Introduction au calcul haute performance

Master GENIAL (Génie Informatique en Alternance), Université de Paris

Pierre Kestener

CEA-Saclay, DRF/IRFU/DEDIP/LILAS
Laboratoire d'ingénieurie Logicielle pour les applications scientifiques

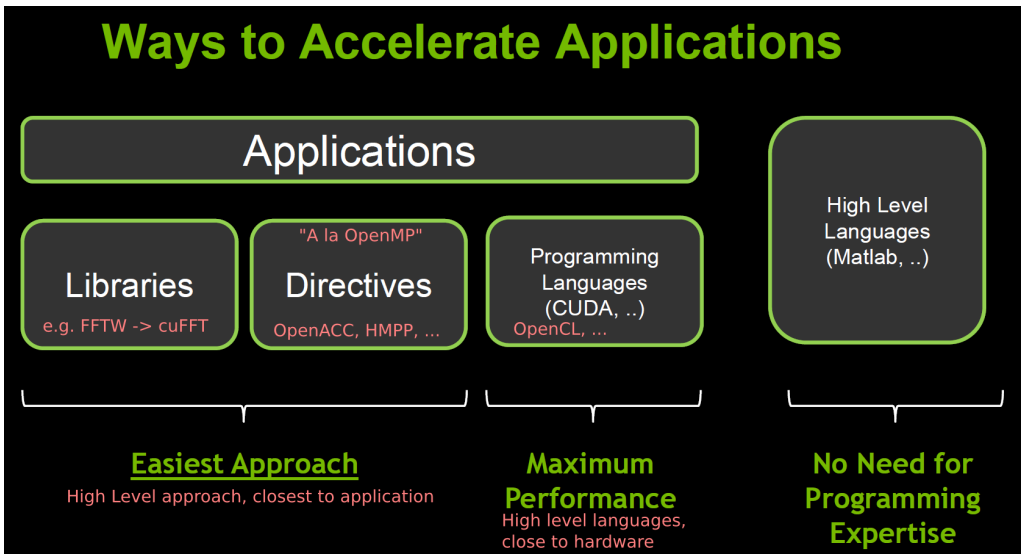
Université de Paris, Grands Moulins, Mai 2021

INFORMATIQUE
Sciences
Université de Paris

Irfu - CEA Saclay
Institut de recherche sur les lois fondamentales de l'Univers

Content

- **Short historical view:** from graphics processor to GPU accelerator
- **Main differences between CPU and GPU**
- **CUDA Hardware : differences with CPU**
- **CUDA software abstraction / programming model**
  - **SIMT - Single Instruction Multiple Thread**
  - **Memory hierarchy**
- **Performance portability ⇒ Kokkos library**



reference: Axel Koehler, NVIDIA, 2012

website associated to this class :

https://gaufre.informatique.univ-paris-diderot.fr/kestener/m2-genial-hpc

Navigation

- **Short historical view**
  - **Computer animation, Supercomputers**
- **CUDA programming model**
  - **Helloworld in CUDA/C++,**
  - **CPU/GPU comparison, Streaming Multiprocessor (SM),**
  - **CUDA software/hardware,**
  - **bandwidth,**
  - **Cooperative Thread Array / PTX,**
  - **CUDA C/C++,**
  - **CUDA example code,**
  - **profiling, roofline**
- **Additionnal subjects**
  - **books / recommended reading**
  - **lstopo / hwloc**
  - **cuda error handling**
  - **cuda python, OpenAcc**
  - **exascale and hardware trends**
  - **Modern C++ for GPU programming** (performance portability with C++/Kokkos library)

Historique

Plan

- Mini-rappels calcul parallèle / notion de concurrence et parallélisme (devinette de cuisine)
- Quand sont apparus les premiers processeurs graphiques (GPU) ? Pourquoi ?
- Évolution de l'utilisation des GPU: des applications graphiques (jeux video, visualisation) vers le calcul scientifique généraliste (GPGPU : General Purpose GPU computing)
- Les GPU et le calcul haute performance; quels sont les domaines d'application ? Introduction de l'architecture CUDA en 2007

Fun fact on computer animation

- before the 1990's, **parallel computers** were **rare** and available for only the most critical problems
- **Toy Story (1995)** : **first completely computer-generated feature-length film**, processed on a "renderfarm" consisting of 117 Sun(™) SPARC station(™) @100MHz workstations. Computing the 114000 frames (77 minutes) required 800000 computer hours. Each frame consists in 300 MBytes of data (one hard-disk in 1995).
- Computer animation (and rendering, i.e. the process of generating an image from a model or scene description) is where **parallel computing / HPC** and **graphics computing / GPU** meets.
- **Software for off-line rendering** : e.g. RenderMan (`http://renderman.pixar.com`) by Pixar, from modelling to rendering
- **Hardware rendering** : OpenGL low-level API used in real-time rendering (i.e. done in dedicated hardware like **GPU**), gaming industry

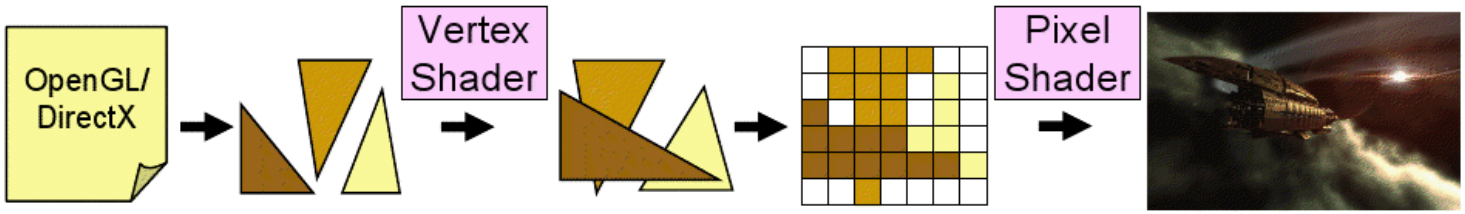
Parallel computing - Rendering - GPU

State of the art in computer generated feature-length movie: Big Hero 6

http://www.tomshardware.fr/articles/big-hero-6-technologie,1-54655.html

- 55000 CPU-core
- 180 days ⇒ **200 Millions computing hours**
- **It's roughly half of CURIE resources during 6 months !! (CURIE is one the most powerfull supercomputer in Europe.)**

GPU evolution: before 2006, i.e. CUDA



- GPU == dedicated hardware for **graphics *pipeline***
- GPU main function : **off-load graphics task from CPU to GPU**
- GPU: **dedicated hardware for specialized tasks**
- **"All processors aspire to be general-purpose."**
  – Tim Van Hook, Graphics Hardware 2001
- 2000's : ***shaders*** (programmable functionalities in the graphics *pipeline*) : low-level vendor-dependent assembly, high-level Cg, HLSL, etc...
- **Legacy GPGPU** (before CUDA, ~ 2004), premises of GPU computing

The Evolution of GPUs for General Purpose Computing,
par Ian Buck
http://www.nvidia.com/content/GTC-2010/pdfs/2275_GTC2010.pdf

Floating-point computation capabilities in GPU ?

Floating point computations capability implemented in GPU hardware

- **IEEE754 standard** written in mid-80s
- Intel 80387 : first floating-point coprocessor IEEE754-compatible
- Value = $(-1)^S \times M \times 2^E$, denormalized, infinity, NaN; rounding algorithms quite complex to handle/implement
- FP16 in 2000
- **FP32 in 2003-2004** : simplified IEEE754 standard, float point rounding are complex and costly in terms of transistors count,
- **CUDA 2007** : rounding computation fully implemented for + and * in 2007, denormalised number not completed implemented
- **CUDA Fermi : 2010** : 4 mandatory IEEE rounding modes; Subnormals at full-speed (Nvidia GF100)
- links:
  http://homepages.dcc.ufmg.br/~sylvain.collange/talks/raim11_scollange.pdf

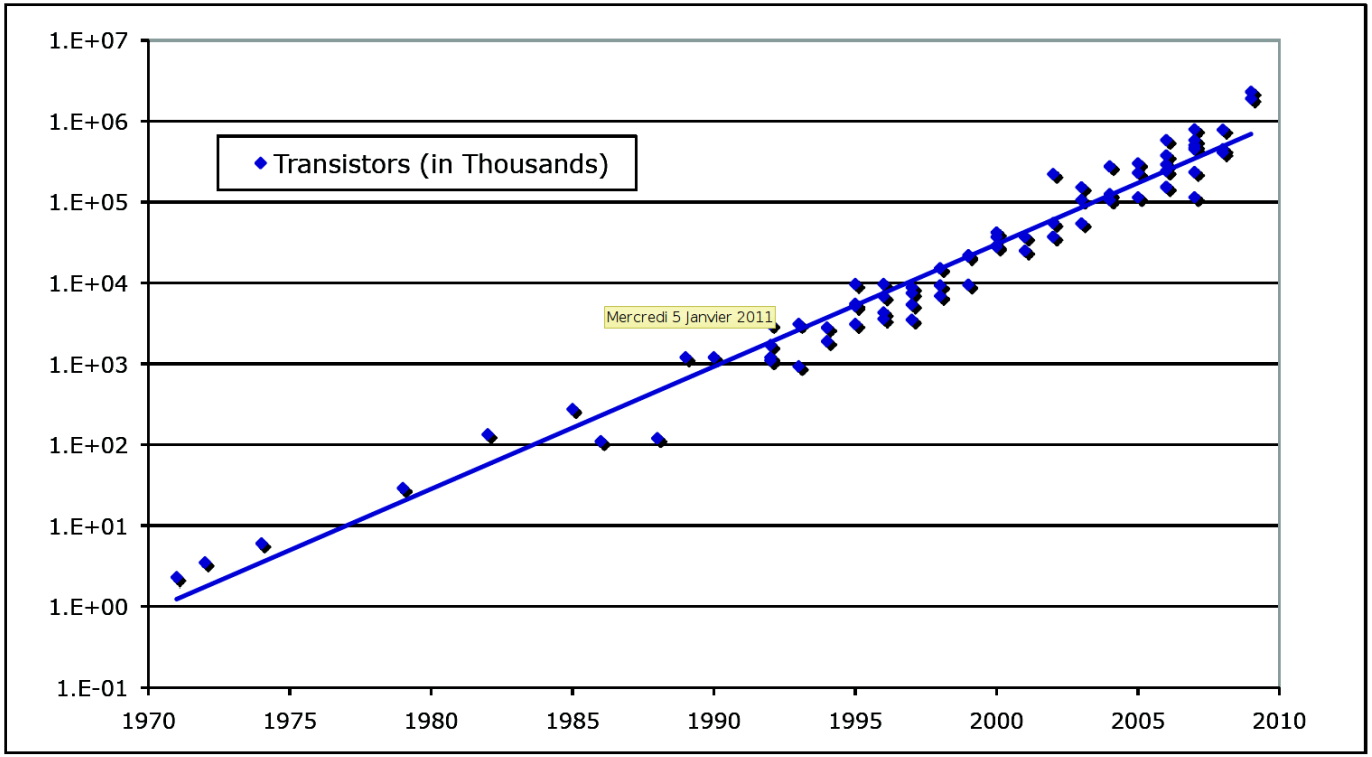
GPU computing - CUDA hardware - 2006

CUDA : Compute Unified Device Architecture

- Nvidia Geforce8800, 2006, introduce a **unified architecture** (only one type of *shader processor*)
- first generation with hardware features designed with GPGPU in mind: almost full support of IEEE 754 standard for single precision floating point, random read/write in external RAM, memory cache controlled by software
- **CUDA ==**
  new hardware architecture +
  new programming model/software abstraction (a *C-like* programming language + development tools : compiler, SDK, librairies like cuFFT)

Moore's law - *the free lunch is over...*

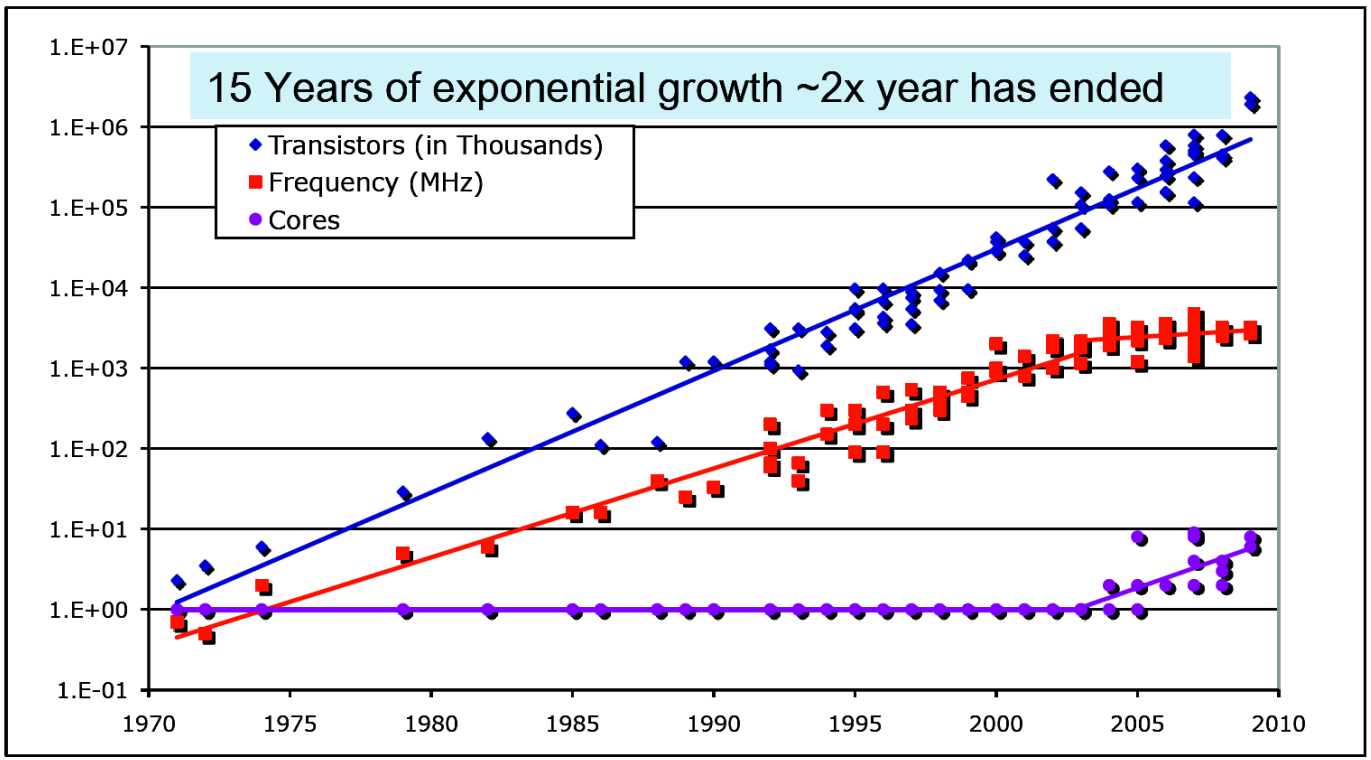
The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years



Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç

Moore's law - *the free lunch is over...*

The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years



Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç

4

Moore's law - *the free lunch is over...*

Moore's Law continues with

- **technology scaling** (32 nm in 2010, 22 nm in 2011),
- improving transistor performance to increase frequency,
- increasing transistor integration capacity to realize complex architectures,
- reducing energy consumed per logic operation to keep power dissipation within limit.

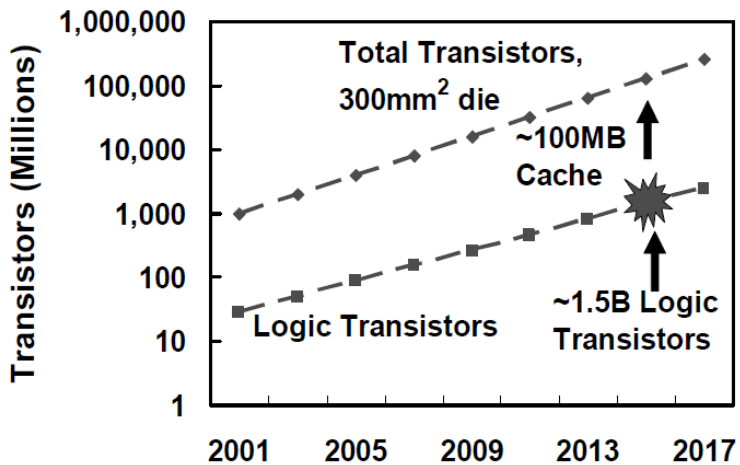


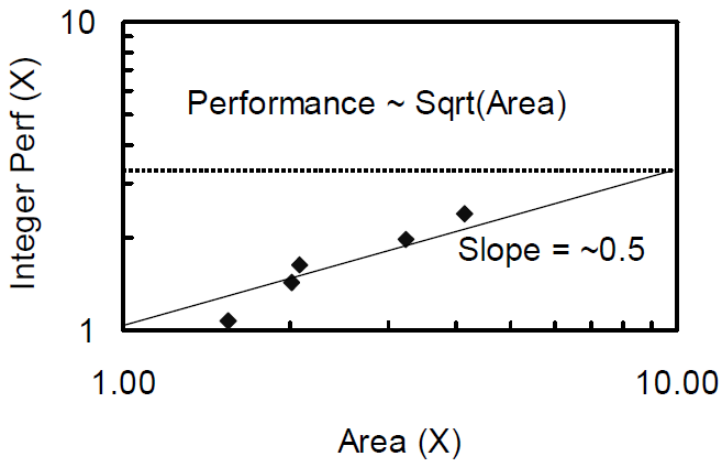
Figure 1: Transistor integration capacity

Shekhar Borkar, *Thousand Core Chips - A Technology Perspective*, in Intel Corp, Microprocessor Technology Lab, 2007, p. 1-4

Moore's law - Towards multi-core architectures

Pollack's rule - Wide adoption of multi-core architectures

- if you **double the logic** in a processor core, then it delivers **only 40% more performance**
- A multi-core microarchitecture has potential to provide near linear performance improvement with complexity and power.
- For example, **two smaller processor cores, instead of a large monolithic processor core, can potentially provide 70-80% more performance, as compared to only 40% from a large monolithic core**



Shekhar Borkar, *Thousand Core Chips - A Technology Perspective*, in Intel Corp, Microprocessor Technology Lab, 2007, p. 1-4

Moore's law - Towards multi-core architectures

- More transistors $\nleftrightarrow$ more computing power !
- **More transistors ? What's the purpose ?** How to use them **efficiently** ?
- **Improve single-core CPU performances:**
  - ☹ keep frequency increasing (watch electric power !)
  - ☺ keep transistor density increasing (more and more difficult) : 32 nm in 2010, 5 nm in 2021
- **Utilize efficiently transistors on chip**
  - ☹ instruction-level parallelism (out-of-order execution, etc...)
  - ☺ data-level parallelism (SIMD, vector units) : SSE, Cell Spe, GPU !
  - ☺ thread-level parallelism: hardware-*multi-threading*, multi-core, many-core ...

http://www.ugrad.cs.ubc.ca/~cs448b/2010-1/lecture/2010-09-09-ugrad.pdf

What is a supercomputer ?

- **Supercomputer:** *A computing system exhibiting high-end performance capabilities and resource capacities within practical constraints of technology, cost, power, and reliability.* Thomas Sterling, 2007.
- **Supercomputer:** *a large very fast mainframe used especially for scientific computations.* Merriam-Webster Online.
- **Supercomputer:** *any of a class of extremely powerful computers. The term is commonly applied to the fastest high-performance systems available at any given time. Such computers are used primarily for scientific and engineering work requiring exceedingly high-speed computations.* Encyclopedia Britannica Online.

Supercomputers architectures - TOP500

A Supercomputer is designed to be at bleeding edge of current technology. Leading technology paths (to exascale) using TOP500 ranks (Nov. 2018)

- **Multicore:** Maintain complex cores, and replicate (Intel x86) (# 4, 8)
- **Manycore/Embedded:** Use many simpler, low power cores from embedded (IBM BlueGene) (# 10)
- **Manycore/Sunway** (# 3)
- **Manycore/Intel XeonPhi (1st and 2nd gen):** Use many simpler cores with wide SIMD instructions, (# 6)
- **Massively Multithread/ GPU:** (# 1, 2, 5, 7, 9)

Next year, we might have supercomputers build with **ARMv8** CPU (From China, Japan, US,...), AMD EPYC CPU, ...

Supercomputers architectures

Multiples levels of hierarchy:

- Need to aggregate the computing power of several 10 000 nodes !
- network efficiency: latency, bandwidth, topology
- memory: on-chip (cache), out-of-chip (DRAM), IO (disk)
- emmerging hybrid programming model: MPI+Multi-thread

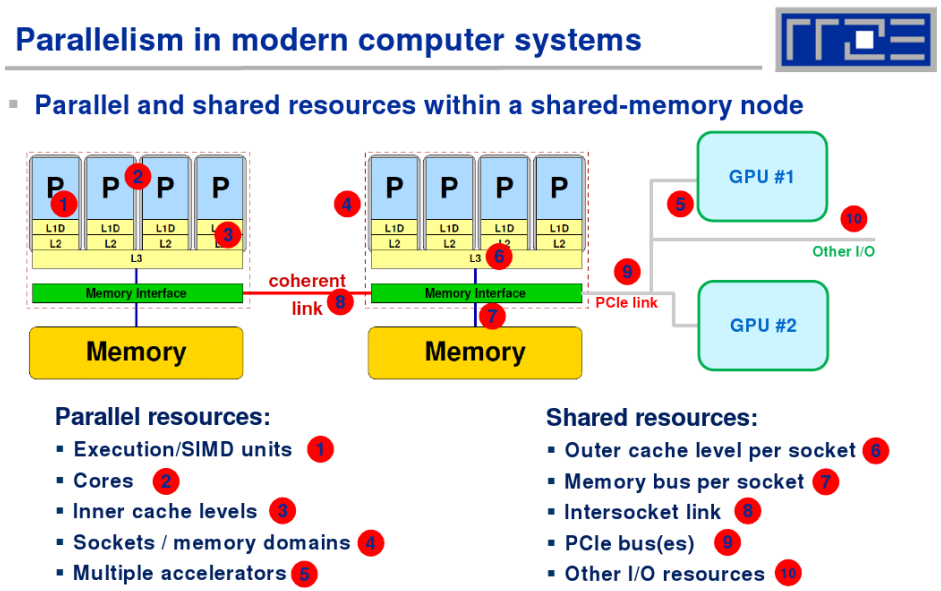


Figure: Multi-core node summary
source: multicore tutorial (SC12) by Georg Hager and Gerhard Wellein

What is a supercomputer ?

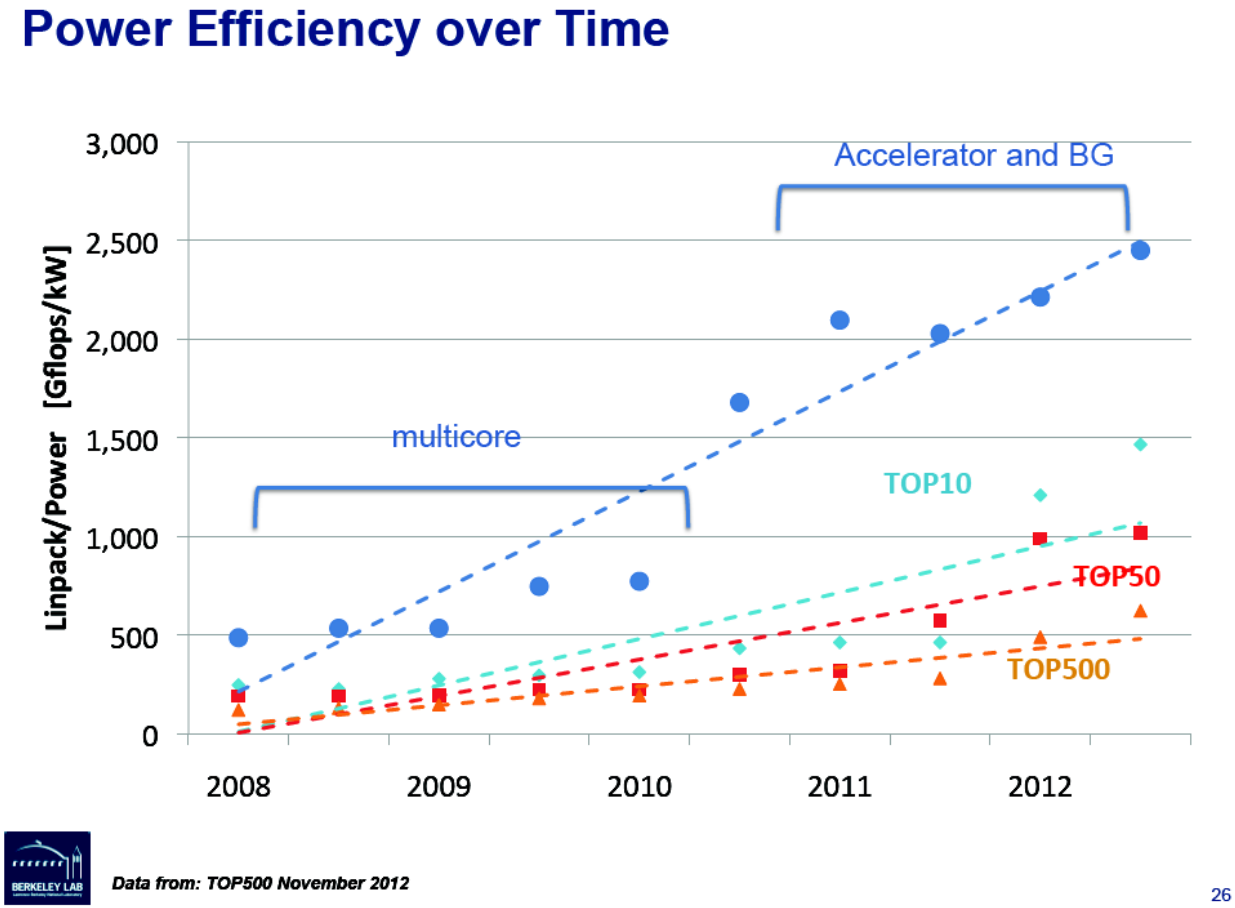
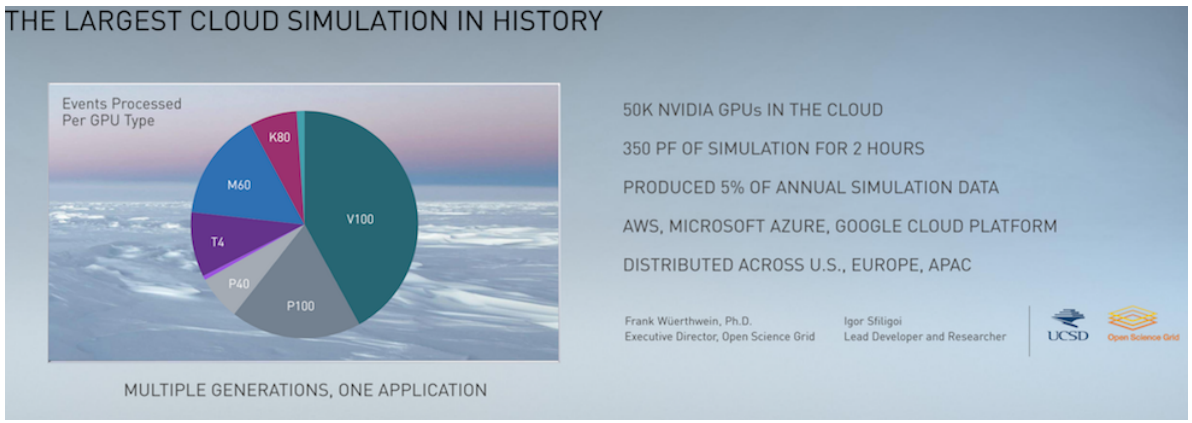


Figure: Horst Simon, LBNL

Supercomputer versus cloud

- What is cloud computing ?
- **More and more GPUs in the cloud**, good for embarassingly parallel computation (see below, neutrino detection)
- **Supercomputer in the cloud** is now coming (up to 800 V100 GPUs, with infiniband)
  azure (Microsoft) cloud news from SC19 conference



50 000 GPUs used in the cloud to detect neutrinos

Getting started with GPU computing...

