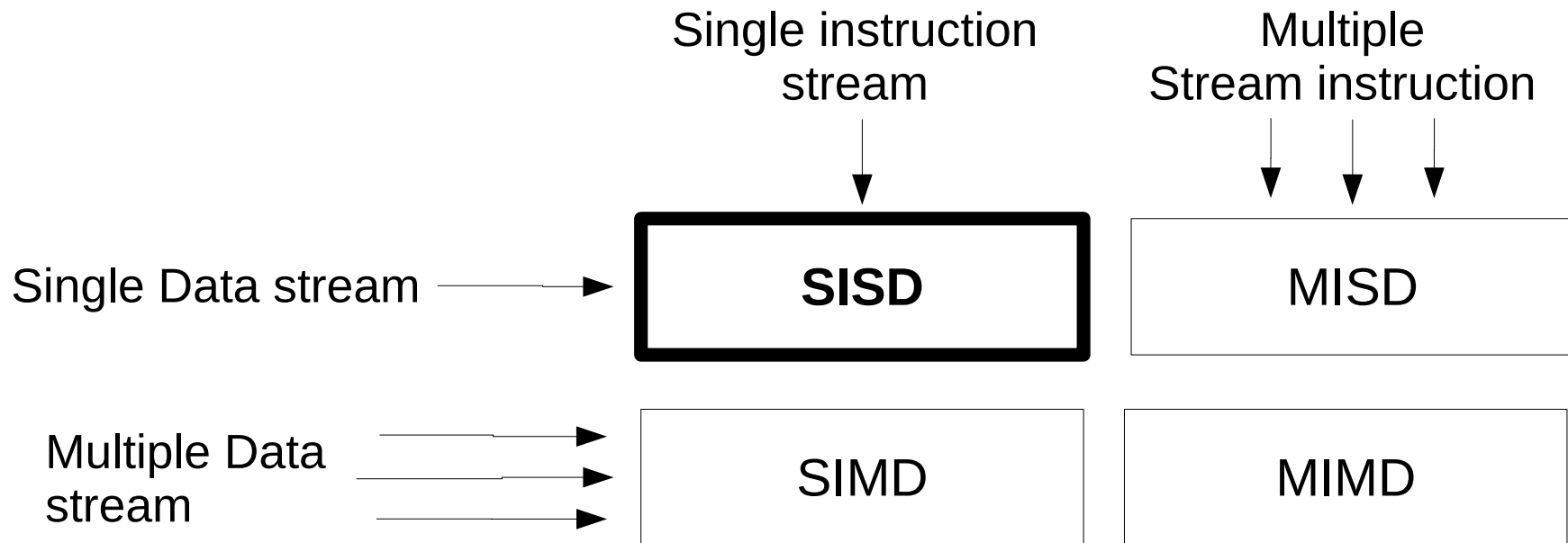
Classification of parallel machines [Flynn 1966]



S: single
M: Multiple
I: instruction
D: data

Classification of parallel machines

[Flynn 1966]

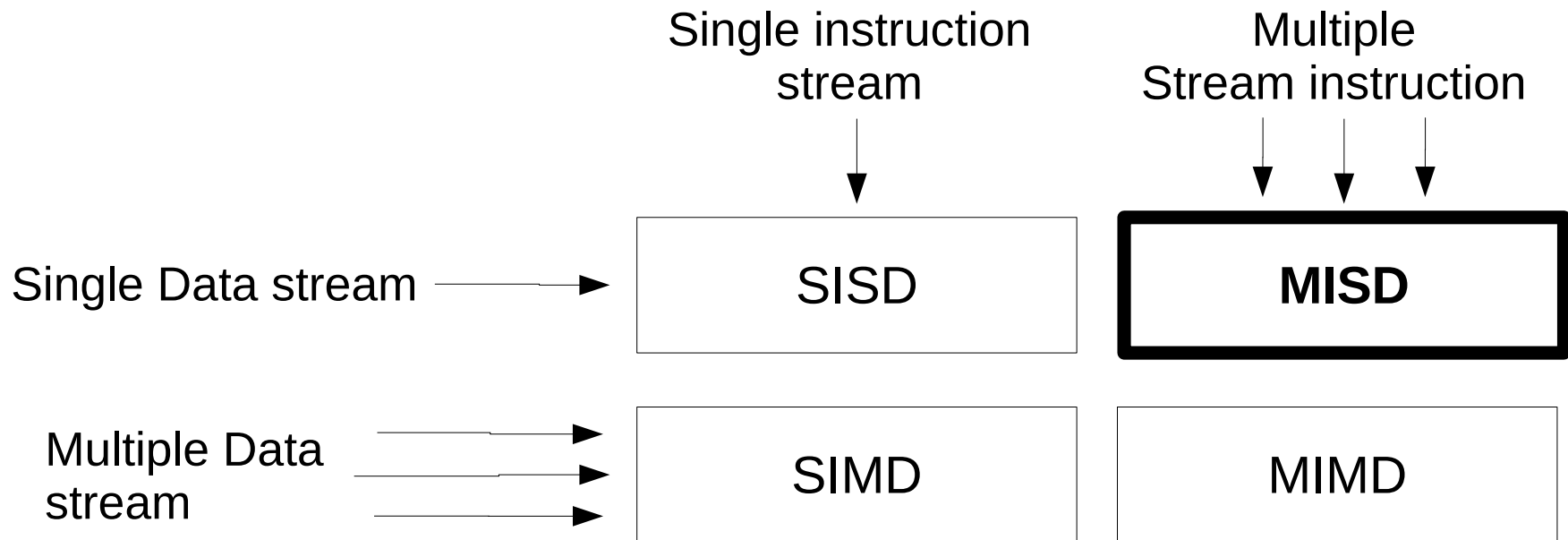


S: single
M: Multiple
I: instruction
D: data

Example: Classic sequential processor

Classification of parallel machines

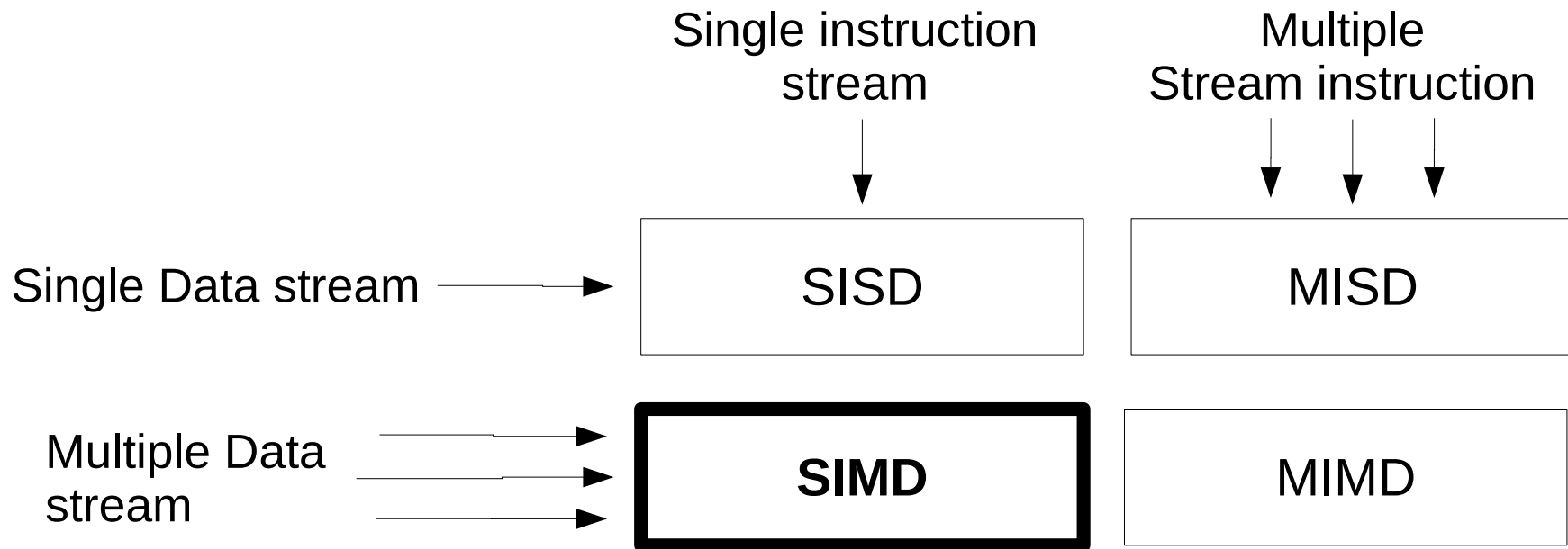
[Flynn 1966]



S: single
M: Multiple
I: instruction
D: data

Example: Pipeline

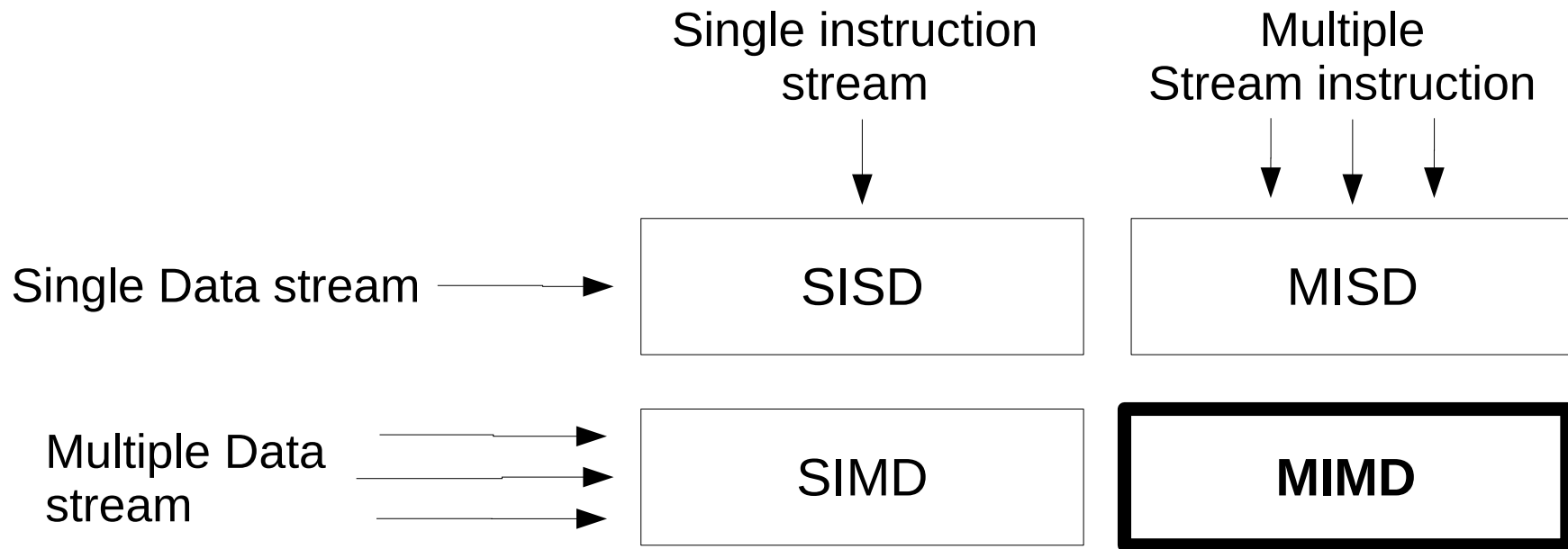
Classification of parallel machines [Flynn 1966]



S: single
M: Multiple
I: instruction
D: data

Example: SSE, AVX , NEON(arm)

Classification of parallel machines [Flynn 1966]



S: single
M: Multiple
I: instruction
D: data

Example: Cluster PC MPI, Xeon Phi

SIMD: Single Instruction Multiple Data

- Study case : Intel SSE Instructions set
- SSE (Streaming SIMD Extensions)
 - Some versions SSE, SSE2, SSE3, SSE4, SSE4a, SSE4.2
 - **SSE2** available on almost all current CPU x86 from Intel, AMD or VIA.
 - Instructions SIMD on 128 bits-wide registers
 - 4 x 32 bits float or integers
 - 2 x 64 bits doubles or long integers
 - 8 x 16 bits short integers

SSE2

- Using in C / C++: `#include<xmmintrin.h>`
- Vectors:
 - 4 float 32 bits: `__m128`
 - 4 int 32 bits : `__m128i`
 - 2 float 64 bits : `__m128d`
- 16 bit Data alignment
 - Under OS 64 bits it is the default alignment
 - Under OS 32 bits must replace `malloc(...)` with :
 - `memalign(16, ...)` on GCC
 - `_align_malloc(...,16)` on MSVC

SSE with 32 bits floats in C/C++

- Declaration

```
__m128 a, c, b={0.0f,1.0f,  
                2.0f,3.0f};
```

- Set contents

- 1 float
- 4 floats

```
a = _mm_set1_ps(5.01f);
```

```
a = _mm_set_ps(0.0f,1.0f,  
               2.0f,3.0f);
```

- From an array

```
float X[1000]={...}; // ~= 250 __m128  
a = _mm_load_ps(X+4*i); // i in [0,250[
```

- Get contents

- To a float
 - To an array

```
float x0=((float*)&a)[0];
```

```
_mm_store_ps(X+4*i,b); // i in [0,250[
```

SSE 32 bits floats in C/C++

- Addition

```
__m128 c = _mm_add_ps(a, b);
```

- Subtraction

```
__m128 c = _mm_sub_ps(a, b);
```

- Product

```
__m128 c = _mm_mul_ps(a, b);
```

- Division

```
__m128 c = _mm_div_ps(a, b);
```

- Square root

```
__m128 c = _mm_sqrt_ps(a);
```

{		{		{
X1,		X2,		X1 op X2,
Y1,		Y2,		Y1 op Y2,
Z1,	op	Z2,	=	Z1 op Z2,
W1		W2		W1 op W2
}		}		}

AVX

- Set of instructions like SSE but working on 256 bits-wide registers
- In C/C++
 - `#include <immintrin.h>`
 - Addition `__m256 c = _mm256_add_ps(a, b);`
 - ...

SSE 32 bits floats in C/C++

- In MS Visual Studio 2008 and Intel C++ Compiler (icc)

```
#include<fvec.h>
```

- Vector Classes

```
F32Vec4 vecA, vecB(0.1f,0.2f,  
0.3f,0.4f);
```

- Operators:

```
F32Vec4 vecC = vecA + vecB;
```

- In g++

- Operators:

```
_m128 c = a + b;
```

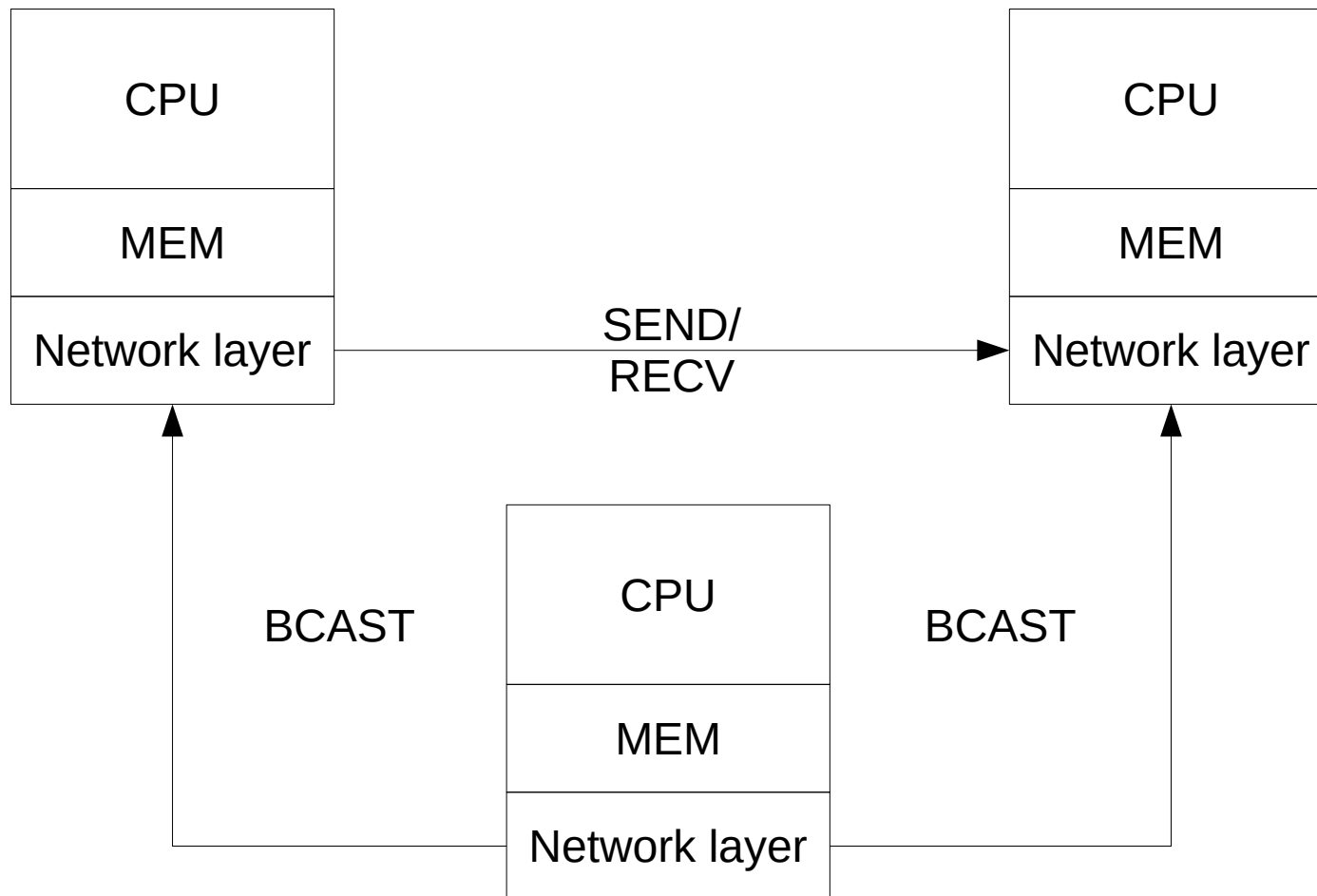
SIMD criticisms

- SIMD take lot of place in silicon, but useful on few real life application => less SIMD and more cores will be better (AMD choice / Intel choice)
 - Linus Torvalds: *"I hope AVX-512 dies a painful death, and that Intel starts fixing real problems instead of trying to create magic instructions to then create benchmarks that they can look good on,"*
<https://www.zdnet.com/article/linus-torvalds-i-hope-intels-avx-512-dies-a-painful-death/>
- There is another lightweight (in terms of footprint and energy consuming) to deal with SIMD in RISC-V vector simple instruction set that the X86 design.
 - RISC-V designers David Patterson and Andrew Waterman: *"The IA-32 instruction set has grown from 80 to around 1400 instructions since 1978, largely fueled by SIMD"*
<https://medium.com/swlh/risc-v-vector-instructions-vs-arm-and-x86-simd-8c9b17963a31>

MIMD : Multiple Instructions Multiple Data

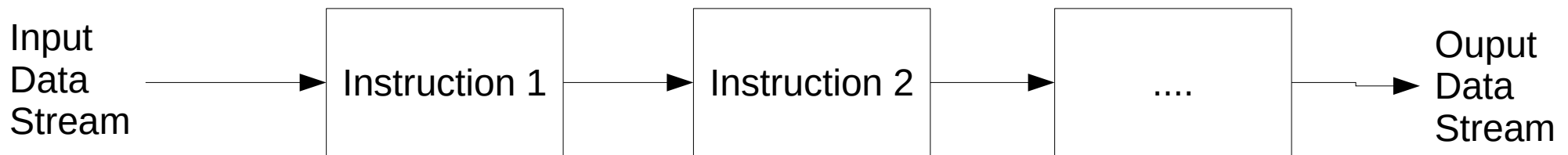
- Asynchronous Processors
- Shared Memory
 - SMP: Symmetric *Multi-processors*
 - Example : Pthread
- Spread Memory
 - MPI *Message Passing Interface*

MPI

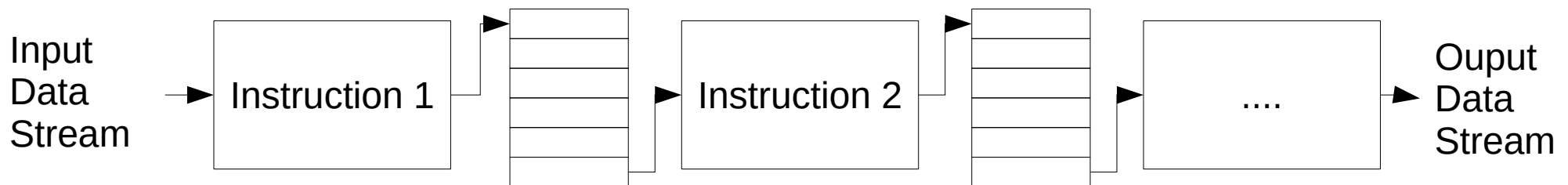


MISD : Multiple Instruction Single Data

- Filters and pipelines
 - Synchronous



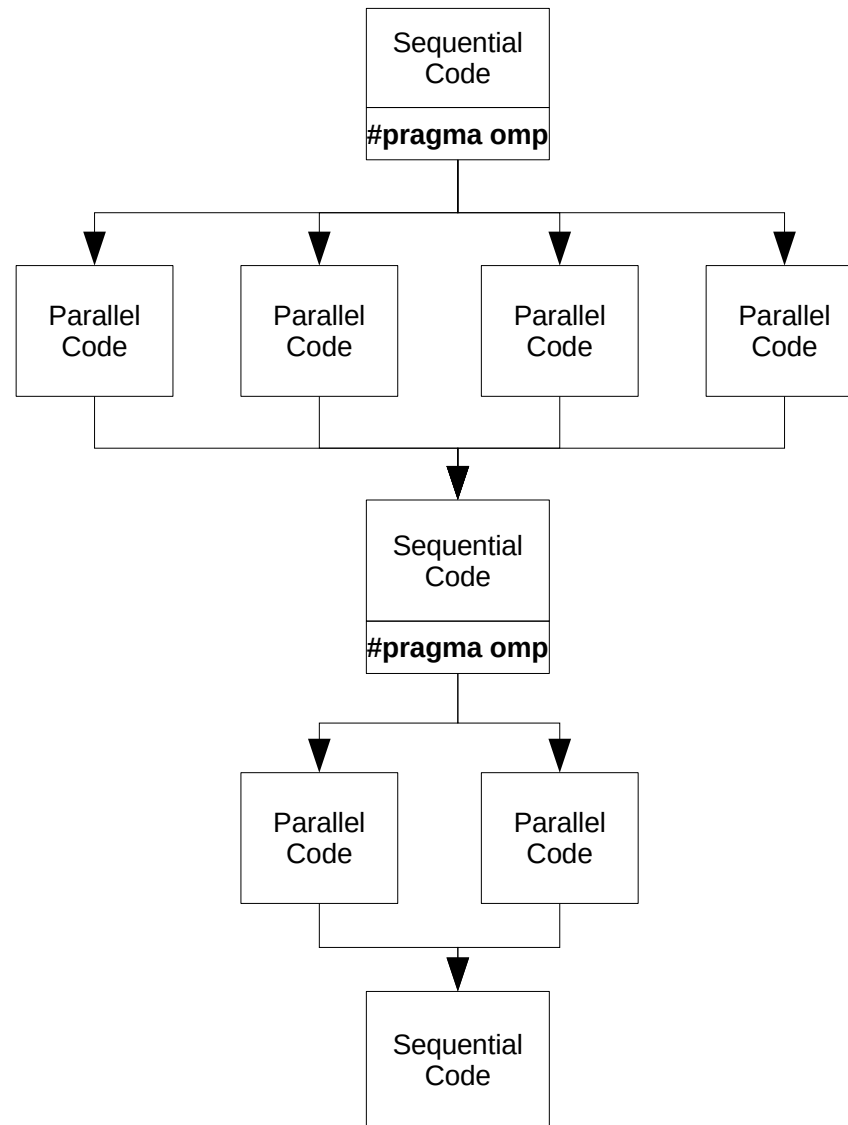
- Asynchronous (with queues)



OpenMP (Open Multi-Processing)

- At Compilation and at runtime
- Add parallel directives to the code
 - Define parallel zones inside a sequential code.
 - In C/C++ by adding compilation directives :
 - `#pragma omp ...`
 - Algorithms
 - Sharing Instructions
 - Sharing memory
 - Reduction

Partial Parallelism with OMP



Sample

```
void a1(int n, float *a, float *b)
{
    int i;

    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Sample

```
void a1(int n, float *a, float *b)
{
    int i;
    #pragma omp parallel for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Sample

```
void a1(int n, float *a, float *b)
{
    int i;
    #pragma omp parallel for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Parallel region

- Code sharing
- Sharing **a** and **b**
- Variable **i** implicitly private

Synchronisation

```
#include <math.h>
void a8(int n, int m, float *a, float *b, float
*y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;
        #pragma omp for
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```

Synchronisation

```
#include <math.h>
void a8(int n, int m, float *a, float *b, float
*y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;
        #pragma omp for
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```

Implicit
synchronisation
barrier

Implicit
synchronisation
barrier

Synchronisation

```
#include <math.h>
void a8(int n, int m, float *a, float *b, float
*y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;
        #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```

Reduction

```
void a31_1(float *x, int *y, int n)
{
    int i, b;
    float a;
    a = 0.0;
    b = 0;

    for (i=0; i<n; i++) {
        a += x[i];
        b ^= y[i];
    }
}
```

Reduction

```
void a31_1(float *x, int *y, int n)
{
    int i, b;
    float a;
    a = 0.0;
    b = 0;
    #pragma omp parallel for private(i) shared(x, y, n) \
        reduction(+:a) reduction(^:b)
    for (i=0; i<n; i++) {
        a += x[i];
        b ^= y[i];
    }
}
```

Define shared variables
and associated operators
in reduction

Reduction

```
void a31_1(float *x, int *y, int n)
{
    int i, b;
    float a;
    a = 0.0;
    b = 0;
    #pragma omp parallel for private(i)
    reduction(+:a) reduction(^:b)
    for (i=0; i<n; i++) {
        a += x[i];
        b ^= y[i];
    }
}
```

Define others
shared variables

shared(x, y, n)

References

- Parallel machines classification http://fr.wikipedia.org/wiki/Taxinomie_de_Flynn
- Parallel architectures http://dio.obspm.fr/PDF/archi_mpi.pdf
- SSE Instruction set http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
- Memory alignment and performance
http://www.gamasutra.com/view/feature/3942/data_alignment_part_1.php?print=1
- Aligned memory allocation with gcc
http://www.gnu.org/s/libc/manual/html_node/Aligned-Memory-Blocks.html
- Specifications OpenMP 2.5 (supported by MS Visual Studio 2008)
<http://www.openmp.org/mp-documents/spec25.pdf>
- Software optimization resources <http://www.agner.org/optimize/>
- Introduction to AVX instructions
<https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>
- intrinsics Intel reference <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>
- ARM Neon Intrinsics
<https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics>

Tutorial material (Only Enssat students) :

- https://gitlab.inria.fr/sed-rennes/formations/tutorials_parallel_computing

First (only one time) :

- ssh-keygen
- ssh-copy-id barn-e-01.enssat.fr

for every connection :

- ssh -XC barn-e-01.enssat.fr

Overview

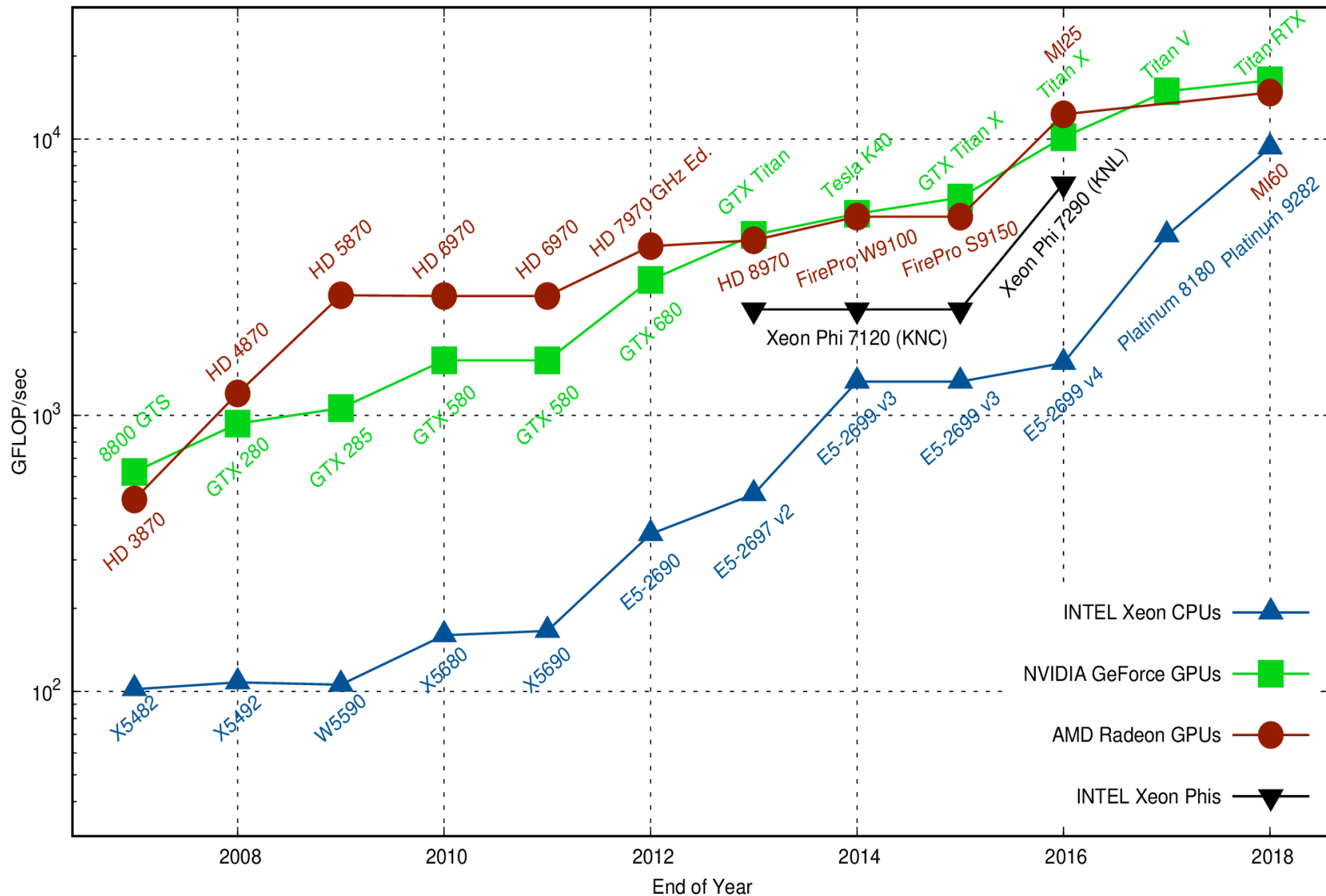
- Part 1 : Basics on parallel machine
- **Part 2 : GPU and CUDA**
- Part 3 : OpenCL & portable tools
- Part 4 : Performance

Part 2: GPU and CUDA

- GPUs architectures
- GP-GPU languages
- Programming CUDA
- PyCuda

Why GPU?