Get a working development environment for application development

- **How do I know if my GPU is a NVIDIA one ?**
- Use command line tool to probe your hardware (under Linux):
  - `lshw | grep -i nvidia`

    ```
    vendor: NVIDIA Corporation
    configuration: driver=nvidia latency=0
    product: NVIDIA Corporation
    vendor: NVIDIA Corporation
    ```
  - `lspci | grep -i nvidia`

    ```
    03:00.0 VGA compatible controller: NVIDIA Corporation GM107GL
    [Quadro K2200] (rev a2)
    ```
- **How do I know Nvidia driver version (Linux) ?**

  `cat /proc/driver/nvidia/version`
- List of NVIDIA GPUs:

  http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

Getting started...

Get a working development environment for application development

- System requirements:
  https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html
- Install `CUDA toolkit` (device driver + compiler) either from your Linux distribution package manager (prefered) or from website
  https://developer.nvidia.com/cuda-downloads
- Documentation:
  https://docs.nvidia.com/cuda/

A glimpse of CUDA programming

- CUDA is heterogeneous programming (single file containing CPU and GPU code)

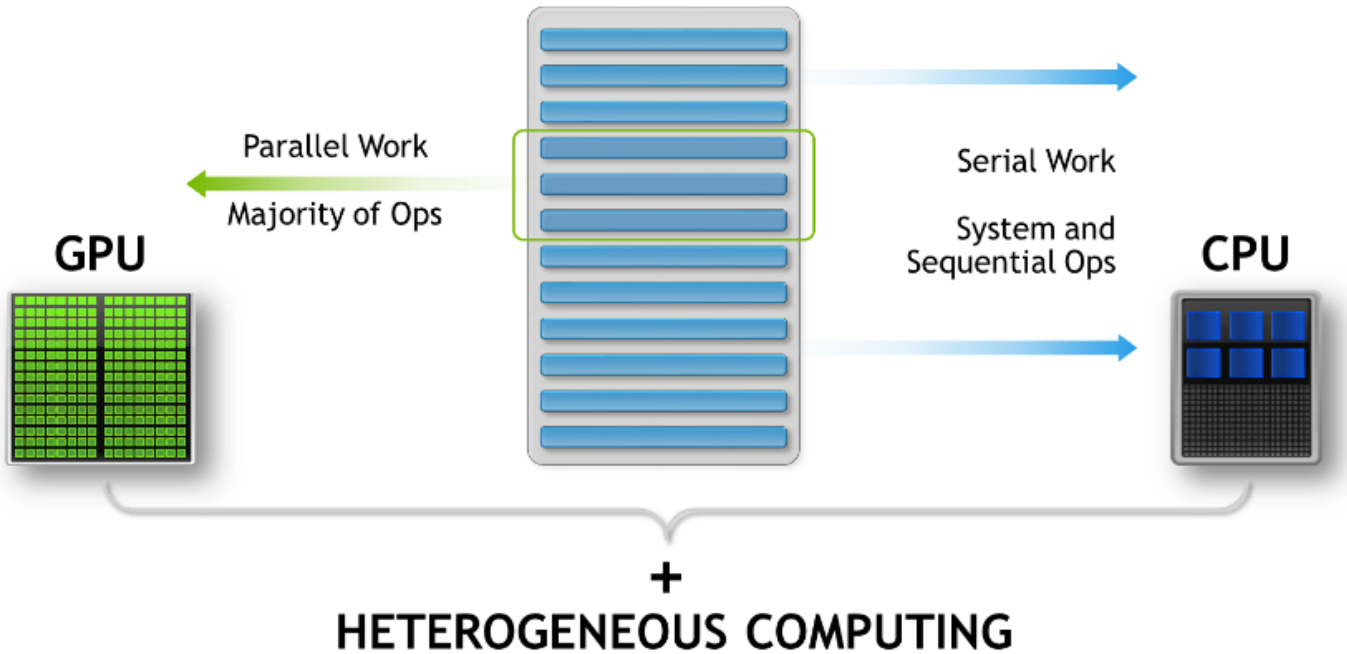
CUDA hello world

```
#include <stdio.h>
#include <stdlib.h>

__global__ void print_from_gpu(void) {
  printf("Hello World from GPU thread %d, block %d !\n",
    threadIdx.x, blockIdx.x);
}
int main(int argc, char* argv[]) {
  printf("Hello from CPU !\n");
  print_from_gpu<<<1,1>>>();
  cudaDeviceSynchronize();
  return EXIT_SUCCESS;
}
```

A glimpse of CUDA programming

- CUDA is heterogeneous programming (single file containing CPU and GPU code)



reference: Learn Cuda programming, Packt, 2019

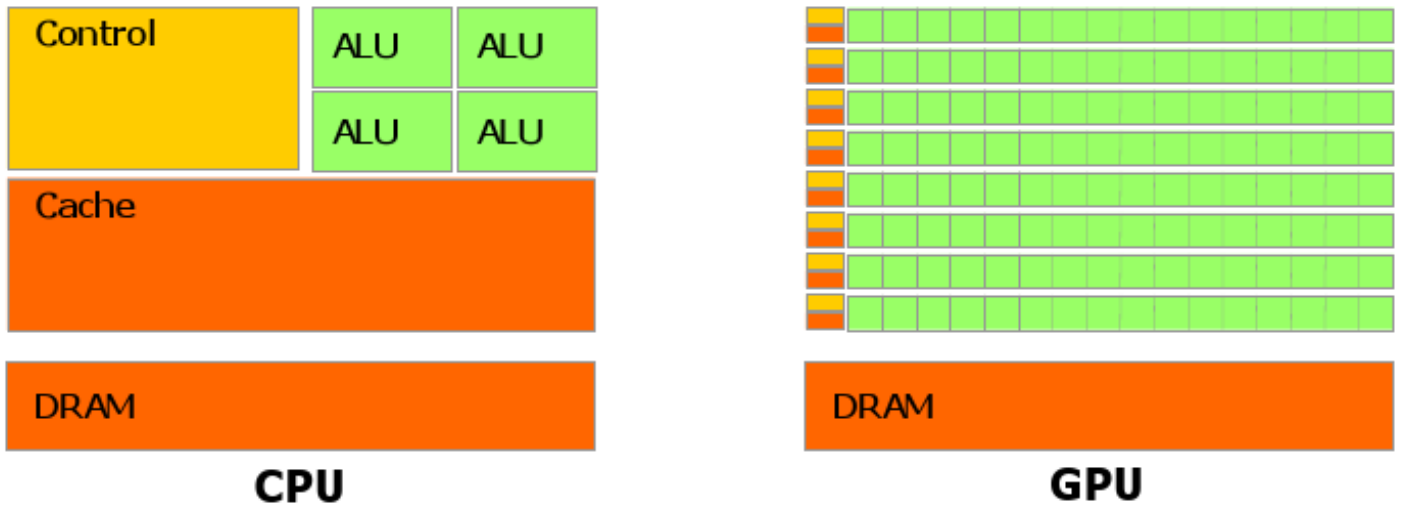
CPU versus GPU: hardware and software comparison

Plan

- What are the major differences between CPUs and GPUs in terms of hardware architecture ?
- What is the meaning of *latency-oriented architecture* ?
- Whait is the meaning of *throughput-oriented architecture* ?

From multi-core CPU to manycore GPU

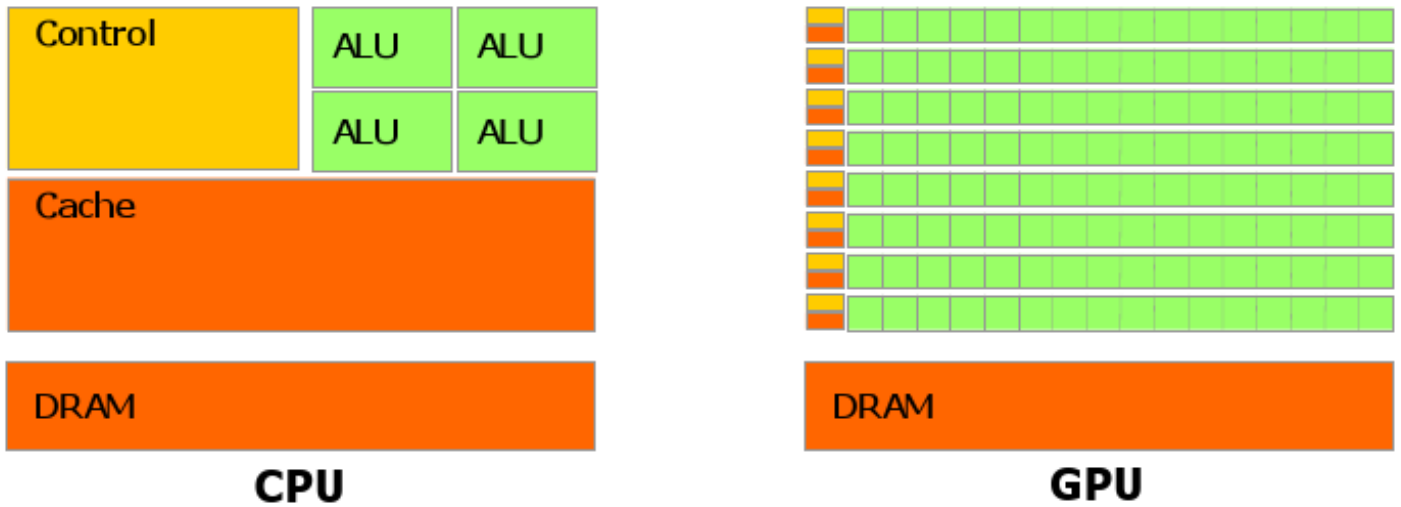
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- **Different goals produce different designs:**
  - **CPU** must be good at everything, parallel or not
  - **GPU** assumes work load is highly parallel

From multi-core CPU to manycore GPU

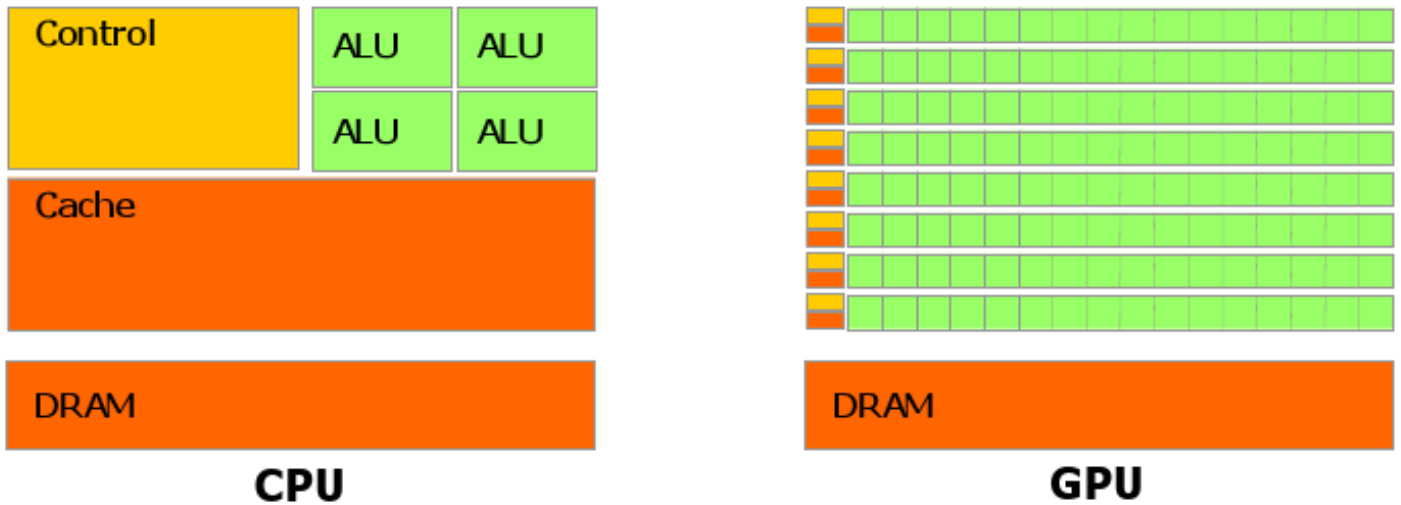
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- **CPU** design goal : optimize architecture for sequential code performance : minimize latency experienced by **1 thread**
  - sophisticated (i.e. large chip area) control logic for instruction-level parallelism (branch prediction, out-of-order instruction, etc...)
  - CPU have large cache memory to reduce the instruction and data access latency

From multi-core CPU to manycore GPU

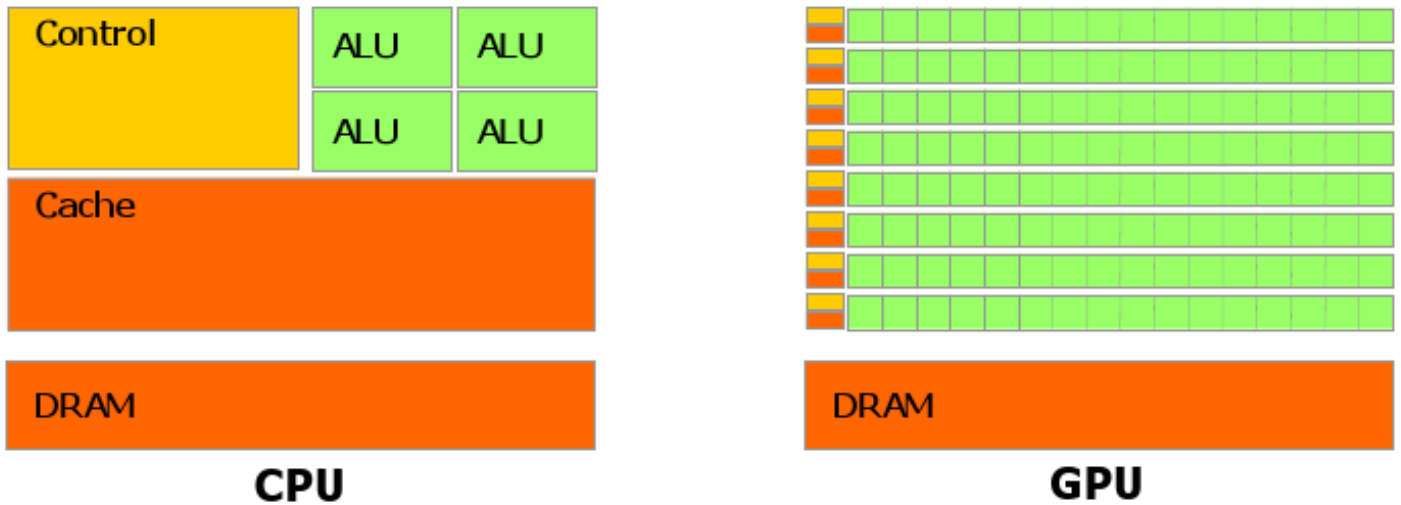
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- **GPU** design goal : maximize throughput of **all threads**
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches
  - share control logic across many threads

From multi-core CPU to manycore GPU

Architecture design differences between manycore GPUs and general purpose multicore CPU ?

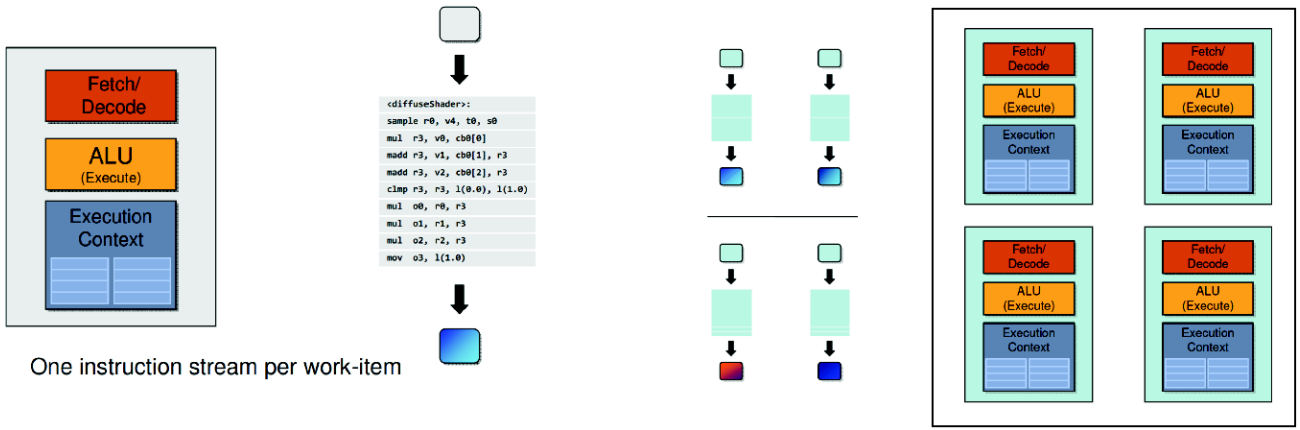


- GPU takes advantage of a **large number of execution threads** to find work to do when other threads are waiting for long-latency memory accesses, thus **minimizing the control logic** required for each execution thread.

From multi-core CPU to manycore GPU

Architecture design differences between manycore GPUs and general purpose multicore CPU ?

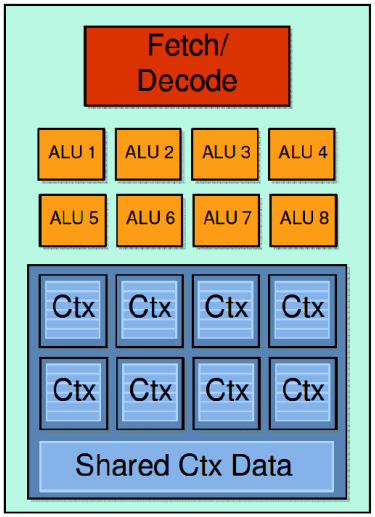
- CPU Conventional Core Quad



- GPU: much more Silicium area dedicated to floating point computations
- GPUs are numeric computing engines that will not perform well on some tasks for which CPU are optimized. Need to take advantage of both !

From multi-core CPU to manycore GPU

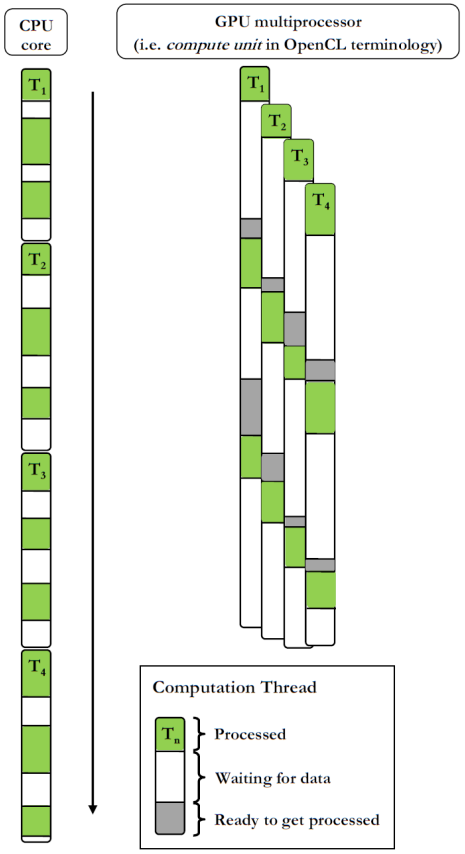
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- SIMD
- GPU: Amortize cost / complexity of managing an instruction stream across many ALUs / hardware resources mutualized.

From multi-core CPU to manycore GPU

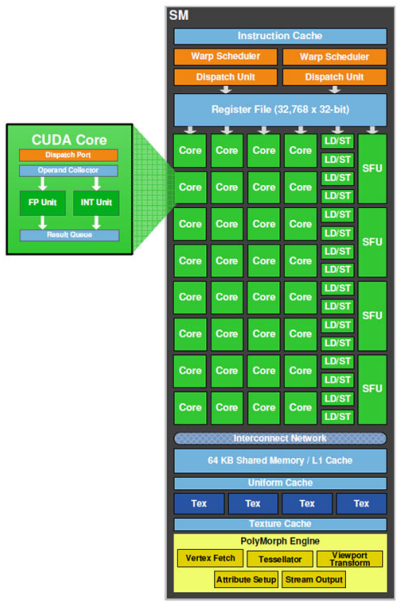
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- **Hiding memory latency** by having a large number of thread per core.

Nvidia Fermi hardware (2010)

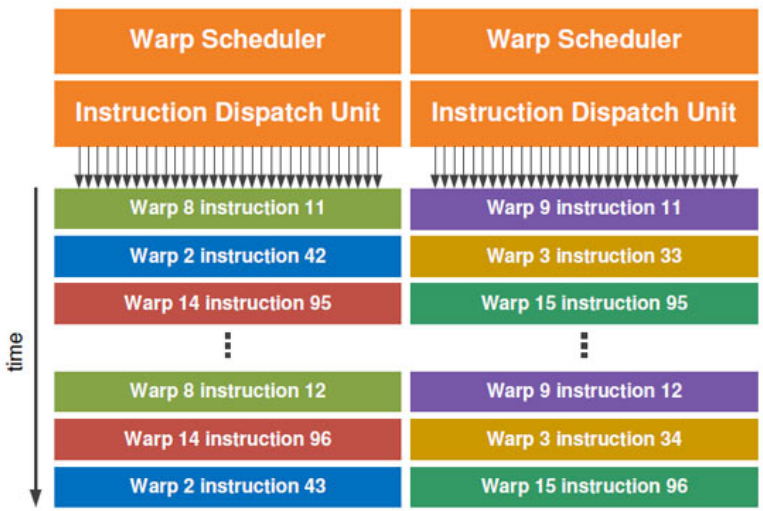
- **Streaming Multiprocessor** (32 cores), hardware control, queuing system
- **GPU = scalable array of SM** (up to 16 on Fermi)
- **warp: vector of 32 threads**, executes the same instruction in lock-step
- **throughput limiters**: finite limit on warp count, on register file, on shared memory, etc...



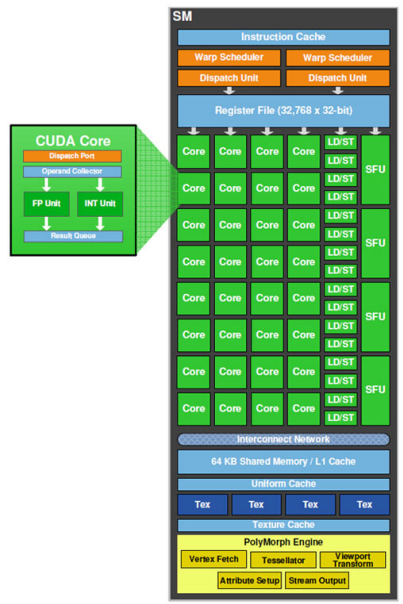
Streaming Multiprocessor (SM)

Nvidia Fermi hardware (2010)

- **Streaming Multiprocessor** (32 cores), hardware control, queuing system
- **GPU = scalable array of SM** (up to 16 on Fermi)
- **warp: vector of 32 threads**, executes the same instruction in lock-step
- **throughput limiters**: finite limit on warp count, on register file, on shared memory, etc...



Dual warp scheduling



Streaming Multiprocessor (SM)

Nvidia Fermi hardware (2010)

CUDA Hardware (HW) key concepts

- Hardware thread management
  - HW thread launch and monitoring
  - HW thread switching
  - up to 10 000's lightweight threads
- SIMT execution model
- Multiple memory scopes
  - Per-thread private memory : (register)
  - Per-thread-block shared memory
  - Global memory
- Using threads to hide memory latency
- Coarse grained thread synchronization

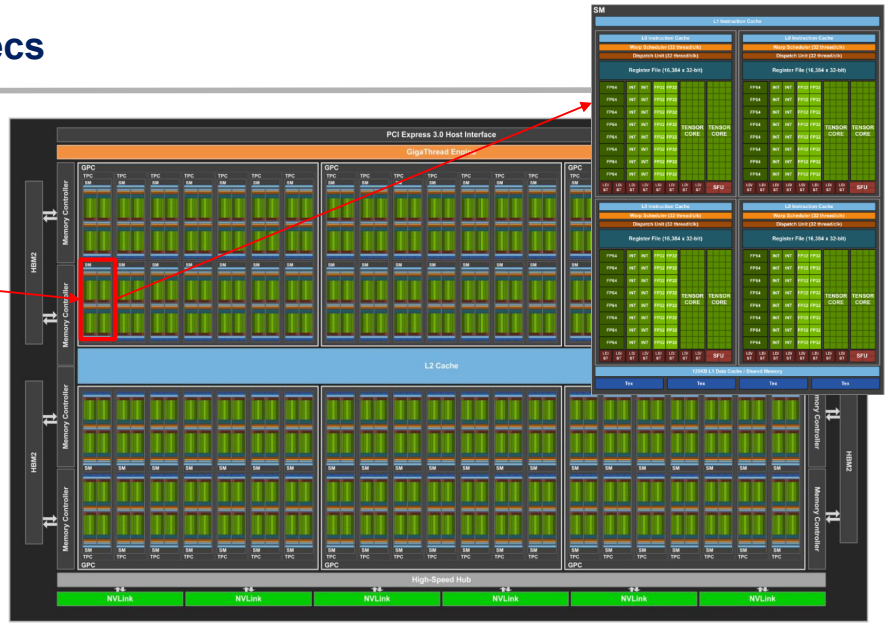
From multi-core CPU to manycore GPU

Compring brut performance (FLOPS and Bandwidth) for CPU and GPU:

Nvidia V100 “Volta” specs

Architecture

- 21.1 B Transistors
- ~ 1.4 GHz clock speed
- ~ 80 “SM” units
  - 64 SP “cores” each (FMA)
  - 32 DP “cores” each (FMA)
  - 8 “Tensor Cores” each
  - 2:1 SP:DP performance
- ~7 TFlop/s DP peak
- 6 MiB L2 Cache
- 4096-bit HBM2
- MemBW ~ 900 GB/s (theoretical)
- MemBW ~ 830 GB/s (measured)



© Nvidia

$$P^{DP}_{peak} = n_{SM} \cdot n_{core} \cdot n_{FP} \cdot f$$

# SMs	# CUDA cores/SM	# FP ops/cy

$$n_{SM} = 80$$
$$n_{core} = 32$$
$$n_{FP} = 2\frac{\text{flops}}{\text{cy}}$$
$$f = 1.4\frac{\text{Gcy}}{\text{s}}$$

reference: Node-level performance engineering, SC19, Tutorial

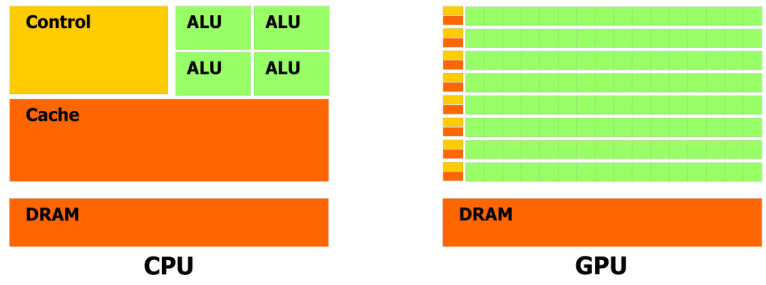
From multi-core CPU to manycore GPU

Compring brut performance (FLOPS and Bandwidth) for CPU and GPU:

Trading single thread performance for parallelism:
GPGPUs vs. CPUs

GPU vs. CPU
light speed estimate
(per device)

MemBW	~ 8-10x
Peak	~ 5-10x

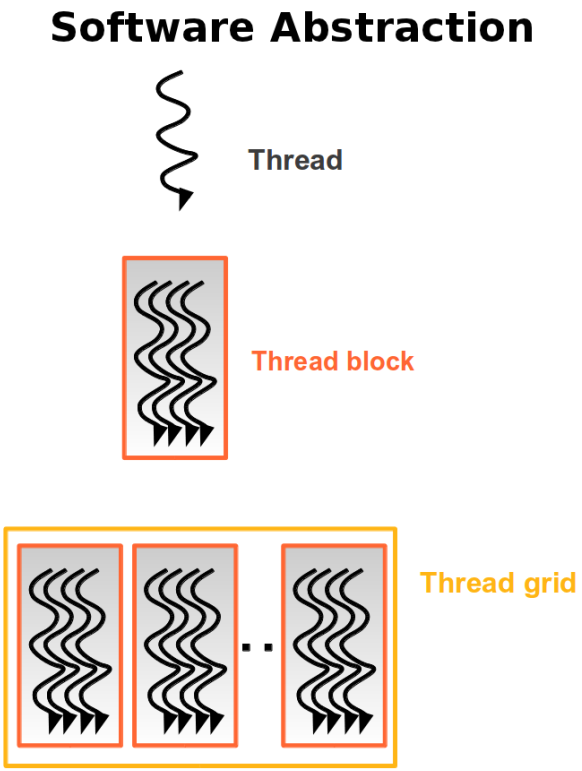


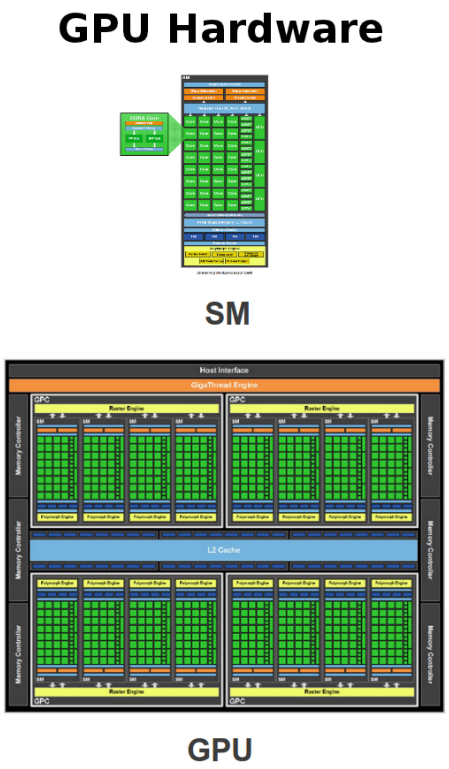
	2x Intel Xeon Platinum 8160	NVidia Tesla V100 “Volta”
Cores@Clock	2 x 24 @ ≥2.1 GHz	80 SMs @ ~1.4 GHz
SP Performance/core	≥ 134 GFlop/s	~179 GFlop/s
Threads@STREAM	~20	> 20000
SP peak	≥ 6 TFlop/s	~14 TFlop/s
Stream BW (meas.)	2 x 105 GB/s	830 GB/s
Transistors / TDP	~2x8 Billion / 2x150 W	21 Billion/250 W

reference: Node-level performance engineering, SC19, Tutorial

CUDA - connecting program and execution model

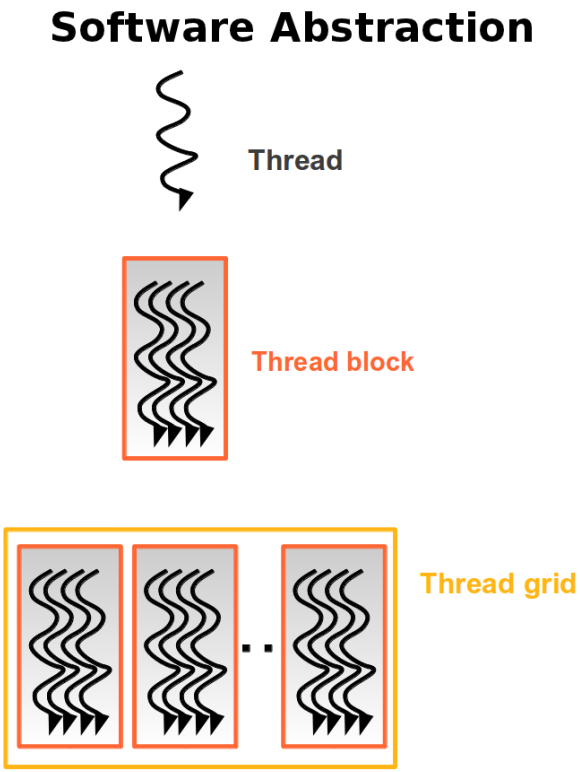
- **Need a programming model to efficiently use such hardware**; also provide **scalability**
- Provide a simple way of partitioning a computation into fixed-size blocks of threads

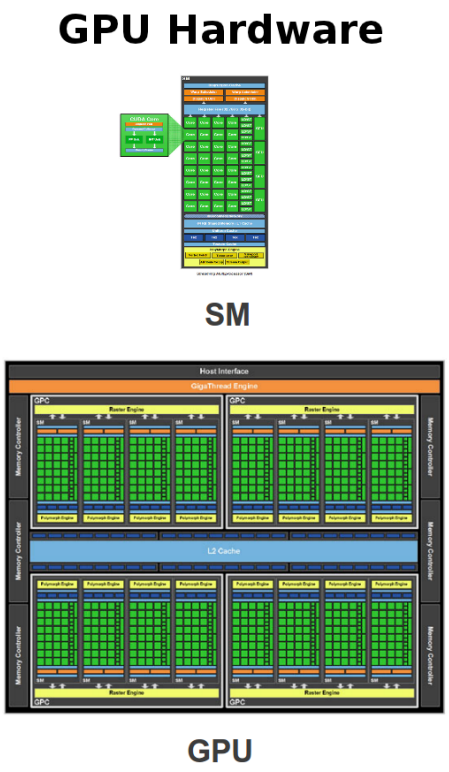




CUDA - connecting program and execution model

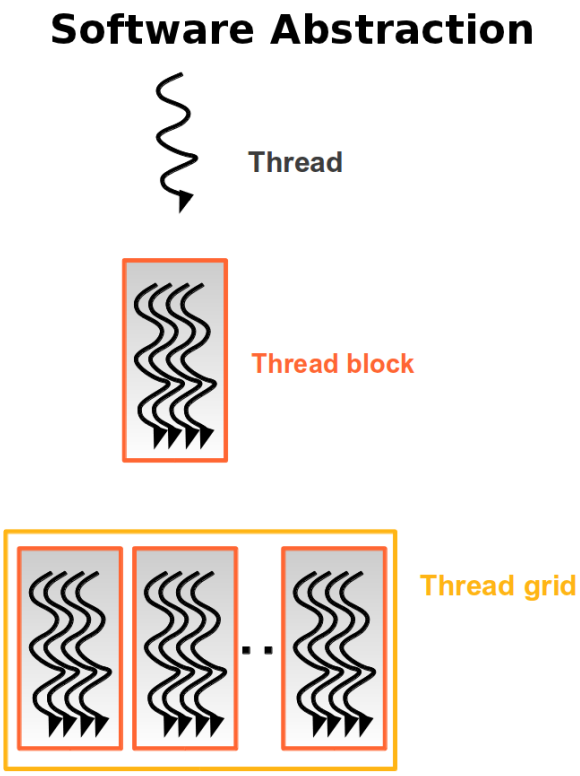
- **Total number of threads** must/need be quite larger than number of cores
- **Thread block** : **logical array of threads**, large number to hide latency
- **Thread block size** : control by program, specify at runtime, better be a multiple of warp size (i.e. 32)

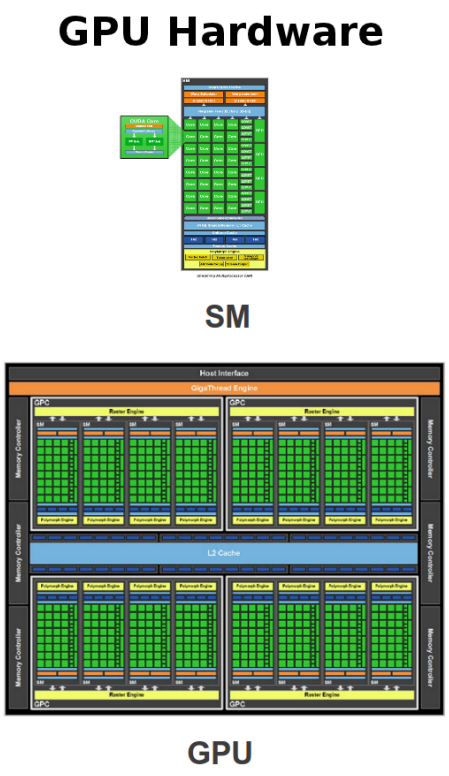




CUDA - connecting program and execution model

- **Must give the GPU enought work to do !** : if not enough thread blocks, some SM will remain idle
- **Thread grid** : **logical array of thread blocks** distribute work among SM, several blocks / SM
- **Thread grid** : chosen by program at runtime, can be the total number of thread / thread block size or a multiple of # SM

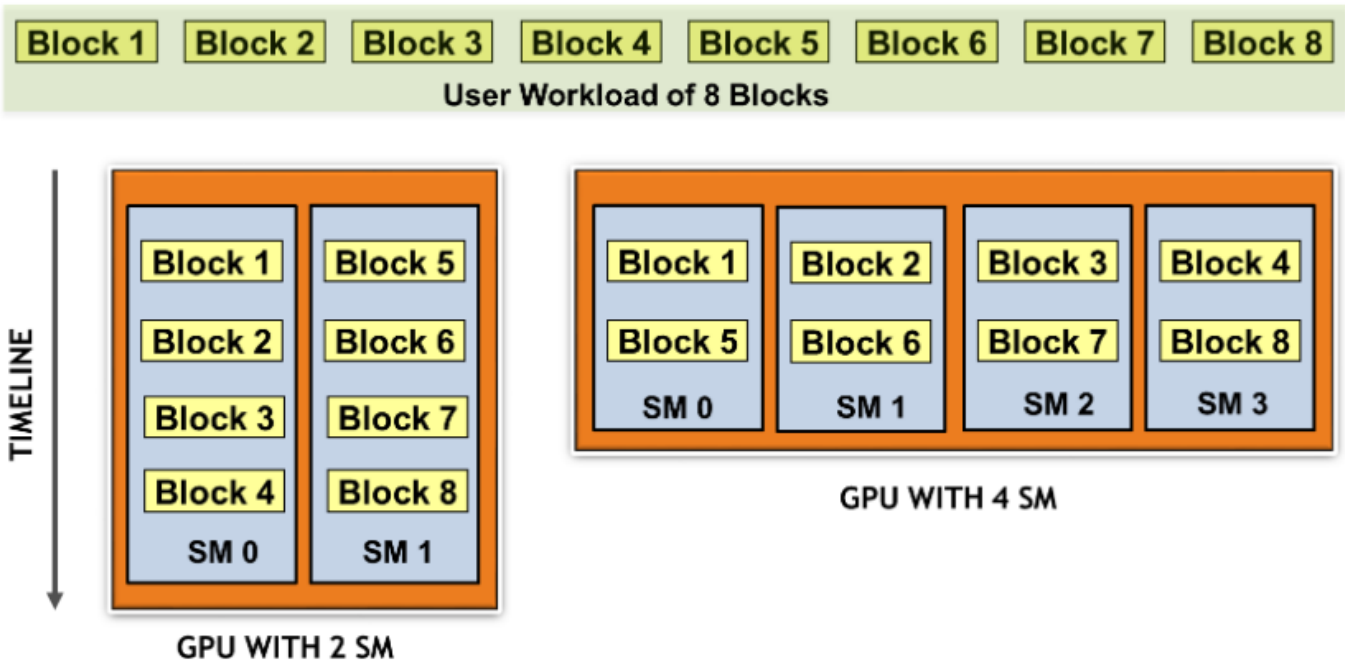




CUDA - connecting program and execution model

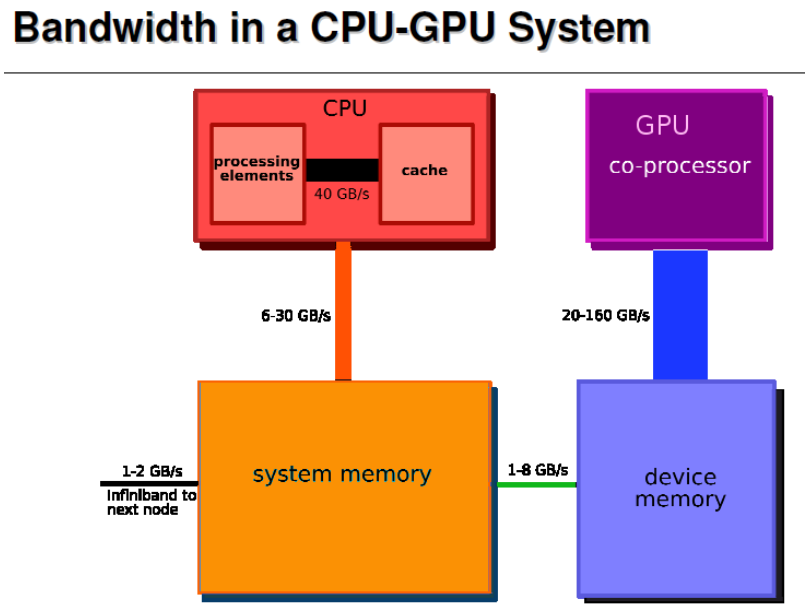
Why two levels of hierarchy block / grid ?

- scale/adapt to different GPUs (make code independent from hardware details, e.g. number of SM)



GPU is an auxilliary processor : memory bandwidth

- **CPU-GPU link**, Pci-express bus x16, Gen2 :
  $BP = 16 * 2 * 250\text{MBytes/s} =$ **8 GBytes/s**
- **CPU-local** : DDR memory ($f = 266\text{MHz}$)
  BP= $266 * 10^6$*4 (transferts/cycle) * 8 (bytes) = **8.5 GBytes/s**
- **GPU-local (carte GTX280)** : 512-bit bus, DDR@$f = 1100\text{MHz}$,
  BP= $2 * 1100 * 10^6$(transferts/s) * 512/8 (bytes/transfert)= **140 GBytes/s**

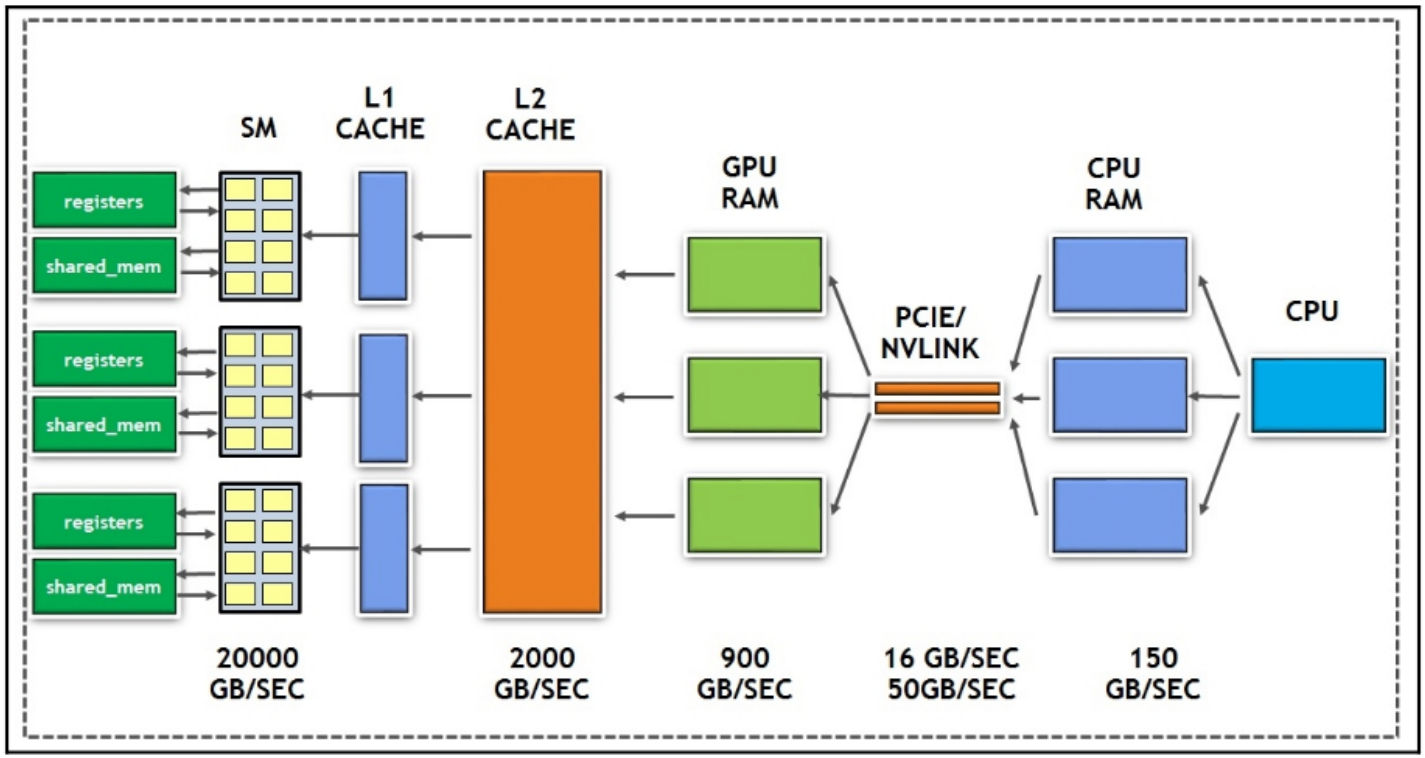


(Digression) Evaluating Peak Memory bandwidth

How to determine the **peak** hardware memory bandwidth of your compute platform ?

- **Multicore CPU** (e.g. Intel Skylake):
  - Memory type ? e.g. DDR4-2666
  - Number of channels ? e.g. 6
  - Max BW = # NbOfChannel × Frequency(GHz) × BusWidth/8 (Bytes) × # NbOfSockets
  - e.g. on TGCC/IRENE, BW = 6 × 2.6 × 64/8 × 2 = 256 GBytes per node
- **Manycore CPU** (e.g. Intel KNL):
  - depends on HBM configuration (CACHE, FLAT, HYBRID)
  - e.g. KNL on TGCC/IRENE configured in CACHE mode, BW ⩾ 400 GBytes/s
- **NVIDIA GPU** (e.g. Pascal P100):
  - Use CUDA SDK `deviceQuery` to retrieve hardware spec (**TODO as an exercise**).
  - # Memory Clock rate: 715 Mhz
  - # Memory Bus Width: 4096-bit
  - BW = 732.1 Gbytes/s
- **NVIDIA GPU** (e.g. Pascal V100):
  - BW = 898.0 Gbytes/s

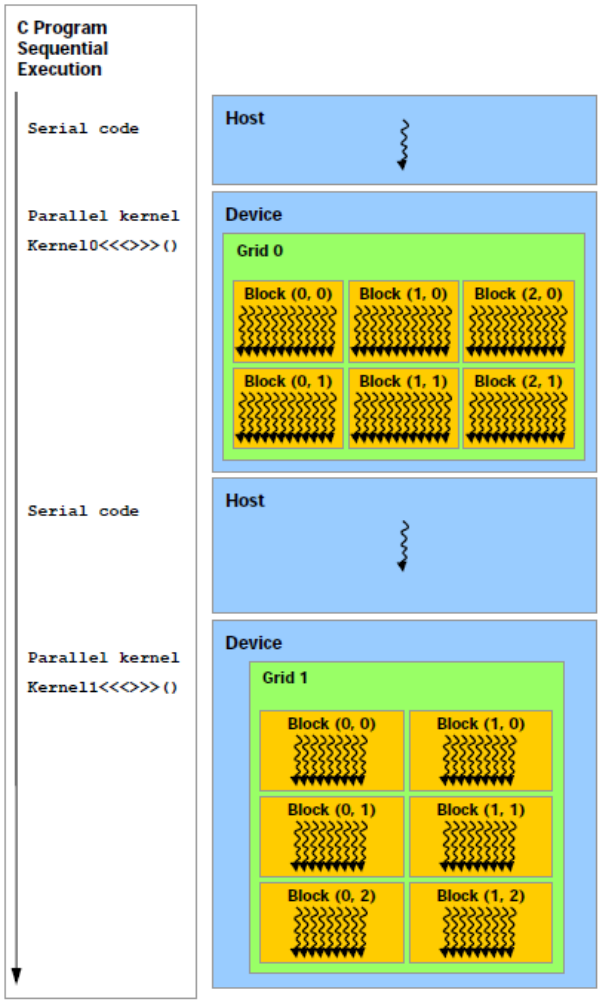
CPU / GPU memory bandwidth / bottleneck



All the different memory bandwidth can actually be measured, see cuda sdk bandwidth test.

reference: Learn Cuda programming, Packt, 2019

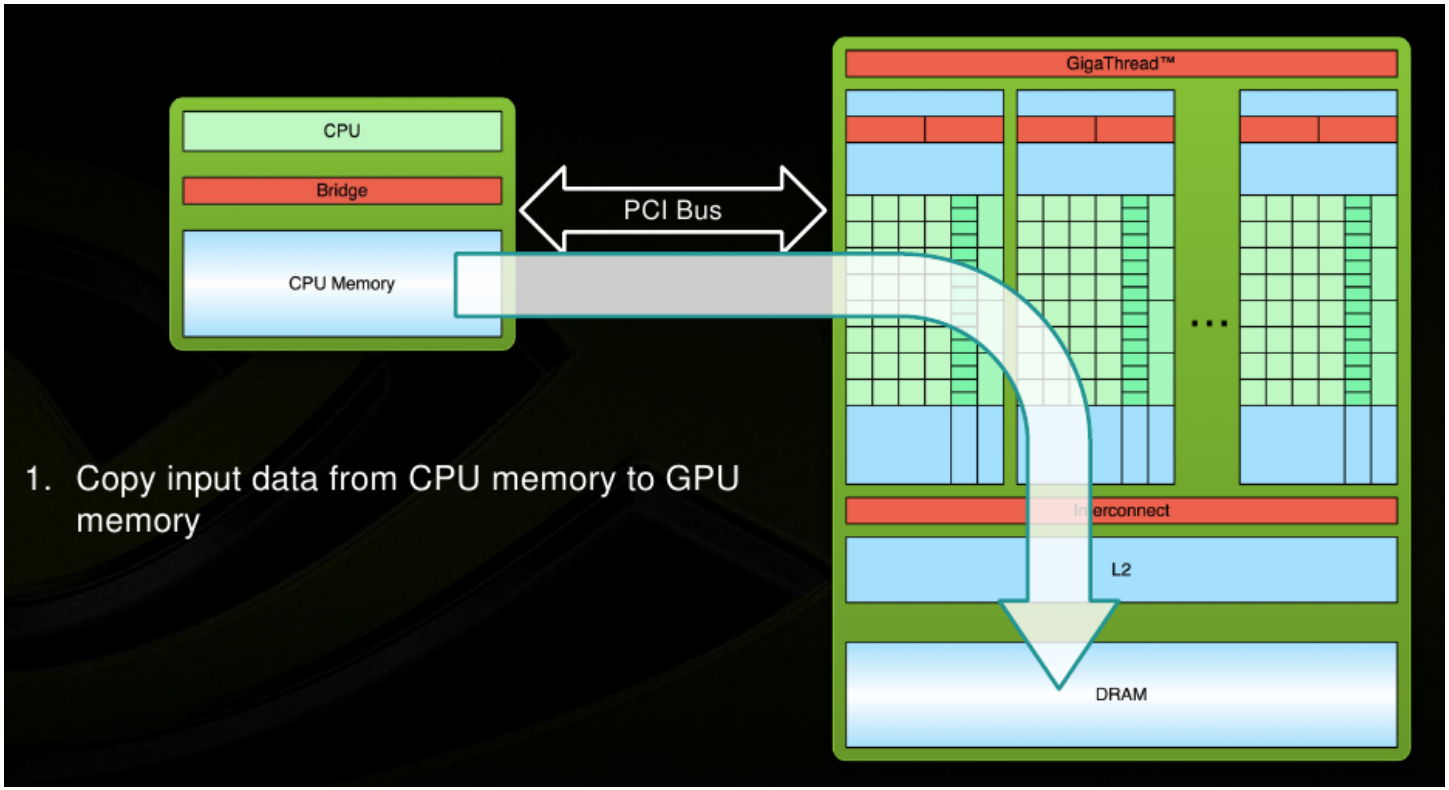
CUDA : heterogeneous programming



- heterogeneous systems : CPU and GPU have physically separated memory spaces (*host* and *device*)
- Unified Virtual Memory (most efficient since Pascal architecture)
- CPU code and GPU code can be in the same program / file (pre-processor will separate CPU/GPU code)
- the programmer focuses on code parallelization (algorithm level) not on how he was to schedule blocks of threads on multiprocessors.