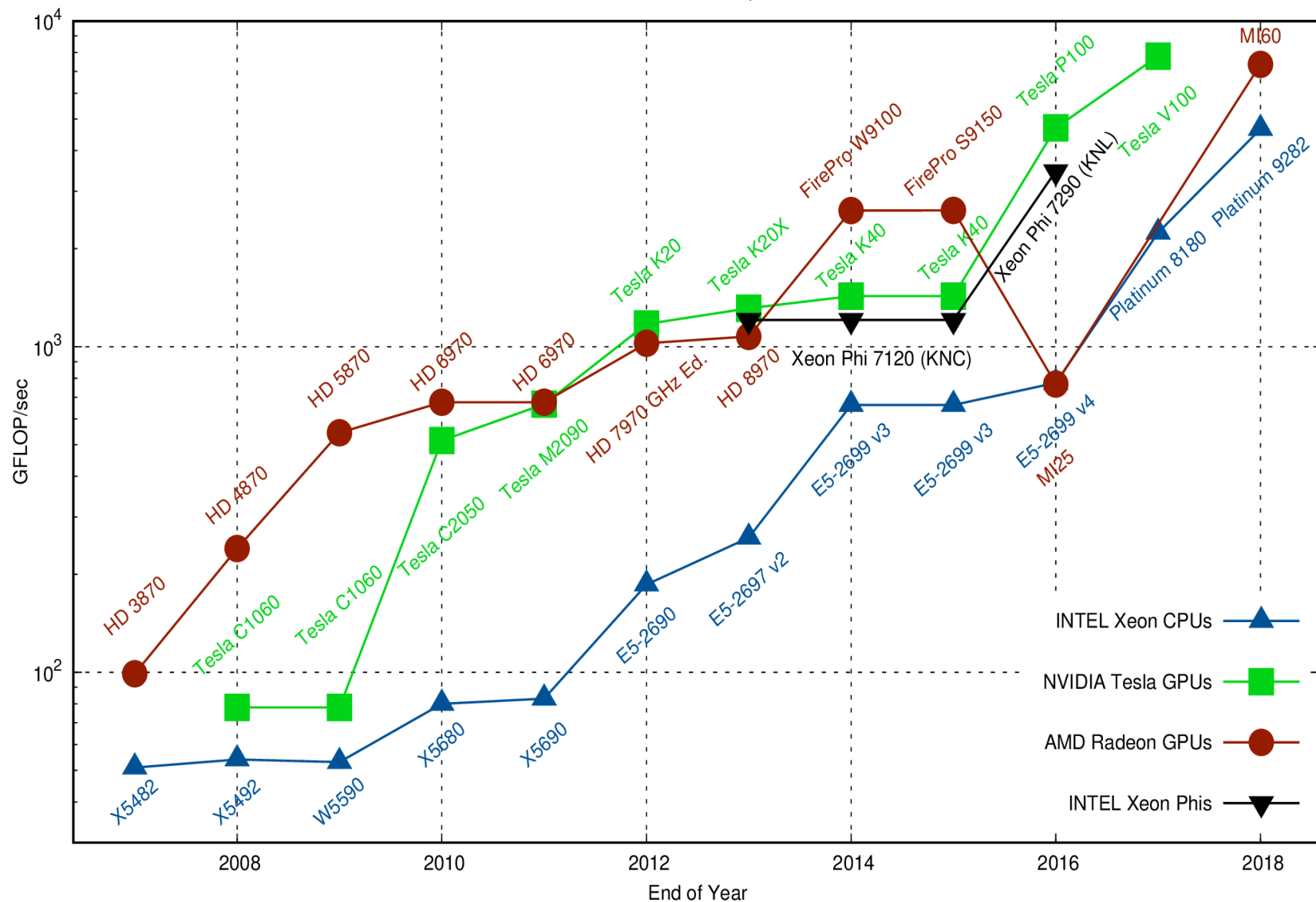
Theoretical Peak Performance, Single Precision



Source : <http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

Why GPU?

Theoretical Peak Performance, Double Precision



Source : <http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

Why GPGPU?

- **Massively parallel** units
- Very large RAM memory bandwidth
- 32 bits and 64 floating point arithmetic (IEEE 754, after NVIDIA G200)
- High and low Level generic programming languages:
 - OpenCL & CUDA / PTX on NVIDIA
 - OpenCL / CAL on AMD
 - C++ X11 parallel for (experimental on cuda >7.5)
- Mass production => Low cost
- Energy efficiency : up to 44 Gflops / watts

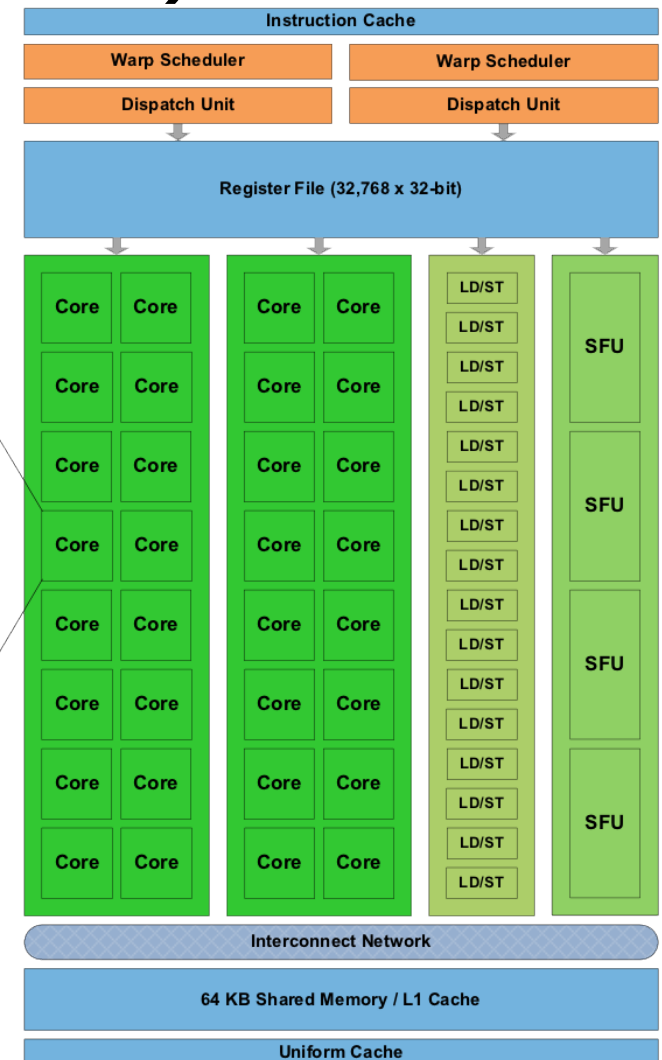
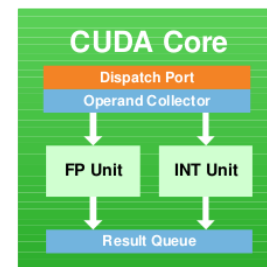
Architecture NVIDIA GTX 580 (GF110)

- 16 = 4 x 4 Streaming Multiprocessors (SM)
 - 512 scalar ALUs = 4 x 4 x 32
 - 64 SFU = 4 x 4 x 4
 - 1544 Mhz=> **1,581 TFLOPS** (32 bits)
- Memory
 - Up to 3 Go RAM (GDDR5)
 - Bus width 384 bits
 - 4008 Mhz=> bandwidth **192,4 GB/s**



Streaming Multiprocessor (SM)

- 32 Cuda Cores
 - 1 Cuda Core (1 scalar)
 - 1 Float IEEE 754-2008 (32 bits) unit or
 - 1 Integer unit (32 bits)
 - 1 Conditional branch unit
- 16 LD/ST (Load/Store units) 64 bits
- 4 SFU (Special Function Units) :
- Sin, cos , sqrt, rsqrt
- 32x2 (FMA) FLOPS 32 bits
- 16x2 (FMA) FLOPS 64 bits
 - 64 Ko shared memory or
 - Cache L1 (48/16 or 16/48)
- 32K registers 32 bits.
- Dual Warp Scheduler (up to 2 warps in //)



Fermi Streaming Multiprocessor (SM)

Conditional execution on SM

- 32 Cuda Cores share same instruction
- How to execute this code ?

```
if( X[i] > 0)
    Y[i]=f(i);
else:
    Y[i]=g(i);
```

X	-1	-2	0	2	3	-1	-5	...	4	6	2	-5	-6
Y	g	g	g	f	f	g	g	...	f	f	f	g	g

Expected result

Conditional execution on SM

- 32 Cuda Cores share same instruction
- How to execute this code ?

```
if( X[i] > 0 )  
    Y[i]=f(i);  
else:  
    Y[i]=g(i);
```

X	-1	-2	0	2	3	-1	-5	...	4	6	2	-5	-6
MASK	0	0	0	1	1	0	0	...	1	1	1	0	0

Conditional execution on SM

- 32 Cuda Cores share same instruction
- How to execute this code ?

```
if( X[i] > 0)
    Y[i]=f(i);
else:
    Y[i]=g(i);
```

X	-1	-2	0	2	3	-1	-5	...	4	6	2	-5	-6
MASK	0	0	0	1	1	0	0	...	1	1	1	0	0
exec	f	f	f	f	f	f	f	...	f	f	f	f	f
Y				f	f			...	f	f	f		

Conditional execution on SM

- 32 Cuda Cores share same instruction
- How to execute this code ?

```
if( X[i] > 0)
    Y[i]=f(i);
else:
    Y[i]=g(i);
```

X	-1	-2	0	2	3	-1	-5	...	4	6	2	-5	-6
MASK	0	0	0	1	1	0	0	...	1	1	1	0	0
exec	g	g	g	g	g	g	g	...	g	g	g	g	g

Conditional execution on SM

- 32 Cuda Cores share same instruction
- How to execute this code ?

```
if( X[i] > 0)
    Y[i]=f(i);
else:
    Y[i]=g(i);
```

X	-1	-2	0	2	3	-1	-5	...	4	6	2	-5	-6
MASK	0	0	0	1	1	0	0	...	1	1	1	0	0
exec	g	g	g	g	g	g	g	...	g	g	g	g	g
Y	g	g	g			g	g	...				g	g

Conditional execution on SM

- 32 Cuda Cores share same instruction
- How to execute this code ?

```
if( X[i] > 0)
    Y[i]=f(i);
else:
    Y[i]=g(i);
```

X	-1	-2	0	2	3	-1	-5	...	4	6	2	-5	-6
Y	g	g	g	f	f	g	g	...	f	f	f	g	g

Conclusion: conditional execution reduce parallelism by serialisation and waste some power

Conditional execution on SM

- 32 Cuda Cores share same instruction
- How to execute this code ?

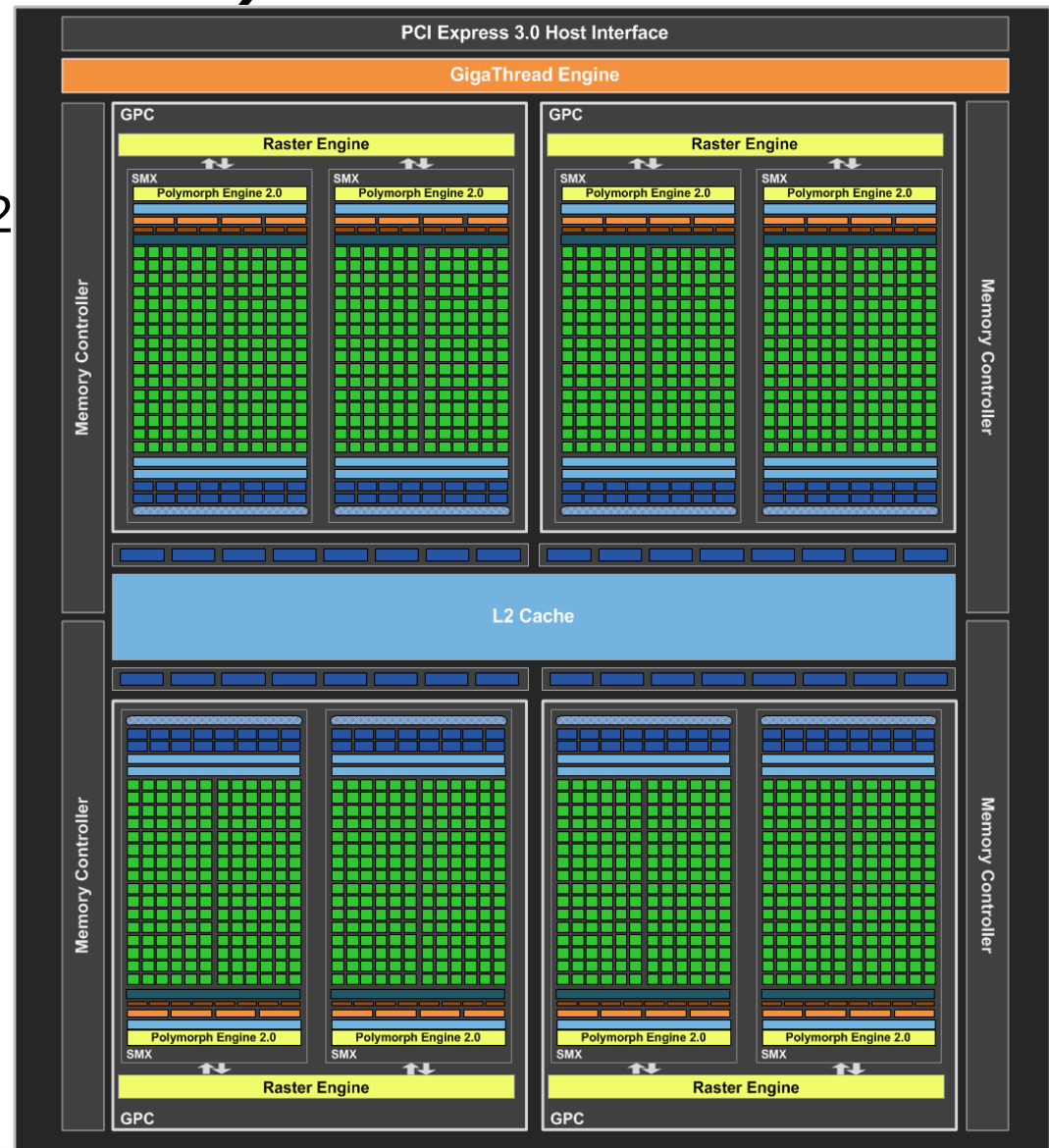
```
if( X[i] > 0 )  
    Y[i]=f(i);  
else:  
    Y[i]=g(i);
```

X	-1	-2	-2	-2	-5	-1	-5	...	-6	0	-2	-5	-1
MASK	0	0	0	0	0	0	0	...	0	0	0	0	0
Y	g	g	g	g	g	g	g	g	g	g	g	g	g

But if ALL mask values are uniform :
Direct result g() execution in Y without exec of f()

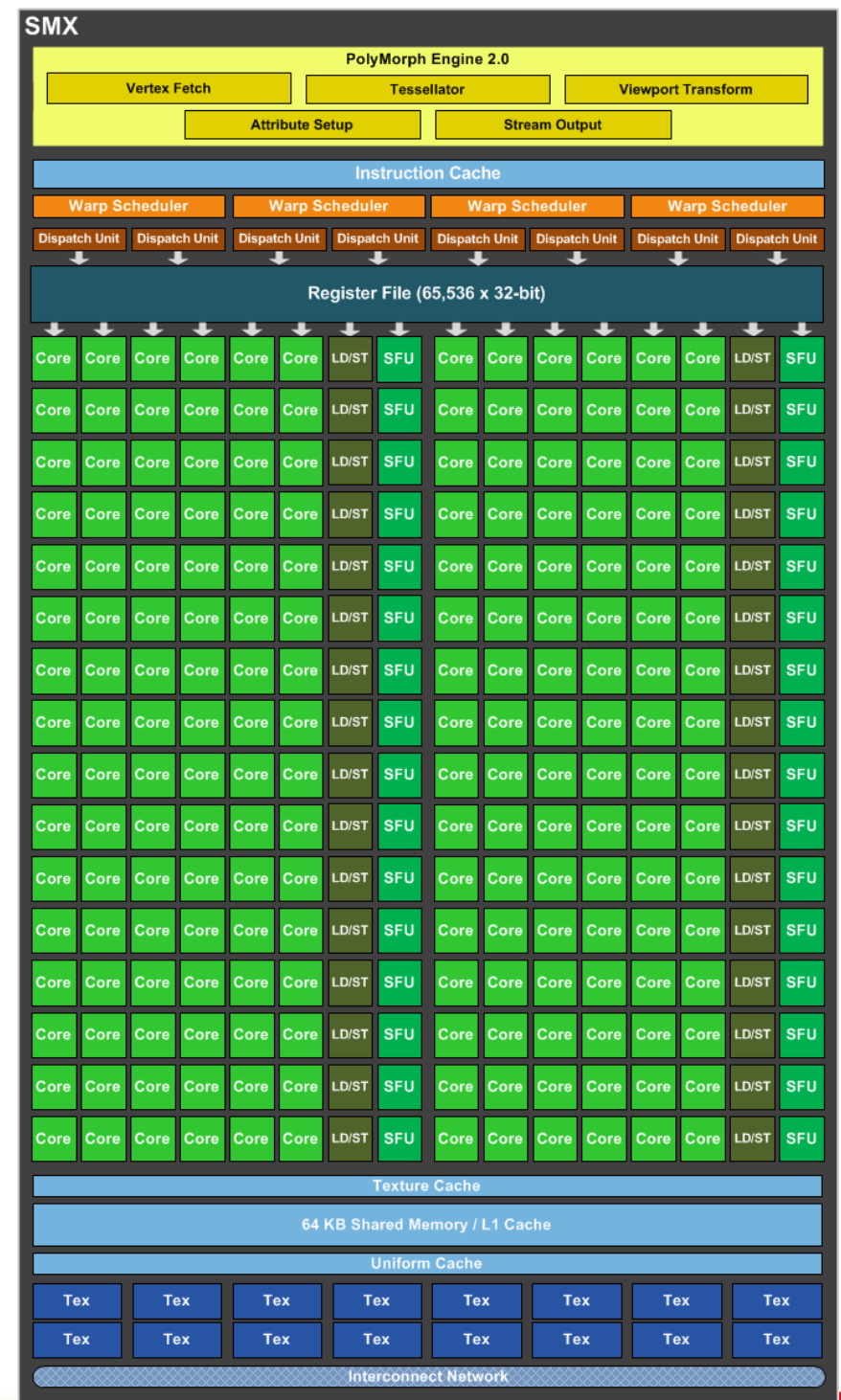
Architecture NVIDIA GTX 680 (GK104)

- 8 = 4 x 2 new Streaming Multiprocessors (SMX)
 - 1536 scalar ALUs = 4 x 2 x 192
 - 256 SFU = 4 x 2 x 64
 - 1006 Mhz \Rightarrow **3,09 TFLOPS** (32 bits)
- Memory
 - Up to 2 Go RAM (GDDR5)
 - Bus mémoire 256 bits
 - 6008 Mhz \Rightarrow Bandwidth **192,26 GB/s**



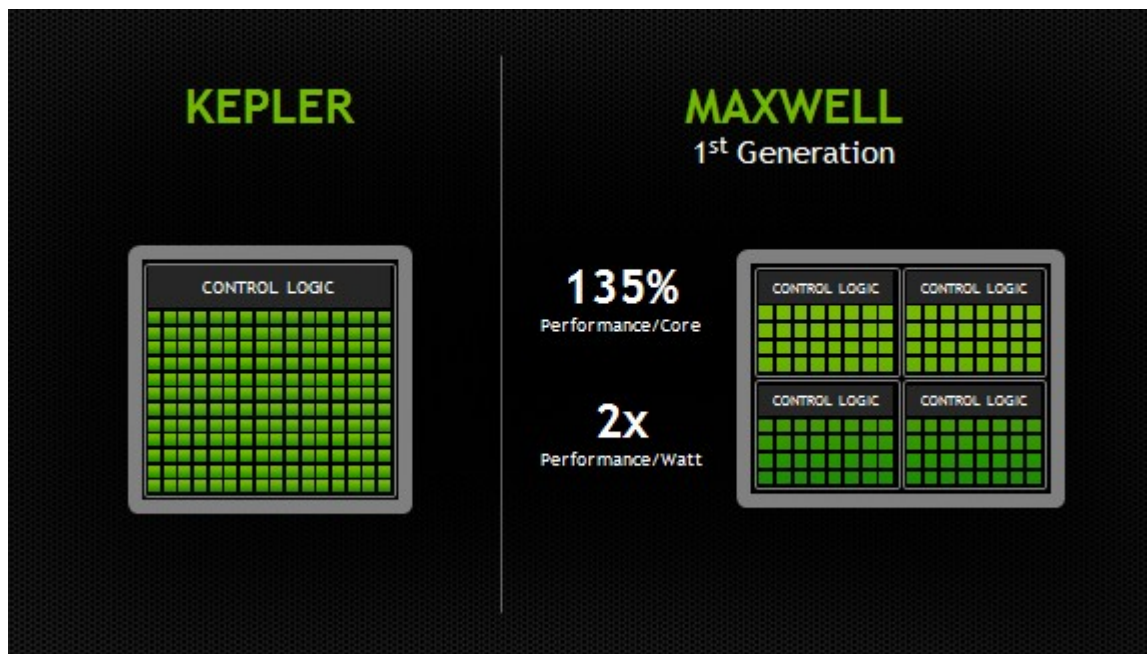
New Streaming Multiprocessor

- **192** Cuda cores
 - 1 Cuda core (1 scalar)
 - 1 Float IEEE 754-2008 (32 bits) or
 - 1 Integer unit (32 bits)
 - 1 Conditional branch unit
- **32** LD/ST (Load/Store units) 64 bits
- **32** SFU (Special Function Units) :
- Sin, cos , sqrt, rsqrt
- **192** (FMA) FLOPS 32 bits
- ??? (FMA) FLOPS 64 bits
- 64 Ko share memory or Cache L1
- 64K registers 32 bits.
- 4 Warp Scheduler et 8 Dispatch Units



Evolution on Maxwell Architecture

- Reduce the number of instructions issued per clock cycle
- Improve :
 - Energy efficiency
 - Cores density
 - Occupancy for existing codes (x2 number of active threads)
 - L2 cache size => 2Mb



Architecture NVIDIA GTX 980 (GM204)

- 4 GPC (Graphics Processing Clusters) x 4 SMM blocks of 4 x 32 cores
 - 2048 scalar ALUs
 - 512 SFU
 - 1126 Mhz

=> **4,612** TFLOPS (32 bits)
- Memory
 - Up to 4 Go RAM
 - Bus width 256 bits
 - 1753 Mhz x4 (GDDR5)

=> Bandwidth **224,3** GB/s



Architecture NVIDIA GTX Titan X (GM200-400)

- 6 GPC / 1075 Mhz
- RAM 12 Go / 384 bits / 7010 Mhz
=> 6,4 TFLOPS / 334,5 GB/S



Architecture NVIDIA PASCAL

GTX 1080 (GP104-400)

- CUDA units
 - 2560 scalar ALUs
 - 320 SFU
 - 1607 MHz

=> **8,9** TFLOPS (32 bits)
- Memory
 - 8 Go RAM (GDDR5X)
 - Bus width 256 bits
 - 1251 Mhz

=> Bandwidth **320** GB/s

Titan Xp (GP102-450)

- CUDA units
 - 3840 scalar ALUs
 - 480 SFU
 - 1405 MHz

=> **12** TFLOPS (32 bits)
- Memory
 - 12 Go RAM (GDDR5X)
 - Bus width 384 bits
 - 1425 Mhz

=> Bandwidth **547** GB/s

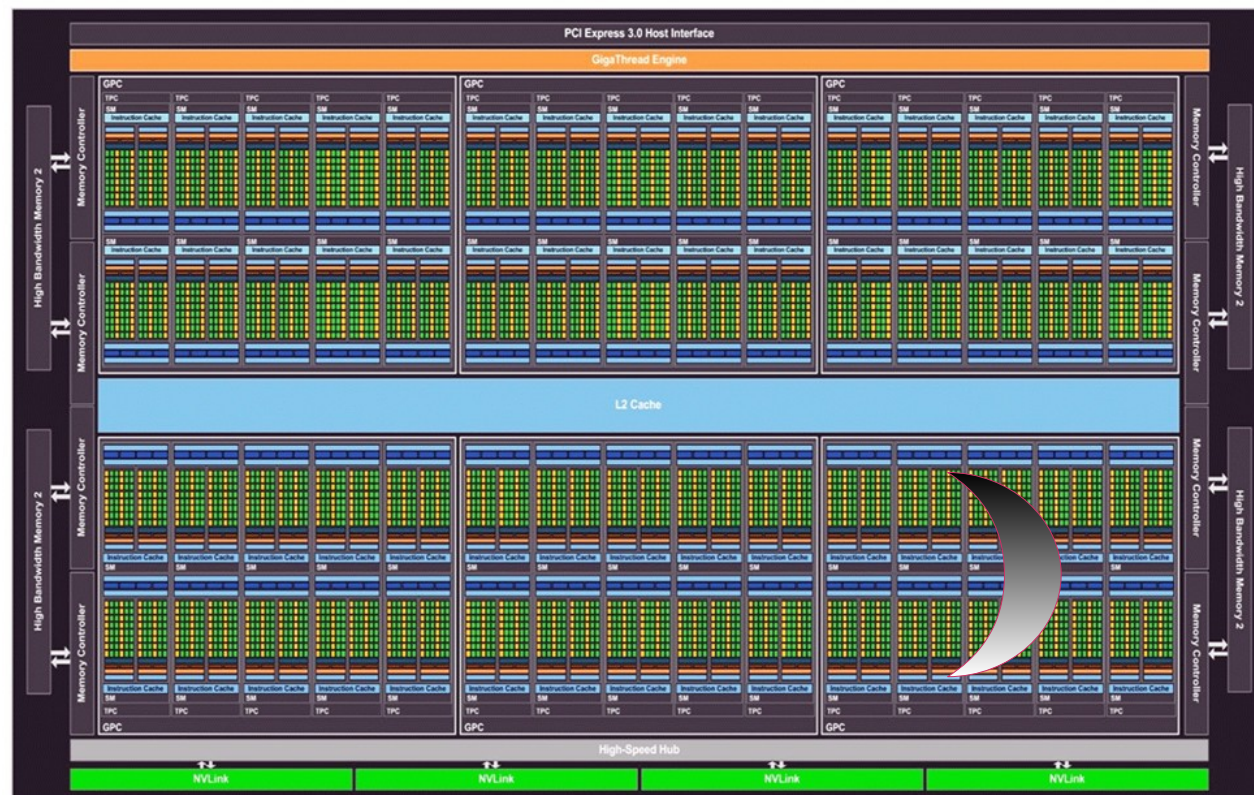
Architecture NVIDIA PASCAL

Tesla P100 PCIe 16GB (GP100)

- CUDA units
 - 3584 scalar ALUs (FP32)
 - 1792 scalar ALUs (FP64)
 - 400 SFU
 - 1607 MHz

=> **10.6** TFLOPS (32 bits)
21.2 TFLOPS (16 bits)
5.3 TFLOPs (64 bits)
- Memory
 - 16 Go RAM (HBM2)
 - Bus width 4096 bits
 - 704 Mhz

=> Bandwidth **721** GB/s



Architecture NVIDIA Turing

RTX TITAN (TU102-400-A1)

- CUDA units
 - 4608 scalar ALUs
 - 1152 SFU
 - 1350 MHz

=> **12,44** TFLOPS (32 bits)
- 576 Tensor cores
 - 64 mixed FMA (FP16 mult + FP32 add)
- Memory
 - 24 Go RAM (GDDR6)
 - Bus width 384 bits
 - 1251 Mhz