CUDA : heterogeneous programming

Current GPU execution flow

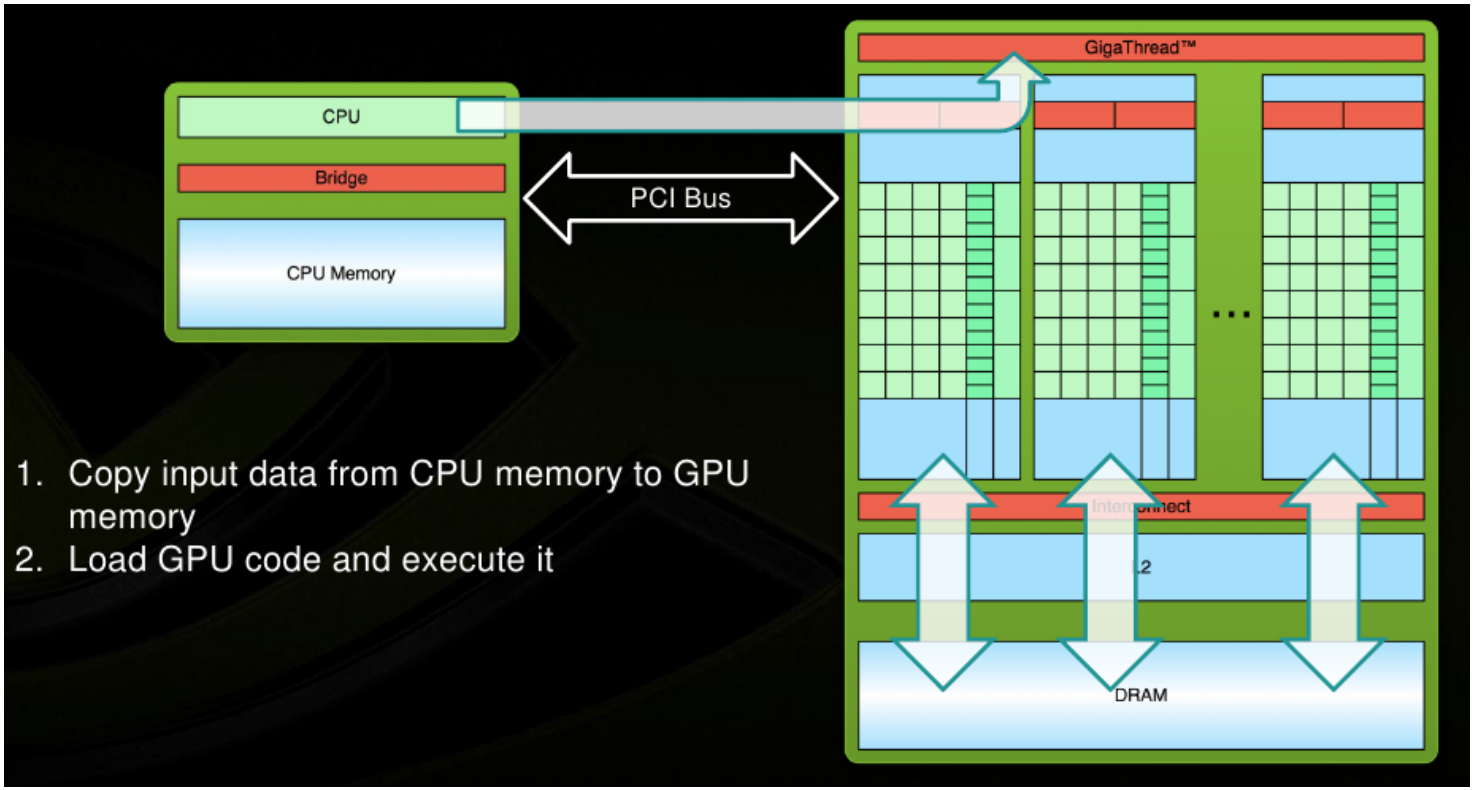


reference: Introduction to CUDA/C, GTC 2012

update (2016): Things are changing / improving with architecture Pascal with a better Unified Memory: larger virtual memory address space, page migration, oversubscription, ...

CUDA : heterogeneous programming

Current GPU execution flow

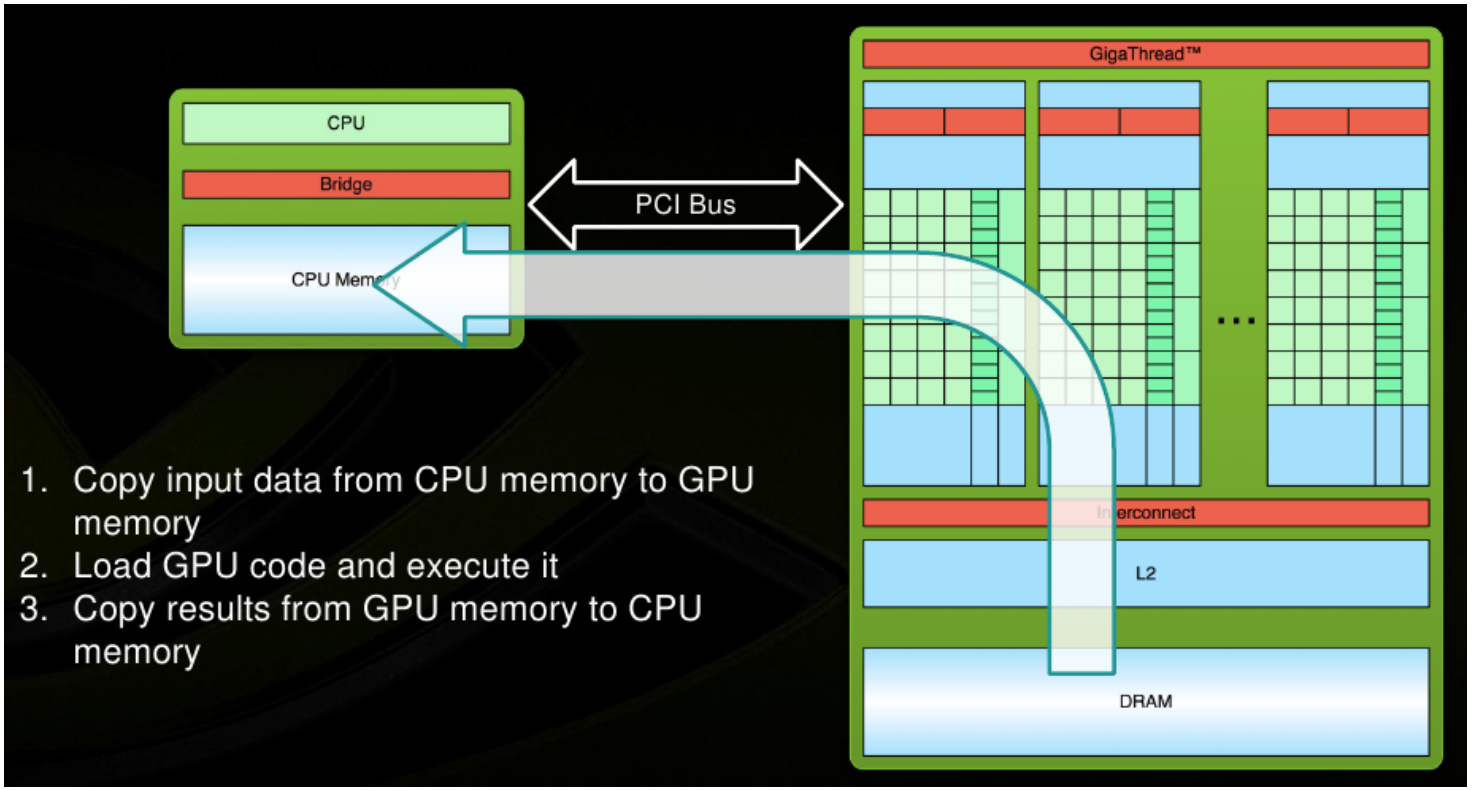


reference: Introduction to CUDA/C, GTC 2012

update (2016): Things are changing / improving with architecture Pascal with a better Unified Memory: larger virtual memory address space, page migration, oversubscription, ...

CUDA : heterogeneous programming

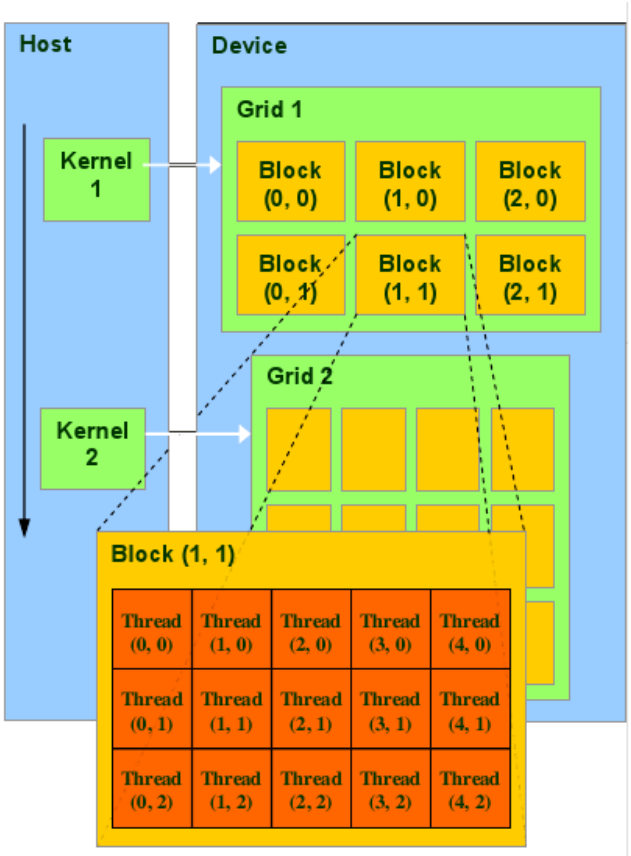
Current GPU execution flow



reference: Introduction to CUDA/C, GTC 2012

update (2016): Things are changing / improving with architecture Pascal with a better Unified Memory: larger virtual memory address space, page migration, oversubscription, ...

CUDA : programming model (PTX)



- a block of threads is a **CTA (Cooperative Thread Array)**
- **Threads are indexed inside a block; use that index to map memory**
- write a program once for a *thread*
- run this program on multiple *threads*
- **block** is a logical array of threads indexed with *threadIdx* (**built-in variable**)
- **grid** is a logical array of blocks indexed with *blockIdx* (**built-in variable**)

CUDA C/C++

- **CUDA C/C++**
  - **Large subset of C/C++ language**
  - CPU and GPU code in the same file; preprocessor to filter GPU specific code from CPU code
  - Small set of extensions to enable heterogeneous programming: new keywords
  - **A runtime/driver API**
    - **Memory management:** `cudaMalloc`, `cudaFree`, ...
    - **Device management:** `cudaChooseDevice`, probe device properties (# SM, amount of memory , ...)
    - **Event management:** profiling, timing, ...
    - **Stream management:** overlapping CPU-GPU memory transfert with computations, ...
    - **...**
- **Terminology**
  - **Host**: CPU and its memory
  - **Device**: GPU and its memory
    - **kernel**: routine executed on GPU

CUDA : C-language extensions and run-time API

- Function and type qualifiers

```
__global__ void KernelFunc(...); // kernel callable from host
__device__ void DeviceFunc(...); // function callable on device
__device__ int  GlobalVar;       // variable in device memory
__shared__ int  SharedVar;       // shared in PDC by thread block
__host__   void HostFunc(...);   // function callable on host
```

- built-in variables : *threadIdx* and *blockDim*, *blockIdx* and *gridDim* (*read-only registers*)
- kernel function launch syntax

```
KernelFunc<<<500, 128>>>(...); // launch 500 blocks w/ 128 threads each
```

  «< .. »> is used to set grid and block sizes (can also set shared mem size per block)

- synchronisation *threads* inside bloc

```
__syncthreads(); // barrier synchronization within kernel
```

- *libc*-like routine (e.g.: memory allocation, CPU/GPU data transfer, ...)

CUDA Code walkthrough

- **Data parallel model**
- **Use intrinsic variables `threadIdx` and `blockIdx` to create a mapping between threads and actual data**

CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a,b,N);
}
```

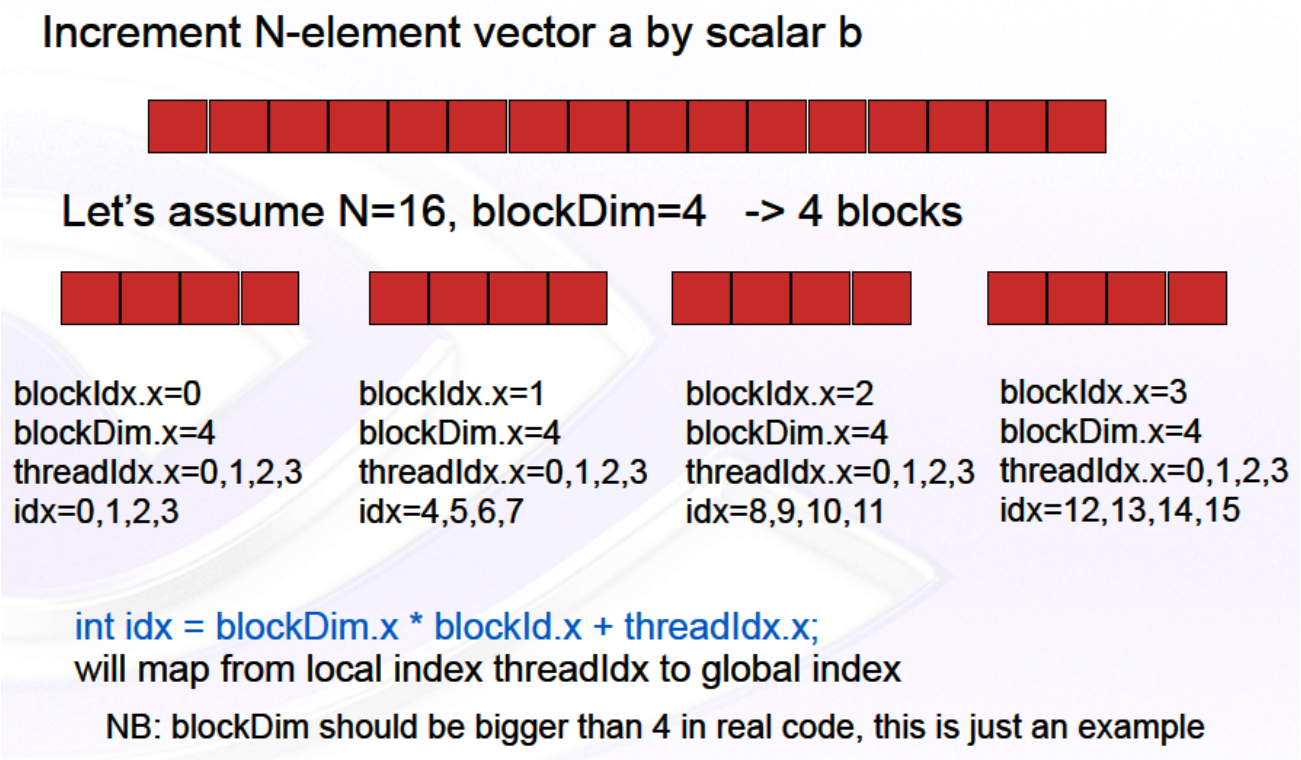
CUDA program

```
__global__ void increment_gpu(float *a, float b)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = a[idx] + b;
}
```

```
void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid (N/blocksize);
    increment_gpu<<<dimGrid, dimBlock>>>(a,b);
}
```

CUDA Code walkthrough

- **Data parallel model**
- **Use intrinsic variables `threadIdx` and `blockIdx` to create a mapping between threads and actual data**



CUDA Code walkthrough

- **Data parallel model**
- **Use intrinsic variables `threadIdx` and `blockIdx` to create a mapping between threads and actual data**

```
/*
 * nvcc -m64 -gencode arch=compute_20,code=sm_20 --ptxas-options -v
 * -o scalarAdd scalarAdd.cu
 */
#include <stdio.h>

/**
 * a simple CUDA kernel
 *
 * \param[inout]  a input integer pointer
 * \param[in]     b input integer
 * \param[in]     n input array size
 */
__global__ void add( int *a, int b, int n ) {

  int idx = threadIdx.x + blockIdx.x*blockDim.x;

  if (idx<n)
    a[idx] = a[idx] + b;
}
```

CUDA Code walkthrough

- **Data parallel model**
- **Use intrinsic variables `threadIdx` and `blockIdx` to create a mapping between threads and actual data**

```
/*
 * main
 */
int main( void ) {
  // array size
  int N = 16;

  // host variables
  int *a;  int  b;

  // device variables
  int *dev_a;

  // CPU memory allocation
  a = (int *) malloc(N*sizeof(int));
  b = N;
  // CPU memory initialization
  for (int i=0; i<N; i++) a[i]=i;
```

```
// GPU device memory allocation
cudaMalloc( (void**)&dev_a,
            N*sizeof(int) ) ;

// GPU device memory initialization
cudaMemcpy( dev_a, a, N*sizeof(int),
            cudaMemcpyHostToDevice ) ;

// perform computation on GPU
int nbThreads = 8;
dim3 blockSize(nbThreads,1,1);
dim3 gridSize((N+1)/nbThreads,1,1);
add<<<gridSize,blockSize>>>( dev_a, b, N );

// get back computation result
// into host CPU memory
cudaMemcpy( a, dev_a, N*sizeof(int),
            cudaMemcpyDeviceToHost ) ;
```

CUDA Code walkthrough

- **Data parallel model**
- **Use intrinsic variables `threadIdx` and `blockIdx` to create a mapping between threads and actual data**

```
  // do something !

  // de-allocate CPU host memory
  free(a);

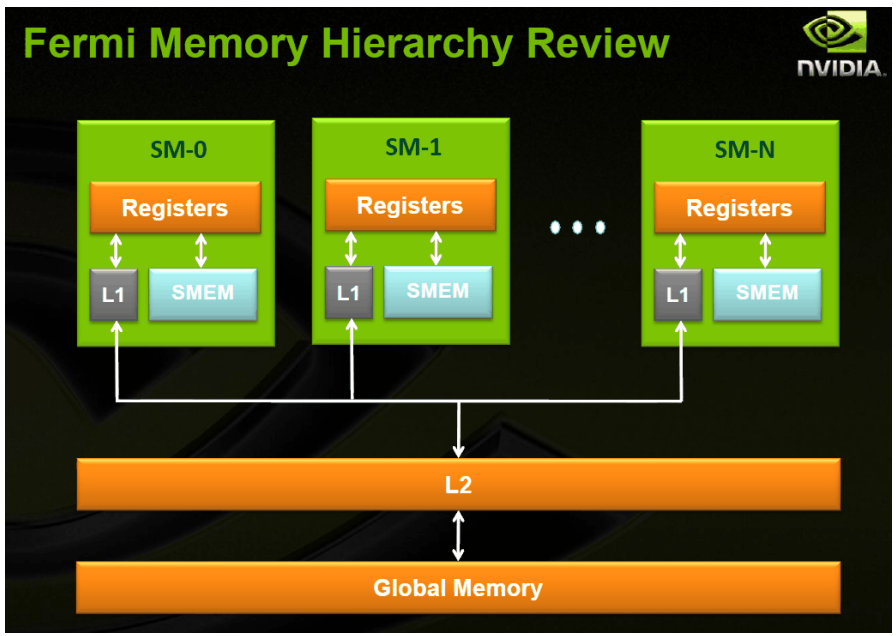
  // de-allocate GPU device memory
  cudaFree( dev_a ) ;

  cudaDeviceSynchronize();
  cudaDeviceReset();

  return 0;
}
```

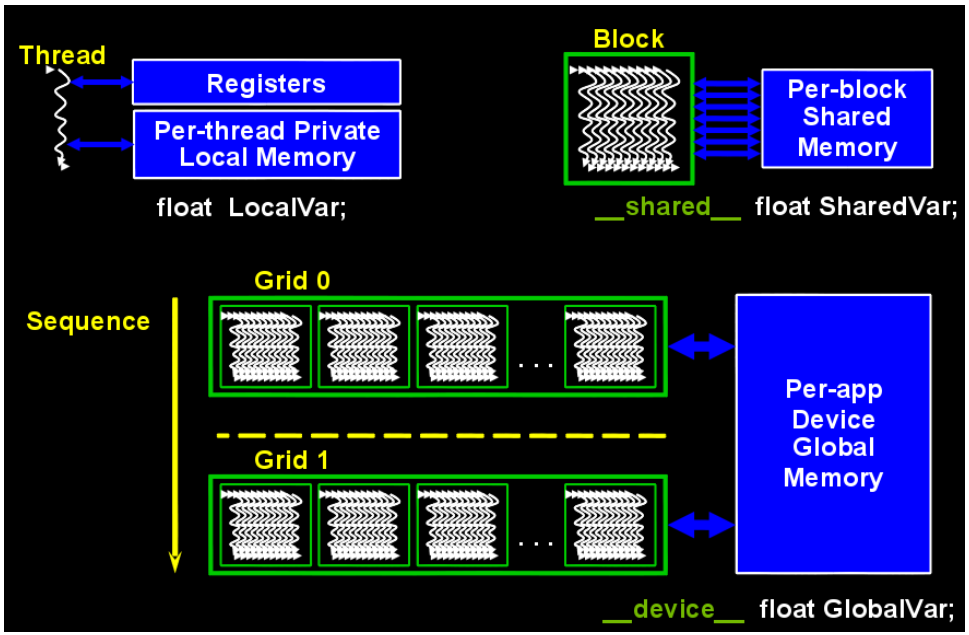
CUDA memory hierarchy: software/hardware

- **hardware (Fermi) memory hierarchy**
  - **on chip memory** : low latency, fine granularity, small amount
  - **off-chip memory** : high latency, coarse granularity (coalescence constraint, ...), large amount
  - shared memory: kind of cache, controlled by user, data reuse inside a thread block
  - need practice to understand how to optimise global memory bandwidth



CUDA memory hierarchy: software/hardware

- **software memory hierarchy**
  - **register** : for variables private to a thread
  - **shared** : for variables private to a thread block, public for all thread inside block
  - **global** : large input data buffer



CUDA kernel calls argument

- **How data are or can be passed to a CUDA kernel ?**

```
cuda_kernel<<<grid, threads>>> ( ... );
```

- Reminder: CUDA kernels are call from the host
- By default, **arguments are passed by value** to a CUDA kernel (CUDA runtime transfer them from CPU to GPU memory), e.g. :

```
int data = 12;
// no need to copy data from CPU to GPU
cuda_kernel<<<grid, threads>>> ( data );
```

- A C language struct (POD) without pointer members can safely be passed by copy.
- If the C struct contains a pointer member, pointers must be *cudaMalloc'ed*

```
struct SomeData {
  int size;
  float *data; // this has to be cudaMalloc'ed !
};
SomeData some_data;
cuda_kernel<<<grid, threads>>> ( some_data );
```

CUDA kernel calls argument

- **How data are or can be passed to a CUDA kernel ?**

```
cuda_kernel<<<grid, threads>>> ( ... );
```

- **Can I pass a short array of data ?**

```
using array_t = int[5];
array_t an_array{0, 1, 2, 3, 4};
cuda_kernel<<<grid, threads>>> ( an_array );
```

⇒ this won't work since actually `an_array` is a pointer of type `int *`
⇒ only the address of `an_array` is transfered, but not its contents !

CUDA kernel calls argument

- **How data are or can be passed to a CUDA kernel ?**

```
cuda_kernel<<<grid, threads>>> ( ... );
```

- **Can I pass a short array of data ? solution**: change into a POD struct

```
struct array_t {
  int data[5];
}

// kernel signature - we will pass an array_t by value
__global__
void cuda_kernel(array_t a) {printf("%d\n",a.data[3]);};

array_t a = {0, 1, 2, 3, 4};
// a is passed by value
cuda_kernel<<<grid, threads>>> ( a );
```

- Could I have used an `std::array` instead ? no, not a POD.

CUDA kernel calls argument

- **How data are or can be passed to a CUDA kernel ?**

```
cuda_kernel<<<grid, threads>>> ( ... );
```

- **Can I pass a short array of data ? solution**: change into a POD struct, with a bit of sugar

```
struct array_t {
  int data[5];
  // you can access data with operator [] on host and device
  __host__ __device__
  int operator[] (int i) {return data[i];}
}

// kernel signature - we will pass an array_t by value
__global__
void cuda_kernel(array_t a) {printf("%d\n",a[3]);};

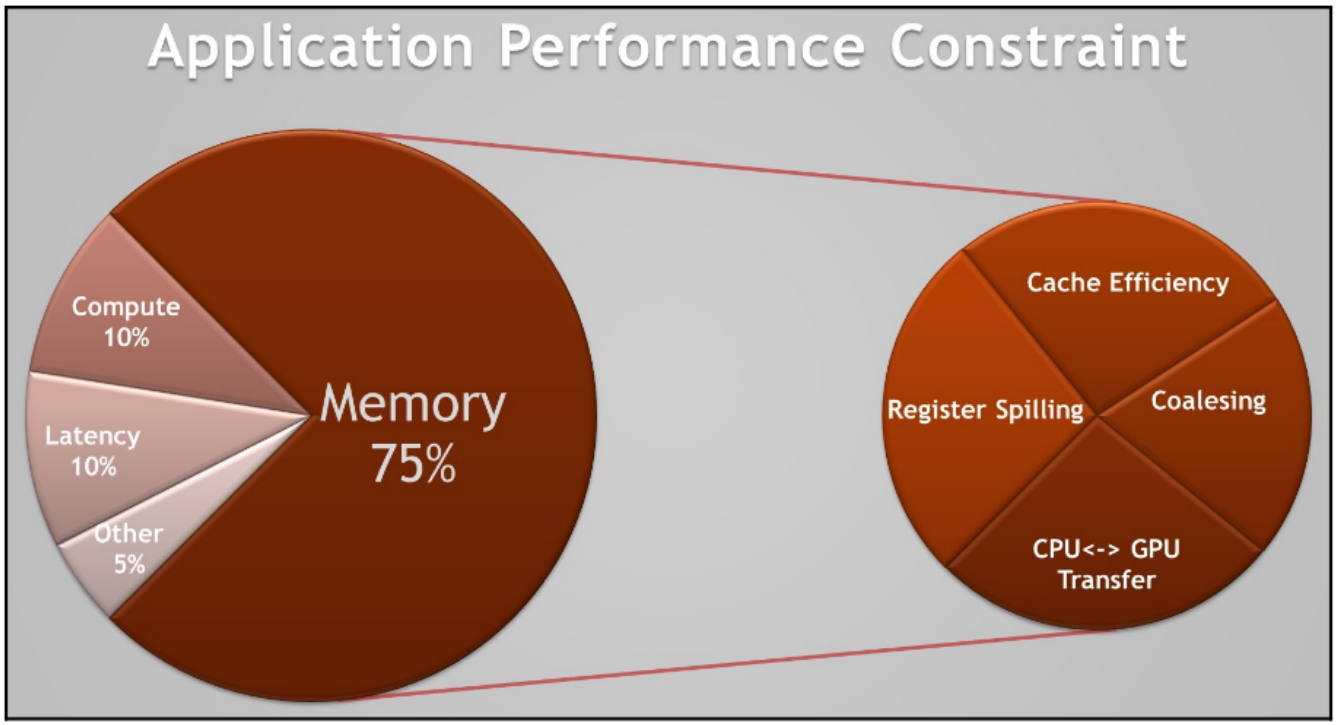
array_t a = {0, 1, 2, 3, 4};
// a is passed by value
cuda_kernel<<<grid, threads>>> ( a );
```

Performance tuning thoughts

- **Threads are free**
  - Keep threads short and balanced
  - HW can (must) use LOTs of threads (several to 10s thousands) to hide memory latency
  - HW launch ⇒ near zero overhead to create a thread
  - HW thread context switch ⇒ near zero overhead scheduling
- **Barriers are cheap**
  - single instruction: `_syncthreads();`
  - HW synchronization of thread blocks
- **Get data on GPU**, and let them there as long as possible
- **Expose parallelism**: give the GPU enough work to do
- **Focus an data reuse**: avoid memory bandwidth limitations

ref: M. Shebanon, NVIDIA

Performance tuning thoughts

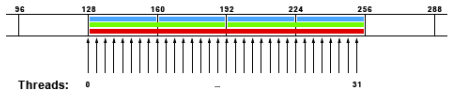


- Optimizing memory access is of prime importance.
- What is coalescing memory access ?
- What is register spiling ? How bad does it hurt performance ? How to avoid it ?

CUDA optimisation : coalescence

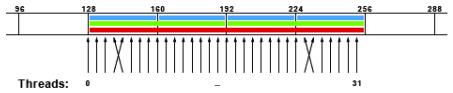
/usr/local/cuda-10.1/doc/pdf/CUDA_C_Best_Practices_Guide.pdf, SDK 10.1 (section 9.2, Device memory spaces)

Aligned and sequential



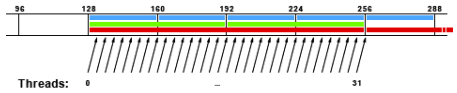
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	1x 64B at 128 1x 64B at 192	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Aligned and non- sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	8x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Mis- aligned and sequential

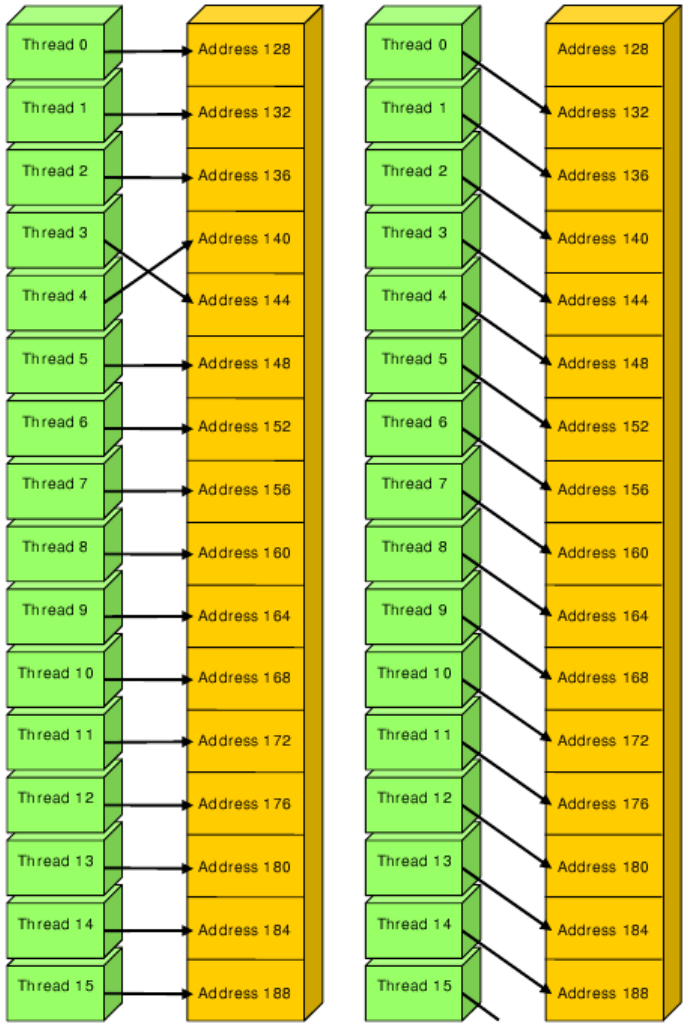


Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	7x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224 1x 32B at 256	1x 128B at 128 1x 64B at 192 1x 32B at 256	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

CUDA optimisation : coalescence

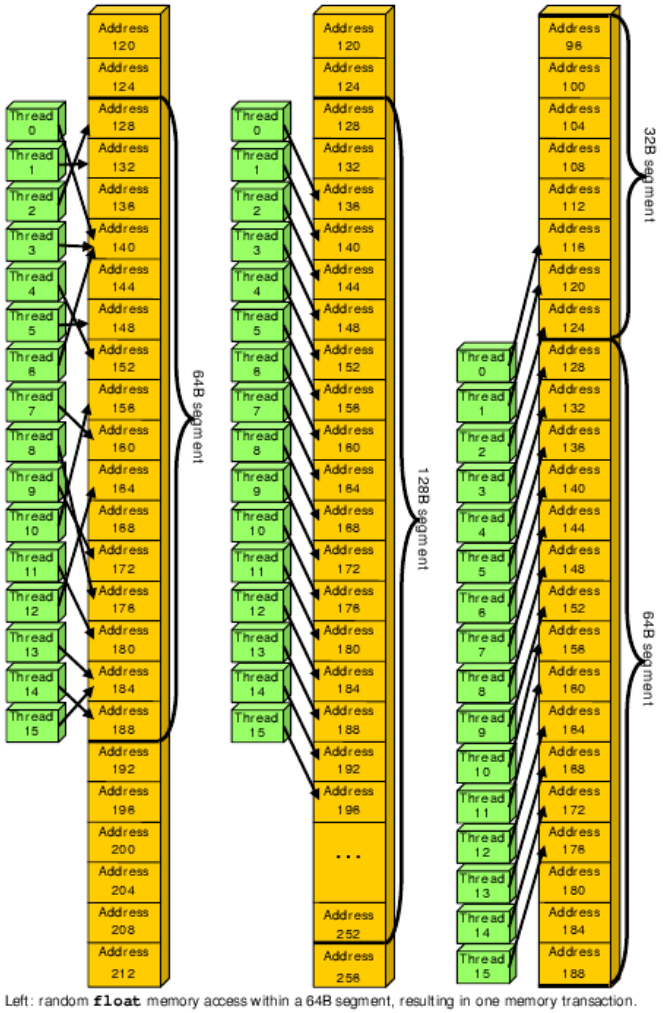
Left: coalesced **`float`** memory access, resulting in a single memory transaction.
Right: coalesced **`float`** memory access (divergent warp), resulting in a single memory transaction.

CUDA optimisation : coalescence



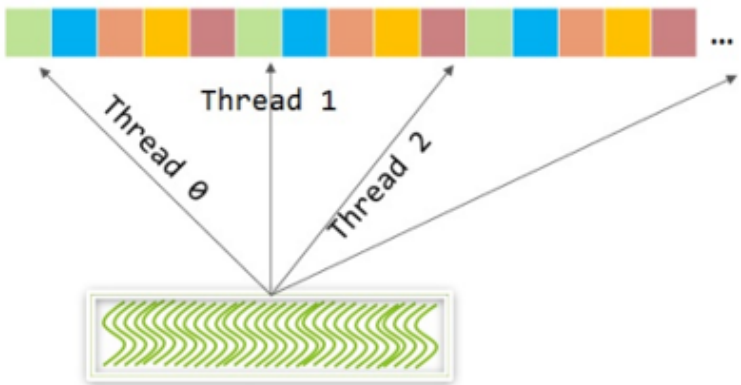
Left: non-sequential **`float`** memory access, resulting in 16 memory transactions.
Right: access with a misaligned starting address, resulting in 16 memory transactions.

CUDA optimisation : coalescence



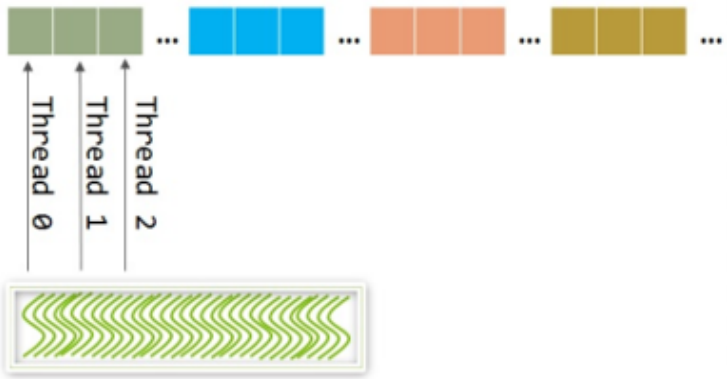
Left: random **float** memory access within a 64B segment, resulting in one memory transaction.
Center: misaligned **float** memory access, resulting in one transaction.
Right: misaligned **float** memory access, resulting in two transactions.

CUDA optimisation : coalescence




```
double u0 = gridData[threadIdx.x].r;
```

ARRAY OF STRUCTURES ACCESS PATTERN




```
double u0 = gridData.r[threadIdx.x];
```

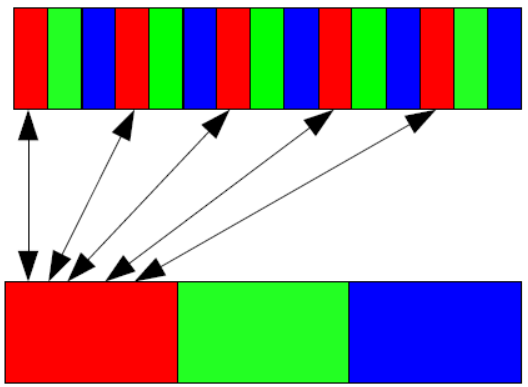
STRUCTURES OF ARRAYS ACCESS PATTERN

reference: Learn Cuda programming, Packt, 2019

CUDA optimisation : coalescence and SoA/AoS

- Array of Structures (AoS) ☹
  - 3 colors / pixel → alignment
  - complex access *pattern* to global memory
- Structure of Arrays (SoA) ☺
  - coalescence constraint by *design*
  - coalescence constraint still OK if you add more field/arrays; e.g. RGB → RGBA

```
struct Pixel {
    float r, g, b;
};
Pixel image_AoS[480][640];
```



```
struct Image {
    float R[480][640];
    float G[480][640];
    float B[480][640];
};
Image image_SoA;
```

http://perso.ens-lyon.fr/sylvain.collange/talks/calcul_gpu_scollange.pdf

Other subjective thoughts

- **tremendous rate of change in hardware from cuda 1.0 to 2.0 (Fermi) and 3.0/3.5 (Kepler)**

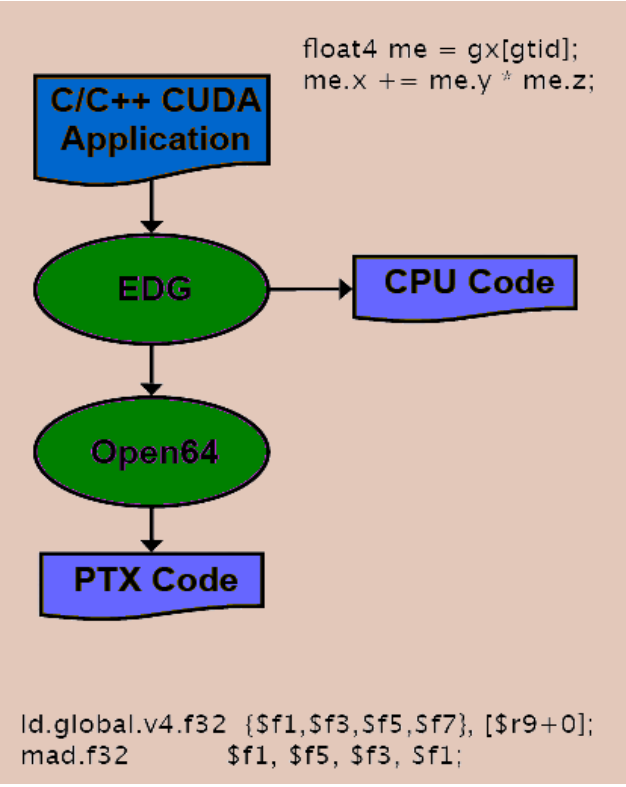
CUDA HW version	Features
1.0	basic CUDA execution model
1.3	double precision, improved memory accesses, atomics
2.0 (Fermi)	Caches (L1, L2), FMAD, 3D grids, ECC, P2P (unified address space), funtion pointers, recurs
3.5 (Kepler GK110) [1]	Dynamics parallelism, object linking, GPU Direct RemoteDMA, new instructions, read-only cache, Hyper-Q

- memory constraint like coalescence were very strong in cuda HW 1.0
  ⇒ large perf drop in memory access pattern was not coaslescent
- **Obtaining functional CUDA code can be easy but optimisation might require good knowledge of hardware (just to fully understand profiling information)**

[1] as seen in slides CUDA 5 and Beyond from GTC2012

CUDA : compilation workflow

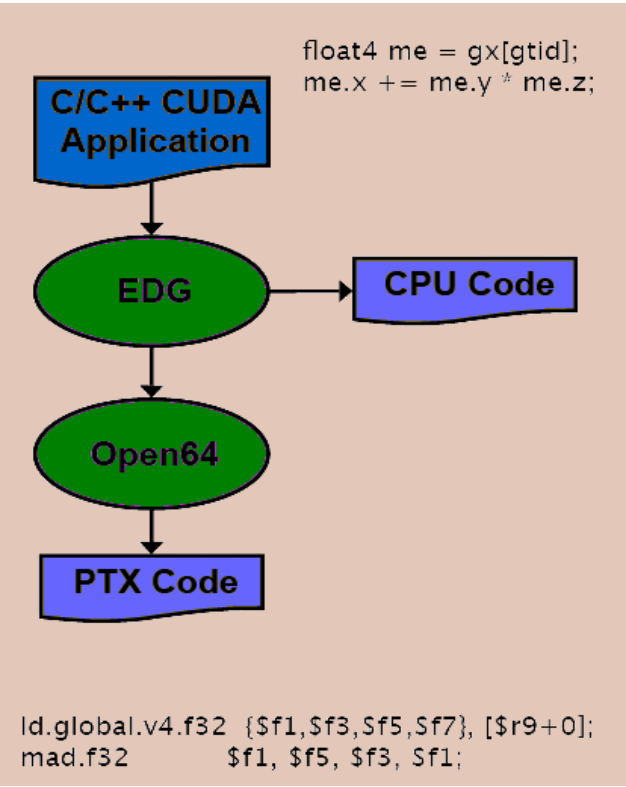
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf



- **NVCC : *compiler driver*** : call behind nvopencc (open64), gcc/g++, ...)
- **PTX : *Parallel Thread eXecution***
- PTX defines an **ISA (*Instruction Set Architecture*)** and a low-level **machine virtuelle** providing hardware abstraction (portability across GPU hardware evolution, GPU generations, scalability accross GPU sizes and number of SM)

CUDA : compilation workflow

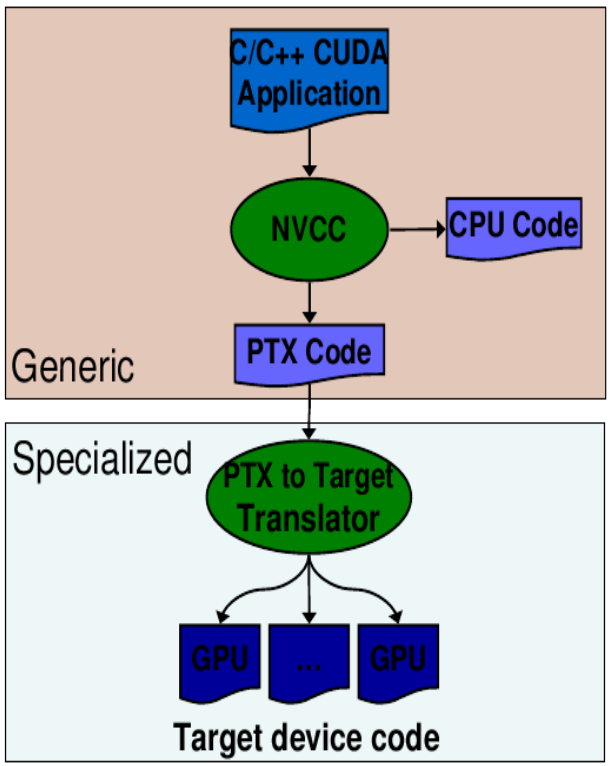
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf



- **NVCC : *compiler driver*** : call behind nvopencc (open64), gcc/g++, ...)
- **PTX : *Parallel Thread eXecution***
- High-level language compilers (e.g. nvcc) generate PTX instructions, which in a second stage, are optimized and translated into native hardware instructions (depending hardware capability).
- Possibility to define other high-level languages to target the same ISA (e.g. CUDA-Fortran)

CUDA : compilation workflow

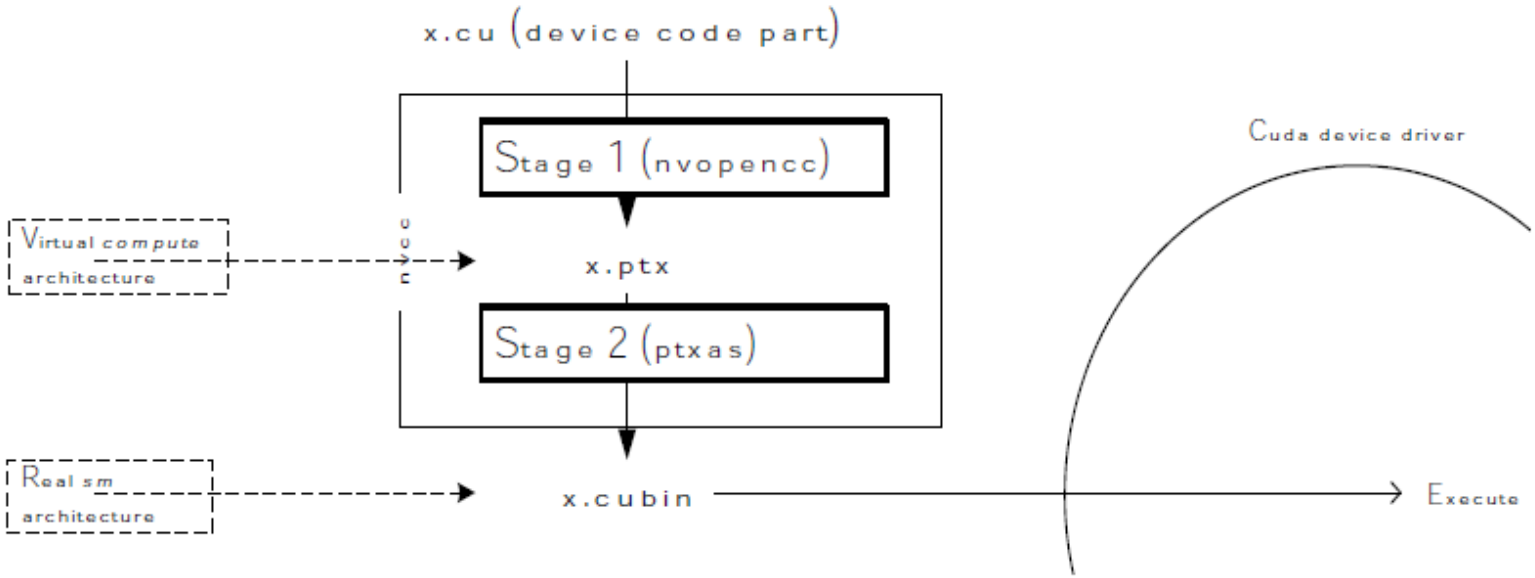
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf



- second stage uses the **ptxas tool: PTX assembly to cubin** (low-level machine instruction)
- graphics driver can also convert PTX into CUBIN (***Just-In-Time* optimisation**) and issue a PCI-express upload to GPU.

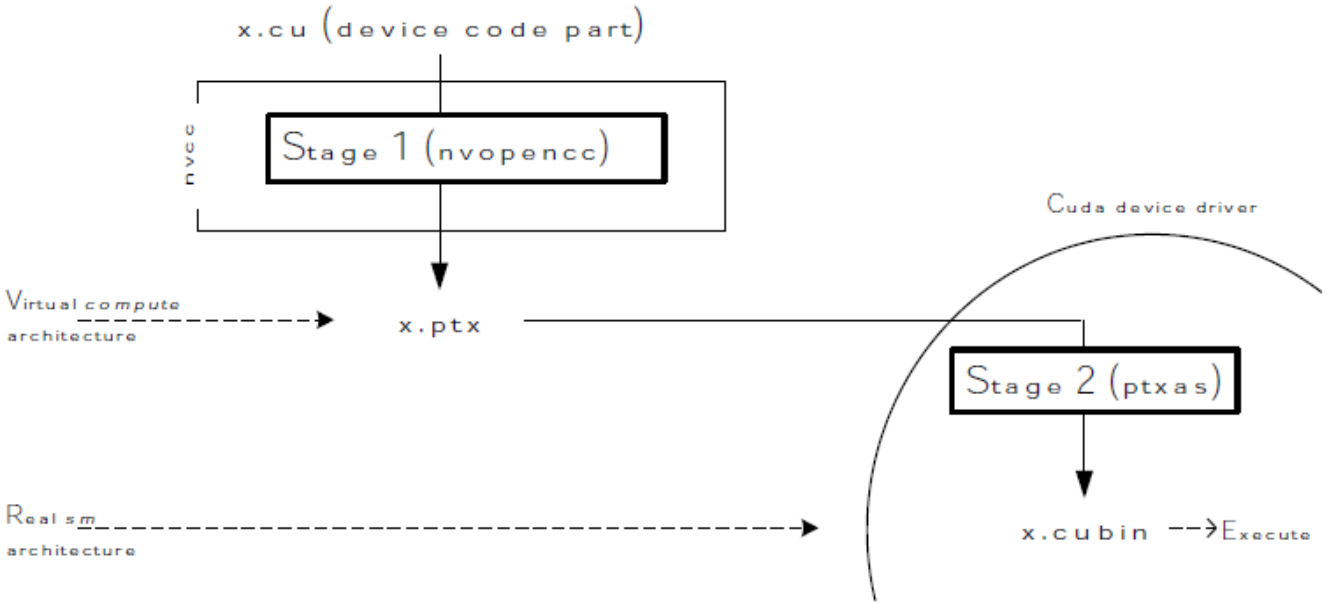
CUDA : compilation workflow

nvcc documentation : CUDA_Compiler_Driver_NVCC.pdf



CUDA : compilation workflow

nvcc documentation : CUDA_Compiler_Driver_NVCC.pdf



CUDA/C development tools

CUDA toolkit = nvcc compiler + runtime library

- any source code file with CUDA/C language extensions (.cu) needs to be compiled with `nvcc`
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
  - PTX (Parallel Thread eXecution) code
  - object code directly
- An executable with CUDA code requires:
  - The CUDA core library (cuda)
  - The CUDA runtime library (cudart)

CUDA/C++ compiler: nvcc

CUDA toolkit = nvcc compiler + runtime library

- Important flags:
  - `-arch sm_60, sm_70, sm_75, sm_80`
  - `-G` Enable debug for device code
  - `--ptxas-options=-v` Show register and memory usage
  - `-use_fast_math` Use fast math library (single precision only)
  - `--maxrregcount <N>` Limit the number of registers

Example:

```
nvcc -arch=sm_80 --ptxas-options -v -o scalarAdd scalarAdd.cu

helloworld_array helloworld_array.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z3addPiS_S_i' for 'sm_80'
ptxas info    : Function properties for _Z3addPiS_S_i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
```

Cuda profiling / performance analysis tools

- **before 2019:** use `nvprof` or `nvvp` but these tools are deprecated;
- **since 2019:** use `nsight-sys` instead (**recommended, but need recent hardware**)
- **Currently you need admin rights to run** `nvprof` or `nsight-sys` see nvidia-development-tools-solutions-err-nvgpuctrperm-nvprof website to fix and allows regular user to use profiling (a bit complex at first)
- Summary to allow permanently regular users to access profiling counters
  - As admin, edit file `/etc/modprobe.d/nvidia-settings.conf` and add `options nvidia "NVreg_RestrictProfilingToAdminUsers=0"`
  - Rebuild initial ram disk `sudo update-initramfs -u`
  - Reboot system
  - check that changes are taken into account (RmProfilingAdminOnly should be 0): `cat /proc/driver/nvidia/params | grep RmProf`

Cuda profiling / performance analysis tools

- On Ubuntu 18.04, you will surely need to install `openjdk-8` (instead of `openjdk-11`) to use `nvvp`.
- ```
  # install openjdk version 8
  sudo apt install openjdk8-jre
  # then make it the default jdk, select jdk8 among all option
  sudo update-alternatives --config java
  ```


  ```

GPU : floating point computation support

Floating point computations capability implemented in GPU hardware

- **IEEE754 standard** written in mid-80s
- Intel 80387 : first floating-point coprocessor IEEE754-compatible
- Value = $(-1)^S \times M \times 2^E$, denormalized, infinity, NaN; rounding algorithms quite complex to handle/implement
- FP16 in 2000
- **FP32 in 2003-2004** : simplified IEEE754 standard, float point rounding are complex and costly in terms of transistors count,
- **CUDA 2007** : rounding computation fully implemented for + and * in 2007, denormalised number not completed implemented
- **CUDA Fermi : 2010** : 4 mandatory IEEE rounding modes; Subnormals at full-speed (Nvidia GF100)
- links: http://perso.ens-lyon.fr/sylvain.collange/talks/raim11_scollange.pdf

GPU : floating point computation support

Floating point computations capability implemented in GPU hardware

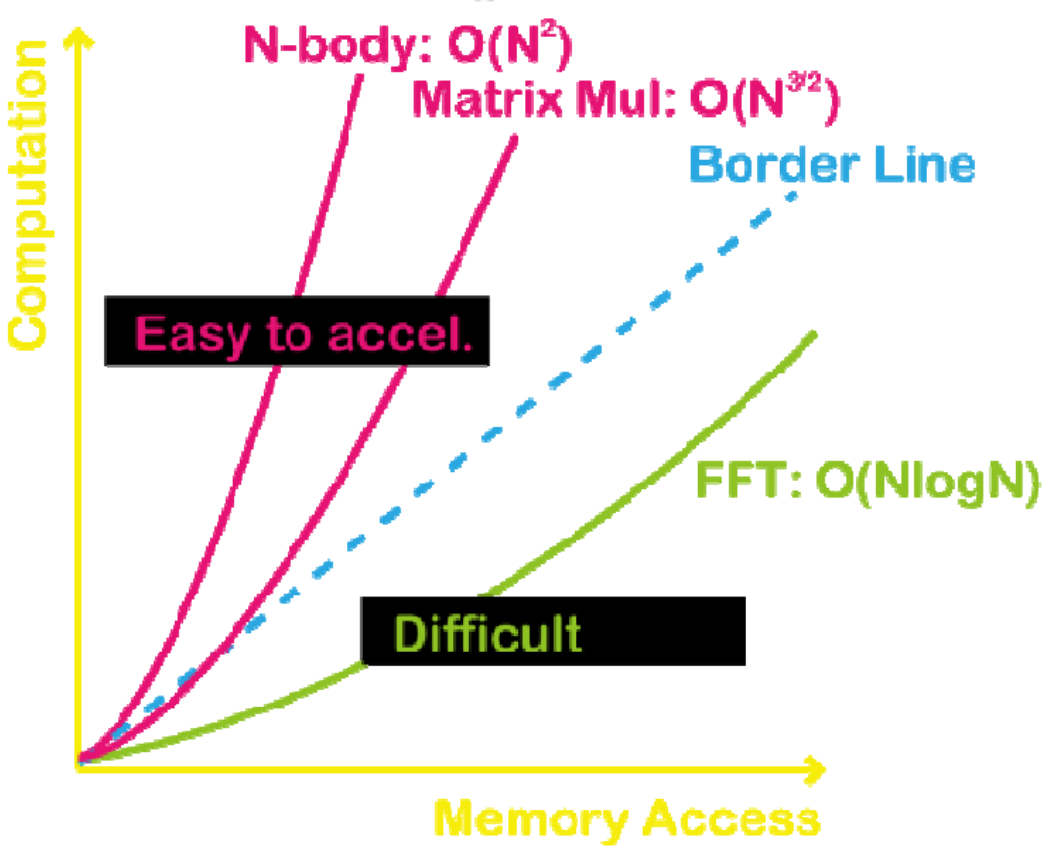
- **Why are my floating results different on GPU from CPU ?**
  - cannot expect always the same results from different hardware
  - **one algorithm**, but 2 different software implementations for 2 different hardware CPU and GPU
  - **Hardware differences:**
    - CPU floating point unit (FPU) might use x87 instructions (with 80 bits precision used internally); SSE operations use 32 or 64 bits values.
    - Compiler dependency: values kept in register ? or spilled to external memory ?
  - **Fused Multiply-Add from IEEE754-2008**
    - $a * b + c$ in a single rounding step; better precision
    - implemented on all GPU from CUDA hardware ≥ 2.0 but not all CPU (AMD/Bulldozer/2011, Intel/Haswell/2013,...)
    - use `nvcc` flags `-fmad=false` to tell compiler not to use FMAD instructions
    - see code snipnet in hands-on
- CUDA doc : Floating_Point_on_NVIDIA_GPU_White_Paper.pdf