=> Bandwidth **672** GB/s

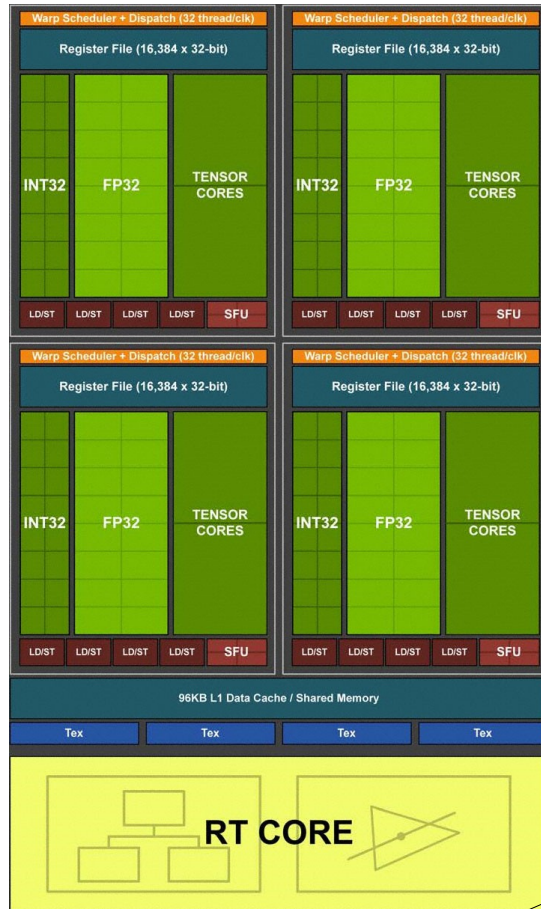
RTX 2080 Ti (TU102)

- CUDA units
 - 4352 scalar ALUs
 - 1088 SFU
 - 1350 MHz

=> **11,75** TFLOPS (32 bits)
- 576 Tensor cores
 - 64 mixed FMA (FP16 mult + FP32 add)
- Memory
 - 11 Go RAM (GDDR6)
 - Bus width 352 bits
 - 1425 Mhz

=> Bandwidth **616** GB/s

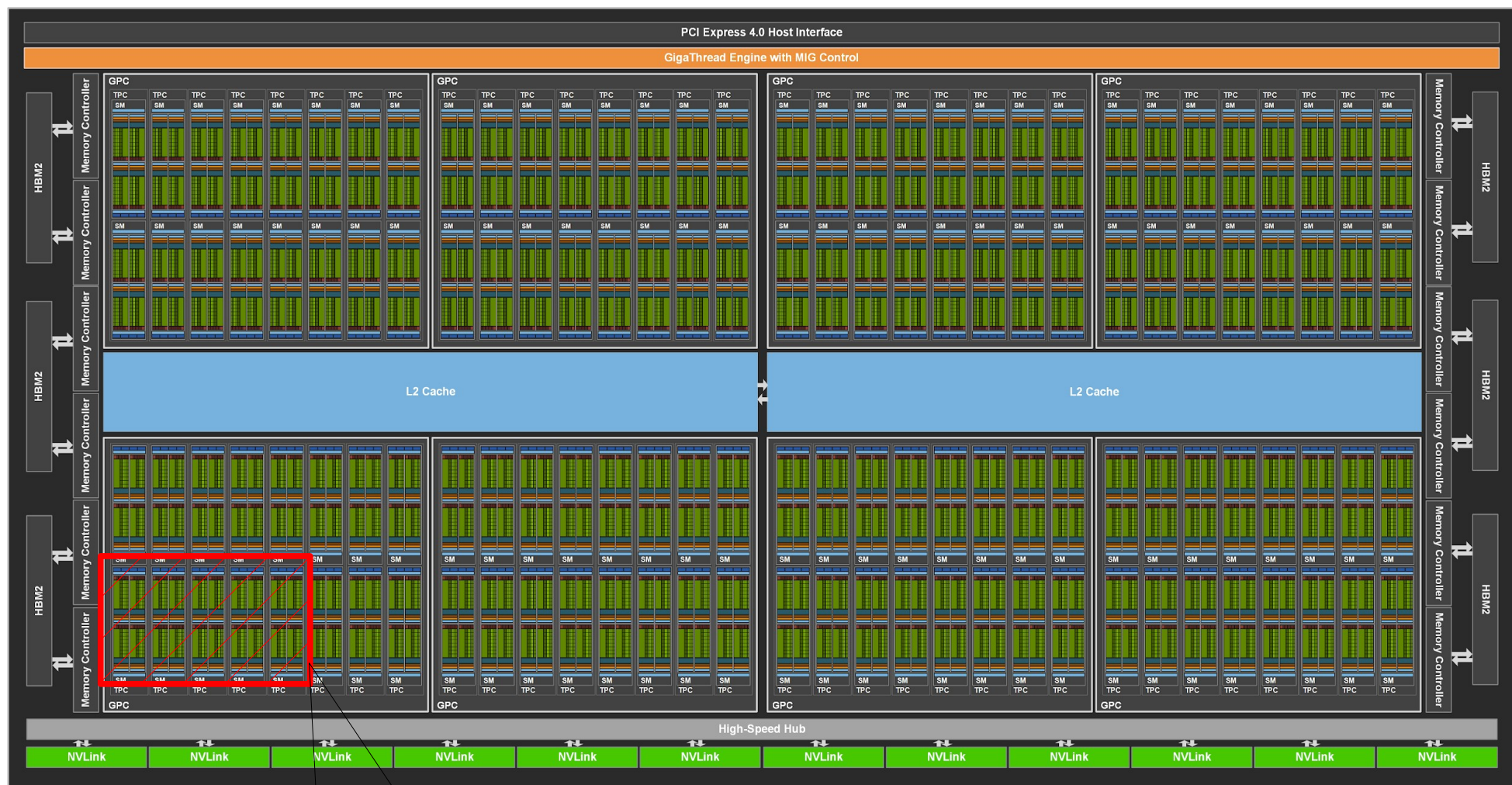
Architecture NVIDIA Turing Titan RTX / RTX 2080 Ti



Disable in GTX 2080 Ti



Architecture NVIDIA Ampere GA100



Disable in A100

Architecture NVIDIA Ampere A100

A100 (GA100)

- 108 SM
 - CUDA units
 - 6912 scalar ALUs FP32
 - 1728 SFU
 - 1410 MHz
- => **19,5 TFLOPS (32 bits)**
- 432 Tensor cores
 - 256 mixed FMA (FP16 mul + FP32 add)
 - Memory
 - 40 Go RAM (HBM2)
 - Bus width 5120 bits
 - 2.4Gbit/s
- => Bandwidth **1555 GB/s**



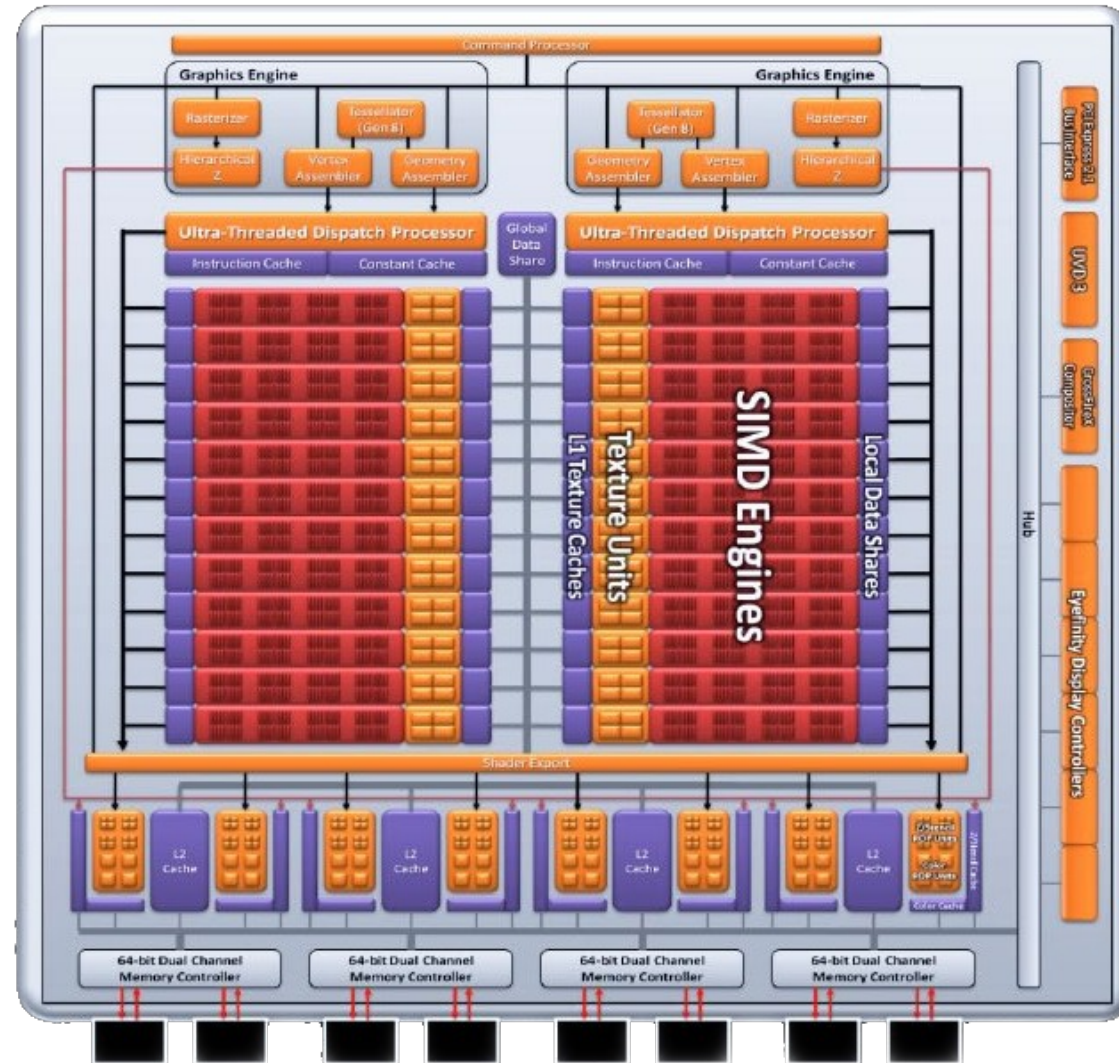
Architecture AMD Radeon HD 6970 (Cayman)

- 24 SIMD cores
 - 16 VLIW4 :
 - [3 ALU ou 1 SFU] + 1 ALU
 - 1536 ALUs = $24 \times 16 \times 4$
 - 880 Mhz

=> **2,703 TFLOPS**

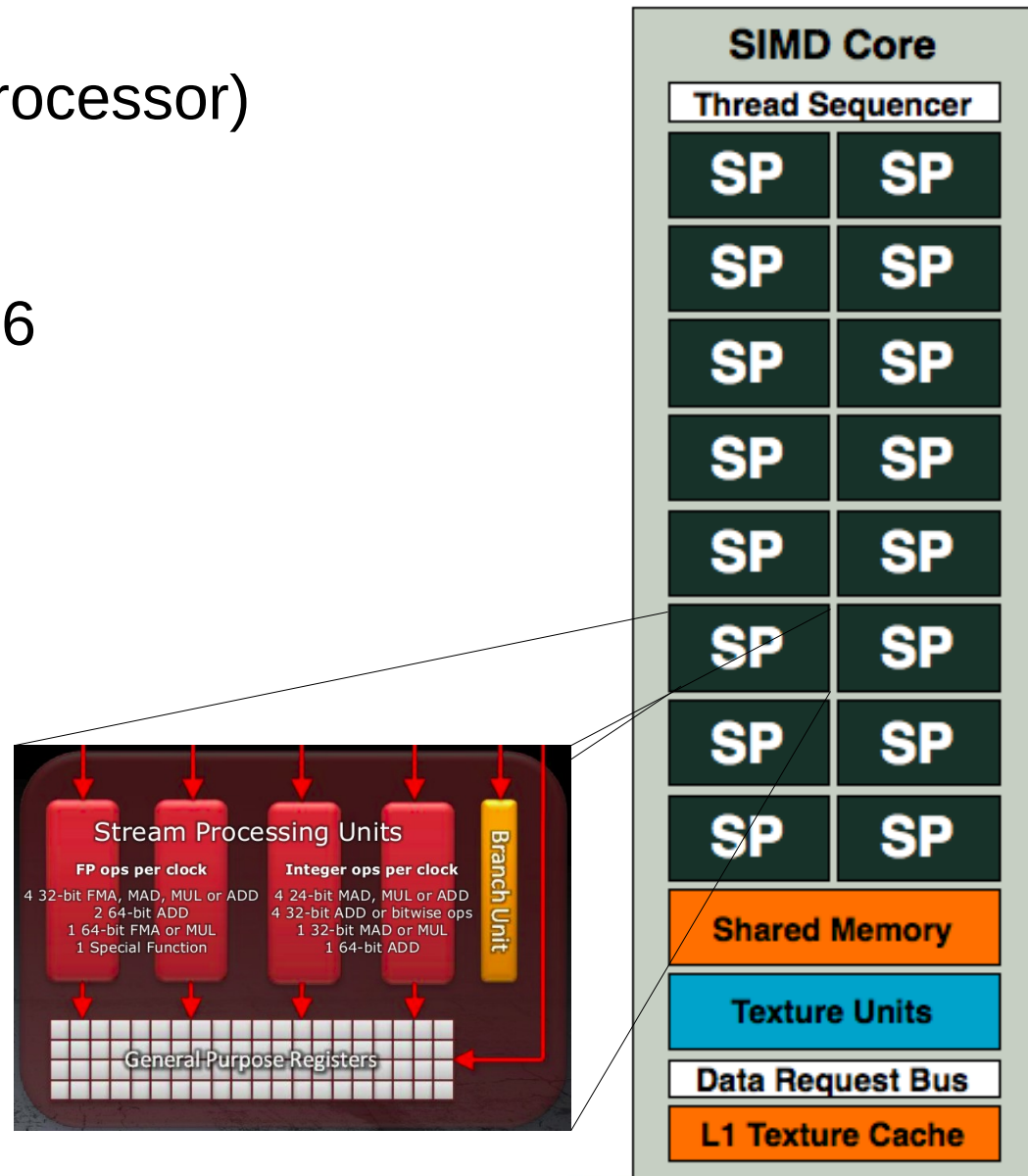
- Memory :
 - 2 Go RAM (GDDR5)
 - 1375 Mhz
 - Bus 256 bits

=> Bandwidth **176 GB/s**



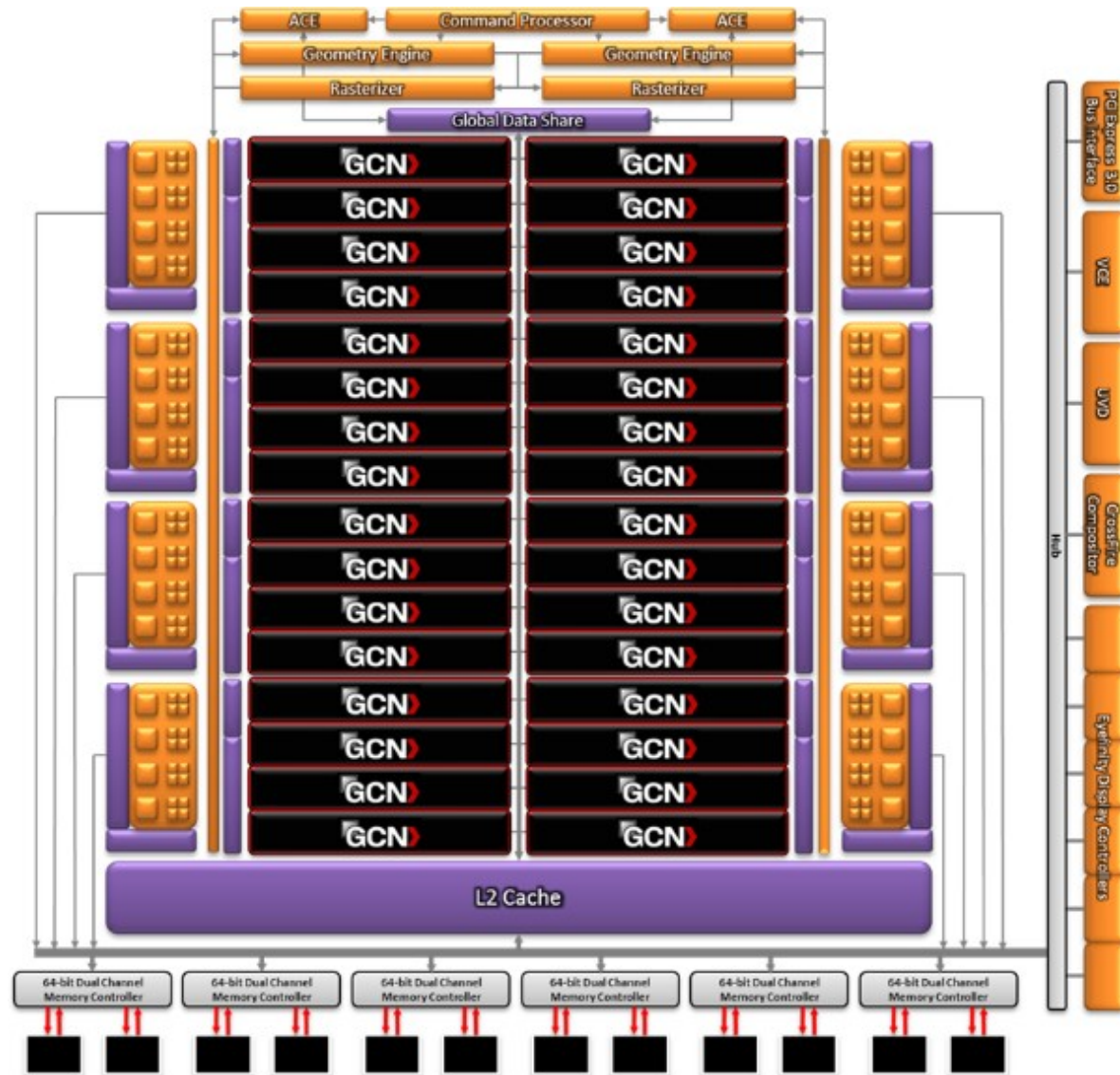
AMD basic unit: SIMD Engine 16-ways

- 16 SP (Stream multi-processor)
- 32Ko Shared Memory
- 8ko Texture L1 Cache
- Wavefront 4 cycles x 16
- 1 SP :
 - VLIW4
 - Registers :
 - 256 x 128 bits



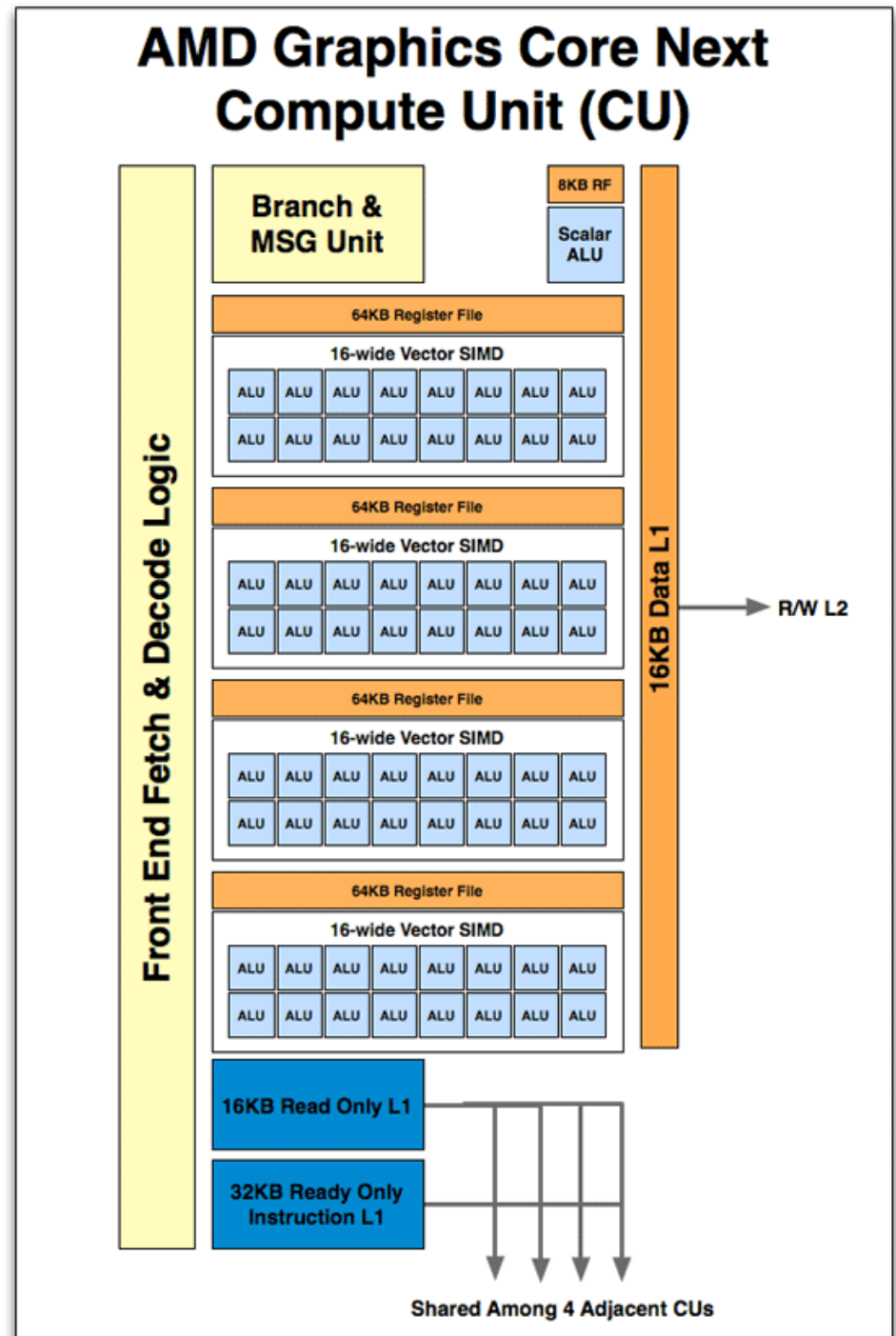
Architecture ATI Radeon HD 7970 (Tahiti)

- 32 Compute Unit (CU)
 - 4 cores SIMD 16-ways
- 2048 ALUs = $32 \times 16 \times 4$
- 925 Mhz
=> **3,79 TFLOPS**
- Memory:
 - 3 Go RAM (GDDR5)
 - 1375 Mhz
 - Bus 384 bits
=> Bandwidth **264 GB/s**
- Cache L2 768 Ko ?



Compute Unit (CU)

- 4 x SIMD Engine 16-ways
- direct access R/W au cache L2
- **16 Kb** de cache L1
- Wavefront 4 cycles x 16 x 4
- Registers :
 - 4 x 64 KB
- 1 Scalar ALU



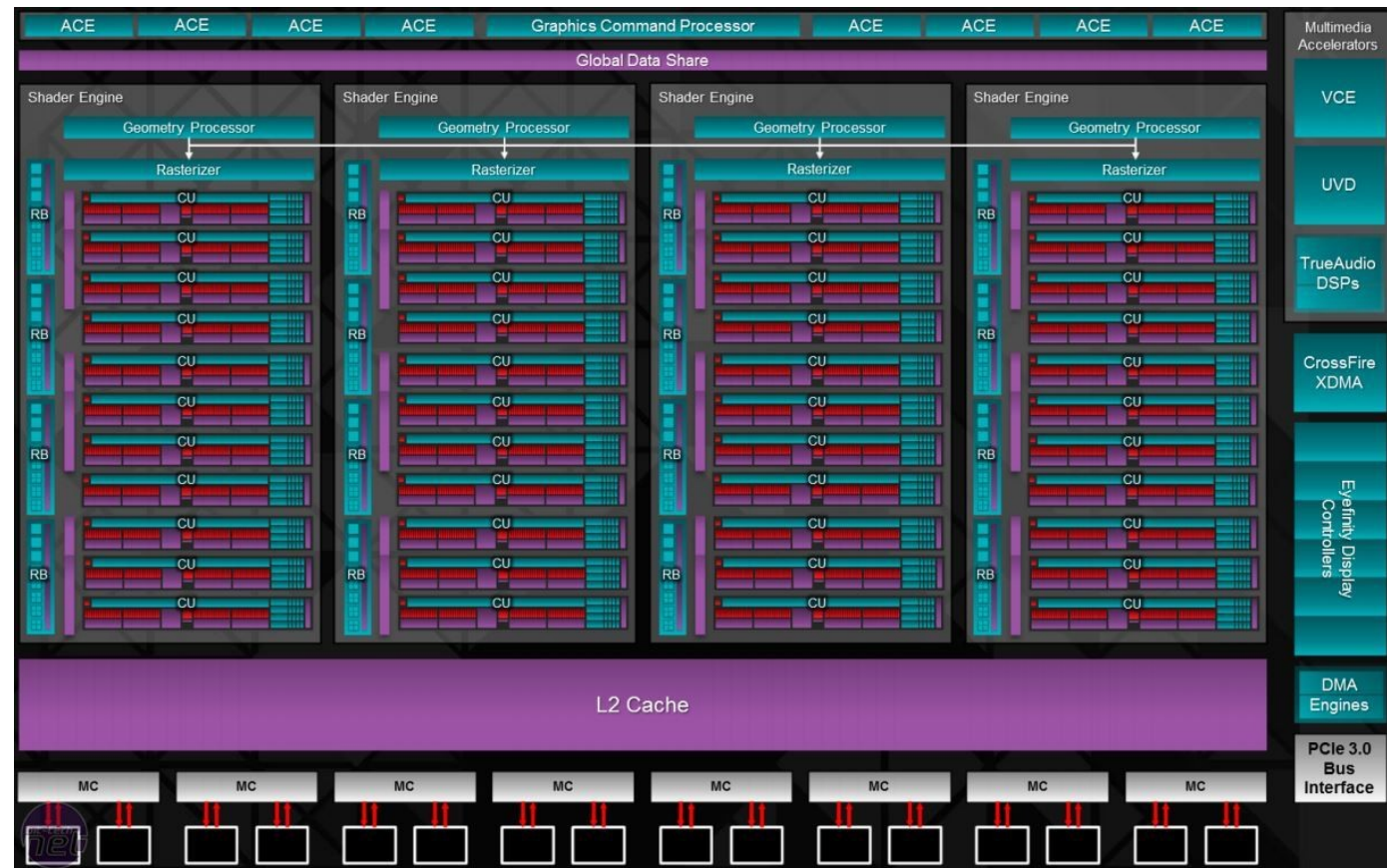
Architecture AMD Radeon R9 290X (Hawaii XT)

- 44 Compute Unit (CU)
 - 4 cores SIMD 16-ways

- 2816 ALUs

$$= 44 \times 16 \times 4$$

- 1000 Mhz
=> **5.63 TFLOPS**
- Memory:
 - 4 Go RAM (GDDR5)
 - 1375 Mhz
 - Bus 512 bits
=> Bandwidth **320 GB/s**
- Cache L2 1 Mo



Architecture AMD Radeon RX 480 (Polaris 10)

- 36 Compute Unit (CU)
 - 4 cores SIMD 16-ways

- 2304 ALUs

$$= 36 \times 16 \times 4$$

- 1266 MHz

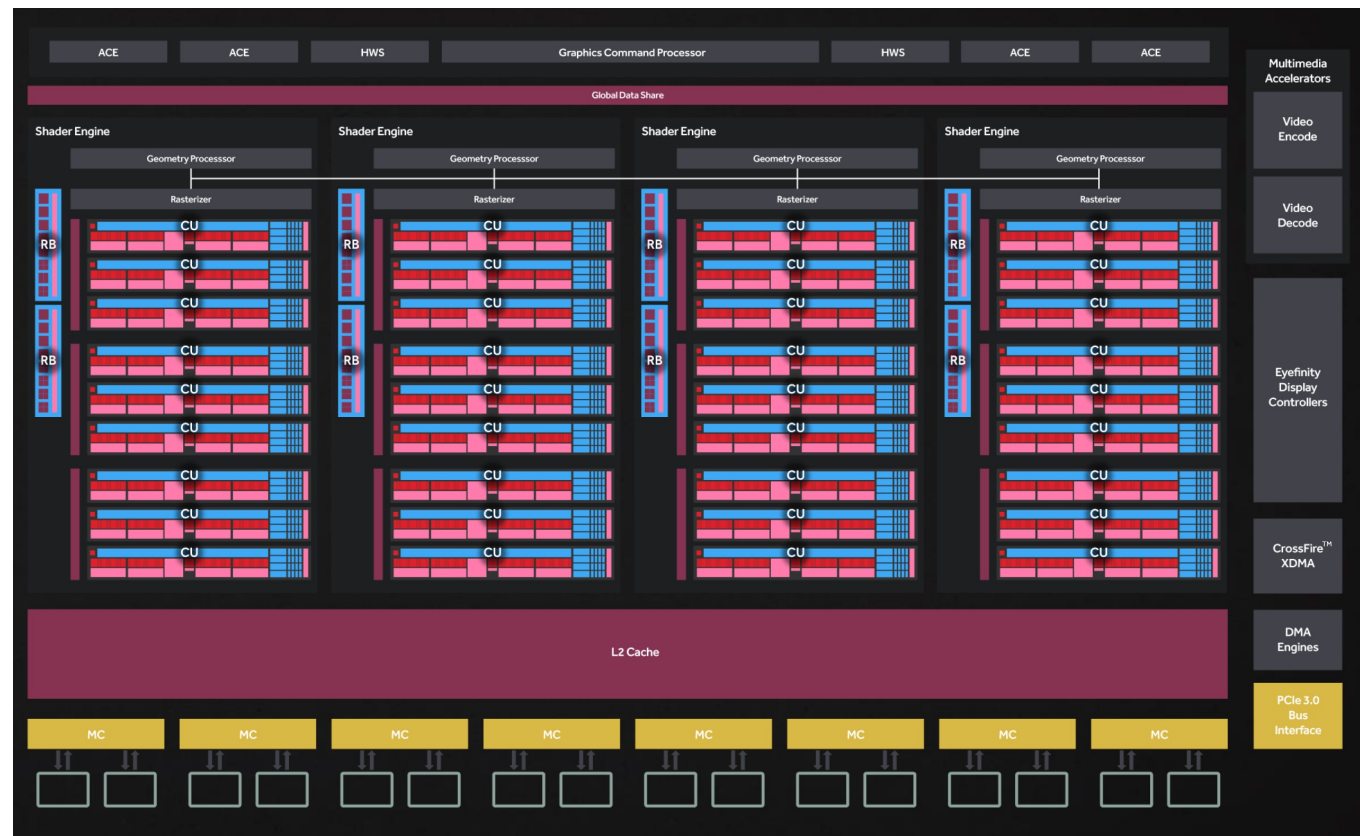
=> **5.834** TFLOPS

- Memory:

- 8 Go RAM
- 7-8 Gbps GDDR5
- Bus 256 bits

=> Bandwidth **256** GB/s

- Cache L2 2 Mo



Architecture AMD Radeon Radeon RX Vega 64 Liquid

- 64 Compute Unit (CU)
 - 4 cores SIMD 16-ways

- 4096 ALUs

$$= 64 \times 16 \times 4$$

- 1677 MHz x 2 Ops/Cycle

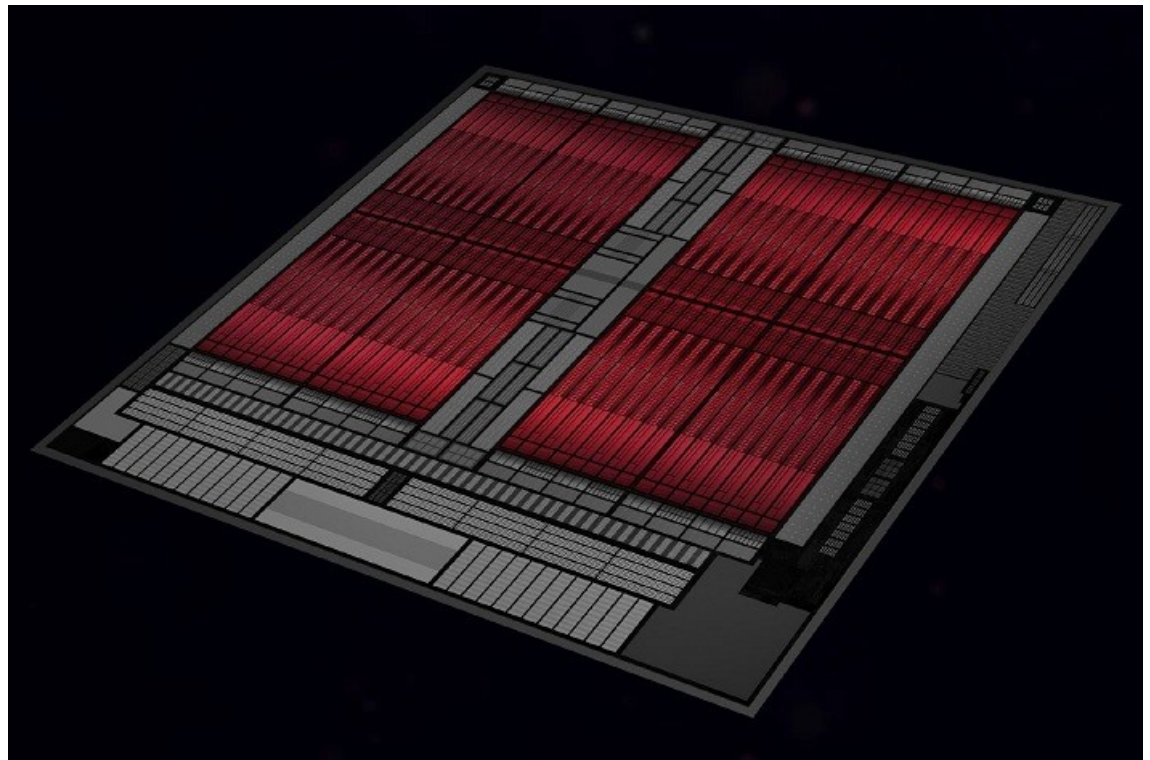
=> **13.7** TFLOPS

- Memory:

- 8 Go RAM
- 7-8 Gbps GDDR5
- Bus 256 bits

=> Bandwidth **256** GB/s

- Cache L2 2 Mo



Architecture AMD

Radeon Instinct MI60 (Vega 20 GL)

- 64 Compute Unit (CU)
 - 4 cores SIMD 16-ways

- 4096 ALUs

$$= 64 \times 16 \times 4$$

- 1800 MHz x 2 Ops/Cycle FP32

=> **14.7** TFLOPS

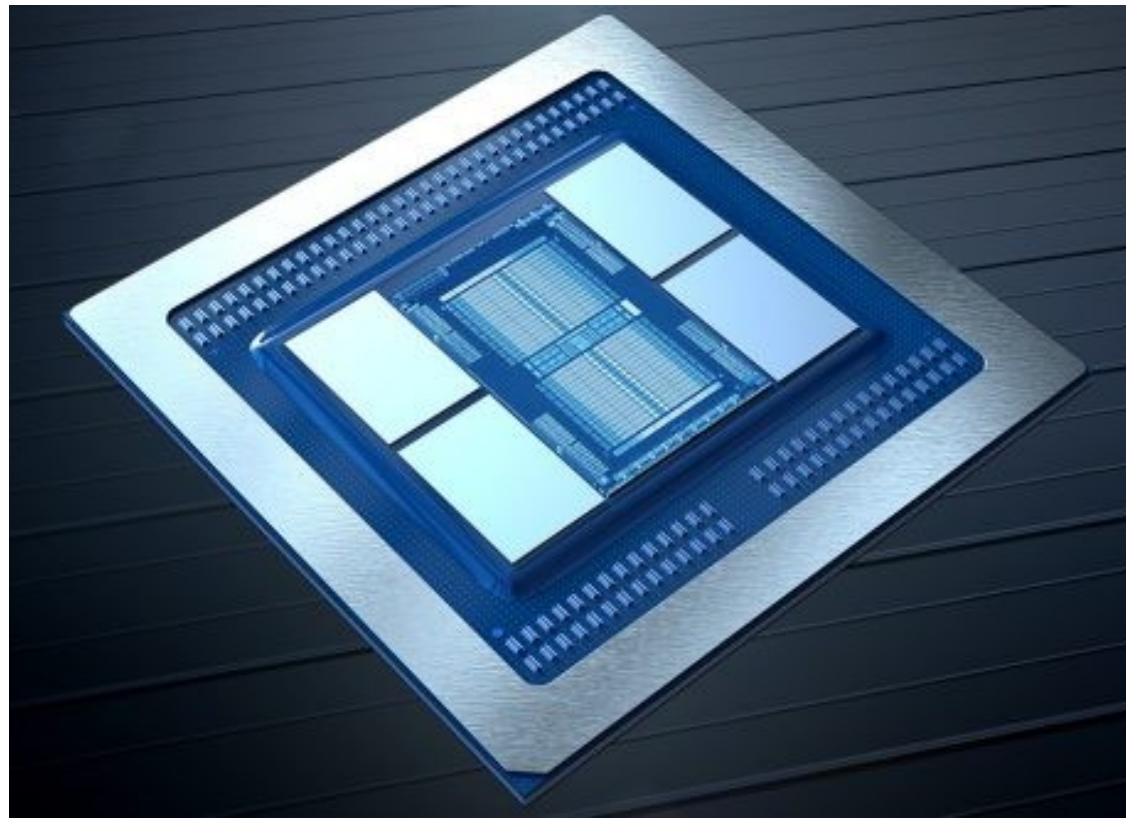
- FP 16 => 29.49 TFLOPS

- Memory:

- 32 Go RAM HBM2 @ 2000 Hz
- Bus 4096 bits

=> Bandwidth **1024** GB/s

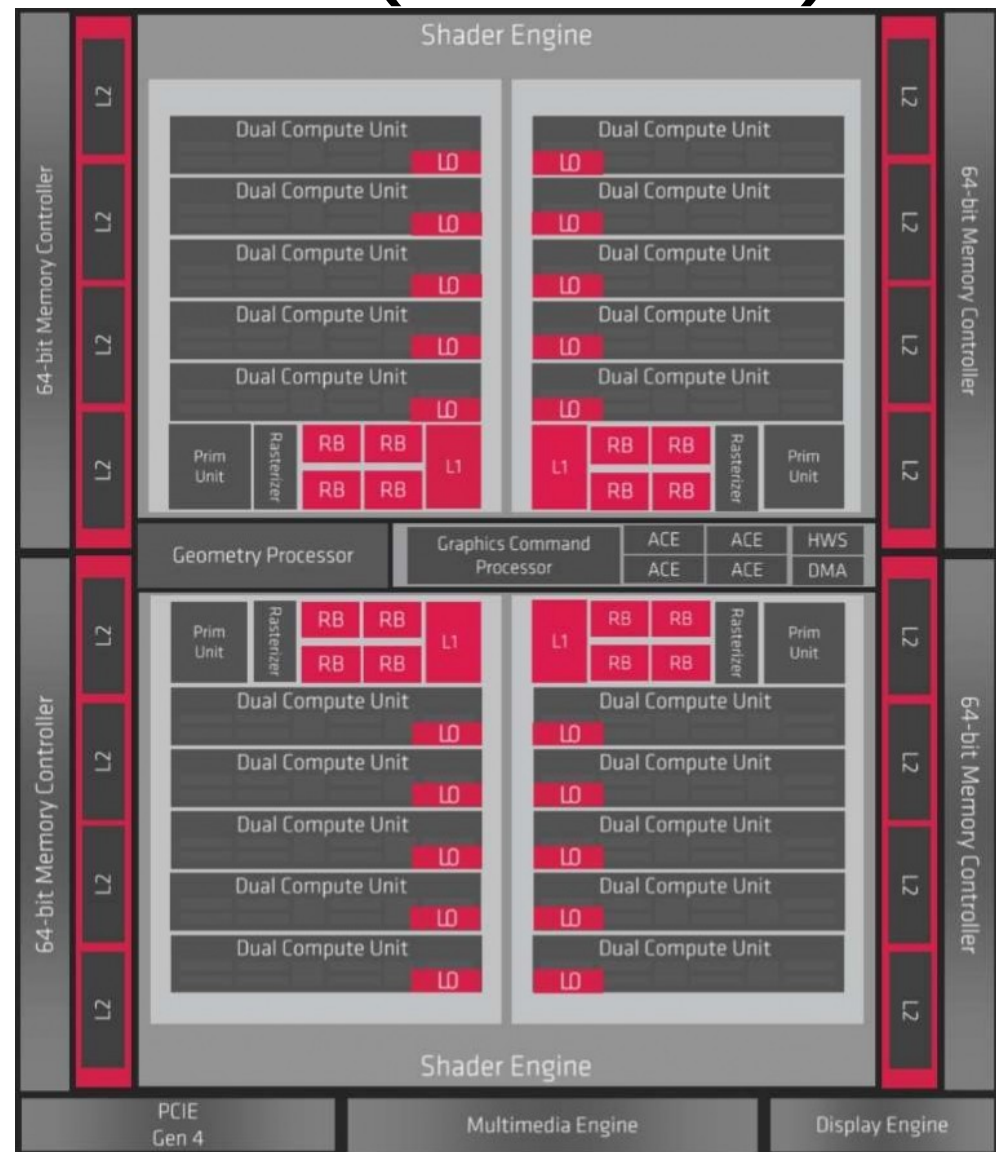
- PCIe 4.0



Architecture AMD

Radeon RX 6900 XT (Navi 21)

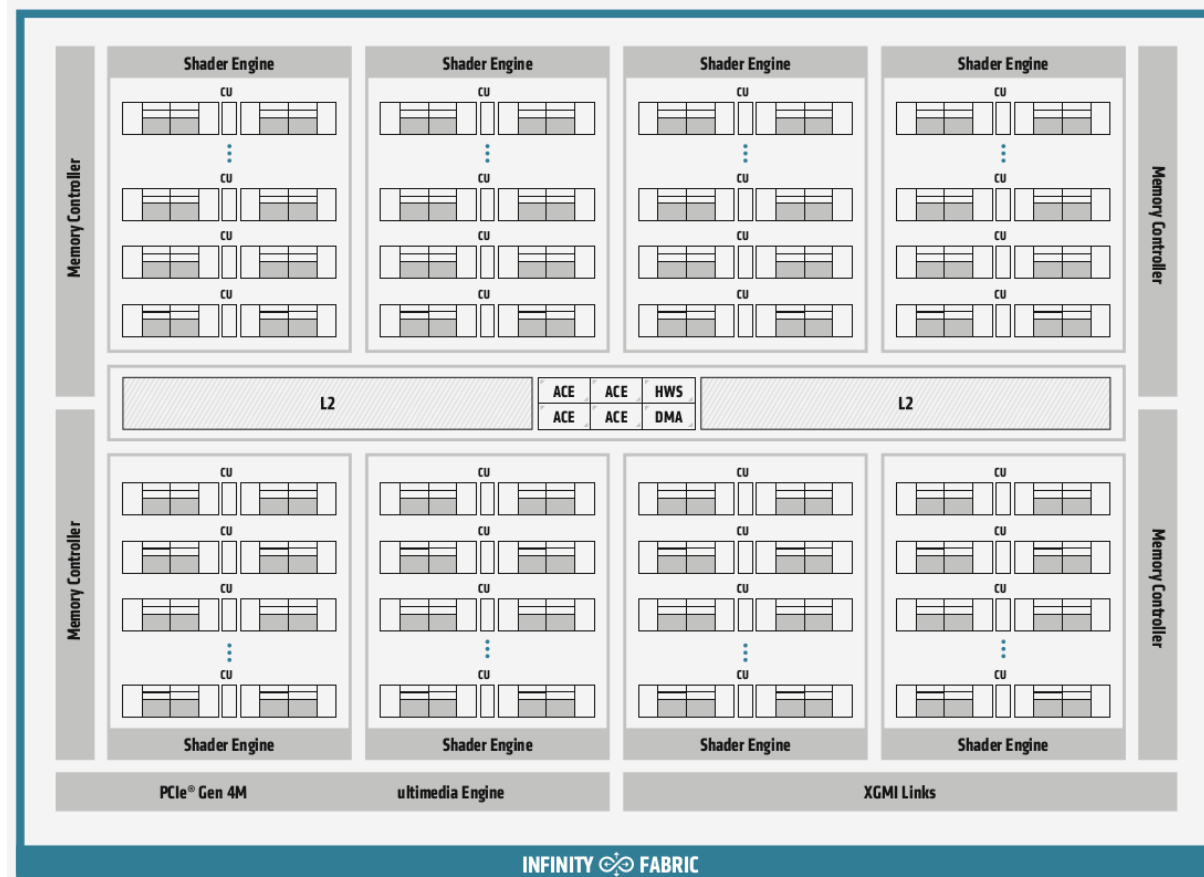
- 80 Compute Unit (CU)
 - 4 cores SIMD 16-ways
- 5120 ALUs
 $= 80 \times 16 \times 4$
- 1825 MHz x 2 Ops/Cycle FP32
 \Rightarrow **18.7 TFLOPS**
- FP 16 \Rightarrow 37 TFLOPS
- Memory:
 - 32 Go RAM GDDR6@ 2000 Hz
 - Bus 256 bits \Rightarrow Bandwidth **512 GB/s**
- PCIe 4.0



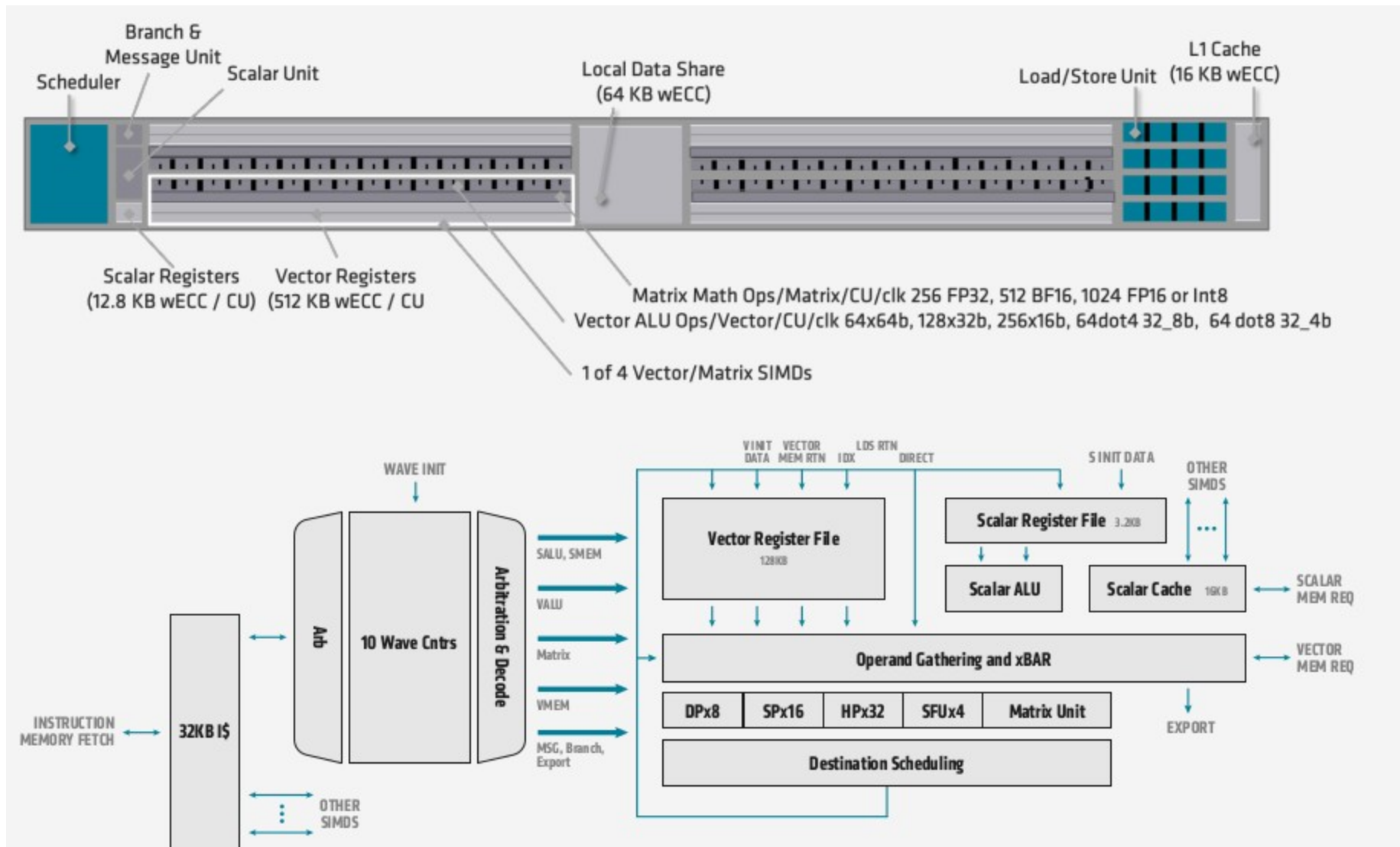
Architecture AMD

Radeon Radeon Instinct MI100

- 120 Compute Unit (CU)
 - 4 cores SIMD 16-ways
- 7680 ALUs
 $= 120 \times 16 \times 4$
- 1.5 GHz x 2 Ops/Cycle FP32
 \Rightarrow **23.1 TFLOPS**
- FP 16 \Rightarrow 184.6 TFLOPS
- Memory:
 - 32 Go RAM HBM2@ 2400 Hz
 - Bus 4096 bits
- \Rightarrow Bandwidth **1.20 TB/s**
- PCIe 4.0

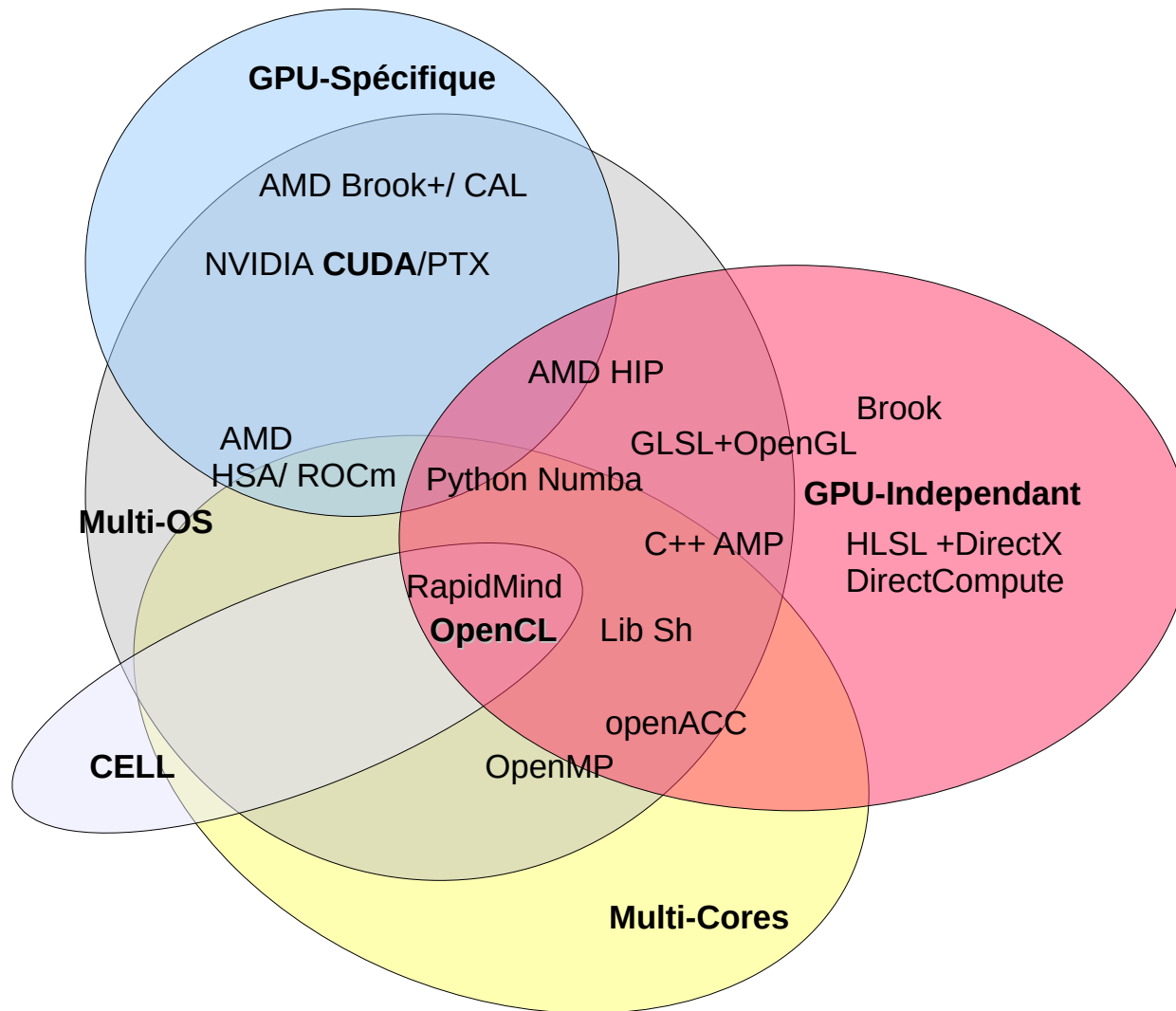


AMD Enhanced Compute Unit (CU) CDNA architecture



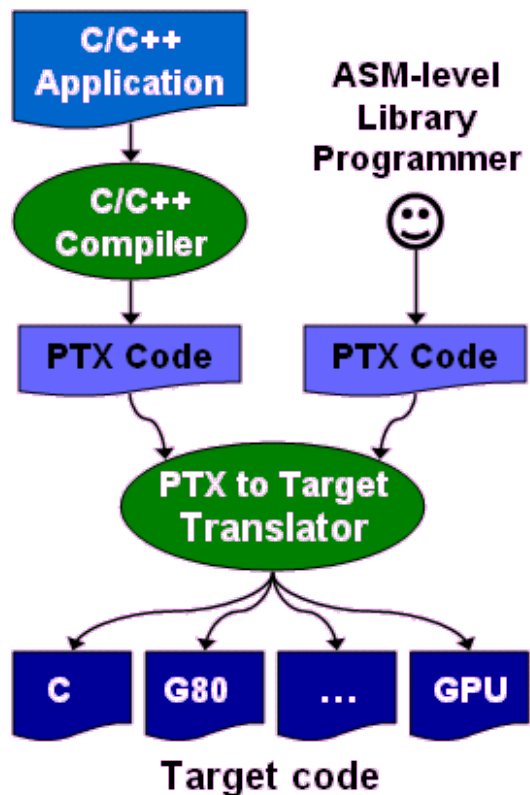
<https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf>

GP-GPU programming languages

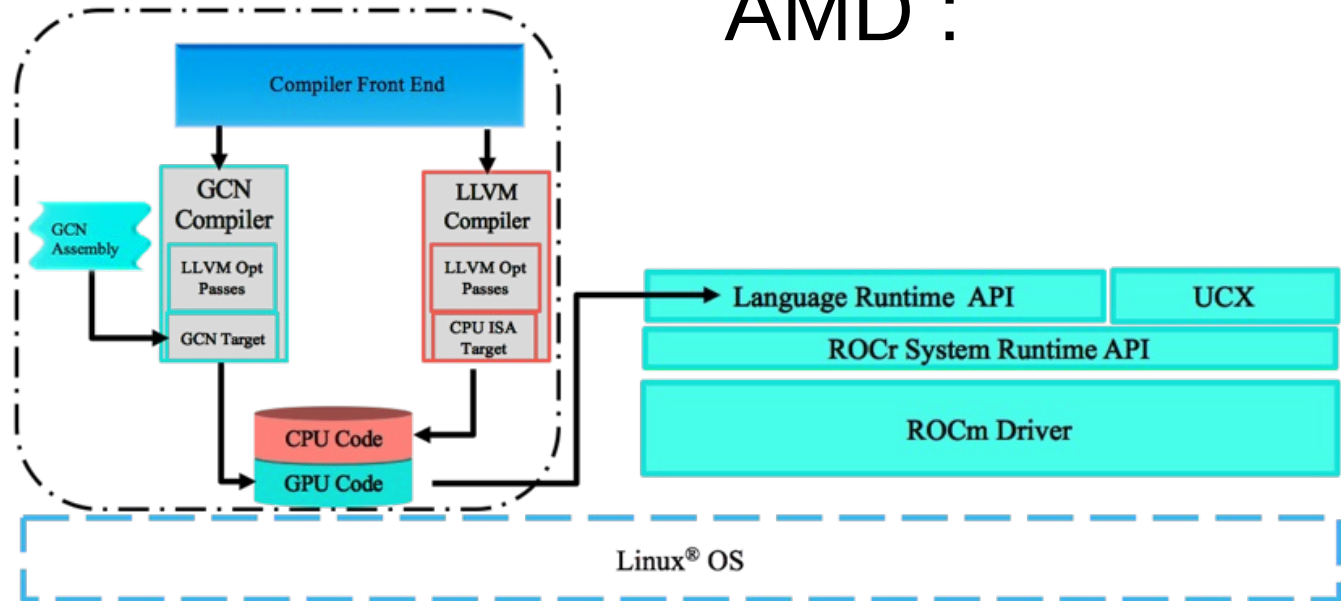


Production du programme

NVIDIA:



AMD :



CUDA

```
__global__ void sum(float* A, float *B,  
                  float* C, int width)  
{  
    unsigned int idx = threadIdx.y * width + threadIdx.x;  
    C[idx] = A[idx] + B[idx];  
}  
int main()  
{  
    ...  
    unsigned int mem_size=10*10*sizeof(float)  
    float *a; cudaMalloc((void**)&a, mem_size);  
    float *b; cudaMalloc((void**)&b, mem_size);  
    float *c; cudaMalloc((void**)&c, mem_size);  
    float input_a[100]; // size = 10*10  
    float input_b[100];  
    float output_c[100];  
    ...  
    cudaMemcpy(a, input_a, mem_size, cudaMemcpyHostToDevice);  
    cudaMemcpy(b, input_b, mem_size, cudaMemcpyHostToDevice);  
    dim3 dimBlock(10, 10);  
    sum<<<1, dimBlock>>>(a, b, c, width);  
    cudaMemcpy(output_c, c, mem_size, cudaMemcpyDeviceToHost);  
    ...  
}
```

Define kernel to run on
GPU

CUDA

```
__global__ void sum(float* A, float *B,  
                  float* C, int width)  
{  
    unsigned int idx = threadIdx.y * width + threadIdx.x;  
    C[idx] = A[idx] + B[idx];  
}  
int main()  
{  
    ...  
    unsigned int mem_size=10*10*sizeof(float);  
    float *a; cudaMalloc((void**)&a, mem_size);  
    float *b; cudaMalloc((void**)&b, mem_size);  
    float *c; cudaMalloc((void**)&c, mem_size);  
    float input_a[100]; // size = 10*10  
    float input_b[100];  
    float output_c[100];  
    ...  
    cudaMemcpy(a, input_a, mem_size, cudaMemcpyHostToDevice);  
    cudaMemcpy(b, input_b, mem_size, cudaMemcpyHostToDevice);  
    dim3 dimBlock(10, 10);  
    sum<<<1, dimBlock>>>(a, b, c, width);  
    cudaMemcpy(output_c, c, mem_size, cudaMemcpyDeviceToHost);  
    ...  
}
```

Allocation of data array to be manipulated on GPU

CUDA

```
__global__ void sum(float* A, float *B,
                   float* C, int width)
{
    unsigned int idx = threadIdx.y * width + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
int main()
{
    ...
    unsigned int mem_size=10*10*sizeof(float);
    float *a; cudaMalloc((void**)&a, mem_size);
    float *b; cudaMalloc((void**)&b, mem_size);
    float *c; cudaMalloc((void**)&c, mem_size);
    float input_a[100]; // size = 10*10
    float input_b[100];
    float output_c[100];
    ...
    cudaMemcpy(a, input_a, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(b, input_b, mem_size,
cudaMemcpyHostToDevice);
    dim3 dimBlock(10, 10);
    sum<<<1, dimBlock>>>(a, b, c, width);
    cudaMemcpy(output_c, c, mem_size, cudaMemcpyDeviceToHost);
    ...
}
```

Send Data to GPU

CUDA

```
__global__ void sum(float* A, float *B,  
                  float* C, int width)  
{  
    unsigned int idx = threadIdx.y * width + threadIdx.x;  
    C[idx] = A[idx] + B[idx];  
}  
int main()  
{  
    ...  
    unsigned int mem_size=10*10*sizeof(float);  
    float *a; cudaMalloc((void**)&a, mem_size);  
    float *b; cudaMalloc((void**)&b, mem_size);  
    float *c; cudaMalloc((void**)&c, mem_size);  
    float input_a[100]; // size = 10*10  
    float input_b[100];  
    float output_c[100];  
    ...  
    cudaMemcpy(a, input_a, mem_size, cudaMemcpyHostToDevice);  
    cudaMemcpy(b, input_b, mem_size, cudaMemcpyHostToDevice);  
    dim3 dimBlock(10, 10);  
    sum<<<1, dimBlock>>>(a, b, c, width);  
    cudaMemcpy(output_c, c, mem_size, cudaMemcpyDeviceToHost);  
    ...  
}
```

Call kernel execution on
GPU

CUDA

```
__global__ void sum(float* A, float *B,  
                  float* C, int width)  
{  
    unsigned int idx = threadIdx.y * width + threadIdx.x;  
    C[idx] = A[idx] + B[idx];  
}  
int main()  
{  
    ...  
    unsigned int mem_size=10*10*sizeof(float);  
    float *a; cudaMalloc((void**)&a, mem_size);  
    float *b; cudaMalloc((void**)&b, mem_size);  
    float *c; cudaMalloc((void**)&c, mem_size);  
    float input_a[100]; // size = 10*10  
    float input_b[100];  
    float output_c[100];  
    ...  
    cudaMemcpy(a, input_a, mem_size, cudaMemcpyHostToDevice);  
    cudaMemcpy(b, input_b, mem_size, cudaMemcpyHostToDevice);  
    dim3 dimBlock(10, 10);  
    sum<<<1, dimBlock>>>(a, b, c, width);  
    cudaMemcpy(output_c, c,  
mem_size, cudaMemcpyDeviceToHost);  
    ...  
}
```

Write-back data in CPU
Ram

Programming CUDA: kernel level

- Vectorial Types :

- {char, short, uint, int, float, long, ulong}{1-4}

example: float4 data[10];
data[1].x=0.1; data[1].y=0.1; data[1].z=0.1; data[1].w=0.2;

- Mathematics functions:

- sqrtf(x), sinf(x), log10f(x), etc.
- Fast Versions : __sinf(x), __log10f(x), etc. (pas de __sqrt(x) !)