GPU : floating point computation support

Floating point computations capability implemented in GPU hardware

- **Why are my floating results different on GPU from CPU ?**
  - **parallel computations rearrange operations**; associativity $((a+b)+c \neq a+(b+c)) \Rightarrow$ different the results; see reduction example
  - **Different ? By how much ?** Can be very hard to distinguish differences coming from parallel computation from a genuine bug in the algorithm. When comparing a reference CPU algorithm with the GPU ported algorithm, **observing differences do not necessarily imply there is bug in your GPU code !!**
  - Take care that single precision uses 23 bits for mantissa ($2^{-23} \sim 10^{-7}$) and double precision 52 bits ($2^{-52} \sim 10^{-16}$): **CPU/GPU differences will accumulate much faster in single precision**
  - **Allways a good idea to use double precision for test/debug**, even if target precision is single precision (e.g use a `typedef real_t`)
- CUDA doc : Floating_Point_on_NVIDIA_GPU_White_Paper.pdf

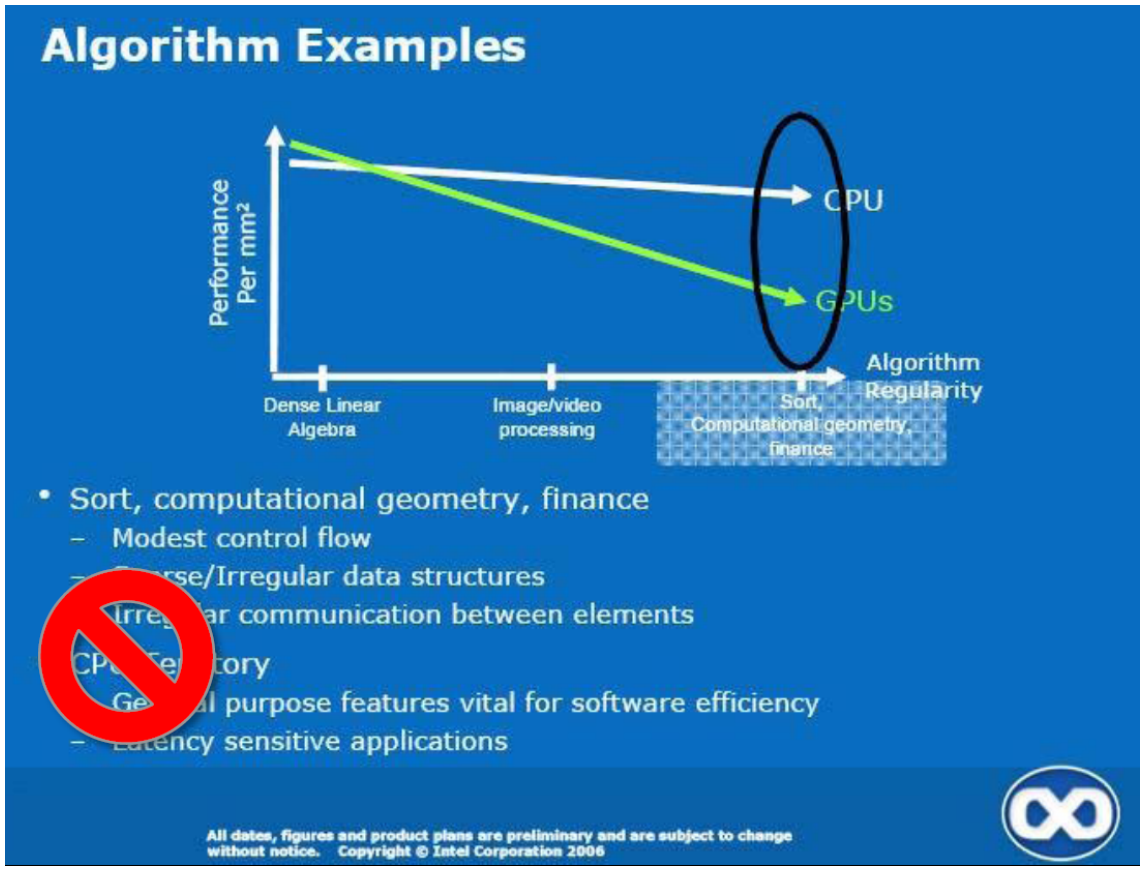
What kinds of computation map well to GPUs ?



Nukada *et al.*, SC08

http://www.nvidia.com/content/GTC-2010/pdfs/2084_GTC2010.pdf

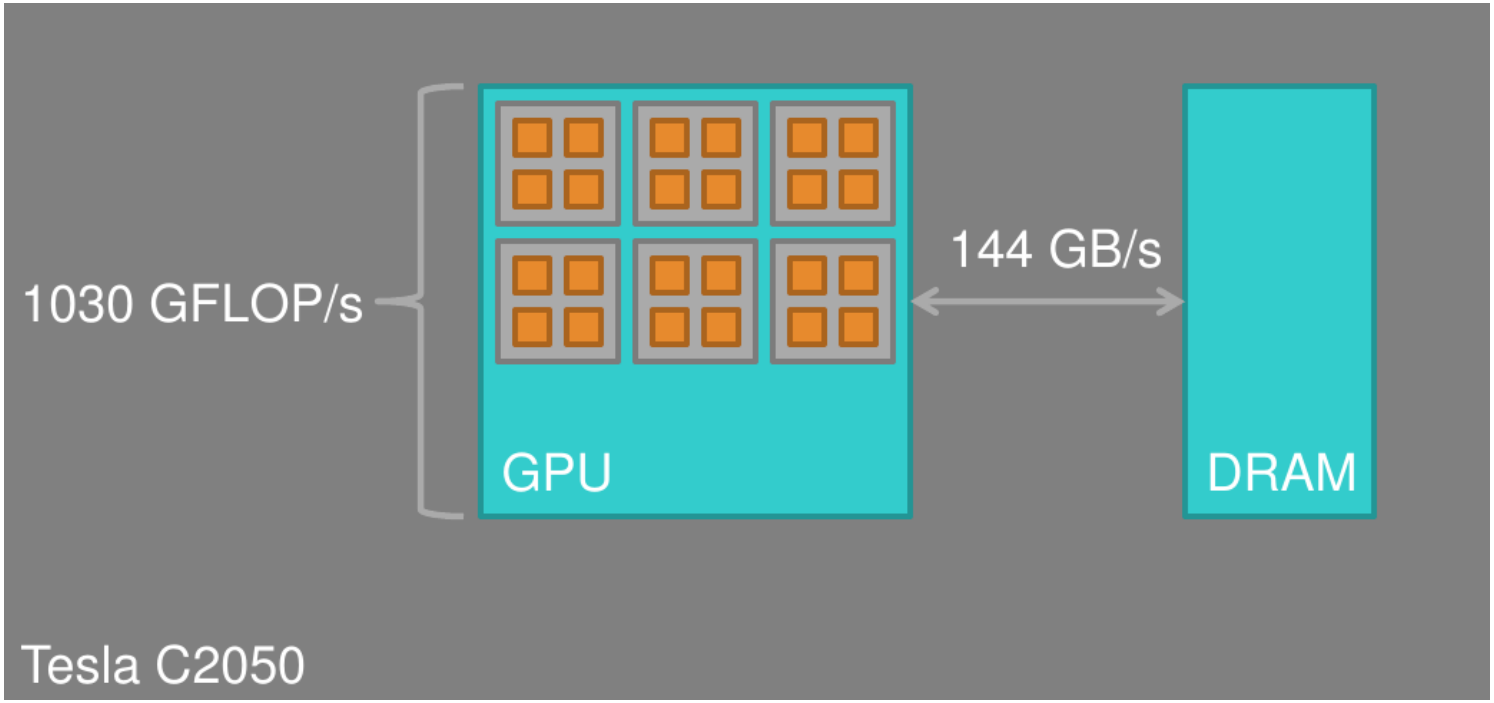
What kinds of computation map well to GPUs ?



Nukada *et al.*, SC08

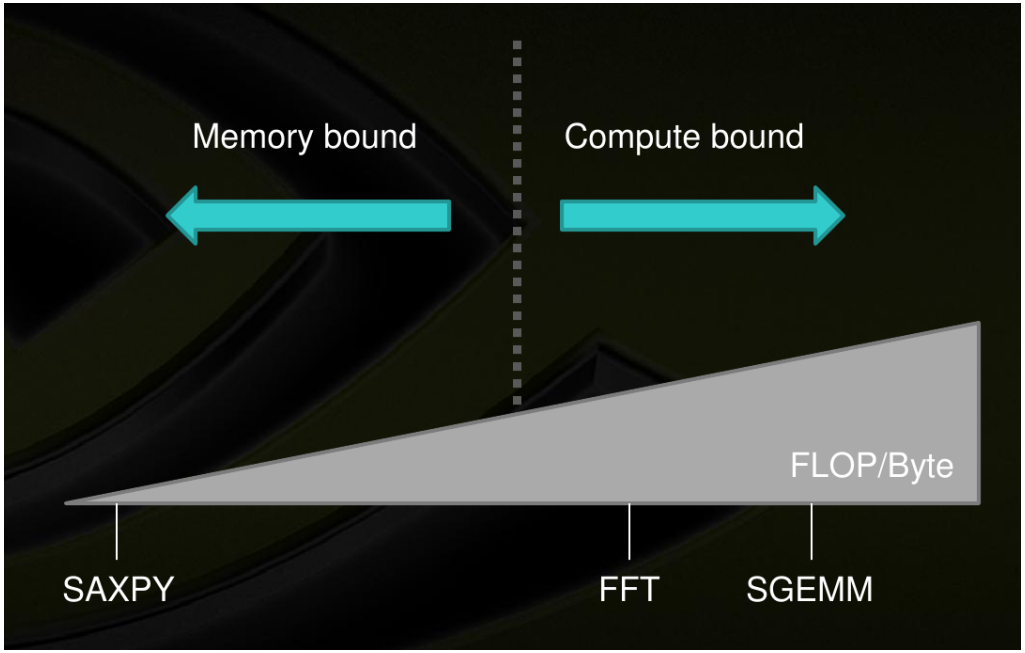
http://www.nvidia.com/content/GTC-2010/pdfs/2084_GTC2010.pdf

What kinds of computation map well to GPUs ?



- **System FLOP/Byte** is quite high for GPU, significantly higher than multi-core CPU
- Actual FLOP/Byte is of course **algorithm depend**

What kinds of computation map well to GPUs ?



- **Memory-bound algorithm:** need special care about external memory handling (coalescence, SAO-SOA, etc...), see CUDA SDA reduction example
- **Compute-bound algorithm:** ideal for GPU data-parallel architecture

What kinds of computation map well to GPUs ?

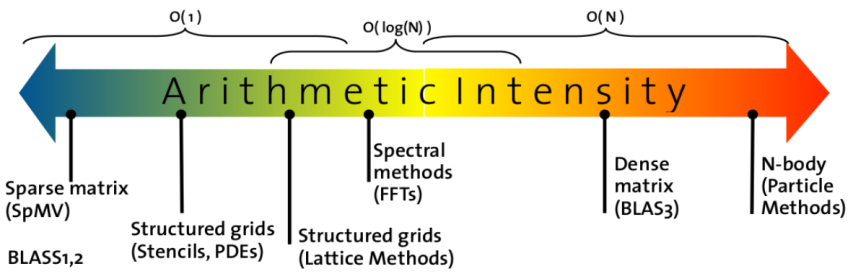
Kernel	FLOP/Byte**
Vector Addition	1 : 12
SAXPY	2 : 12
Ternary Transformation	5 : 20
Sum	1 : 4
Max Index	1 : 12

** excludes indexing overhead

Kernel	FLOP/Byte
GeForce GTX 280	~7.0 : 1
GeForce GTX 480	~7.6 : 1
Tesla C870	~6.7 : 1
Tesla C1060	~9.1 : 1
Tesla C2050	~7.1 : 1

Roofline model

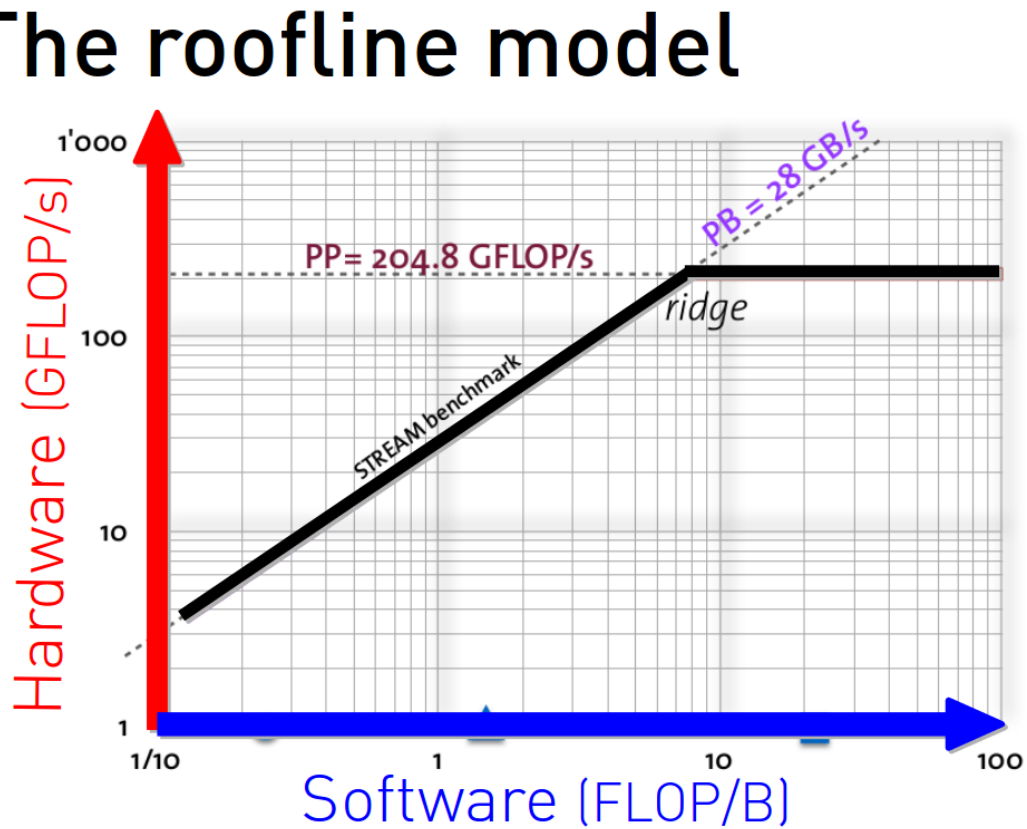
- an increasingly large diversity of architectures
- software challenges to use new architectures:
  - refactoring (when ? where ?); avoid sub-optimal use of hardware/software
  - optimization strategies
- **Roofline model:** a simple way of characterizing hardware performances
  Each algorihm implementation is characterized by
  - **arithmetic intensity (FLOPS/Bytes):** number of FLOP per bytes read/write from external memory
  - **effective memory bandwidth (GB/s):** data moved to/from external RAM



reference:

http://www.nvidiacodesignlab.ethz.ch/news/CoDesignLabWorkshop2013_Rossinelli_Roofline.pdf

Roofline model

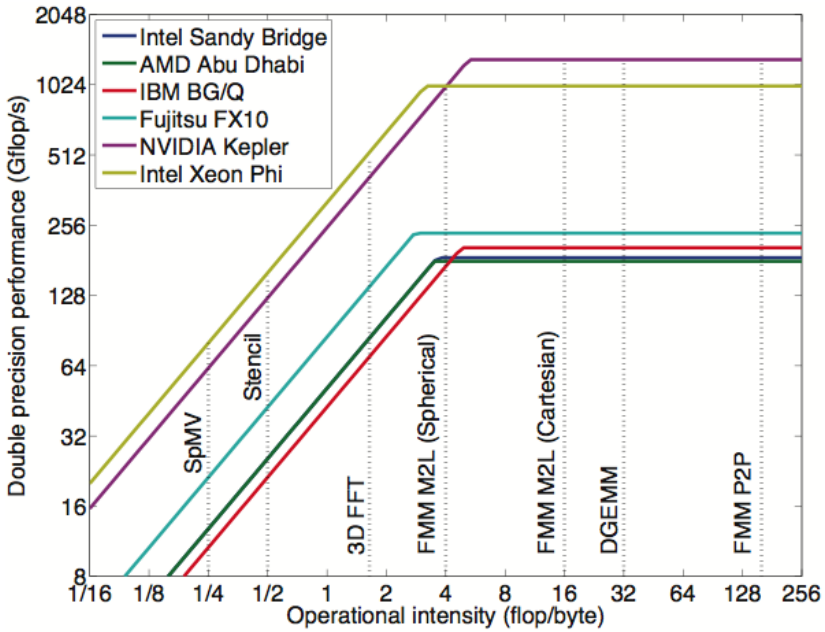


- it visually relates hardware with software
- Performance = min (Peak Bandwidth * Arith Intensity, Peak Flops)

reference:

http://www.nvidiacodesignlab.ethz.ch/news/CoDesignLabWorkshop2013_Rossinelli_Roofline.pdf

About roofline model of current (future ?) hardware



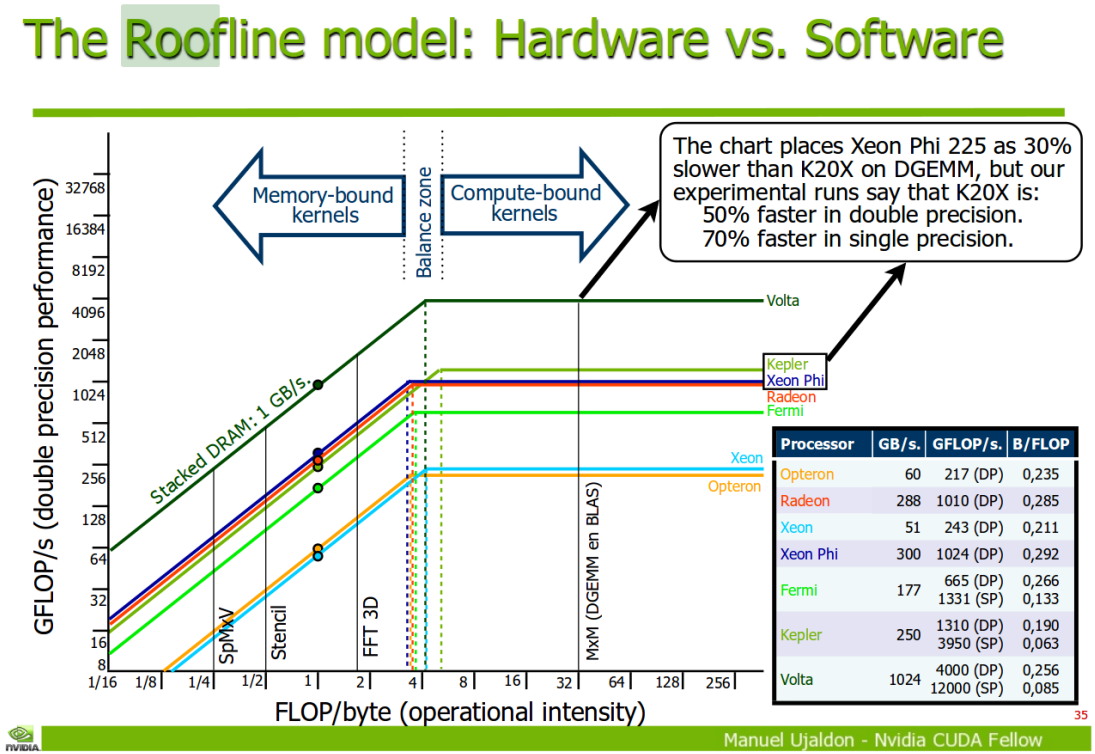
- Understand inherent hardware limitations
- Show priority of optimization

references:

http://lorenabarba.com/news/fast-multipole-method-in-the-exascale-era/

http://icpp2013.ens-lyon.fr/GPUs-ICPP.pdf

About roofline model of current (future ?) hardware



- Understand inherent hardware limitations
- Show priority of optimization

references:

http://lorenabarba.com/news/fast-multipole-method-in-the-exascale-era/

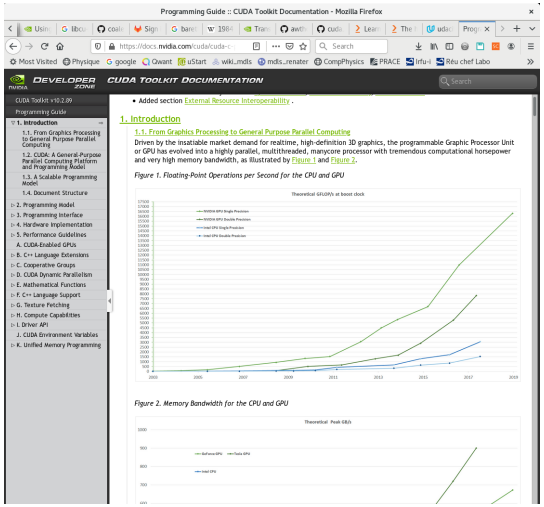
http://icpp2013.ens-lyon.fr/GPUs-ICPP.pdf

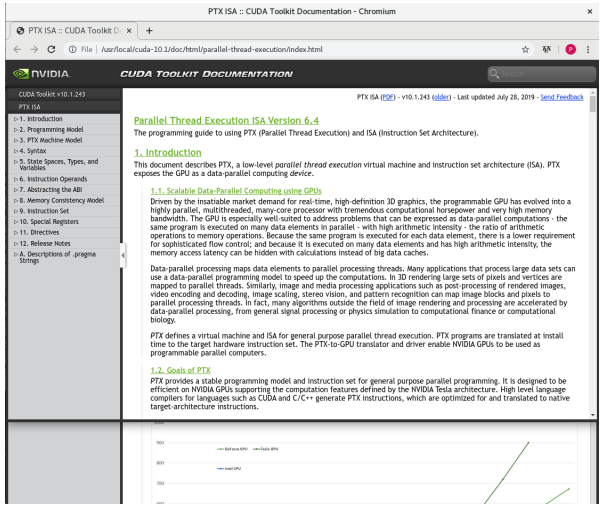
Reduction algorithm

- reduction example: compute $S = \sum_{i=0}^{N-1} data[i]$
- Effcient massively multithread parallel reduction implementation is actuallu a hard job, but fortunately very well documented
- We will study reduction from the CUDA sdk during hands-on.
- Revisit reduction in light of the new cooperative groups concepts introduced in CUDA 9.0, along with recent hardware (Pascal, Turing) which support grid-wide (and even cross-GPU) synchronization.