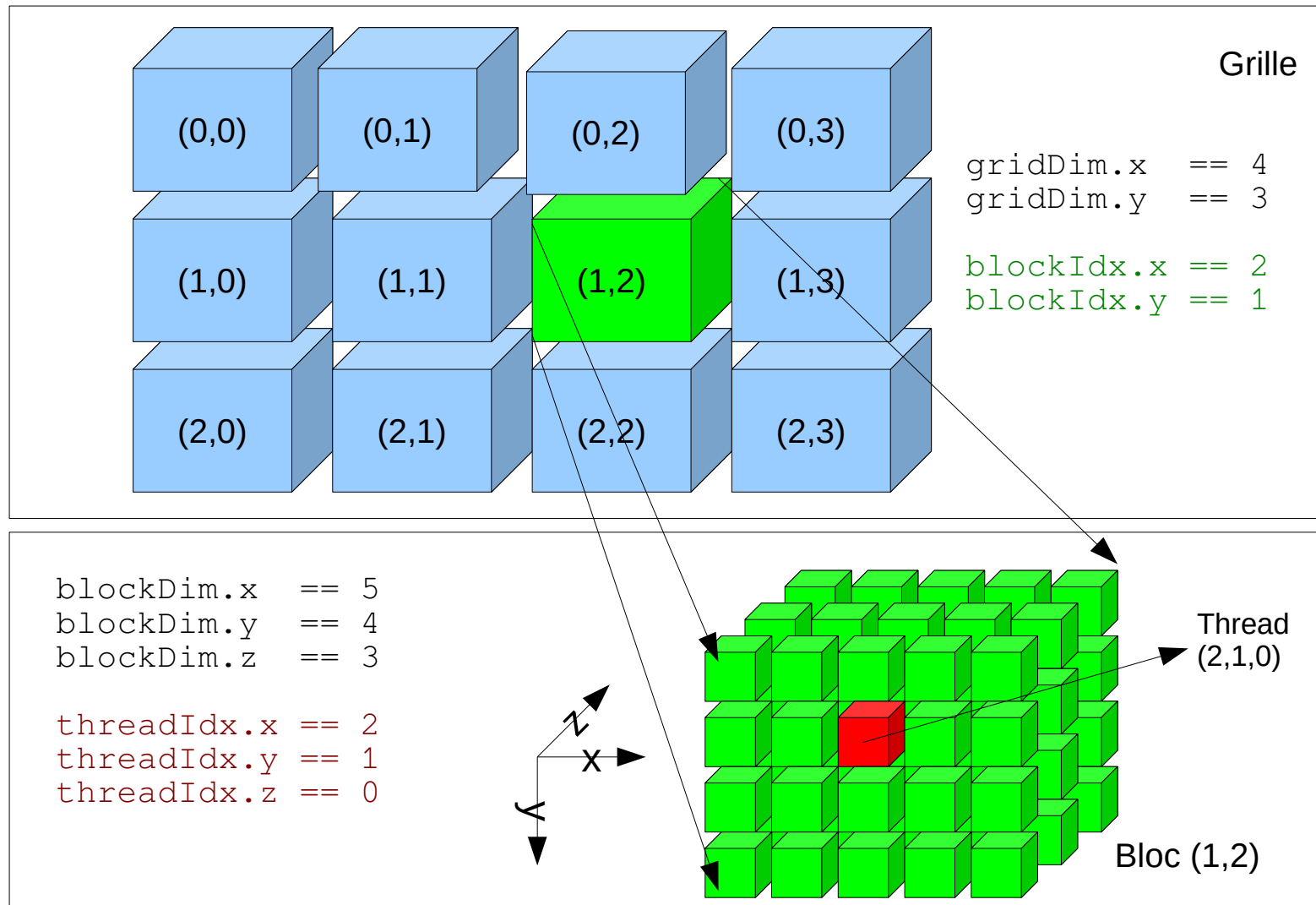
- Branching logic:

- if(), for(; ;), while(), etc.
- unroll loops #pragma unroll (know small size)

- Temps measurement: clock()

- Vote (>G9x) : int __all(int predicate); int __any(int predicate) ;
- Atomic (>G84) : int atomicAdd(int* address, int val);

Grids and blocks



Grids and blocks (Host side level)

```
// execution parameters

dim3 grid(size_x / BLOCK_DIM, size_y /
BLOCK_DIM, 1);

dim3 blocks(BLOCK_DIM, BLOCK_DIM, 1);

// call a kernel « myKernel »

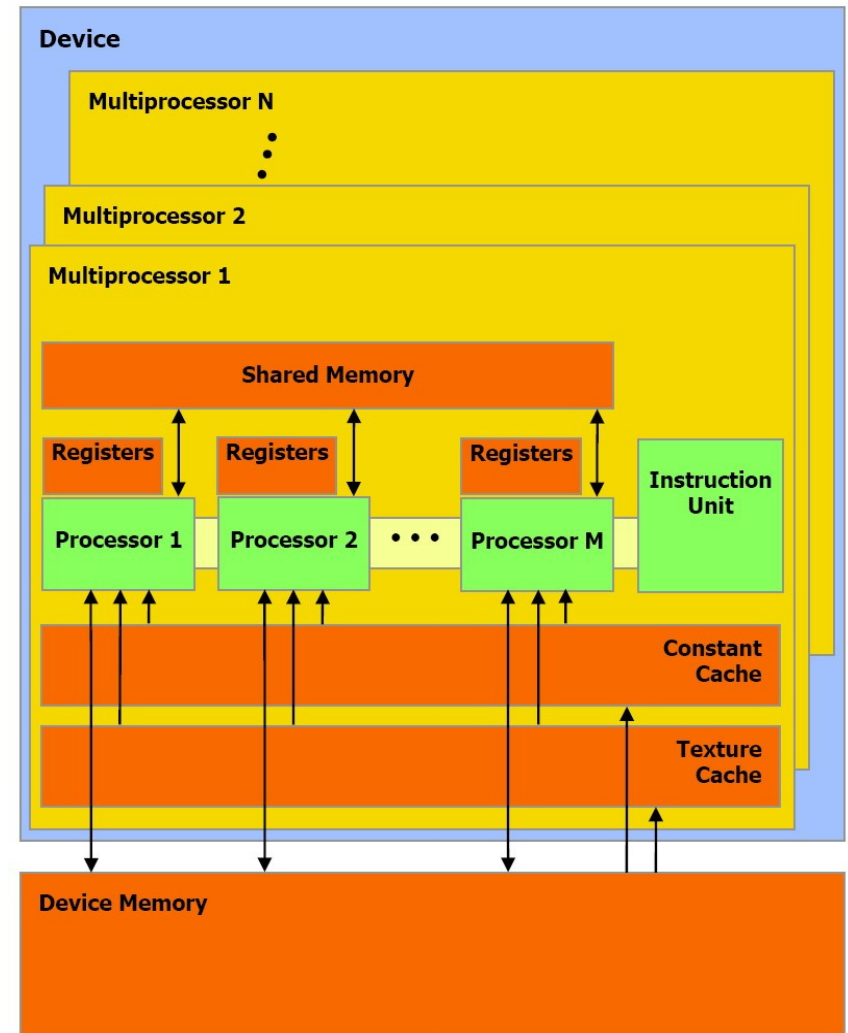
myKernel<<< grid, blocks
>>>(device_out_data, device_in_data,
size_x, size_y);
```

Grids and blocks (kernel level)

```
// kernel declaration of « myKernel »
__global__ void myKernel(float *odata, float*
idata, int width, int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x
+ threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y
+ threadIdx.y;
    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in  = xIndex + width *
yIndex;
        unsigned int index_out = yIndex + height
* xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

Memory

- Every Multi-processor (SM):
 - 1 shared memory (read/write.)
 - 1 texture cache(read only)
 - 1 constants cache (read only)
 - 1 L1 cache (read/write since GF100)
 - 1 instruction cache
- Every ALU:
 - Local registers (read/write)
- direct access to GPU DRAM (*global memory*)



Memory (kernel level)

```
__shared__ float block[BLOCK_DIM]  
[BLOCK_DIM+1];  
  
__constant__ float3 kColorMetric  
= { 1.0f, 1.0f, 1.0f };  
  
texture<unsigned char, 2> tex;
```

Memory (host level)

```
// allocation into the GPU card  
(device)
```

```
float* device_data;
```

```
cudaMalloc( (void**)   
&device_data, mem_size);
```

```
// Release GPU memory
```

```
cudaFree(device_data);
```

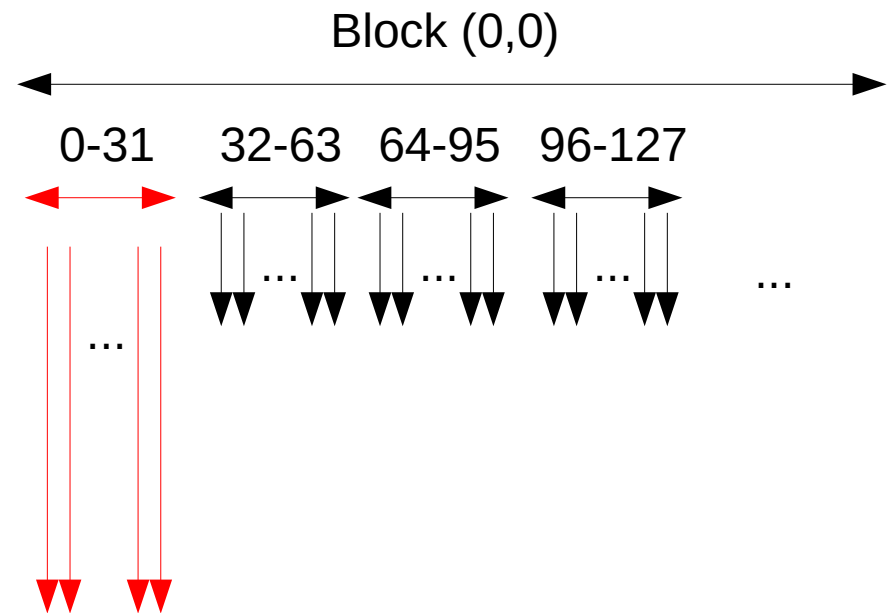
Memory (host level)

```
// copy from CPU memory (host)
// to GPU memory (device)
cudaMemcpy( device_data, host_data, mem_size,
            cudaMemcpyHostToDevice);

// copy from GPU memory (device)
// to CPU memory (host)
cudaMemcpy( host_data, device_data, mem_size,
            cudaMemcpyDeviceToHost);
```

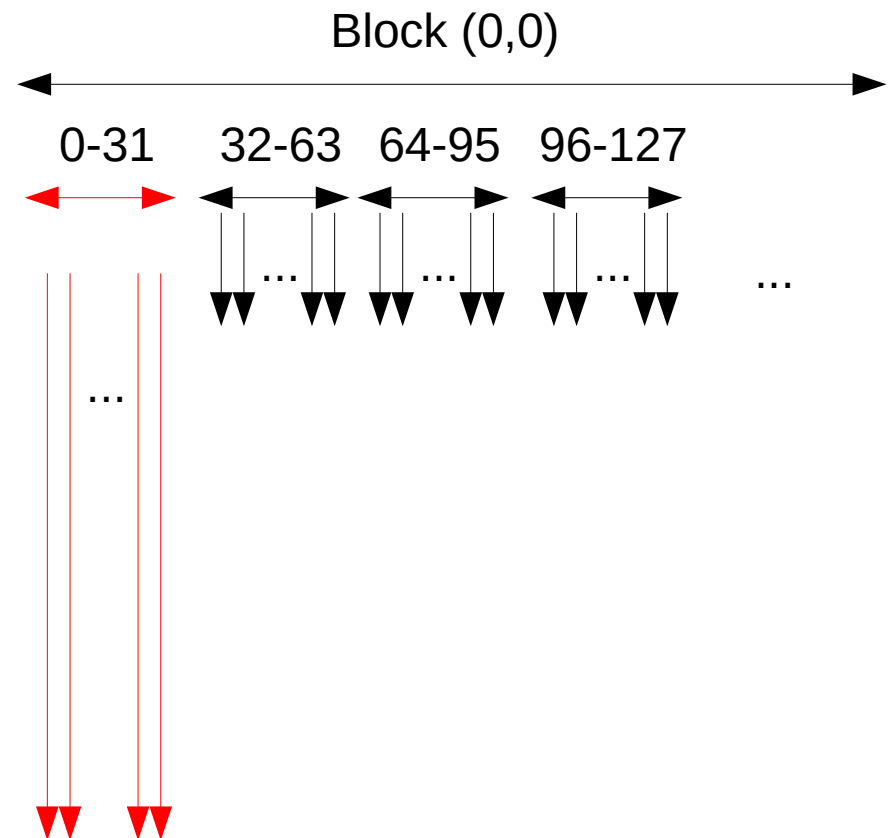
Threads

```
global
myKernel(float
*A, ...)
{
    ...
    float a= x*(x+0.5f)-
    1 ;
    ...
    A[g_index]=a;
    ...
}
```



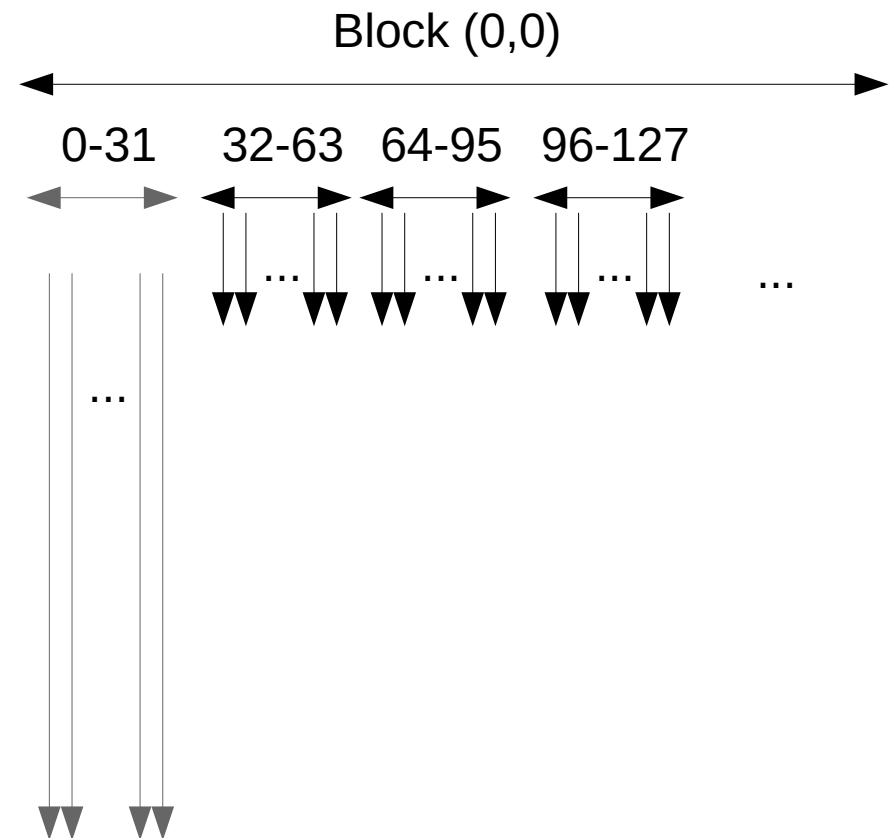
Threads

```
global
myKernel(float
*A, ...)
{
    ...
    float a= x*(x+0.5f) -
    1 ;
    ...
    A[g_index]=a;
    ...
}
```



Threads

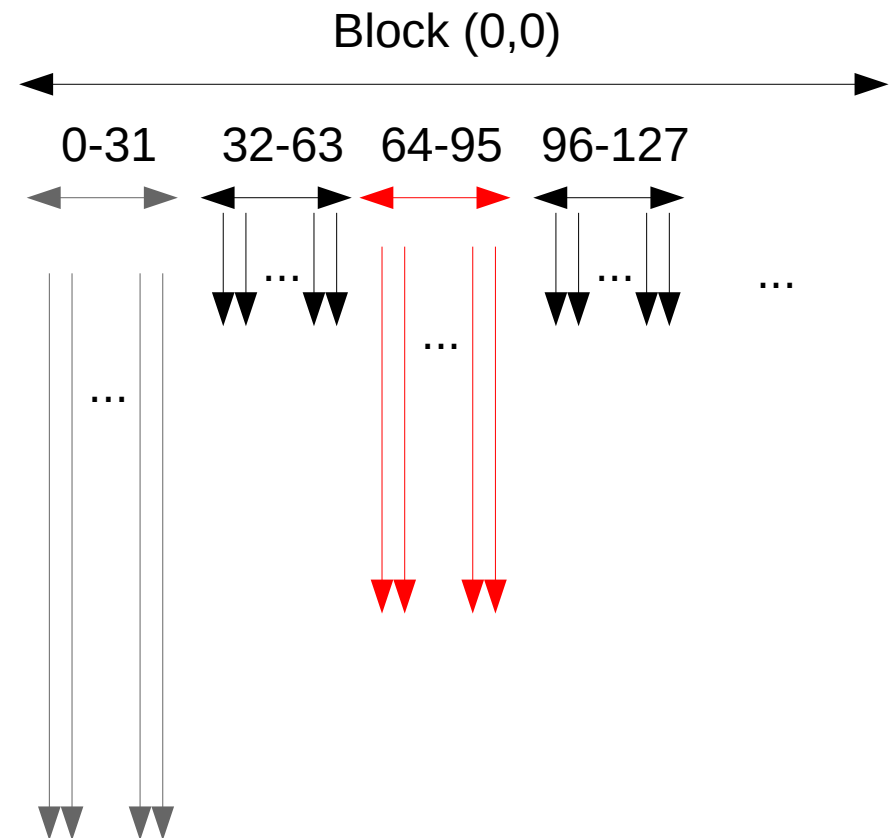
```
global
myKernel(float
*A, ...)
{
    ...
    float a= x*(x+0.5f)-
    1 ;
    ...
    A[g_index]=a;
    ...
}
```



Waiting
~400 cycles

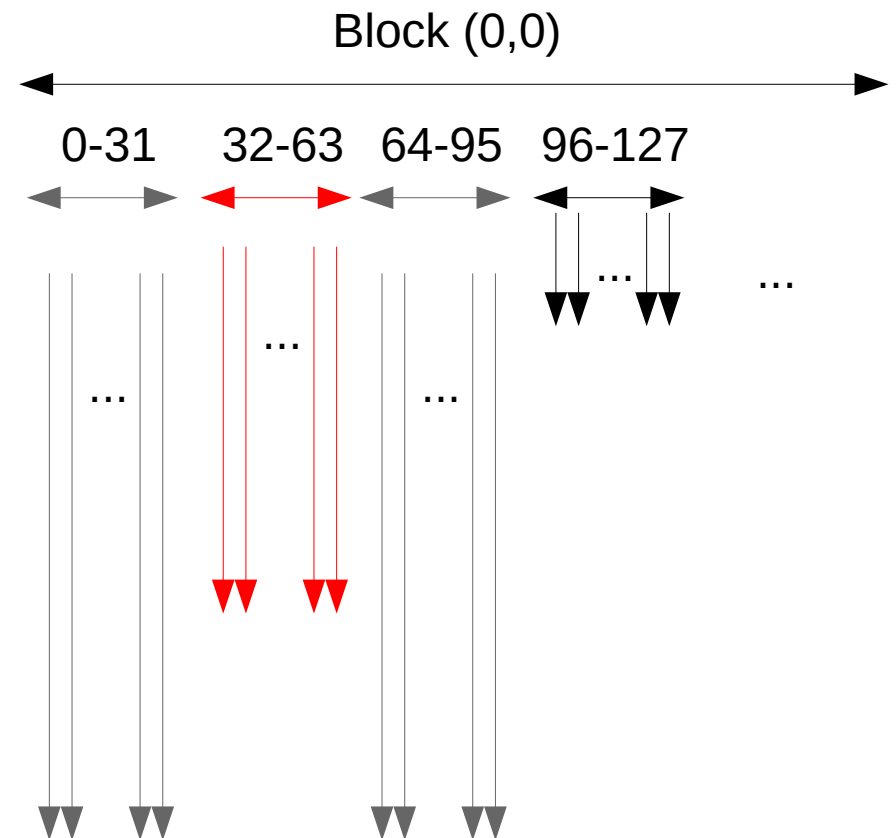
Threads

```
global
myKernel(float
*A, ...)
{
    ...
    float a= x*(x+0.5f) -
    1 ;
    ...
    A[g_index]=a;
    ...
}
```



Threads

```
global
myKernel(float
*A, ...)
{
    ...
    float a= x*(x+0.5f)-
    1 ;
    ...
    A[g_index]=a;
    ...
}
```



Threads

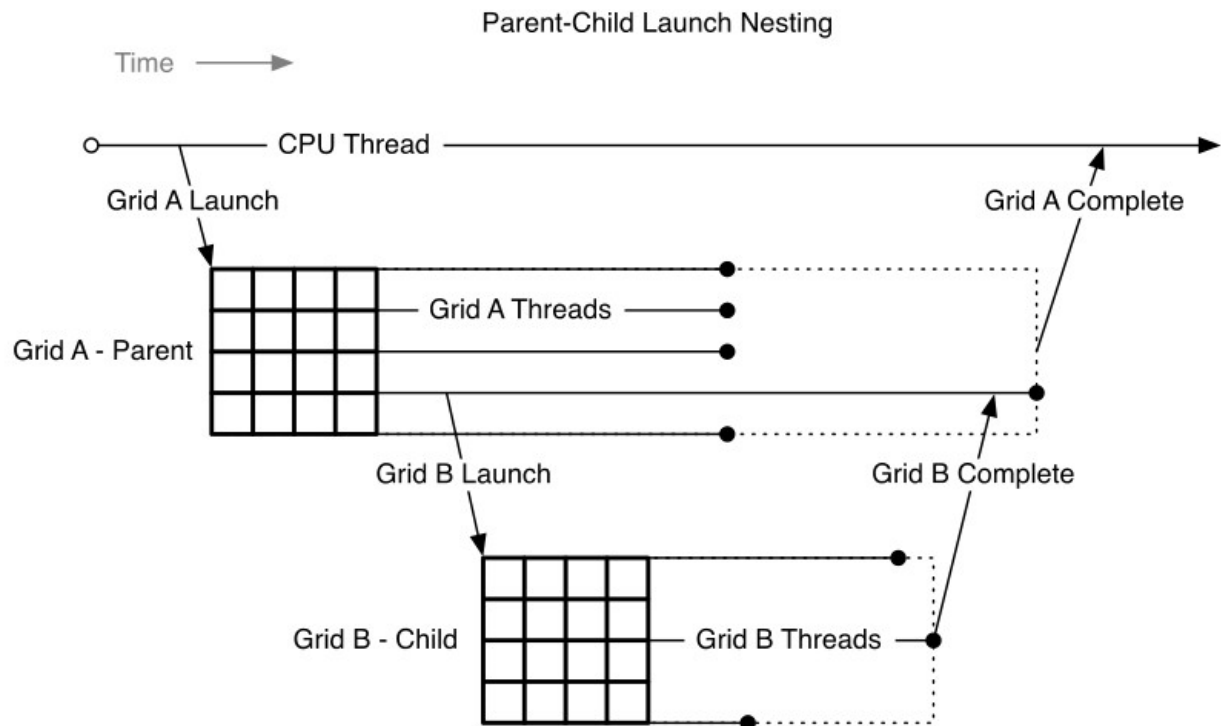
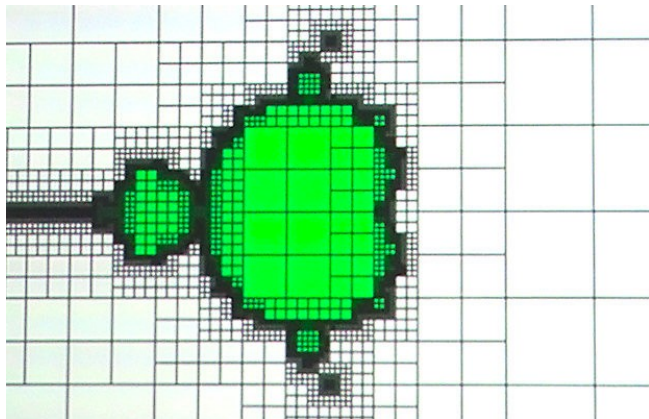
```
__global__ void myKernel(float *odata, float* idata, int width, int
height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM];
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if((xIndex < width) && (yIndex < height))
    {
        unsigned int index_in = yIndex * width + xIndex;
        // copy into shared memory
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }

    __syncthreads();
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if((xIndex < height) && (yIndex < width))
    {
        unsigned int index_out = yIndex * height + xIndex;
        // process
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Cuda 5

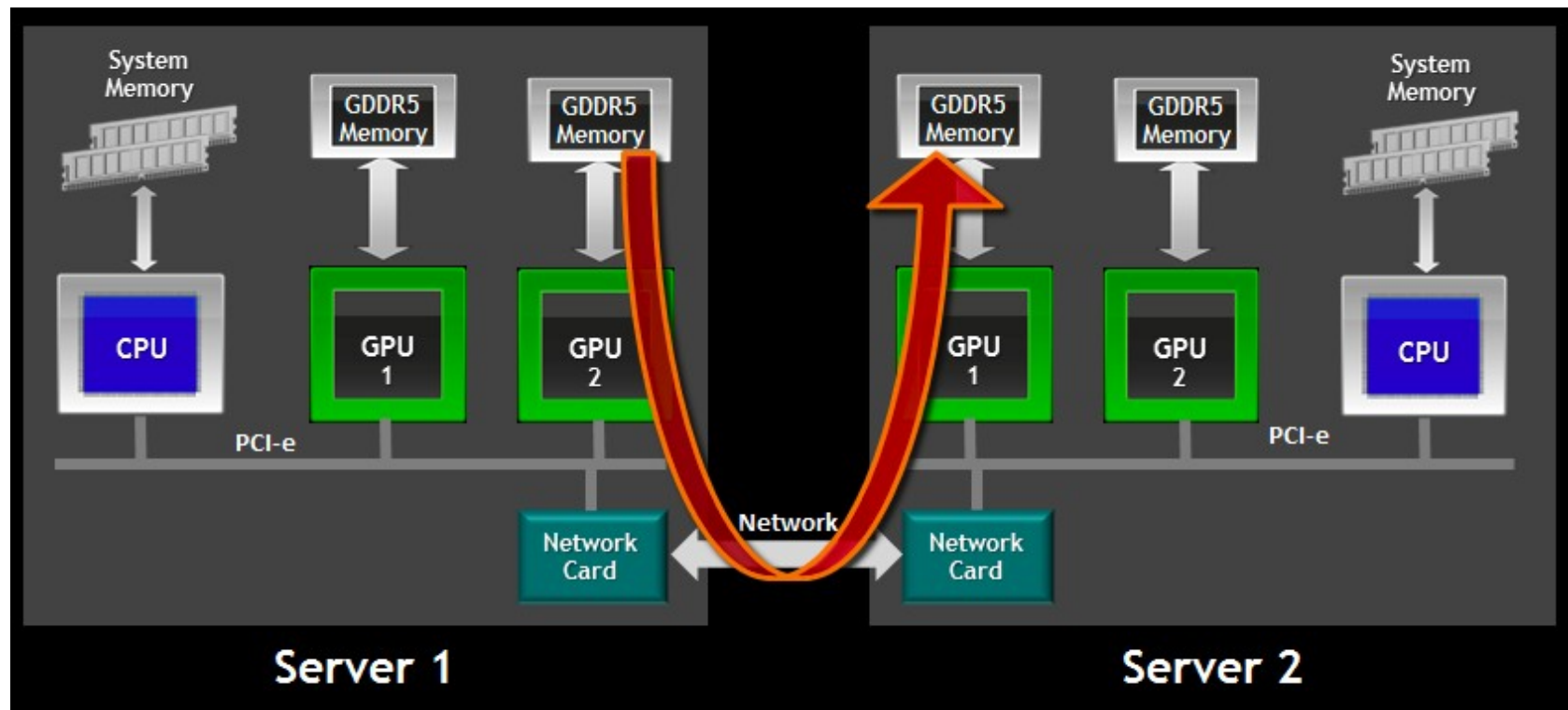
- Dynamic parallelism

```
__global__ RecursiveKernel(void*  
data) {  
    if(continueRecursion == true)  
        RecursiveKernel<<<64, 16>>>(data);  
}
```