CUDA Recommended reading

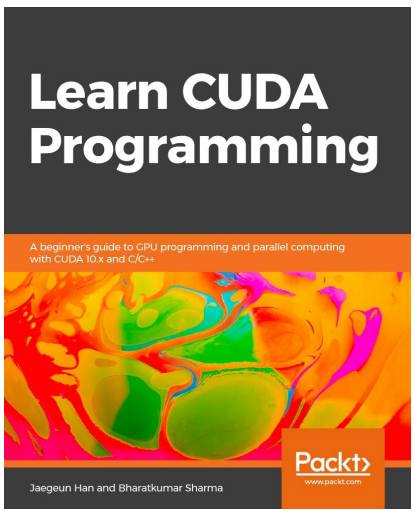
- NVIDIA toolkit 10.1 documentation (pdf and html): cuda-c-programming-guide
- NVIDIA PTX (Parallel Thread Execution) documentation : parallel-thread-execution

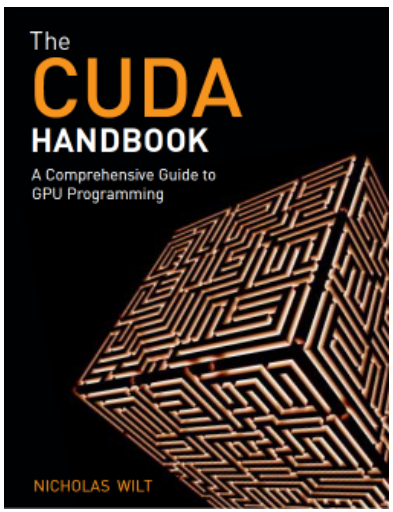




CUDA Recommended reading

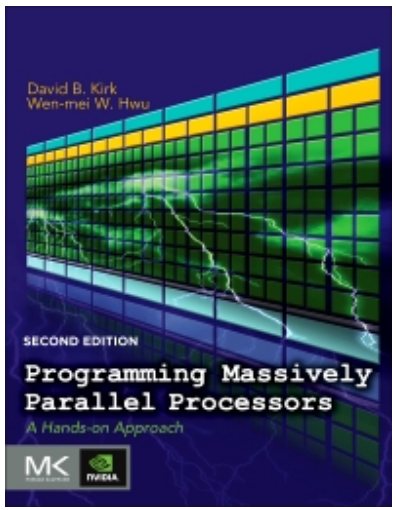
- NVIDIA toolkit 10.1 documentation (pdf and html): cuda-c-programming-guide
- command line `man` is your friend (make sure to append `/usr/local/cuda-10.1/doc/` in env variable `MANPATH`)
- book ***Learn CUDA programming***, 2019 by B. Sharma and J. Han
- book ***The CUDA Handbook***, 2013, by N. Wilt; 2nd edition in 2020

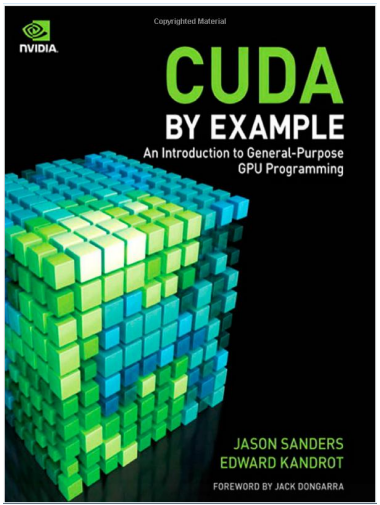


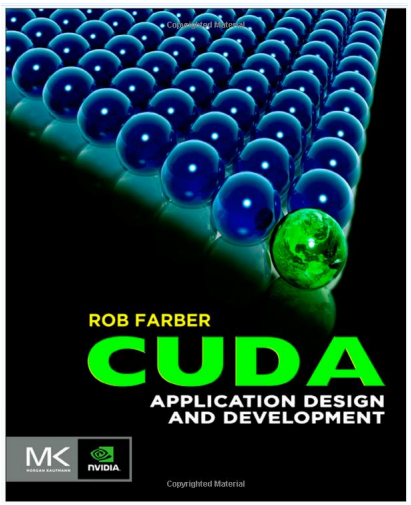


CUDA Recommended reading

- book ***CUDA by example***, 2010, by J. Sanders and E. Kandrot from NVIDIA
- book ***CUDA Application Design and Development***, 2011, by R. Farber
- book ***Programming Massively Parallel Processors: a hands-on approach***, 2nd ed., 2012, by D. Kirk and W.-M. Hwu

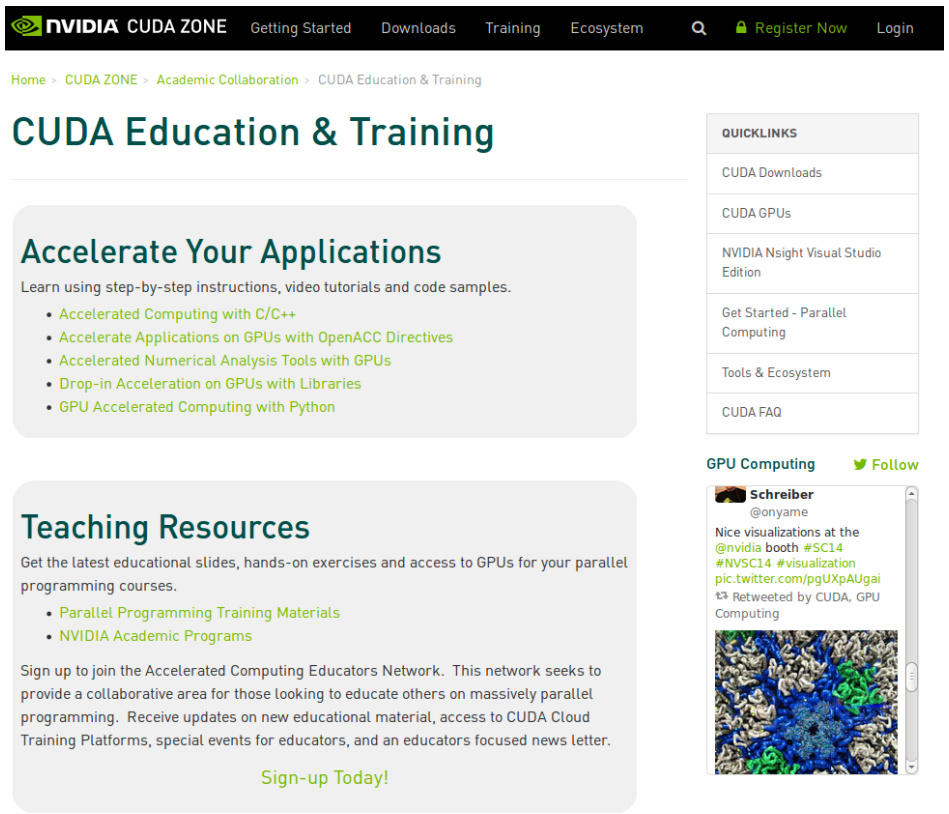






CUDA On-line materials

Learning and teaching GPU programming:
https://developer.nvidia.com/cuda-education-training



- gpu-edu-workshops cuda code on github.

CUDA On-line materials

- GPU Computing webinars:
  http://www.gputechconf.com/resources/gtc-express-webinar-program
- GPU Technology Conference resources : select for example *GTC*, you'll get numerous material about:
  - Algorithms and numerical techniques
  - Astrophysics, Image processing, Computer Vision, Bioinformatics, Climate, Cloud, CFD, Data Mining
  - Development tools, libraries
- https://moderngpu.github.io/intro.html: a very insightful presentation by S. Baxter
- List / comparison of Nvidia GPUs:
  http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

Other books for a computational scientist

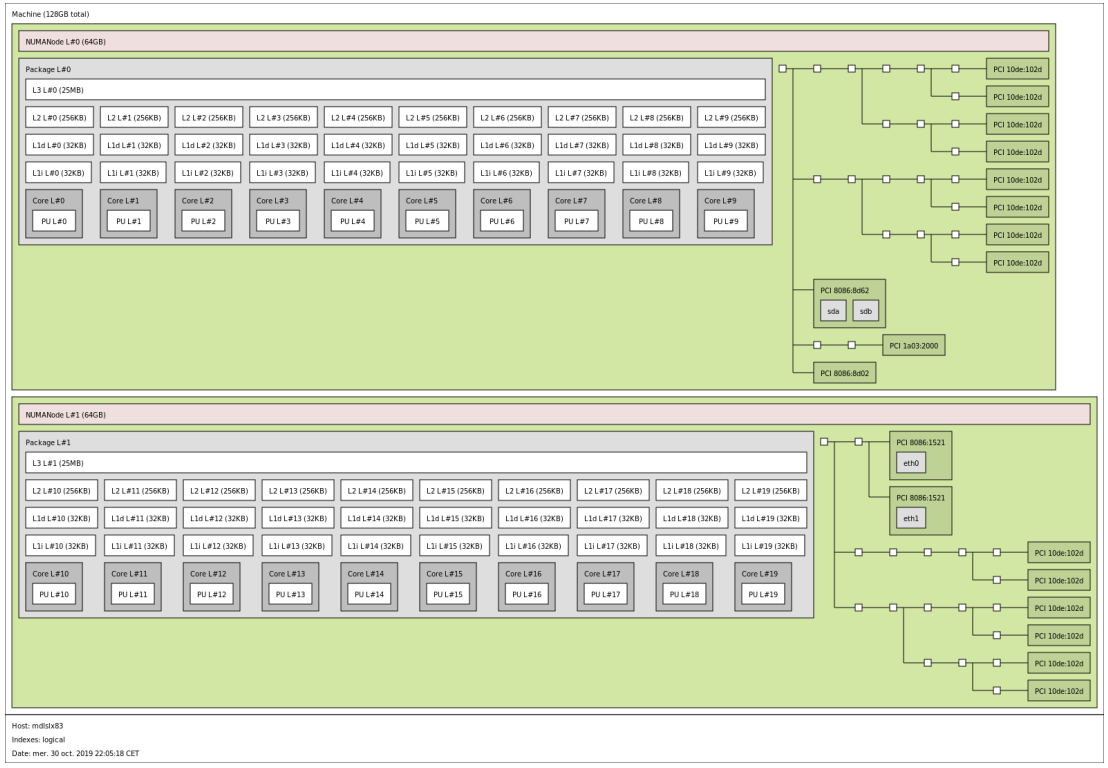
- A book on several aspect of HPC:
  Introduction to High-Performance Scientific Computing by Victor Eijkhout
- Computational Physics and Scientific computing in C++ by KONSTANTINOS N. ANAGNOSTOPOULOS, National Technical University of Athens

Other resources for a computational scientist

- Node-level performance engineering slides (advanced level, very insightful)

Probe CPU hardware information

Use hwloc / lstopo to probe the number of CPU sockets, number of cores per socket, cache sizes, etc ...



Probe GPU hardware information

Use `deviceQuery` sample code from
`/usr/local/cuda-10.1/samples/1_Utilities`

```
pkestene@mdlslx79:~/NVIDIA_CUDA-10.1_Samples/1_Utilities/deviceQuery $ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

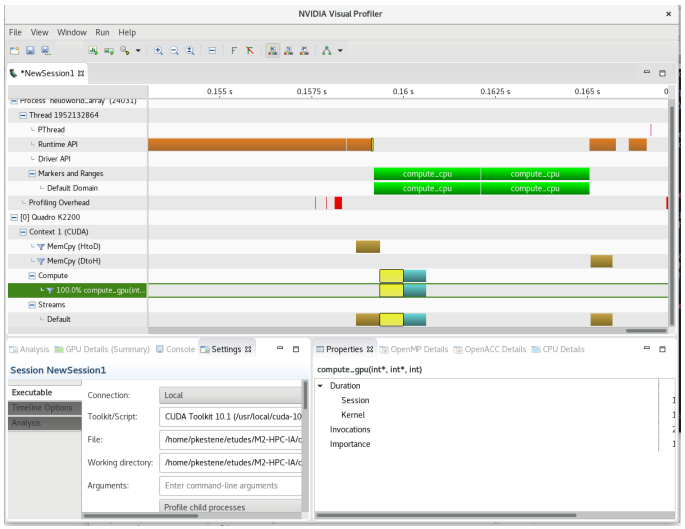
Detected 1 CUDA Capable device(s)

Device 0: "Quadro K2200"
  CUDA Driver Version / Runtime Version          10.2 / 10.1
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 4043 MBytes (4239785984 bytes)
  ( 5) Multiprocessors, (128) CUDA Cores/MP:     640 CUDA Cores
  GPU Max Clock rate:                            1124 MHz (1.12 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 3 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.2, CUDA Runtime Version = 10.1, NumDevs = 1
Result = PASS
pkestene@mdlslx79:~/NVIDIA_CUDA-10.1_Samples/1_Utilities/deviceQuery $
-rw-r--r--  1 root root 60244 Aug  9 21:47 EULA.txt
```

Cuda error handling

- **Important reminder :** GPU execution computation is **asynchronous** wrt CPU, i.e. returns to host rightaway
- As consequence, **error reporting** from a GPU execution to the host (CPU) is also **asynchronous**
  - error status can be probed and retrieve at some point later, after the kernel execution has finished
  - or get it rightaway, but the price to pay is a synchronization barrier



Look at the green box (**CPU**) and yellow one (**GPU**)

```
// the following call is asynchronous
//  wrt the CPU
compute_gpu<<<gridSize,blockSize>>>( ... );
// perform computation on CPU, that
// run in parallel with the CPU
compute_cpu( ... );
```

Cuda error handling / reporting

Two types of error reporting : API and kernel launch

```
#include "cuda_error.h"

// check status of a CUDA runtime API call
CUDA_API_CHECK( cudaMalloc( (void**)&dev_a, N*sizeof(int) ) );

// check status of a CUDA kernel execution
compute_gpu<<<gridSize,blockSize>>>( ... );
CUDA_KERNEL_CHECK("compute_gpu");
```

CUDA API error reporting

```
/**
 * Preprocessor macro helping to retrieve the exact code
 * location where the error was emitted.
 */
#define CUDA_API_CHECK(value) cuda_api_check((value), #value, __FILE__, __LINE__)

/**
 * Check CUDA API call status (e.g. cudaMemcpy for memory allocation)
 * see https://docs.nvidia.com/cuda/cuda-runtime-api/index.html
 */
static void cuda_api_check(cudaError_t       status,
                           const char *const func,
                           const char *const file,
                           const int         line)
{

  if (status != cudaSuccess) {
    fprintf(stderr, "CUDA API error at %s:%d code=%d(%s) \"%s\" \n",
    file, line,
    static_cast<unsigned int>(status),
    cudaGetErrorName(status), func);

    // Make sure we call CUDA Device Reset before exiting
    cudaDeviceReset();
    exit(EXIT_FAILURE);
  }

} // cuda_api_check
```

CUDA kernel execution error reporting

```
/**
 * Check last CUDA kernel call status.
 *
 * \param[in] errstr error message to print
 * \param[in] file source filename where error occured
 * \param[in] line line number where error occured
 * \param[in] sync integer, 0 means no device synchronization
 */
static void cuda_kernel_check(const char* errstr,
                              const char* file,
                              const int   line,
                              const int   sync)
{
  // optionally : sync device to force GPU ends computation
  // before probing status
  if (sync or FORCE_SYNC_GPU) {
    cudaDeviceSynchronize();
  }

  auto status = cudaGetLastError();
  if (status != cudaSuccess) {
    fprintf(stderr,
            "%s(%i) : getLastCudaError() CUDA error :"
            " %s : (%d) %s.\n",
            file, line, errstr, static_cast<int>(status),
            cudaGetErrorString(status));
  }
} // cuda_kernel_check
```

CUDA and PYTHON

There are multiple ways to do GPU computing in python:

1. Drop-in replacement for simple kernels (with numpy interoperability)
   - numba, cupy, pycuda
2. Inlining CUDA kernels as strings + JIT compilation
   - requires CUDA/C++ knowledge
   - numba, cupy, pycuda
3. C/C++ extension
   - see `https://docs.python.org/3/extending/extending.html`
   - swig (a bit deprecated), cython, pybind11, cppyy
   - graalpython ? legate ?

ref: CUDA in your Python: Effective Parallel Programming on the GPU video on YouTube from Pytexas2019 conference.

Using GPU hardware inside python app : CuPy

Using cupy as a drop-in replacement for numpy:

Python example on CPU with numpy:

```
import numpy as np
x = np.random.randn(10000000).astype(np.float32)
y = np.random.randn(10000000).astype(np.float32)
z = x + y
```

Using GPU hardware inside python app: CuPy

Using cupy as a drop-in replacement for numpy:

Python example on GPU with cupy

```
import cupy as cp
x = cp.random.randn(10000000).astype(np.float32)
y = cp.random.randn(10000000).astype(np.float32)
z = x + y
```

Using GPU hardware inside python app : Numba

What is Numba ?

- Translation of python functions to machine code at runtime using the LLVM compiler library
- Designed to be used with NumPy arrays
- Options to parallelize code for CPUs and GPUs and automatic SIMD Vectorization
- Support for both NVIDIA's CUDA and AMD's ROCm driver allowing to write parallel GPU code from Python.

Accelerate python app with Numba

Serial CPU version - pur python

```
def axpy(x,y,a):
    for i in range(x.shape[0]):
        x[i] = a*x[i] + y[i]
```

Accelerate python app with Numba

Serial CPU version - compiled to machine (no python interpreter)

```
import numba

@numba.jit(nopython=True)
def axpy(x,y,a):
    for i in range(x.shape[0]):
        x[i] = a*x[i] + y[i]
```

Accelerate python app with Numba

Parallel CPU version - multithreading + SIMD

`range` changed into `prange`

```
import numba

@numba.jit(nopython=True, parallel=True)
def axpy(x,y,a):
    for i in prange(x.shape[0]):
        x[i] = a*x[i] + y[i]
```

ref: https://github.com/numba/numba-examples/blob/master/notebooks/threads.ipynb

Accelerate python app with Numba and CUDA

Parallel GPU version - CUDA

`range` changed into `prange`

```
import numba

@numba.cuda.jit('void(float32[:],float32[:])')
def axpy(x,y,a):
    i = cuda.grid(1)
    # i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if i < x.shape[0]:
        x[i] = a*x[i] + y[i]
```

ref: https://github.com/numba/numba-examples/blob/master/notebooks/threads.ipynb

Numba reduction on CPU

```
import numba

@numba.jit(nopython=True)
def reduce(x):
    x_sum = 0.0
    for i in range(x.shape[0]):
        x_sum += x[i]
    return x_sum
```

Numba reduction on GPU

```
import numpy
from numba import cuda

@cuda.reduce
def sum_reduce(a, b):
    return a + b

A = (numpy.arange(1234, dtype=numpy.float64)) + 1
expect = A.sum()        # numpy sum reduction
got = sum_reduce(A)     # cuda sum reduction
assert expect == got
```

https://numba.pydata.org/numba-doc/dev/cuda/reduction.html

Accelerate python app with pycuda

Inlining Cuda/C++ as python string with pycuda

```
mod = SourceModule("""
    void __globall kernel_add_arrays(float *a, float *b, float *
      int gid = threadIdx.x + blockDim.x*blockIdx.x;
      while (gid < N) {
        c[gid] = a[gid] + b[gid];
        gid += blockDim.x*gridDim.x;
      }
    }
""")
```

then gets a callable object (this is where cuda kernel is compiled) for launching GPU computation

```
func = mod.get_function("kernel_add_arrays")
```

Additional resources

- template project for Cuda/python bindings using swig or cython : `https://github.com/pkestene/npcuda-example`
- template project for Cuda/python bindings using pybind11 and modern cmake : `https://github.com/pkestene/pybind11-cuda`

Numba + Cupy

A very good starting point:
https://github.com/ContinuumIO/gtc2020-numba

1. install miniconda
2. `conda install jupyter notebook`
3. install cupy (be sure to use pip from miniconda): `pip install pip install cupy-cuda101`
4. run `jupyter notebook` and open one of the tutorial notebook

Numba + Cupy

Additionnal notes:

- By default, you don't need to install `cudatoolkit` from conda, if your Linux OS already has cuda in `/usr/local/cuda`

Numba snippets

minimal example in numba/cuda

```
import numpy as np
from numba import cuda

# create a CPU numpy array
arr = np.arange(1000)

# allocate a GPU array, and copy from host
d_arr = cuda.to_device(arr)

# cuda kernel launch
my_kernel[100, 100](d_arr)

# copy back results on host
result_array = d_arr.copy_to_host()
```

Numba snippets - device functions

Numba equivalent to `__device__ function` in cuda/c++:

device function in numba

```
from numba import cuda

@cuda.jit(device=True)
def a_device_function(a, b):
    return a + b
```

Reminder: device functions are functions that can only be call inside a CUDA kernel or inside another device function

Numba snippets - cuda kernels

CUDA kernel : sum of two 1D array

```
@cuda.jit
def max_example(a,b,c):
    """c = a + b"""
    tid = cuda.threadIdx.x
    bid = cuda.blockIdx.x
    bdim = cuda.blockDim.x

    start = (bid * bdim) + tid
    stride = cuda.blockDim.x * cuda.gridDim.x
    size = a.shape[0]

    for i in range(start, size, stride):
        c[i] = a[i] + b[i]
```

Numba snippets - cuda kernels for reduction

CUDA kernel : reduce example

```
"""https://numba.pydata.org/numba-doc/dev/cuda/reduction.html"""
@cuda.reduce
def sum_reduce(a, b):
    return a + b

A = (numpy.arange(1234, dtype=numpy.float64)) + 1
expect = A.sum()        # numpy sum reduction
got = sum_reduce(A)     # cuda sum reduction
assert expect == got
```

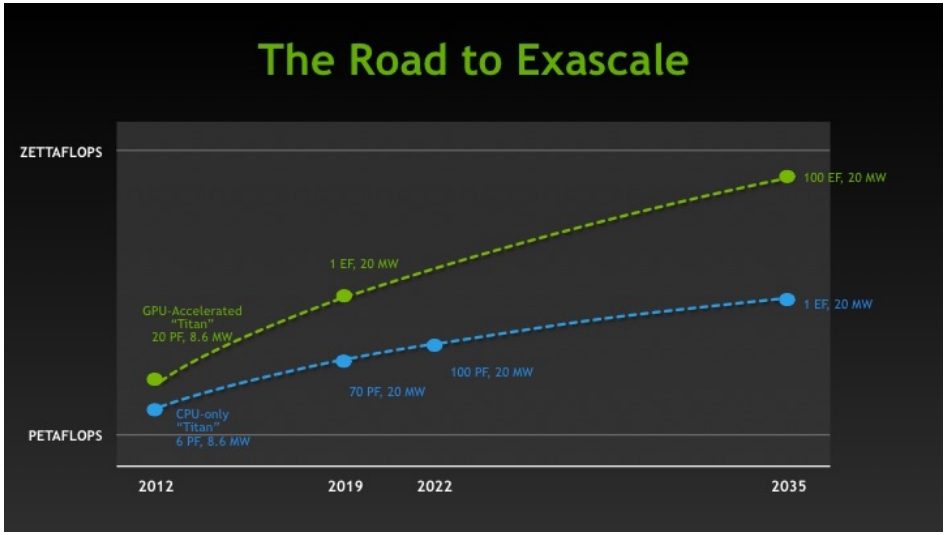
Others approach for CUDA/Python

E.g. in AI community:

- pytorch is using pybind11 to design extension modules written in C++ and CUDA, see
  https://pytorch.org/tutorials/advanced/cpp_extension.html
- tensorFlow is in evolving frow swig to pybind11 :
  https://github.com/tensorflow/community/blob/master/rfcs/201

Exascale : days of future past...

- At SC2011, Nvidia CEO talk:



- Document The International Exascale Software Project Roadmap by Dongarra et al. (DOI:10.1177/1094342010391989)
  1. Make a thorough assessment of needs, issues and strategies,
  2. Develop a coordinated software roadmap,
  3. Encourage and facilitate collaboration in education and training.
- **Artificial Intelligence (AI) was not even in the scope...**

Exascale race: technological challenge(s)

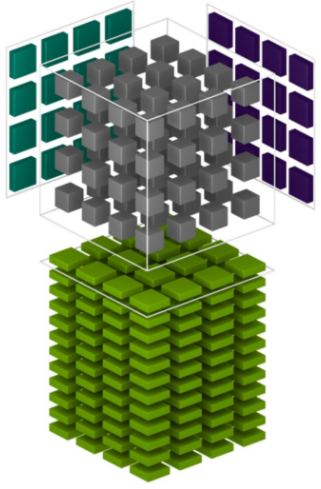
Many technical challenges...

- **Most challenging constraint:** fitting the **electrical power envelop** ($P \in [20-40]$ MW)
- hardware technologies (**interconnect, storage, processor architectures,...**)

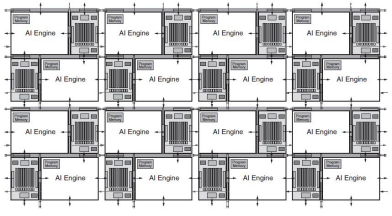
...also about building an economic ecosystem

- **New architectures** for **several markets** : HPC, AI, IoT, near-sensor computing, automotive...
- **Hardware vendors already designing/optimizing new architectures for AI** (always back and forth between general purpose and application specific):e.g
  - Nvidia (Tensor Core),
  - Xilinx (Alveo / Versal),
  - Intel (BFLOAT16, for future CooperLake), ...
- **Semantic shift: HPC = simulation + AI**
- **Cost of designing a new chip skyrocketting, ...**

Nvidia Tensor Core, Volta (2017)



Xilinx AI Engine array (2019)



Pre-Exascale machines - architecture diversity !

- **US**: Summit , Sierra ⇒ mostly OpenPower (IBM P9 + Nvidia V100), GPU-based architecture, #1 and #2 @top500; exascale machines announced
  - Aurora (Argonne NL): Intel Xe GPU
  - Frontier (Oak Ridge NL): AMD EPYC + Radeon GPU
- **China:** 3 machines
  - Phytium FT2000/64 ARM chips + Matrix2000 GPDSP accelerators ⇒ #4 @top500, Tianhe-2A, 61 Pflops
  - 260-core Shenwei, **homegrow technology** hardware + software (C++/fortran compiler + OpenACC) ⇒ #3 @top500 , Sunway TaihuLght, 93 PFlops
  - Dhyana, AMD-licenced x86 multicore (300 M$!), identical to AMD EPYC
- **Japan:** **Post K**(Fujitsu, ARM, RIKEN) A64FX ARM (**home grown**, started in 2014, 900 M$), GPU, etc ...
- **Europe** : lagging behind but new organization EuroHPC (2019), EC H2020 budget (~ 500 M€)
  home grown ARM and/or RISC-V architecture, early stage

Nvidia / AMD portability

- Cuda Language provides abstractions / concepts based on CUDA hardware (grid of block, block of threads, warp, ...)
- is Cuda language specific to Nvidia ? yes
- OpenCL was created to absorb most of the CUDA programming model concepts and make more generic / vendor neutral, and above all cross-platform.
- OpenCL is more than a language, it is supported by the Khronos group, i.e. there is an open standard with a specification and multiple compiler vendors can implement it
- OpenCL is as *low-level* as CUDA ⇒ **high-level programming approaches** (directive-based or library-based) have emerged over the past ~10 years

Nvidia / AMD portability

- AMD ROCm platform provides a way to compile code either for
  - AMD GPUs using hcc or clang compiler
  - NVIDIA GPUs using nvcc compiler
- HIP (Heterogeneous-compute Interface for Portability) is a C++ library / programming model such that code written with HPI can be compiled (with Clang compiler) for AMD GPUs or converted into regular CUDA/C++ and then compiled for NVIDIA GPUs.

http://www.admin-magazine.com/HPC/Articles/Discovering-ROCm

AMD_GPU_HIP_training_20190906.pdf

Performance portability and software productivity

- **Developing / maintaining** a **separate implementation** of an application for each **new hardware platform** (Intel KNL, Nvidia GPU, ARMv8, ...) **requires lots of efforts**
- **Identical code** will never perform **optimally** on all platforms [2]
- Is it possible to have a **single set of source codes** that can be compiled for different hardware targets ?

  – high level of abstraction for the end user/developper
  – low-level for hardware optimization

- **Performance portability** should be understood as a single source code base with
  - **good** performance on different architectures
  - a relatively **small amount of effort** required to tune app performance from one architecture to another.

  source https://performanceportability.org/
- **Performance portability** is achieved when software implementation
  - compiles and runs on **multiple architectures**,
  - obtains performant **memory access patterns** across architectures,
  - can **leverage architecture-specific features** where possible.

[2] source: Matt Norman, WACCPD 2016

Performance portability and software productivity

ARCHITECTURAL DETAILS

How much architectural detail must be revealed to programmers?