CUDA 5

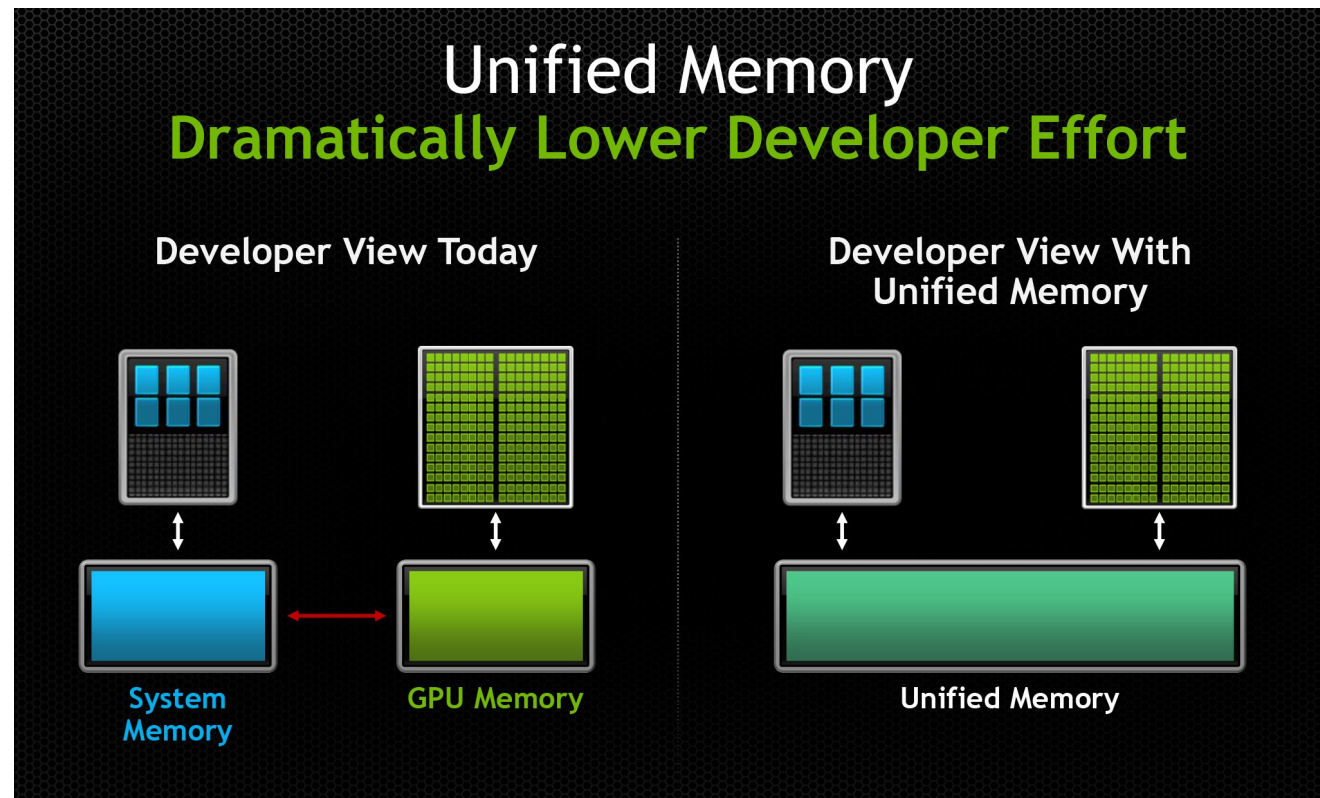
- DirectGPU with RDMA
(MPI Send/Receive without copy in RAM
CPU)



CUDA 6

- Unified Memory

```
void sortfile(FILE *fp, int N) {  
    char*data;  
    cudaMallocManaged(&data, N);  
    fread(data, 1, N, fp);  
    qsort<<<...>>>(data,N,1,compare);  
    cudaDeviceSynchronize();  
    use_data(data);  
    cudaFree(data);  
}
```



CUDA 7.5

- Support C++ 11
 - Functions Lambda
 - auto

CUDA 8

- Native FP16 and INT8

<https://devblogs.nvidia.com/parallelforall/new-features-cuda-7-5/>

<https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/>

CUDA 9.0

- Support C++ 14
- Cooperative Groups

```
__global__ void particleSim(Particle *p, int N) {  
    grid_group g = this_grid();  
    // phase 1  
    for (i = g.thread_rank(); i < N; i += g.size())  
        integrate(p[i]);  
    g.sync() // Sync whole grid  
    // phase 2  
    for (i = g.thread_rank(); i < N; i += g.size())  
        collide(p[i], p, N);  
}
```

CUDA 9.0 Tensor Core Operations

```
#include <mma.h>
using namespace nvcuda;
...
const int WMMA_M = 16;
const int WMMA_N = 16;
const int WMMA_K = 16;
```

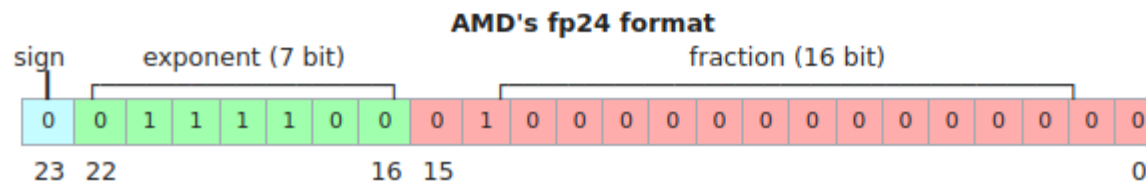
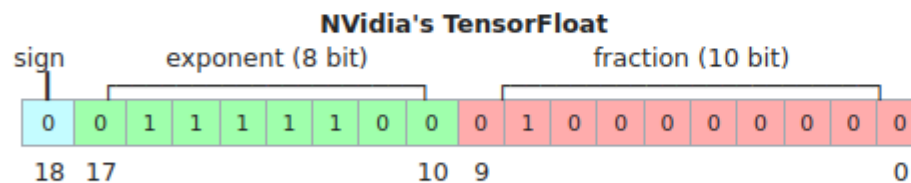
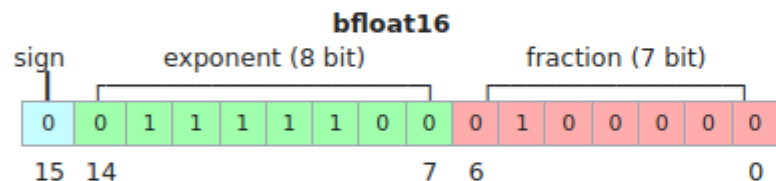
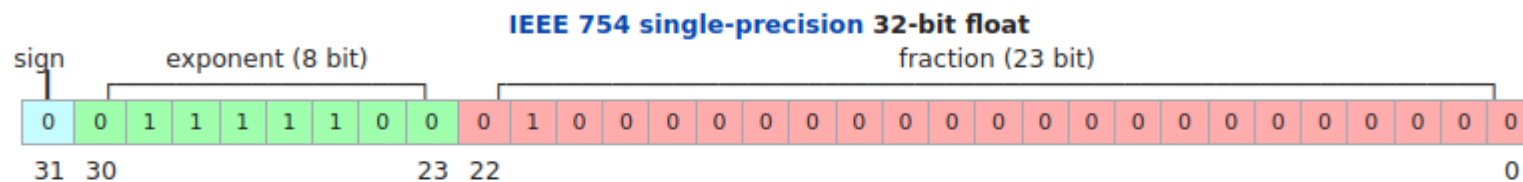
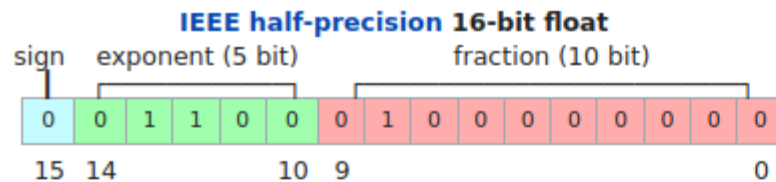
```
...
    wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half,
wmma::col_major> a_frag;
    wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half,
wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K,
float> acc_frag;
    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K,
float> c_frag;
...
wmma::fill_fragment(acc_frag, 0.0f);
...
wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda);
    wmma::load_matrix_sync(b_frag, b + bRow + bCol *
ldb, ldb);
...
wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
...
wmma::store_matrix_sync(c + cRow + cCol * ldc, c_frag, ldc,
wmma::mem_col_major);
...
```

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma>

BF16 floating point



PyCuda

- Hight level programming with Python
- Manage data in CPU with NumPy library
- Simple but complete access to functions of CUDA API

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule

mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))

print dest-a*b
```

Simple Init devices context

```

import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule

mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))

print dest-a*b

```

Inline kernel code as multi-line python string
Automatically compiled

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))
print dest-a*b
```

Simple data generation with numpy

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))
print dest-a*b
```

Simple data generation with numpy

```

import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))
print dest-a*b

```

Automatic data transfer before kernel call

```

import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))
print dest-a*b

```

Kernel call with grid & block parameters

```

import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))
print dest-a*b

```

Automatic copy-back result

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))
print dest-a*b
```

Automatic copy-back result

```

import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))

```

print dest-a*b

Check result with numpy primitives

References

- PyCuda documentation <http://documen.tician.de/pycuda/>
- Study Card Nvidia GTX 480 et GTX 470 <http://techreport.com/articles.x/18682>
- Architecture GPU NVIDIA GF100 <http://techreport.com/articles.x/17670>
- Architecture GPU AMD RV870
<http://www.bit-tech.net/hardware/graphics/2009/09/30/ati-radeon-hd-5870-architecture-analysis/>
- AMD's Radeon HD 6970 & Radeon HD 6950: Paving The Future For AMD
<http://www.anandtech.com/show/4061/amds-radeon-hd-6970-radeon-hd-6950>
- A Jump Start to OpenCL <https://www.cis.upenn.edu/~cis565/LECTURE2010/OpenCL.pdf>
- List of Nvidia GPUs : http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units
- List of AMD GPUs : http://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units

Overview

- Part 1 : Basics on parallel machine
- Part 2 : GPU and CUDA
- **Part 3 : OpenCL**
- Part 4 : Performance

Part 3 : OpenCL

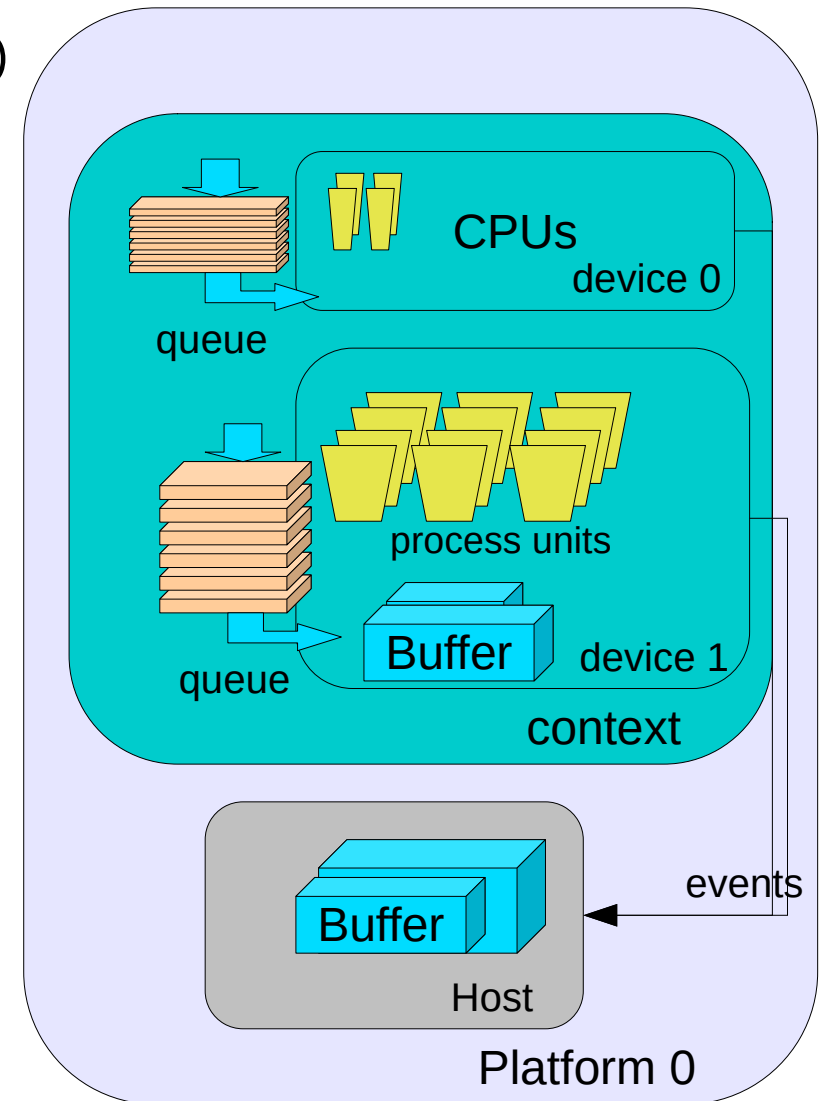
- Concepts
- STDCL
- PyOpenCL

OpenCL

- Open Standard originally supported by Apple
 - Maintained by KHRONOS Group, a companies consortium: AMD, Intel, Apple, NVIDIA, ARM, IBM...
- C-language common API and C99-language for kernels
 - Execution Target to CPUs, GPUs or CELL/BE
 - Multiple contexts management (GPUs, GPUs + CPUs, ...)
 - Explicit memory transfer API
 - Synchronous or asynchronous execution

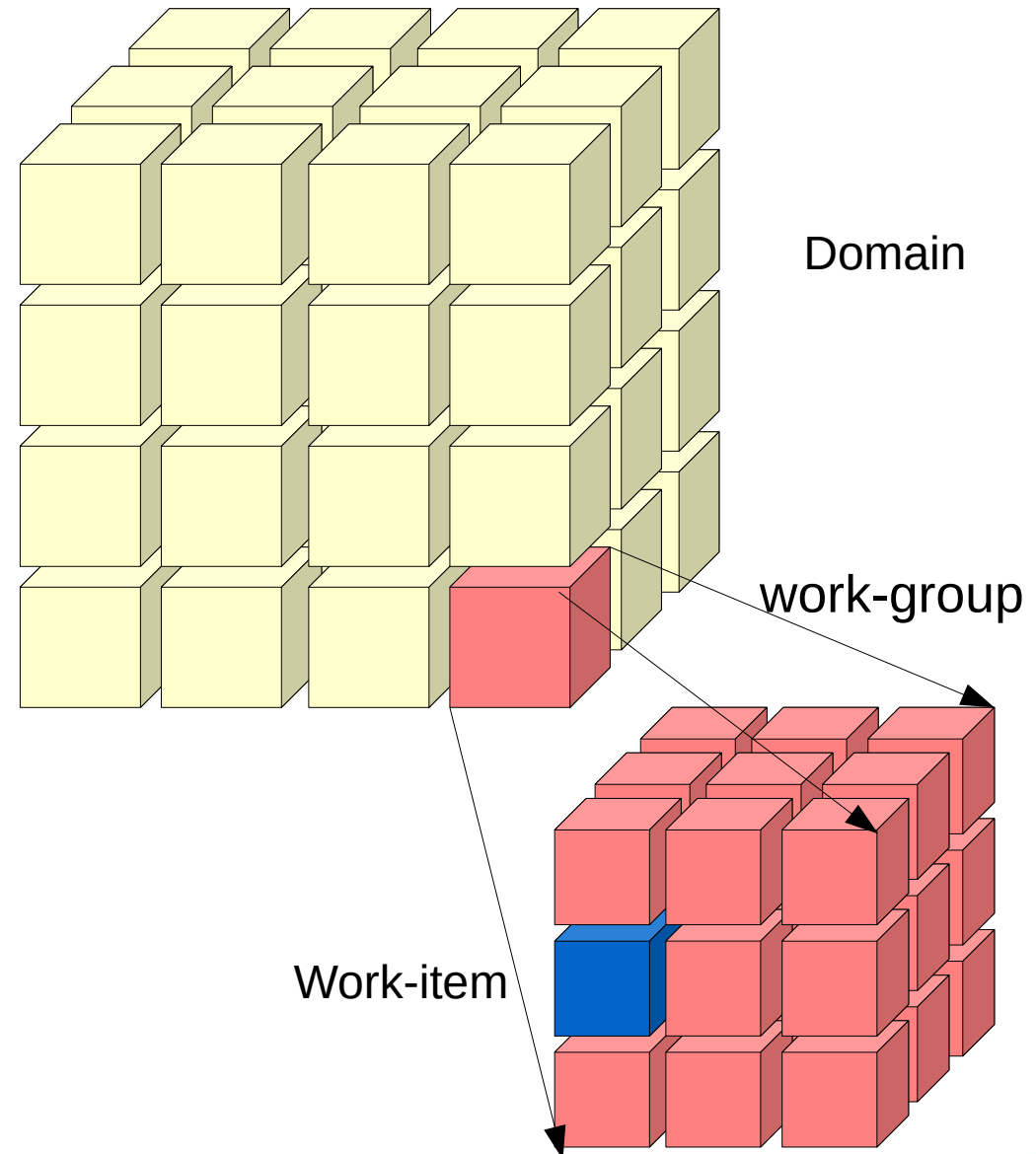
Material Abstraction (almost)

- *context* : collection of compute devices (*devices*)
 - GPUs
 - CPUs (cores + SSE)
 - CELL BE
- *program* : Collection of *kernels*
 - *Kernel* : function run into a *process unit*
- Memories :
 - *Buffers* :1D
 - *Images* : 2D or 3D
- A commands *queue* is associated with every *device*
 - Commands :
 - Run kernels
 - Copy memories
- *events* : handle synchronization between *devices*



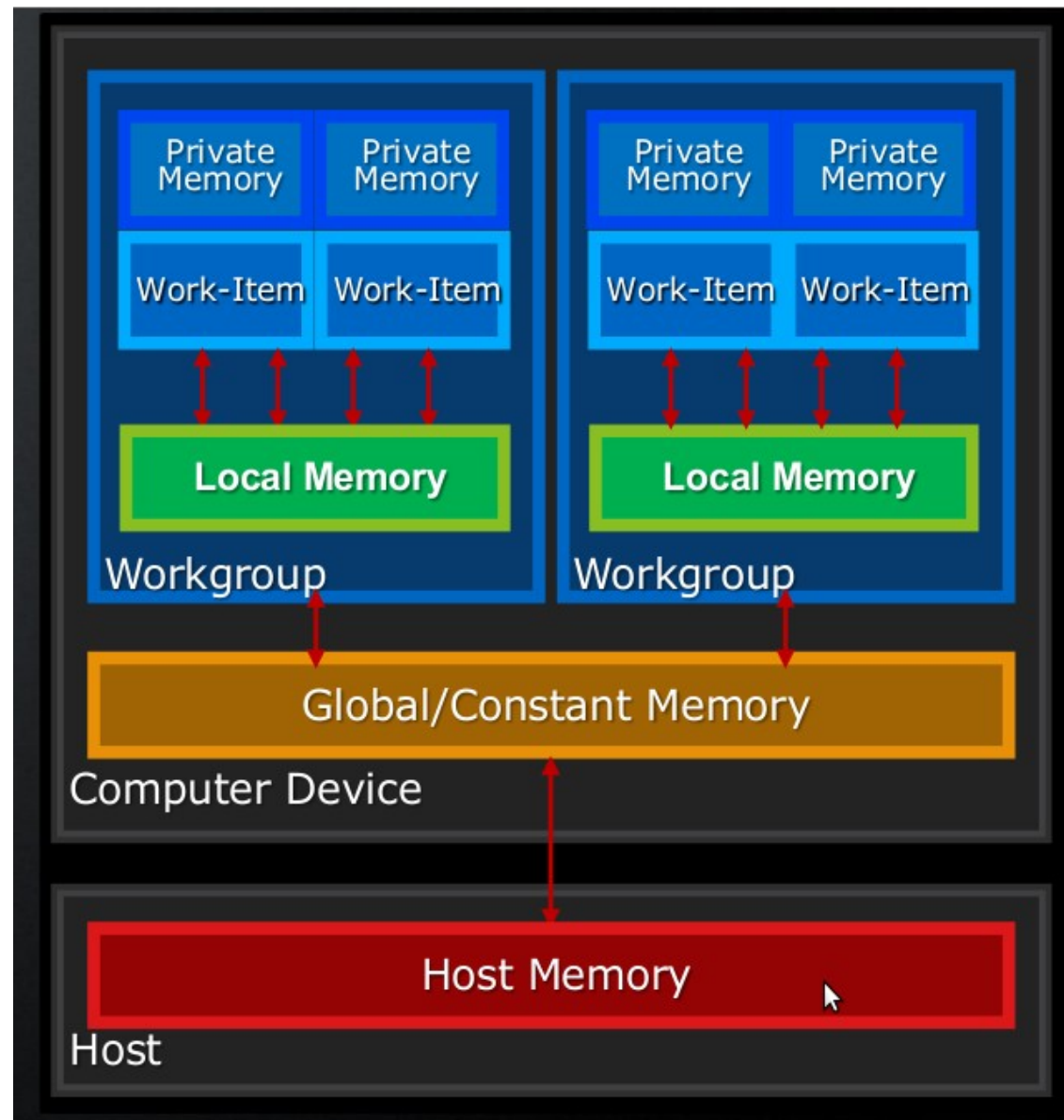
Compute domain

- Compute *domain* with N-dimension ($N_{max}=3$)
 - *global dimensions*
 - Composed by *work-groups*
 - Every *work-group* has *Local dimensions* and is composed by *Work-items*



Memory model

- *private memory* to every *work-item* (registers)
- *Local memory* is shared between *work-items* in same *work-group*
- *Global memory* of device
- *Host memory* (PC RAM)



OpenCL helloworld

```
// OpenCL Kernel Code
__kernel void
vectorAdd(__global const float * a,
          __global const float * b,
          __global float * c)
{
    // Vector element index
    int nIndex = get_global_id(0);
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

OpenCL helloworld

```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0,
CL_DEVICE_TYPE_GPU,
0, 0, 0);
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
0, 0, &nContextDescriptorSize);
cl_device_id * aDevices =
malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
nContextDescriptorSize, aDevices, 0);
```

OpenCL helloworld

```
// create a command queue for first device the
context reported
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext,
aDevices[0], 0, 0);
// create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1,

sProgramSource, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0) ;
// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "vectorAdd",
0);
```

OpenCL helloworld

```
// allocate host vectors
float * A = new float[n];
float * B = new float[n];
float * C = new float[n] ;
// allocate device memory
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             n * sizeof(cl_float),
                             A,
                             0) ;
hDeviceMemB = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             n * sizeof(cl_float),
                             B,
                             0) ;
hDeviceMemC = clCreateBuffer(hContext,
                             CL_MEM_WRITE_ONLY,
                             n * sizeof(cl_float),
                             0, 0);
```

OpenCL helloworld

```
// setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void
*)&hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void
*)&hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void
*)&hDeviceMemC);
// execute kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                        &n, 0, 0, 0, 0);
// copy results from device back to host
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0,
                    n * sizeof(cl_float), C, 0, 0, 0);
```

OpenCL helloworld

```
delete[] A;  
delete[] B;  
delete[] C;  
clReleaseMemObj (hDeviceMemA) ;  
clReleaseMemObj (hDeviceMemB) ;  
clReleaseMemObj (hDeviceMemC) ;
```

OpenCL kernel with local memory

```
__kernel void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height,
    __local float *a_local)
{
    int base_idx_a    =
        get_group_id(0) * get_local_size(0) +
        get_group_id(1) * a_width;
    int base_idx_a_t =
        get_group_id(1) * get_local_size(1) +
        get_group_id(0) * a_height;

    int glob_idx_a    = base_idx_a + get_local_id(0) + a_width *
get_local_id(1);
    int glob_idx_a_t = base_idx_a_t + get_local_id(0) + a_height *
get_local_id(1);

    a_local[get_local_id(1)*get_local_size(1)+get_local_id(0)] = a[glob_idx_a];

    barrier(CLK_LOCAL_MEM_FENCE);

    a_t[glob_idx_a_t] =
a_local[get_local_id(0)*get_local_size(0)+get_local_id(1)];
}
```

Boost.Compute

- A thin C++ wrapper over the OpenCL API
- STL-like interface providing common algorithms
 - transform(), accumulate(), sort(), ...
- Standard C++
- Native C99 Kernels also
- header-only library => simple compilation:
 - `g++ -I/path/to/compute/include sort.cpp -lOpenCL`
- Open Source
- Accepted as an official Boost library in January 2015

Example

```
#include <boost/compute/container/vector.hpp>
namespace compute = boost::compute;

...
int main()
{
    ...
    // get default device and setup context
    compute::device device = compute::system::default_device();
    compute::context context(device);
    compute::command_queue queue(context, device);
    // declare host vectors for data
    std::vector<float> host_A(data_size);
    std::vector<float> host_B(data_size);
    std::vector<float> host_C(data_size);
    ...
    // create a vectors on the device
    compute::vector<float> device_A(host_A.size(), context);
    compute::vector<float> device_B(host_B.size(), context);
    compute::vector<float> device_C(host_C.size(), context);
    // transfer data from the host to the device
    compute::copy(host_A.begin(), host_A.end(), device_A.begin(), queue);
    compute::copy(host_B.begin(), host_B.end(), device_B.begin(), queue);
    // define the kernel
    const char source[] = BOOST_COMPUTE_STRINGIZE_SOURCE(
        __kernel void add(__global const float * a, __global const float * b, __global float *c)
        {
            int nIndex = get_global_id(0);
            c[nIndex] = a[nIndex] + b[nIndex];
        });
    // create and build the add program
    compute::program program = compute::program::build_with_source(source, context);
    // get the add kernel
    compute::kernel add = program.create_kernel("add");
    add.set_arg(0, device_A);
    add.set_arg(1, device_B);
    add.set_arg(2, device_C);
    // launch the kernel
    queue.enqueue_nd_range_kernel(add, dim(0), dim(data_size), dim(block_size));
    queue.finish();
    // copy values back to the host
    compute::copy(device_C.begin(), device_C.end(), host_C.begin(), queue);
    ...
}
```

Example

Host Data as
STL vector

```
#include <boost/compute/container/vector.hpp>
namespace compute = boost::compute;

...
int main()
{
    ...
    // get default device and setup context
    compute::device device = compute::system::default_device();
    compute::context context(device);
    compute::command_queue queue(context, device);
    // declare host vectors for data
    std::vector<float> host_A(data_size);
    std::vector<float> host_B(data_size);
    std::vector<float> host_C(data_size);
    ...
    // create a vectors on the device
    compute::vector<float> device_A(host_A.size(), context);
    compute::vector<float> device_B(host_B.size(), context);
    compute::vector<float> device_C(host_C.size(), context);
    // transfer data from the host to the device
    compute::copy(host_A.begin(), host_A.end(), device_A.begin(), queue);
    compute::copy(host_B.begin(), host_B.end(), device_B.begin(), queue);
    // define the kernel
    const char source[] = BOOST_COMPUTE_STRINGIZE_SOURCE(
        __kernel void add(__global const float * a, __global const float * b, __global float * c)
        {
            int nIndex = get_global_id(0);
            c[nIndex] = a[nIndex] + b[nIndex];
        }
    );
    // create and build the add program
    compute::program program = compute::program::build_with_source(source, context);
    // get the add kernel
    compute::kernel add = program.create_kernel("add");
    add.set_arg(0, device_A);
    add.set_arg(1, device_B);
    add.set_arg(2, device_C);
    // launch the kernel
    queue.enqueue_nd_range_kernel(add, dim(0), dim(data_size), dim(block_size));
    queue.finish();
    // copy values back to the host
    compute::copy(device_C.begin(), device_C.end(), host_C.begin(), queue);
    ...
}
```

Example

Device Data declaration

```
#include <boost/compute/container/vector.hpp>
namespace compute = boost::compute;

...
int main()
{
    ...
    // get default device and setup context
    compute::device device = compute::system::default_device();
    compute::context context(device);
    compute::command_queue queue(context, device);
    // declare host vectors for data
    std::vector<float> host_A(data_size);
    std::vector<float> host_B(data_size);
    std::vector<float> host_C(data_size);
    ...
    // create a vectors on the device
    compute::vector<float> device_A(host_A.size(), context);
    compute::vector<float> device_B(host_B.size(), context);
    compute::vector<float> device_C(host_C.size(), context);
    // transfer data from the host to the device
    compute::copy(host_A.begin(), host_A.end(), device_A.begin(), queue);
    compute::copy(host_B.begin(), host_B.end(), device_B.begin(), queue);
    // define the kernel
    const char source[] = BOOST_COMPUTE_STRINGIZE_SOURCE(
        __kernel void add(__global const float * a, __global const float * b, __global float *c)
        {
            int nIndex = get_global_id(0);
            c[nIndex] = a[nIndex] + b[nIndex];
        }
    );
    // create and build the add program
    compute::program program = compute::program::build_with_source(source, context);
    // get the add kernel
    compute::kernel add = program.create_kernel("add");
    add.set_arg(0, device_A);
    add.set_arg(1, device_B);
    add.set_arg(2, device_C);
    // launch the kernel
    queue.enqueue_nd_range_kernel(add, dim(0), dim(data_size), dim(block_size));
    queue.finish();
    // copy values back to the host
    compute::copy(device_C.begin(), device_C.end(), host_C.begin(), queue);
    ...
}
```

Example

```
#include <boost/compute/container/vector.hpp>
namespace compute = boost::compute;

...
int main()
{
    ...
    // get default device and setup context
    compute::device device = compute::system::default_device();
    compute::context context(device);
    compute::command_queue queue(context, device);
    // declare host vectors for data
    std::vector<float> host_A(data_size);
    std::vector<float> host_B(data_size);
    std::vector<float> host_C(data_size);
    ...
    // create a vectors on the device
    compute::vector<float> device_A(host_A.size(), context);
    compute::vector<float> device_B(host_B.size(), context);
    compute::vector<float> device_C(host_C.size(), context);
    // transfer data from the host to the device
    compute::copy(host_A.begin(), host_A.end(), device_A.begin(),
queue);
    compute::copy(host_B.begin(), host_B.end(), device_B.begin(),
queue);
    // define the kernel
    const char source[] = BOOST_COMPUTE_STRINGIZE_SOURCE(
        __kernel void add(__global const float * a, __global const float * b, __global float *c)
        {
            int nIndex = get_global_id(0);
            c[nIndex] = a[nIndex] + b[nIndex];
        });
    // create and build the add program
    compute::program program = compute::program::build_with_source(source, context);
    // get the add kernel
    compute::kernel add = program.create_kernel("add");
    add.set_arg(0, device_A);
    add.set_arg(1, device_B);
    add.set_arg(2, device_C);
    // launch the kernel
    queue.enqueue_nd_range_kernel(add, dim(0), dim(data_size), dim(block_size));
    queue.finish();
    // copy values back to the host
    compute::copy(device_C.begin(), device_C.end(), host_C.begin(), queue);
    ...
}
```

Data transfer
from host
to device

Example

Kernel code
as string

```
#include <boost/compute/container/vector.hpp>
namespace compute = boost::compute;
...
int main()
{
    ...
    // get default device and setup context
    compute::device device = compute::system::default_device();
    compute::context context(device);
    compute::command_queue queue(context, device);
    // declare host vectors for data
    std::vector<float> host_A(data_size);
    std::vector<float> host_B(data_size);
    std::vector<float> host_C(data_size);
    ...
    // create a vectors on the device
    compute::vector<float> device_A(host_A.size(), context);
    compute::vector<float> device_B(host_B.size(), context);
    compute::vector<float> device_C(host_C.size(), context);
    // transfer data from the host to the device
    compute::copy(host_A.begin(), host_A.end(), device_A.begin(), queue);
    compute::copy(host_B.begin(), host_B.end(), device_B.begin(), queue);
    // define the kernel
    const char source[] = BOOST_COMPUTE_STRINGIZE_SOURCE(
        __kernel void add(__global const float * a,
                          __global const float * b,
                          __global float *c)
        {
            int nIndex = get_global_id(0);
            c[nIndex] = a[nIndex] + b[nIndex];
        });
    // create and build the add program
    compute::program program = compute::program::build_with_source(source, context);
    // get the add kernel
    compute::kernel add = program.create_kernel("add");
    add.set_arg(0, device_A);
    add.set_arg(1, device_B);
    add.set_arg(2, device_C);
    // launch the kernel
    queue.enqueue_nd_range_kernel(add, dim(0), dim(data_size), dim(block_size));
    queue.finish();
    // copy values back to the host
    compute::copy(device_C.begin(), device_C.end(), host_C.begin(), queue);
    ...
}
```

Example

```
#include <boost/compute/container/vector.hpp>
namespace compute = boost::compute;

...
int main()
{
    ...
    // get default device and setup context
    compute::device device = compute::system::default_device();
    compute::context context(device);
    compute::command_queue queue(context, device);
    // declare host vectors for data
    std::vector<float> host_A(data_size);
    std::vector<float> host_B(data_size);
    std::vector<float> host_C(data_size);
    ...
    // create a vectors on the device
    compute::vector<float> device_A(host_A.size(), context);
    compute::vector<float> device_B(host_B.size(), context);
    compute::vector<float> device_C(host_C.size(), context);
    // transfer data from the host to the device
    compute::copy(host_A.begin(), host_A.end(), device_A.begin(), queue);
    compute::copy(host_B.begin(), host_B.end(), device_B.begin(), queue);
    // define the kernel
    const char source[] = BOOST_COMPUTE_STRINGIZE_SOURCE(
        __kernel void add(__global const float * a, __global const float * b, __global float *c)
        {
            int nIndex = get_global_id(0);
            c[nIndex] = a[nIndex] + b[nIndex];
        });
    // create and build the add program
    compute::program program = compute::program::build_with_source(source,
context);
    // get the add kernel
    compute::kernel add = program.create_kernel("add");
    add.set_arg(0, device_A);
    add.set_arg(1, device_B);
    add.set_arg(2, device_C);
    // launch the kernel
    queue.enqueue_nd_range_kernel(add, dim(0), dim(data_size), dim(block_size));
    queue.finish();
    // copy values back to the host
    compute::copy(device_C.begin(), device_C.end(), host_C.begin(), queue);
    ...
}
```

Build and prepare
the kernel
(can do once,
not at every call)

Example

```
#include <boost/compute/container/vector.hpp>
namespace compute = boost::compute;

...
int main()
{
    ...
    // get default device and setup context
    compute::device device = compute::system::default_device();
    compute::context context(device);
    compute::command_queue queue(context, device);
    // declare host vectors for data
    std::vector<float> host_A(data_size);
    std::vector<float> host_B(data_size);
    std::vector<float> host_C(data_size);
    ...
    // create a vectors on the device
    compute::vector<float> device_A(host_A.size(), context);
    compute::vector<float> device_B(host_B.size(), context);
    compute::vector<float> device_C(host_C.size(), context);
    // transfer data from the host to the device
    compute::copy(host_A.begin(), host_A.end(), device_A.begin(), queue);
    compute::copy(host_B.begin(), host_B.end(), device_B.begin(), queue);
    // define the kernel
    const char source[] = BOOST_COMPUTE_STRINGIZE_SOURCE(
        __kernel void add(__global const float * a, __global const float * b, __global float *c)
        {
            int nIndex = get_global_id(0);
            c[nIndex] = a[nIndex] + b[nIndex];
        });
    // create and build the add program
    compute::program program = compute::program::build_with_source(source, context);
    // get the add kernel
    compute::kernel add = program.create_kernel("add");
    add.set_arg(0, device_A);
    add.set_arg(1, device_B);
    add.set_arg(2, device_C);
    // launch the kernel
    queue.enqueue_nd_range_kernel(add, dim(0), dim(data_size),
    dim(block_size));
    queue.finish();
    // copy values back to the host
    compute::copy(device_C.begin(), device_C.end(), host_C.begin(), queue);
    ...
}
```

Call the kernel

Example

```
#include <boost/compute/container/vector.hpp>
namespace compute = boost::compute;

...
int main()
{
    ...
    // get default device and setup context
    compute::device device = compute::system::default_device();
    compute::context context(device);
    compute::command_queue queue(context, device);
    // declare host vectors for data
    std::vector<float> host_A(data_size);
    std::vector<float> host_B(data_size);
    std::vector<float> host_C(data_size);
    ...
    // create a vectors on the device
    compute::vector<float> device_A(host_A.size(), context);
    compute::vector<float> device_B(host_B.size(), context);
    compute::vector<float> device_C(host_C.size(), context);
    // transfer data from the host to the device
    compute::copy(host_A.begin(), host_A.end(), device_A.begin(), queue);
    compute::copy(host_B.begin(), host_B.end(), device_B.begin(), queue);
    // define the kernel
    const char source[] = BOOST_COMPUTE_STRINGIZE_SOURCE(
        __kernel void add(__global const float * a, __global const float * b, __global float *c)
        {
            int nIndex = get_global_id(0);
            c[nIndex] = a[nIndex] + b[nIndex];
        });
    // create and build the add program
    compute::program program = compute::program::build_with_source(source, context);
    // get the add kernel
    compute::kernel add = program.create_kernel("add");
    add.set_arg(0, device_A);
    add.set_arg(1, device_B);
    add.set_arg(2, device_C);
    // launch the kernel
    queue.enqueue_nd_range_kernel(add, dim(0), dim(data_size), dim(block_size));
    queue.finish();
    // copy values back to the host
    compute::copy(device_C.begin(), device_C.end(),
host_C.begin(), queue);
    ...
}
```

Copy back
The result to
host

Boost.Compute algorithms

```
boost::compute::vector<float> vec;  
  
...  
  
reduce (vec.begin(), vec.end(),  
&result, plus<float>());
```

http://boostorg.github.io/compute/boost_compute/reference.html#boost_compute.reference.api_overview.algorithms

Boot.Compute custom function

```
BOOST_COMPUTE_FUNCTION(int, add_four, (int x),  
{  
    return x + 4;  
});
```

```
boost::compute::transform(input.begin(),  
input.end(), output.begin(), add_four, queue);
```

Custom function and algorithms generate kernels
compiled online

Why Python ?

- General purpose language
 - Object Oriented
 - Dynamic typing
 - Automatic memory management
- High level :
 - Interpreted
 - High performance natives libraries:
 - Numpy (faster than others C linalg packs)
 - ... (lot of others)
- Multi-OS
- OpenSource active community
- <http://docs.python.org/tutorial/introduction.html>

Why PyOpenCL ?

- OpenCL interface for Python language
- Simple interface but complete
- Plus high level functions :
 - Map/ Reduce
 - Random generator
 - FFT via pyfft
- OpenSource licence type MIT

<http://document.tician.de/pyopenccl/>

Example

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf, is_blocking=True)
print (la.norm(a_plus_b - (a+b)))
```

Example

```
import pyopenccl as cl
```

```
import numpy
```

```
import numpy.linalg as la
```

```
a = numpy.random.rand(50000).astype(numpy.float32)
```

```
b = numpy.random.rand(50000).astype(numpy.float32)
```

```
ctx = cl.create_some_context()
```

```
queue = cl.CommandQueue(ctx)
```

```
mf = cl.mem_flags
```

```
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
```

```
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
```

```
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
```

```
prg = cl.Program(ctx, """
```

```
__kernel void sum(__global const float *a,
```

```
__global const float *b, __global float *c)
```

```
{
```

```
    int gid = get_global_id(0);
```

```
    c[gid] = a[gid] + b[gid];
```

```
}
```

```
""").build()
```

```
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
```

```
a_plus_b = numpy.empty_like(a)
```

```
cl.enqueue_copy(queue, a_plus_b, dest_buf, is_blocking=True)
```

```
print (la.norm(a_plus_b - (a+b)))
```

load library in python

Example

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY, size=a.shape[0]*a.itemsize)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY, size=b.shape[0]*b.itemsize)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, size=a.shape[0]*a.itemsize)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a, __global const float *b, __global float *c)
    {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
    }
""").build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf, is_blocking=True)
print (la.norm(a_plus_b - (a+b)))
```

Data initialization(Host side)
with numpy

Example

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, a.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf, is_blocking=True)
print (la.norm(a_plus_b - (a+b)))
```

Initialization and
interactive selecting *devices*

Example

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf, is_blocking=True)
print (la.norm(a_plus_b - (a+b)))
```

Access to associated command *queue* to current *context*

Example

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf, is_blocking=True)
print (la.norm(a_plus_b - (a+b)))
```

Allocate memory and copy data
A and B into *device*

Example

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float
    __global const float *b, __global float *c)
    {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf, is_blocking=True)
print (la.norm(a_plus_b - (a+b)))
```

Allocate memory to receive the
result C into *device*

Example

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf)
print (la.norm(a_plus_b - (a+b)))
```

Define and compile the kernel in C99 language

Example

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf, is_blocking=True)
print (la.norm(a_plus_b - (a+b)))
```

Call kernel into *device*

Example

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
        int gid = get_global_id(0);
        c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf, is_blocking=True)
print (la.norm(a_plus_b - (a+b)))
```

Transfer result to PC-RAM
(host)

References

- OpenCL Officiel web site <http://www.khronos.org/opencv/>
- Welcome to PyOpenCL's documentation! <http://documen.tician.de/pyopencv/>
- Boost.Compute A parallel computing library for C++ based on OpenCL
<https://www.iwocl.org/wp-content/uploads/iwocl-2016-boost-compute.pdf>
- Boost.Compute documentation <http://boostorg.github.io/compute/>
- Introduction to OpenCL by AMD:
 - https://developer.amd.com/wordpress/media/2012/10/1001_final.pdf
 - <http://developer.amd.com/wordpress/media/2012/10/IntroductionToOpenCL-final.pdf>
- OpenCL Programming in Detail
<http://developer.amd.com/wordpress/media/2012/10/OpenCLProginDetailDRichiev3.pdf>
- OpenCL Wrappers <https://streamhpc.com/knowledge/for-developers/opencv-wrappers/>
- OpenCL 1.2 Quick Reference Card
<https://www.khronos.org/registry/cl/sdk/1.2/docs/OpenCL-1.2-refcard.pdf>

Others portables alternatives

- AMD HIP
- OpenACC
- OpenMP 4.0
- C++ AMP
- Python Numba

AMD HIP

- C++ Heterogeneous-Compute Interface for Portability
- Allow easy convert CUDA code to run over AMD GPUs
- Common language to portable GPU programming
- Runtime and code generation tools
- HIP Kernel Language similar to CUDA
- <https://github.com/ROCm-Developer-Tools/HIP>

AMD HIP example

```
hipMalloc(&A_d, Nbytes));  
hipMalloc(&C_d, Nbytes));  
  
hipMemcpy(A_d, A_h, Nbytes, hipMemcpyHostToDevice);  
  
const unsigned blocks = 512;  
const unsigned threadsPerBlock = 256;  
  
hipLaunchKernel(vector_square,    /* compute kernel*/  
                dim3(blocks), dim3(threadsPerBlock), 0/*dynamic  
shared*/, 0/*stream*/,          /* launch config*/  
                C_d, A_d, N);    /* arguments to the compute kernel  
*/  
  
hipMemcpy(C_h, C_d, Nbytes, hipMemcpyDeviceToHost);
```

AMD HIP kernel example

```
template <typename T>
__global__ void
vector_square(T *C_d, const T *A_d, size_t N)
{
    size_t offset = (hipBlockIdx_x * hipBlockDim_x +
hipThreadId_x);
    size_t stride = hipBlockDim_x * hipGridDim_x ;

    for (size_t i=offset; i<N; i+=stride) {
        C_d[i] = A_d[i] * A_d[i];
    }
}
```

OpenACC

- Proposed by CAPS enterprise, Cray, NVIDIA and PGI at end of 2011
- Annotated C, C++ or Fortran for use GPUs and Multicores
- Compiler directives like OpenMP
- Release 2.0 in November 2013
- Release 2.5 in October 2015
- Release 2.7 in November 2018
- Convergence to OpenMP in roadmap

OpenACC compilers

- Caps enterprise with HMPP ... but now is dead
- OpenUH experimental compiler
<https://github.com/uhhpctools/openuh> no changes since Nov. 2015
- OpenARC (closed beta: need to ask for code)
<http://ft.ornl.gov/research/openarc>
- Cray Compiler with Xeon Phi and NVIDIA GPUs support but only for Cray hardware, But end of life support announced in profit of OpenMP
- PGI/NVIDIA free for academic
<https://developer.nvidia.com/openacc-toolkit> support NVIDIA Tesla GPUS
- GCC 9.1 nearly complete support OpenACC 2.5 with NVIDIA GPUs

OpenACC directives

- Parallel Regions

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
    double pi = 0.0f;
    long long i;
    #pragma acc parallel vector_length(vl)
    #pragma acc loop reduction(+:pi)
    for (i=0; i<N; i++) {
        double t= (double)((i+0.5)/N);
        pi +=4.0/(1.0+t*t);
    }
    printf("pi=%11.10f\n", pi/N);
    return 0;
}
```

OpenACC directives

- Loop clauses

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
    double pi = 0.0f;
    long long i;
    #pragma acc parallel vector_length(vl)
    #pragma acc loop reduction(+:pi)
    for (i=0; i<N; i++) {
        double t= (double)((i+0.5)/N);
        pi +=4.0/(1.0+t*t);
    }
    printf("pi=%11.10f\n", pi/N);
    return 0;
}
```

OpenACC directives

- Kernels

```
#pragma acc kernels
{
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
```

OpenACC directives

- Loop clauses

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
    double pi = 0.0f;
    long long i;
    #pragma acc parallel vector_length(vl)
    #pragma acc loop reduction(+:pi)
    for (i=0; i<N; i++) {
        double t= (double)((i+0.5)/N);
        pi +=4.0/(1.0+t*t);
    }
    printf("pi=%11.10f\n", pi/N);
    return 0;
}
```

OpenACC directives

- Data management
 - `#pragma acc data copy(A)`
...
create and copy A to device while program is inside parallel region
copy back data to host when program exit parallel region
 - `#pragma acc data copyin(A)`
same but not copy back
 - `#pragma acc data copyout(A)`
create and just copy when program exit parallel region
 - `#pragma acc data copyin(a[0:nelem])`
define size explicitly

OpenMP 4.0

- SIMD

```
void add(double *a, double *b,  
double *c, int n, )  
{  
    int i;  
    #pragma for omp simd  
    for( i= 0; i < n; i++ )  
        c[i]= a[i] + b[i];  
}
```

C++ AMP (C++ Accelerated Massive Parallelism)

- C++ 11
 - Usage of lambda function
 - Foreach primitive
 - Data proxies (array_view)
 - Or explicit data transfer (array)

C++ AMP example

```
#include <amp.h>
#include <iostream>
using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain, which is the set of threads that are
        // created.
        sum.extent,
        // Define the code to run on each thread on the accelerator.
        [=](index<1> idx) restrict(amp)
        {
            sum[idx] = a[idx] + b[idx];
        }
    );

    // Print the results. The expected output is "7, 9, 11, 13, 15".
    for (int i = 0; i < size; i++) {
        std::cout << sum[i] << "\n";
    }
}
```

C++ AMP example

```
#include <amp.h>
#include <iostream>
using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain, which is the set of threads that are
        // created.
        sum.extent,
        // Define the code to run on each thread on the accelerator.
        [=](index<1> idx) restrict(amp)
        {
            sum[idx] = a[idx] + b[idx];
        }
    );

    // Print the results. The expected output is "7, 9, 11, 13, 15".
    for (int i = 0; i < size; i++) {
        std::cout << sum[i] << "\n";
    }
}
```

C++ AMP example

```
#include <amp.h>
#include <iostream>
using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain, which is the set of threads that are
        // created.
        sum.extent,
        // Define the code to run on each thread on the accelerator.
        [=](index<1> idx) restrict(amp)
        {
            sum[idx] = a[idx] + b[idx];
        }
    );

    // Print the results. The expected output is "7, 9, 11, 13, 15".
    for (int i = 0; i < size; i++) {
        std::cout << sum[i] << "\n";
    }
}
```

Data Proxies

C++ AMP example

```
#include <amp.h>
#include <iostream>
using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain, which is the set of threads that are
        // created.
        sum.extent,
        // Define the code to run on each thread on the accelerator.
        [=](index<1> idx) restrict(amp)
        {
            sum[idx] = a[idx] + b[idx];
        }
    );

    // Print the results. The expected output is "7, 9, 11, 13, 15".
    for (int i = 0; i < size; i++) {
        std::cout << sum[i] << "\n";
    }
}
```

Kernel is a lambda function

Python Numba

- Numba is an open source JIT compiler
 - Use LLVM
- Annotate code via decorators
- SIMD Vectorization (intel SSE, AVX or AVX-512)
- GPUs supported
 - CUDA like code for NVIDIA GPUS
 - HSA like code for AMD GPUs

Numba examples

```
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@njit()
def go_fast(a): # Function is compiled to machine code when called the first time
    trace = 0
    for i in range(a.shape[0]): # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace # Numba likes NumPy broadcasting

print(go_fast(x))
```

Parallel reduction

```
from numba import njit, prange

@njit(parallel=True)
def prange_test(A):
    s = 0
    for i in prange(A.shape[0]):
        s += A[i]
    return s
```

Numba CUDA example

CUDA Kernel

```
from numba import jit, cuda

@cuda.jit
def increment_by_one(an_array):
    # Thread id in a 1D block
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    ty = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    bw = cuda.blockDim.x
    # Compute flattened index inside the array
    pos = tx + ty * bw
    if pos < an_array.size: # Check array boundaries
        an_array[pos] += 1
```

kernel call

```
threadsperblock = 32
blockspergrid = (an_array.size + (threadsperblock - 1)) // threadsperblock
increment_by_one[blockspergrid, threadsperblock](an_array)
```

Numba ROC example

HSA Kernel

```
from numba import jit, roc

@roc.jit
def increment_by_one(an_array):
    # workitem id in a 1D workgroup
    tx = roc.get_local_id(0)
    # workgroup id in a 1D grid
    ty = roc.get_group_id(0)
    # workgroup size, i.e. number of workitem per workgroup
    bw = roc.get_local_size(0)
    # Compute flattened index inside the array
    pos = tx + ty * bw
    # The above is equivalent to pos = roc.get_global_id(0)
    if pos < an_array.size: # Check array boundaries
        an_array[pos] += 1
```

kernel call

```
threadsperblock = 32
blockspergrid = (an_array.size + (threadsperblock - 1)) // threadsperblock
increment_by_one[blockspergrid, threadsperblock](an_array)
```

References

- Usage of OpenACC with gcc

<https://www.mentor.com/embedded-software/multimedia/player/accelerating-your-programs-using-openacc-2-0-with-gcc-58ecec79-ae82-46b3-9ac7-8cfea8ea7bff>

- Microsoft C++ AMP introduction

<https://msdn.microsoft.com/en-us/library/hh265136.aspx>

- Example openACC

<https://devblogs.nvidia.com/parallelforall/openacc-example-part-1/>

- Numba

<https://numba.pydata.org/>

Overview

- Part 1 : Basics on parallel machine
- Part 2 : GPU and CUDA
- Part 3 : OpenCL
- **Part 4 : Performance**

Performance

- Optimisation
 - CPU
 - GPU
- N-body problem (work in teams)
 - Optimisation CPU
 - Registers
 - SSE
 - OMP
 - Optimisation GPU
 - Registers
 - Shared Memory
 - Unroll loops

Optimisation: necessary conditions

- First: Have you code that works fine ?!
 - Check procedure / Tests
 - Reference Code (white box)
- Performance criteria
- Realistic scenarios with representative data
 - Benchmarks & Test-Beds
 - Performance Analysis (*profiling*)
 - Identify critical code

The CPU

- Memory bandwidth versus Instruction bandwidth
 - transfer CPU / RAM at 102 GB/s
 - 1 float 32 bits = 4 b $\Rightarrow 25.5 \cdot 10^9$ float transferred/s
 - CPU Computing
 - AVX float 32 bits 8 operation par cycle @ 3.1 Ghz $\Rightarrow 8 \cdot 3.1 = 24.8 \cdot 10^9$ float processed per seconds = 24.8 Gflops
 - 20 cores $\Rightarrow 20 \cdot 24.8 = 496$ Gflops
- \Rightarrow optimal application: a least 19 operations / float transferred

<http://www.pugetsystems.com/blog/2014/09/08/Memory-Performance-for-Intel-Xeon-Haswell-EP-DDR4-596/>

The GPU

- Memory bandwidth versus Instruction bandwidth
 - transfer GPU / G-RAM at 224 GB/s
 - 1 float 32 bits = 4 o $\Rightarrow 56 * 10^9$ float transferred /s
 - GPU computing
 - 3494 Gflops
- \Rightarrow optimal application : at least 62 Operations / float transferred
- But if transfers is done from GPU to RAM PC at 15.75 Go/s
- \Rightarrow Optimal application : au minimum **222** Operations / float transferred

GPU

- GPU speed versus global memory speed :
 - Unroll loop => more « useful » operation per thread
 - More threads per block => Hide the memory latency
 - But limited by register size
- Work with share memory:
 - Near faster access than register ~ 30 cycle / 400 cycles on global memory access
 - Adapt algorithms to share data locally
 - But limited by shared memory size
- Reduce threads divergence
 - Limit usage of “if”

Optimisation step-by-step sample

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 16M elements: 72 GB/s!

http://www.gpgpu.org/sc2007/SC07_CUDA_5_Optimization_Harris.pdf

Practice: N-body problems

- Application
 - Astronomy
 - Material physics
 - Mechanic simulation
 - Crowd simulation

Definitions

- Particles $n \gg 1$
- interaction evaluation for every particle

$$i \in [0, n-1]$$

$$F(i) = \sum_{j=0}^{j=n-1} f(x_i, x_j)$$

- With (simplified here) :

$$f(x_i, x_j) = x_i - x_j$$

First version CPU

- ```
inline float f(float xi, float xj)
{ return xi - xj; }
```
- ```
void calcul( float* F, float* X, unsigned int
len)
{
    for( unsigned int i = 0; i < len; i++)
    { F[i]=0.0f;
        for( unsigned int j = 0; j < len; ++j)
            F[i] = F[i] + f(X[i],X[j]);
    }
}
```

Experimentations

- Number of particles
 - $n=1024000$
- GPU :
 - NVIDIA GTX 280
- CPU :
 - bi-quadcore Intel Xeon CPU X5482 à 3.20GHz (2x4 cores).
- Execution time measurement :
 - With data transfer from and to GPU.

Results for 1 CPU single core

```
void computeF( float* F,
float*  X, const unsigned
int len)
{
    for(unsigned int i = 0;
        i<len; i++)
    {
        float f_i= 0.0f;
        for(unsigned int j = 0;
            j<len; ++j)
            f_i = f_i + X[i]-X[j];
        F[i] = f_i;
    }
}
```

n=102400	Time <i>ms</i>	<i>Mflops</i>	<i>Band width MB/s</i>
1 core CPU	19749 ,4	1061,9	

Results for 1 CPU core + SSE

```
void computeF_SSE( float* F,
float* X, const unsigned int len)
{
    for( unsigned int i = 0 ; i <
len; i+=4)
    {
        __m128 f_i, x_i;
        f_i=_mm_set_ps1(0.0f);
        x_i=_mm_load_ps(X+i);
        for( unsigned int j = 0; j < len;
++j)
        {
            __m128 x_j=_mm_set_ps1(X[j]);
            f_i = f_i + x_i - x_j;
        }
        _mm_store_ps( (F+i), f_i );
    }
}
```

n=102400	Time <i>ms</i>	Mflops	Band width <i>MB/s</i>
1 core CPU	19749,4	1061,9	
1 core CPU + SSE	4934,6	4249,9	

Results for 8 CPU cores OMP+ SSE

```
void computeF_SSE( float*  F, float*
X, const unsigned int len)
{
    for( unsigned int i = 0 ; i < len;
i+=4)
    {
        __m128 f_i, x_i;
        f_i=_mm_set_ps1(0.0f);
        x_i=_mm_load_ps(X+i);
#pragma omp parallel for
reduction(+:f_i) shared(X)
        for( unsigned int j = 0;j < len;+
+j)
        {
            __m128 x_j=_mm_set_ps1(X[j]);
            f_i = f_i + x_i - x_j;
        }
        _mm_store_ps((F+i), f_i );
    }
}
```

n=102400	Time <i>ms</i>	Mflops	Band width <i>MB/s</i>
1 core CPU	19749 ,4	1061,9	
1 core CPU + SSE	4934,6	4249,9	
8 core CPU OMP + SSE	698,8	30010,8	

Results for Version 0 CUDA

```
__global__ void
Cal_F_v0 ( float * X,
unsigned int n, float *
F)
{
    unsigned int i =
blockDim.x * blockIdx.x
+ threadIdx.x ;

    F[i] = 0.0f ;

    for(int j = 0; j < n;
j++)

        F[i] = F[i] +
f(X[i],X[j]) ;
}
```

n=102400	Time <i>ms</i>	<i>Mflops</i>	<i>Band width MB/s</i>
OMPx8 + SSE	698,8	30010,8	
Block size 32	943,3	22232,1	
Block size 64	847,3	24751,0	
Block size 128	854,7	24536,7	
Block size 256	854,0	24556,8	
Block size 512	862,6	24312,0	

Result for Version 0 CUDA

```

__global__ void
Cal_F_v0 ( float * X,
unsigned int n, float *
F)
{
    unsigned int i =
blockDim.x * blockIdx.x
+ threadIdx.x ;

    F[i] = 0.0f ;

    for(int j = 0; j < n;
j++)

        F[i] = F[i] +
f(X[i], X[j]) ;
}

```

n=102400	Time <i>ms</i>	<i>Mflops</i>	<i>Band width MB/s</i>
OMPx8 + SSE	698,8	30010,8	
Block size 32	943,3	22232,1	133392,9
Block size 64	847,3	24751,0	148506,5
Block size 128	854,7	24536,7	147220,7
Block size 256	854,0	24556,8	147341, 4
Block size 512	862,6	24312,0	145872,4

Results for Version 1 CUDA