Conceal – Hide the feature

instruction-level parallelism

Virtualize – Reveal existence, hide details

vectorized loops

Reveal – All details revealed to programmer

SIMD intrinsics

PGI

6 NVIDIA

source : M. Wolfe, PGI/Nvidia, DOE Perf. Port. Meeting, April 2019

Performance portability and software productivity

THREE EX'S OF PERFORMANCE PORTABILITY

How much detail must the programmer know in order to …

Expose – in the choice of algorithm and data structure

Express – in the choice of program constructs

Exploit – in the choice of how to schedule

PGI

7 NVIDIA

source : M. Wolfe, PGI/Nvidia, DOE Perf. Port. Meeting, April 2019

Performance portability issue

Developper productivity versus optimization

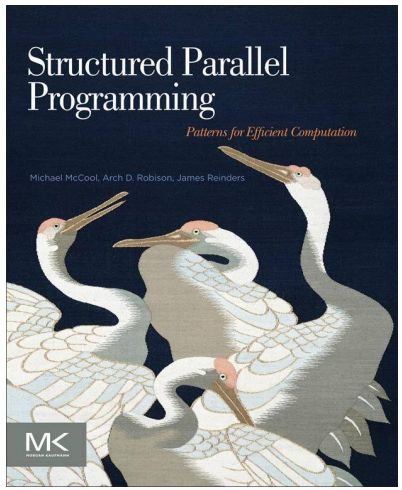
- **Taking into account hardware details is a hard job**
  - CPU vector length: 256 bits (8 *vector threads*)
    Heavily cache-based
  - KNL vector length: 512 bits x 2 (16-32 *vector threads*)
    Moderately cache-based, some latency/bandwidth hiding
  - GPU vector length: 65 536 bit (2048 *GPU vector threads*)
    Less cache-based, heavy on latency/bandwidth hiding
- **Find ways of writing codes that avoid optimization blockers**
  ⇒ Constructs that can be used to express operations without going into details

Performance portability issue : algorithmic patterns

- **Low-level native language:** OpenCL, CUDA, ...
- **Directive approach (code annotations)** for multicore/GPU, ...:
  - OpenMP 4.5 (Clang, GNU, PGI, ...)
  - OpenACC 2.5 (PGI, GNU, ...)
- **Other high-level library-based approaches** (mostly c++, à la TBB):
  - Some provide STL-like algorithmics patterns (e.g. Thrust is CUDA-based with backends for other archs, lift, arrayFire (numerical libraries, language wrappers, ...))
  - Kokkos, RAJA, Alpaka, Dash-project, agency, ...
  - Cross-platform frameworks
    - Chamm++: message-driven execution, task and data migration, distributed load-balancing, ...
    - hpx (heavy use of new c++ standards (11,14,17): `std::future`, `std::launch::async`, distributed parallelism, ...)
    - SYCL (Khronos Group *standard*), one implementation by CodePlay, by Keryell/Xilinx, ..., parallel STL, wide hardware targets (CPU, GPU, FPGA, ...), Intel OneAPI (Q4 2019)
- **Use an embedded Domain Specific Language (DSL)**
  - Halide (for image processing),
  - NABLA (for HPC, developped at CEA, PDE mesh+particules apps)

SC16-compareSYCL-Michael-Wong_Nov16.pdf

Programming with structured parallel patterns

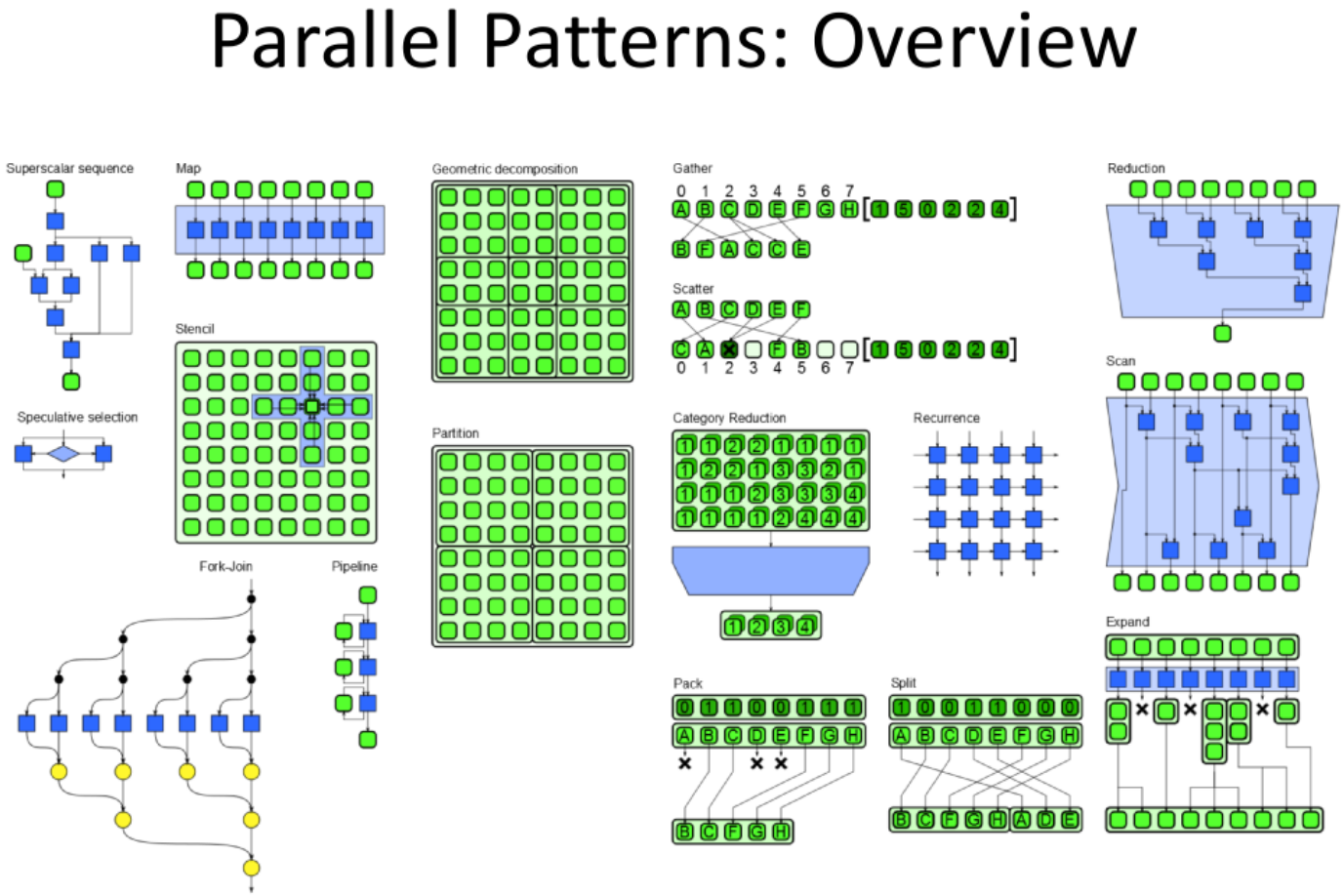


- **pattern** : **a basic structural entity of an algorithm**
- book Structured Parallel Programming: Patterns for Efficient Computation
- implementation: Intel TBB, OpenMP, OpenACC and many others
- OpenMP/OpenAcc for GPU/XeonPhi: pattern-based comparison: map, stencil, reduce, scan, fork-join, superscalar sequence, parallel update

reference:

A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing

Programming with structured parallel patterns

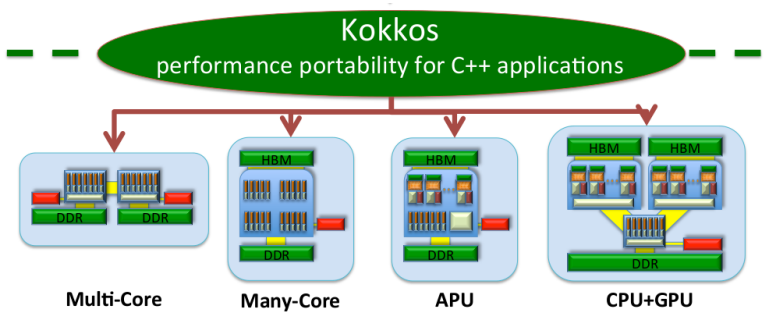


reference: Structured Parallel Programming with Patterns, SC13 tutorial, by M. Hebenstreilt, J. Reinders, A. Robison, M. McCool

Kokkos: a programming model for perf. portability

- **Kokkos** is a **C++ library** for **node-level parallelism** (i.e. **shared memory**) providing:
  - **parallel algorithmic patterns**
  - **data containers**
- Implementation relies heavily on **meta-programming** to derive native low-level code (OpenMP, Pthreads, CUDA, ...) and adapt data structure memory layout at compile-time
- Core developers at **SANDIA NL** (**H.C. Edwards** [3]**, C. Trott**)

Goal: **ISO/C++ 2020 Standard** subsumes Kokkos abstractions
Make Kokkos a sliding window of future c++ features



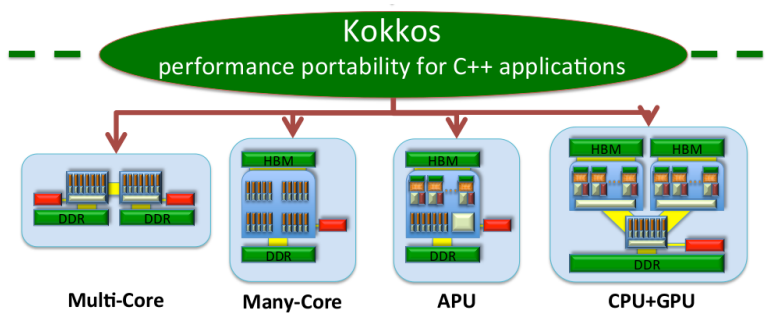
see mdspan proposal https://github.com/kokkos/array_ref

[3] now working @Nvidia

Kokkos: a programming model for perf. portability

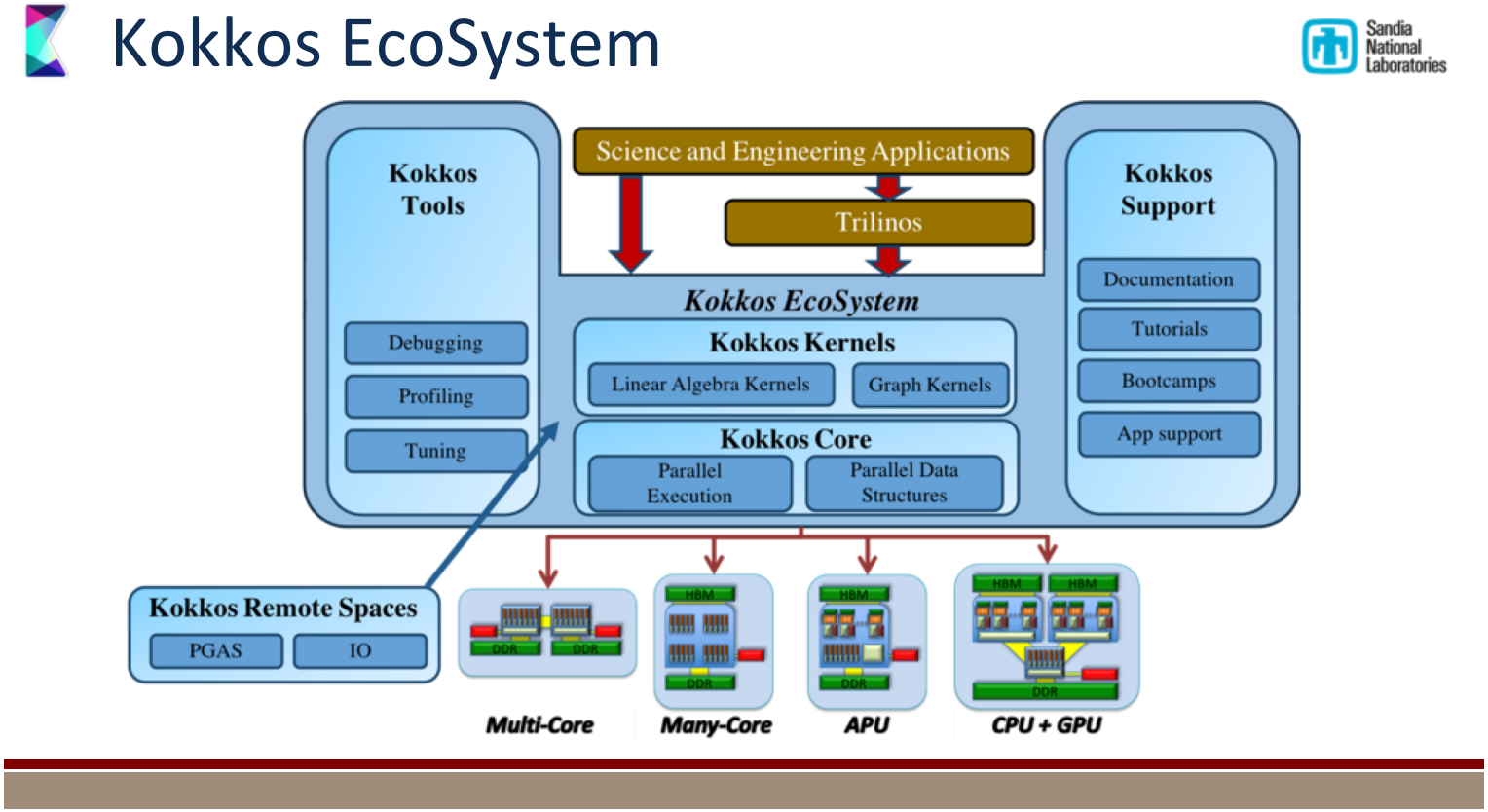
- **Open source**, https://github.com/kokkos/kokkos
- Primarily developped as a base building layer for **generic high-performance parallel linear algebra** in Trilinos
- Also used in
  - LAMMPS (molecular dynamics code),
  - NALU CFD (low-Mach wind),
  - SPARTA/DSMC (rarefied gas flow), SPARC (CFD, RANS, LES, hypersonic flow)
  - Albany (fluid/solid,...)
  - Uintah (structured AMR, combustion, radiation)

Goal: **ISO/C++ 2020 Standard** subsumes Kokkos abstractions
Make Kokkos a sliding window of future c++ features



see mdspan proposal https://github.com/kokkos/array_ref

Kokkos: a programming model for perf. portability



source: C. Trott, DOE Perf. Port. Meeting, April 2019

C++ Kokkos library summary

- Framework for efficient **node-level parallelism (CPU, GPU, ...)**
- Provides
  - **Computationnal parallel patterns** (for, reduce, scan, ...)
  - **Hardware aware memory containers**: e.g. **A multi-dimensionnal data container with hardware adapted memory layout**
- Mostly a header library (C++ metaprogramming)

C++ Kokkos library summary

- **What does it mean hardware aware memory containers ?**
- Most commonly in a C/C++, **multi-dimensionnal array access** is done through **index linearization** (row or column-major in 2D):

$$\mathbf{index} = \mathbf{i} + nx * \mathbf{j} + nx * ny * \mathbf{k}$$

- **Fortran** (column-major format) vs **C/C++** (row-major format)
- There is no reason to favour one layout versus the other
  - column-major is better for vectorization on CPU architecture
  - row-major is better for high througput architecture e.g. GPU (memory coalescence)
  - $\Rightarrow$ **Kokkos allows to chose memory layout at compile time**
- In Kokkos, one should/must avoid this index linearization at the user level, let `Kokkos::View` do this job (**decided at compile-time, hardware adapted**)

$$\text{data}(i, j, k)$$

Multidimensional array and data parallelism

Memory layouts / index linearization: e.g. 2D

row-major

$n_x(n_y-1)$	$n_x(n_y-1)+1$	$\cdots$	$n_x n_y-1$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$2n_x$	$2n_x+1$	$\cdots$	$3n_x-1$
n_x	n_x+1	$\cdots$	$2n_x-1$
0	1	$\cdots$	n_x-1

$$\text{index} = i + n_x j,$$

left layout

fast index on the left

column-major

n_y-1	$2n_y-1$	$\cdots$	$n_x n_y-1$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
2	n_y+2	$\cdots$	$n_y(n_x-1)+2$
1	n_y+1	$\cdots$	$n_y(n_x-1)+1$
0	n_y	$\cdots$	$n_y(n_x-1)$

$$\text{index} = j + n_y i,$$

right layout

fast index on the right

Multidimensional array and data parallelism

Question: Assuming left layout, which loop would you prefer to parallelize (inner or outer) ?

row-major

$n_x(n_y-1)$	$n_x(n_y-1)+1$	$\cdots$	$n_x n_y - 1$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$2n_x$	$2n_x+1$	$\cdots$	$3n_x-1$
n_x	n_x+1	$\cdots$	$2n_x-1$
0	1	$\cdots$	n_x-1

$$\text{index} = i + n_x j,$$

left layout

fast index on the left

```
for(int j=0; j<ny; ++j)
  for(int i=0; i<nx; ++i)
    data[i+nx*j] += 12;
```

Favor memory locality:

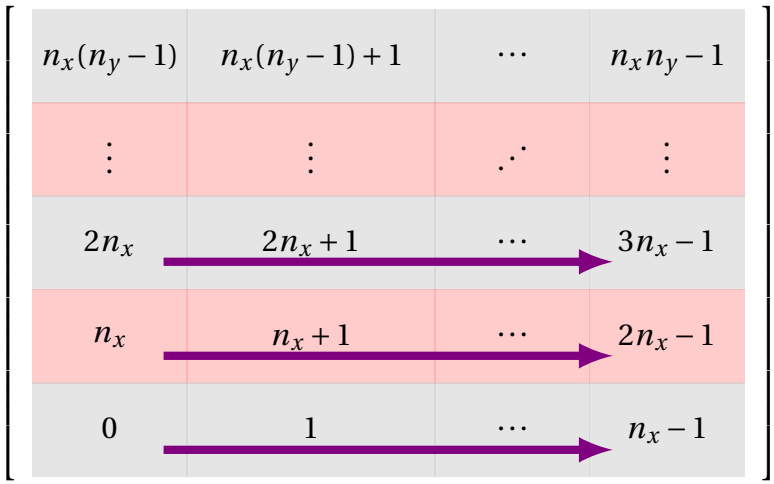
- maximize cache usage for CPU
- maximize memory coalescence on GPU

Different hardware ⇒ different parallelization strategies

Multidimensional array and data parallelism

Question: Assuming left layout, which loop would you prefer to parallelize (inner or outer) ?

row-major



$\text{index} = i + n_x j$,

left layout

fast index on the left

OpenMP // outer loop

each OpenMP thread handles **1 or more row(s)**

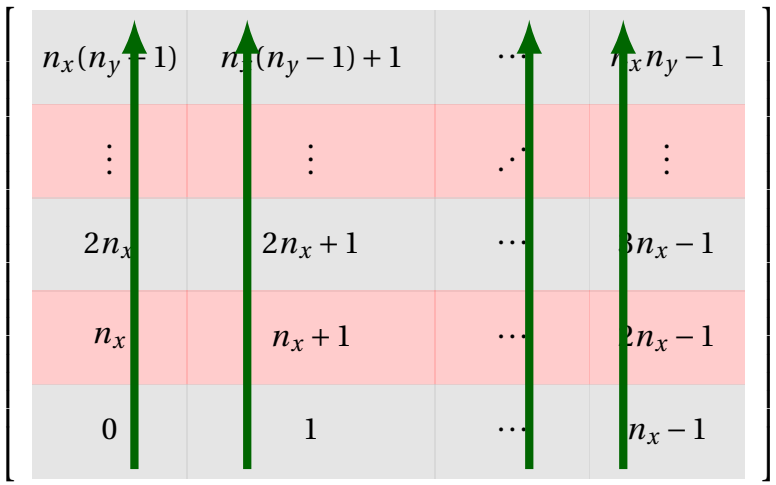
```
#pragma omp parallel
{
  #pragma omp for
  for(int j=0; j<ny; ++j)

    // vectorization loop
    // memory contiguity
    for(int i=0; i<nx; ++i)
      data[i+nx*j] += 12;
}
```

Multidimensional array and data parallelism

Question: Assuming left layout, which loop would you prefer to parallelize (inner or outer) ?

row-major



index $= i + n_x j$, left layout
fast index on the left

CUDA // inner loop
each CUDA thread handles **1 or more col(s)**
memory coalescence

```
__global__ void compute(int *data)
{
  // adjacent memory cells
  // computed by
  // neighboring threads
  int i = threadIdx.x +
      blockIdx.x*blockDim.x;

  for(int j=0; j<ny; ++j)
    data[i+nx*j] += 12;
}
```

Multidimensional array and data parallelism

Conclusion:

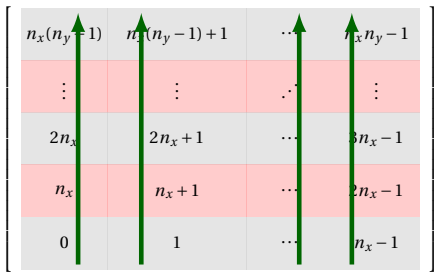
Don't assume layout, let's chose at compile-time !

- **First conclusion:**
  if we keep the same memory layout, **OpenMP** and **CUDA** **disagree** on which loop should be parallelized to optimize for their respective hardware target.
- **How can we make portable code ?**
- Note that swapping memory layout and `for loops` is **involutive**
- **Kokkos answer:** make memory layout abstract (since a good memory layout is hardware dependent), fixed at compile-time access $data(i, j)$
  - On **OpenMP** $data(i, j)$ actually means accessing $dataPtr[Ny * i + j]$
  - On **Cuda** $data(i, j)$ actually means accessing $dataPtr[i + Nx * j]$

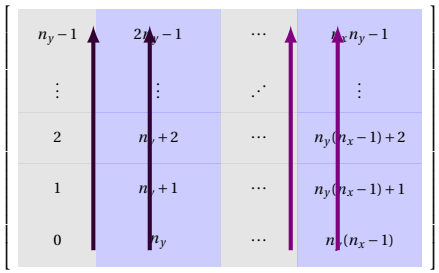
Multidimensional array and data parallelism

Don't assume layout, let's chose at compile-time !
Make it hardware aware.

left layout / CUDA



right layout / OpenMP



Kokkos parallel version for both CUDA/OpenMP

```
Kokkos::parallel_for(nx,
    KOKKOS_LAMBDA(int i) {
      for (int j=0; j<ny; ++j)
        data(i,j) += 12;
    }
);
```

7-point Stencil kernel with Kokkos - 1

- A **single high-level parallel programming model for shared memory** architectures (CPU, GPU, ...) ⇒ **developper more productive**
- 3d stencil kernel - **SERIAL**

```
// CPU version
for(int i=1; i<nx-1; ++i)
  for(int j=1; j<ny-1; ++j)
    for(int k=1; k<nz-1; ++k) {

      int index = k + j*nz + i*ny*nz

      y[index] = -5*x[index] +
        ( x[index-1]     + x[index+1] +
          x[index-nz]    + x[index+nz] +
          x[index-nz*ny] + x[index+nz*ny] );
    }
```

7-point Stencil kernel with Kokkos - 2

- A **single high-level parallel programming model for shared memory** architectures (CPU, GPU, ...) ⇒ **developper more productive**
- 3d stencil kernel - **parallel KOKKOS - naive**

```
// naive Kokkos kernel - for CPU, GPU, ...
Range3d range ( {{0,0,0}}, {{nx,ny,nz}} );

parallel_for(range, KOKKOS_LAMBDA(int i,
                                  int j,
                                  int k) {

  y(i,j,k) = -5*x(i,j,k) +
    ( x(i-1,j  ,k  ) + x(i+1,j  ,k  ) +
      x(i  ,j-1,k  ) + x(i  ,j+1,k  ) +
      x(i  ,j  ,k-1) + x(i  ,j  ,k+1) );
});
```

7-point Stencil kernel with Kokkos - 3

- A **single high-level parallel programming model for shared memory** architectures (CPU, GPU, ...) ⇒ **developper more productive**
- 3d stencil kernel - **parallel KOKKOS - HIERARCHICAL (CPU / GPU)**

```
parallel_for(team_policy_t(nbTeams, AUTO),
   KOKKOS_LAMBDA(const thread_t& thread) {
     int i = thread.league_rank();

     parallel_for(TeamThreadRange(thread, 1, n-1),
       [=](const int j) {

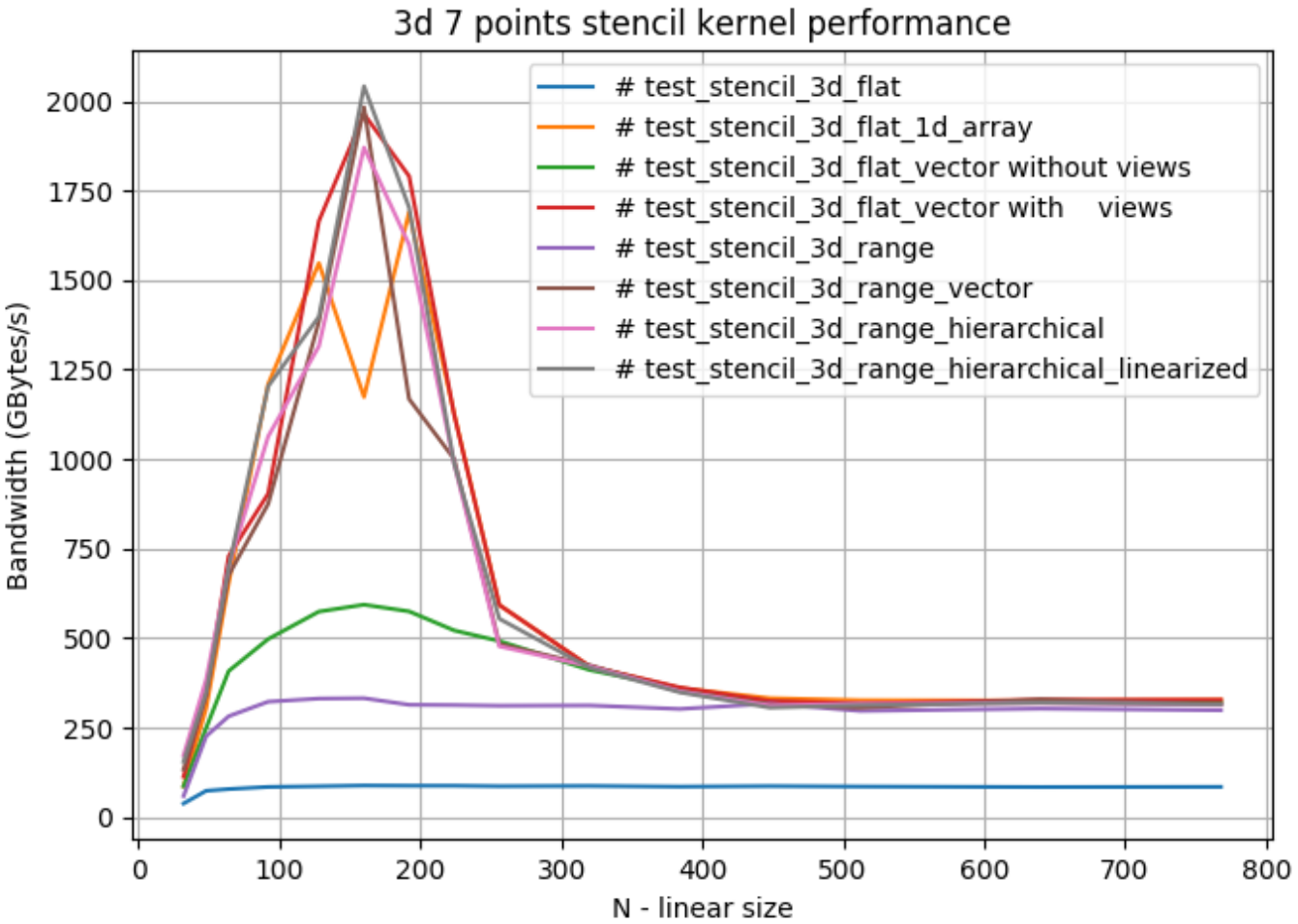
         parallel_for(ThreadVectorRange(thread, 1, n-1),
           [=](const int k) {

             if ( ... ) {
               y(i,j,k) = -5*x(i,j,k) +
               (x(i-1,j,k) + x(i+1,j,k) +
                x(i,j-1,k) + x(i,j+1,k) +
                x(i,j,k-1) + x(i,j,k+1));
             }
           }); // end vector range
       }); // end thread range
   }); // end team policy
```

ref: https://github.com/pkestene/kokkos-proj-tmpl

7-point Stencil kernel with Kokkos - 4

Intel CPU Skylake 2×18 cores, gold



7-point Stencil kernel with Kokkos - 4

Nvidia GPU P100, ouessant@IDRIS