```
__global__ void Cal_F_v0 (
float * X, unsigned int n,
float * F)
{
    unsigned int i =
blockDim.x * blockIdx.x +
threadIdx.x ;

    float f_i= 0.0f ;
    float x_i= X[i];
    for(int j = 0; j < n; j+
+)
        f_i = f_i + f(x_i,X[j])
;
    F[i]=f_i;
}
```

n=102400	Time ms	Mflops	Band width MB/s
OMPx8 + SSE	698,8	30010,8	
Block size 32	568,6	36882,73	73766,9
Block size 64	337,8	62082,65	124167,7
Block size 128	284,2	73791,41	147585,7
Block size 256	283,4	73999,72	148002,3
Block size 512	282,4	74261,76	148526,4

Memory usage in version 1

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
---	---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

X

X_i

for i=0

Memory usage in version 1

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														

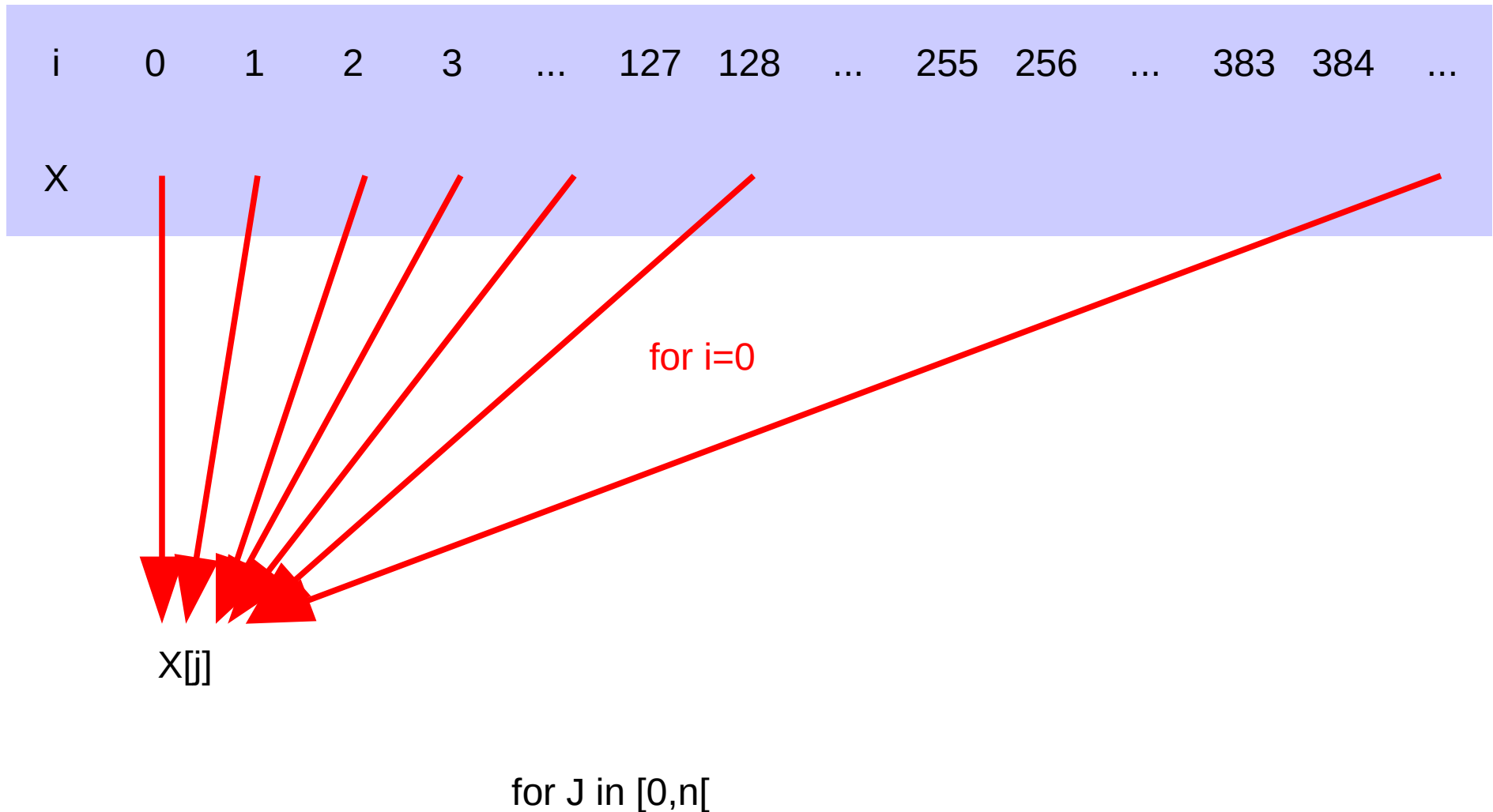


Global Memory READ access

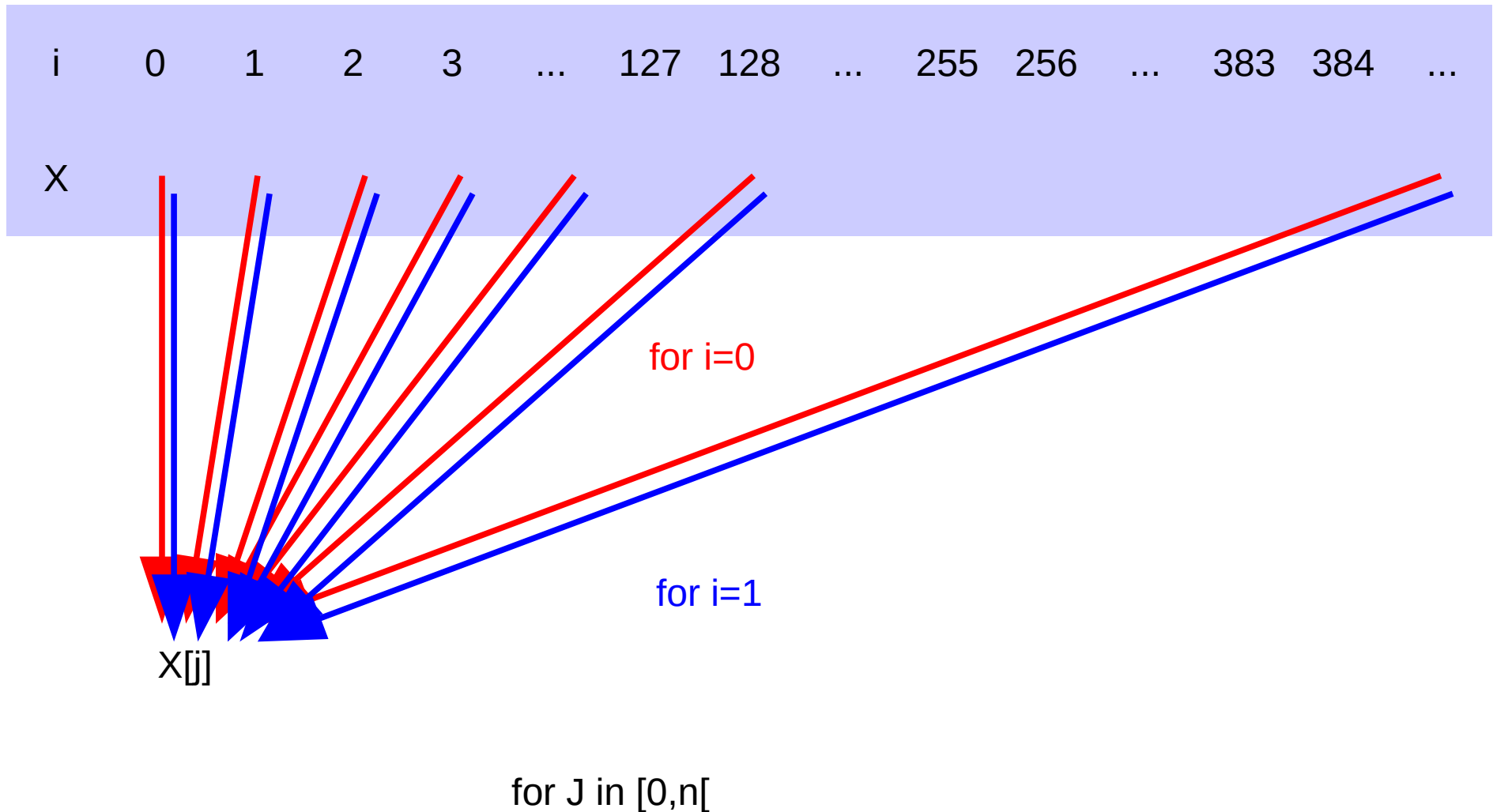
X_i

for i=0

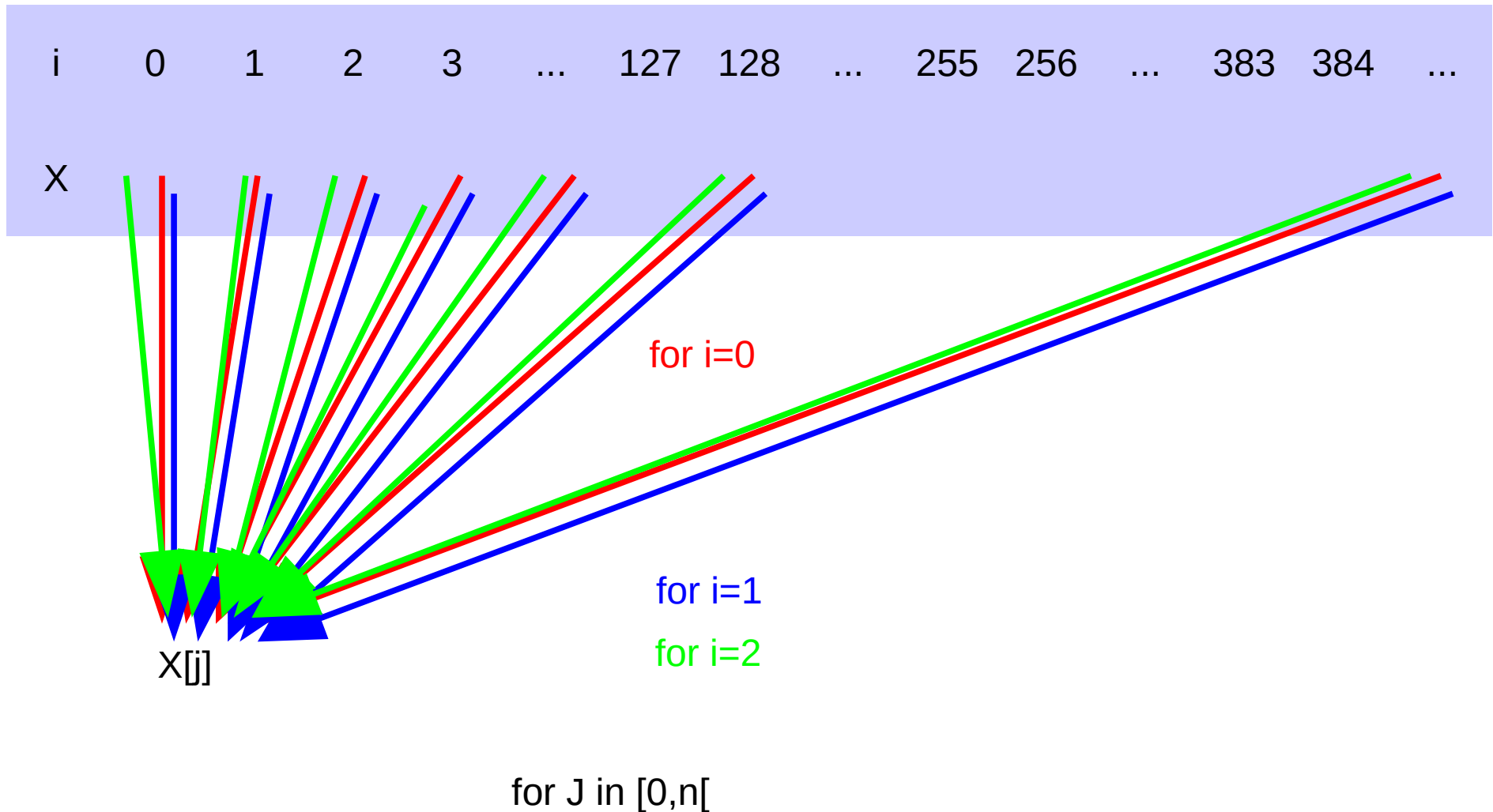
Memory usage in version 1



Memory usage in version 1



Memory usage in version 1

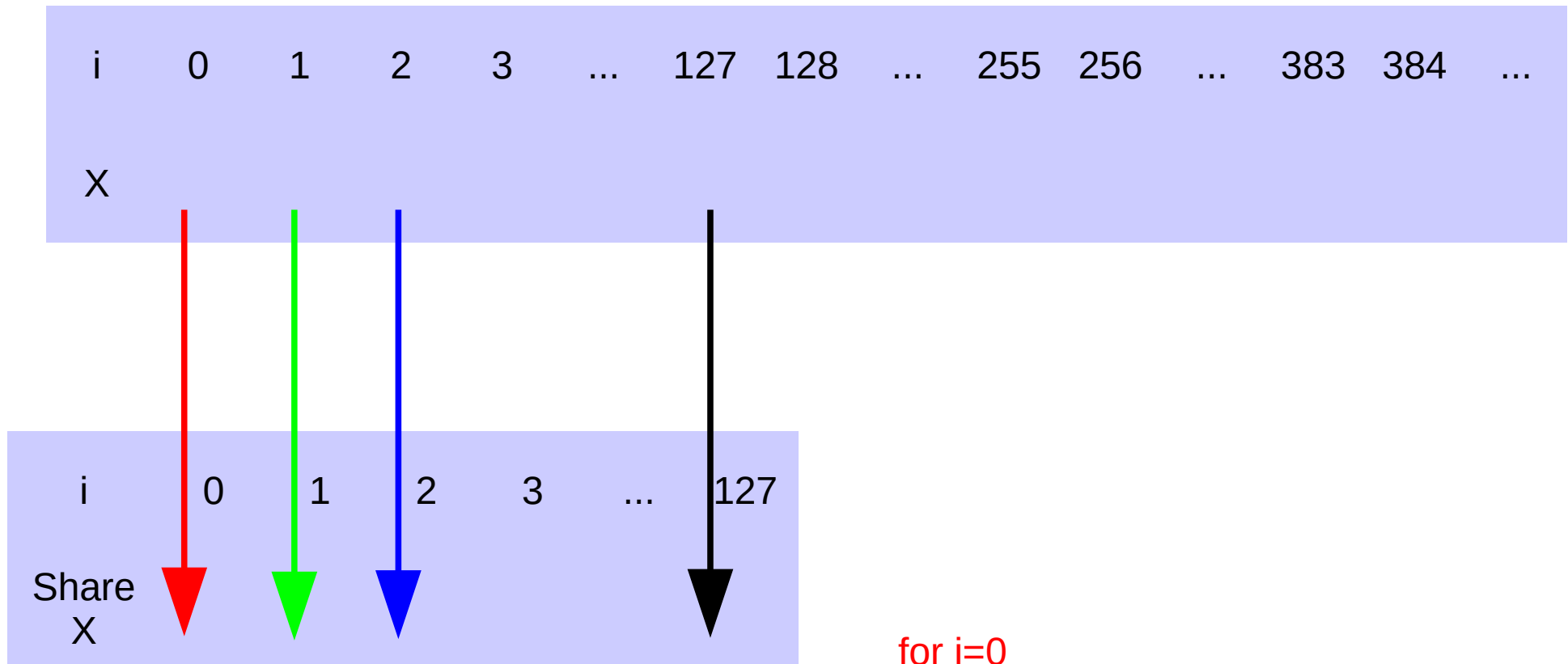


Shared memory usage

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														

i	0	1	2	3	...	127
Share						
X						

Shared memory usage



for `i=0`

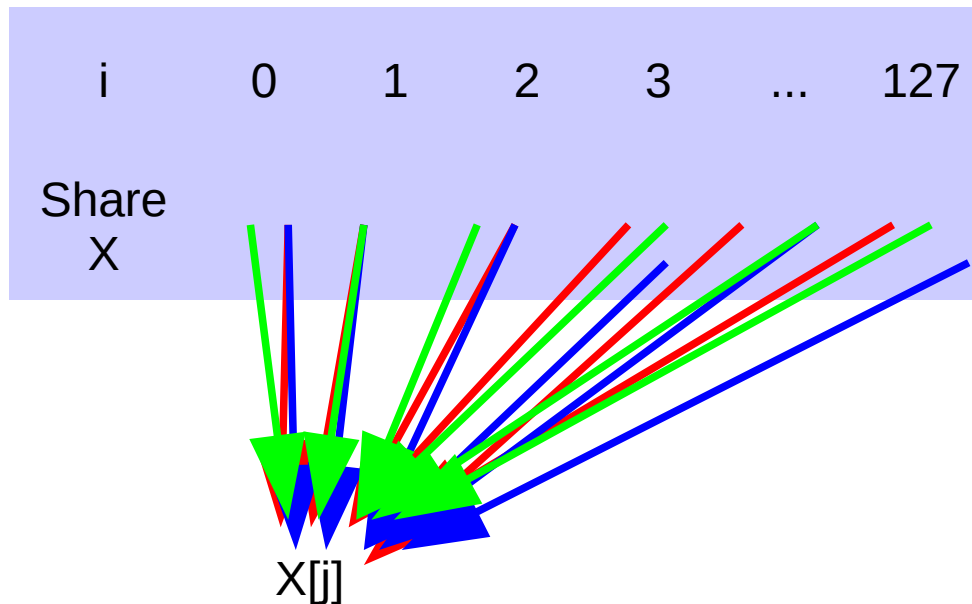
for `i=1`

for `i=2`

...
for `i=127`

Shared memory usage

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														



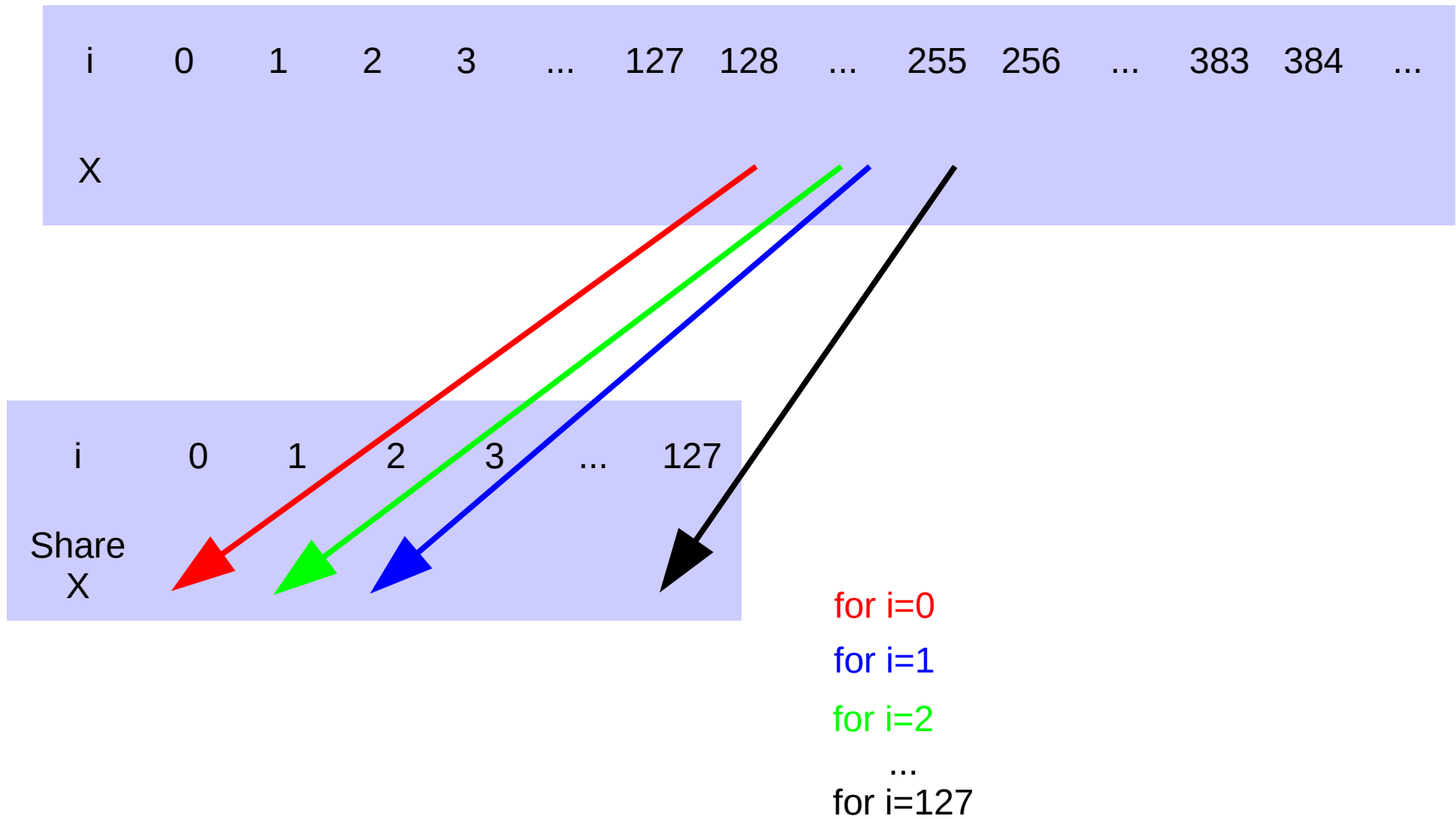
for i=0

for i=1

for i=2

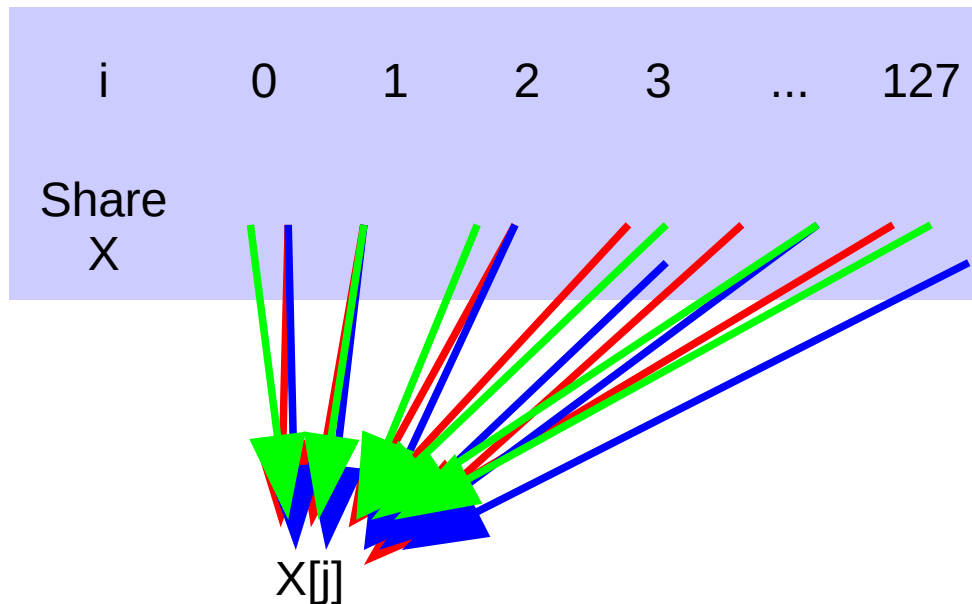
for j in [0,127]

Shared memory usage



Shared memory usage

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														



for $i=0$

for $i=1$

for $i=2$

for j in $[128, 255]$

Results for Version 2 CUDA

```
__global__ void Cal_F_v2 ( float * X,
unsigned int n, float * F)
{
    extern __shared__ float shareX[];
    unsigned int i = blockDim.x *
blockIdx.x + threadIdx.x ;
    float f_i= 0.0f ;
    float x_i= X[i];
    for(unsigned int k = 0; k < n;
k+=blockDim.x)
    {
        shareX[threadIdx.x]=X[k+threadIdx.x];
        __syncthreads();
        for(unsigned int j = 0; j <
blockDim.x; j++)
            f_i = f_i + f(x_i, shareX[j]) ;
        __syncthreads();
    }
    F[i]=f_i;
}
```

n=102400	Time <i>ms</i>	<i>Mflops</i>	<i>Band width MB/s</i>
OMPx8 + SSE	698,8	30010,8	
Block size 32	204,1	102751,2	6422,1
Block size 64	193,3	108492,1	3390,4
Block size 128	188,1	111491,3	1742,1
Block size 256	193,8	108212,2	845,4
Block size 512	193,3	108492,1	423,8

Unroll loops

CUDA code

```
for(unsigned int j = 0; j <
blockDim.x; j++)
    f_i = f_i + f(x_i,shareX[j]) ;
```

Generated PTX code

```
$Lt_0_14:
.loc 16 73 0
ld.shared.f32 %f4, [%rd12+0];
shareX+0x0
sub.f32 %f5, %f1, %f4;
add.f32 %f2, %f2, %f5;
add.u32 %r11, %r11, 1;
add.u64 %rd12, %rd12, 4;
setp.ne.u32 %p3, %r11, %r1;
@%p3 bra $Lt_0_14;
```

Unroll loops

CUDA code

```
for(unsigned int j = 0; j < blockDim.x; j++)  
    f_i = f_i + f(x_i, shareX[j]) ;
```

« productive » operations

«necessary» operations
for Loop maintain
but « unproductive »

Generated PTX code

```
$Lt_0_14:  
.loc 16 73 0  
ld.shared.f32 %f4,  
[%rd12+0]; shareX+0x0  
sub.f32 %f5, %f1, %f4;  
add.f32 %f2, %f2, %f5;  
add.u32 %r11, %r11, 1;  
add.u64 %rd12, %rd12, 4;  
setp.ne.u32 %p3, %r11, %r1;  
@%p3 bra $Lt_0_14;
```

Unroll loops

CUDA code version 2 bis

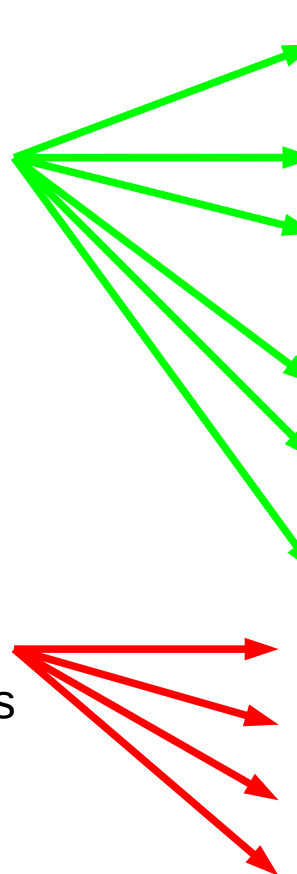
```
for(unsigned int j = 0; j < blockDim.x;  
j+=2)  
{  
    f_i = f_i + f(x_i,shareX[j]) ;  
    f_i = f_i + f(x_i,shareX[j+1]) ;  
}
```

Doubling
« productive » operations

Same number
Of «necessary» operations
for Loop maintain

Generated PTX code

```
$Lt_2_14:  
.loc 16 128 0  
ld.shared.f32    %f4,  
[%rd12+0];  
sub.f32    %f5, %f1, %f4;  
add.f32    %f2, %f2, %f5;  
.loc 16 129 0  
ld.shared.f32    %f6,  
[%rd12+4]; shareX+0x0  
sub.f32    %f7, %f1, %f6;  
add.f32    %f2, %f2, %f7;  
add.u32    %r16, %r16, 2;  
add.u64    %rd12, %rd12, 8;  
setp.lt.u32    %p3, %r16,  
%r1;  
@%p3 bra $Lt_2_14;
```



Results for Version 2 bis CUDA

```
__global__ void Cal_F_v2bis ( float *
X, unsigned int n, float * F)
{
    extern __shared__ float shareX[];
    unsigned int i = blockDim.x *
blockIdx.x + threadIdx.x ;
    float f_i= 0.0f ;
    float x_i= X[i];
    for(unsigned int k = 0; k < n;
k+=blockDim.x)
    {
shareX[threadIdx.x]=X[k+threadIdx.x];
    __syncthreads();
    for(unsigned int j = 0; j <
blockDim.x; j+=2)
        f_i = f_i + f(x_i,shareX[j]) ;
        f_i = f_i + f(x_i,shareX[j+1]) ;
    __syncthreads();
    }
    F[i]=f_i;
}
```

n=102400	Time ms	Mflops	Band width MB/s
OMPx8 + SSE	698,8	30010,8	
Block 128 Ver. 2	188,1	111491,3	1742,1
Block 128 Ver. 2 bis	138,0	151956,5	2374,4

Results for Version 3 CUDA

(automatic unroll the whole loop)

```
__global__ void Cal_F_v3 ( float * X,
unsigned int n, float * F)
{
    extern __shared__ float shareX[];
    unsigned int i = blockDim.x *
blockIdx.x + threadIdx.x ;
    float f_i= 0.0f ;
    float x_i= X[i];
    for(unsigned int k = 0; k < n;
k+=blockDim.x)
    {
shareX[threadIdx.x]=X[k+threadIdx.x];
    __syncthreads();
    #pragma unroll 128
    for(unsigned int j = 0; j <
blockDim.x; j++)
        f_i = f_i + f(x_i,shareX[j]) ;
    __syncthreads();
    }
    F[i]=f_i;
}
```

n=102400	Time <i>ms</i>	<i>Mflops</i>	<i>Band width MB/s</i>
OMPx8 + SSE	698,8	30010,8	
Block 128 Ver. 2	188,1	111491,3	1742,1
Block 128 Ver. 2 bis	138,0	151956,5	2374,4
Block 128 Ver. 3	90,6	231473,7	3616,8