Programming Graphics Processing Units (GPUs)

Who? Lokmane ABBAS TURKI and **Roman IAKYMCHUK**
lokmane.abbas_turki@sorbonne-universite.fr
roman.iakymchuk@sorbonne-universite.fr

When? October 2020

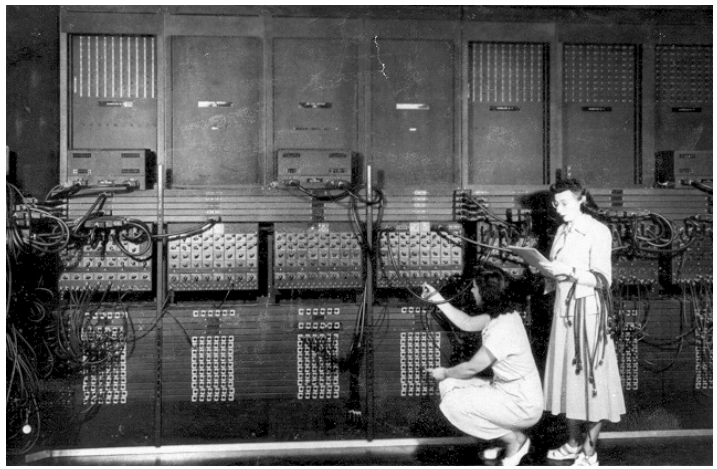
Plan

Plan

From 1946 to 70s

ENIAC 1946

- Was the first electronic general-purpose machine
- Was a parallel machine
- Later became the first Von Neumann machine
- Was used for Monte Carlo simulation
- Although developed for ballistic research, it was first used for hydrogen bomb computations



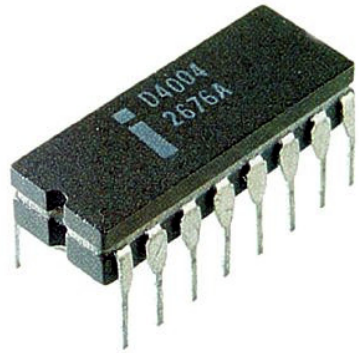
Parallel machines till the 70s

- Continued to exist as the real solution for heavy computations
- Used by specialists and dedicated essentially to military applications
- Some well known: ILLIAC IV (1971) and Cray 1 (1976)

From 70s to 2000

Becoming sequential

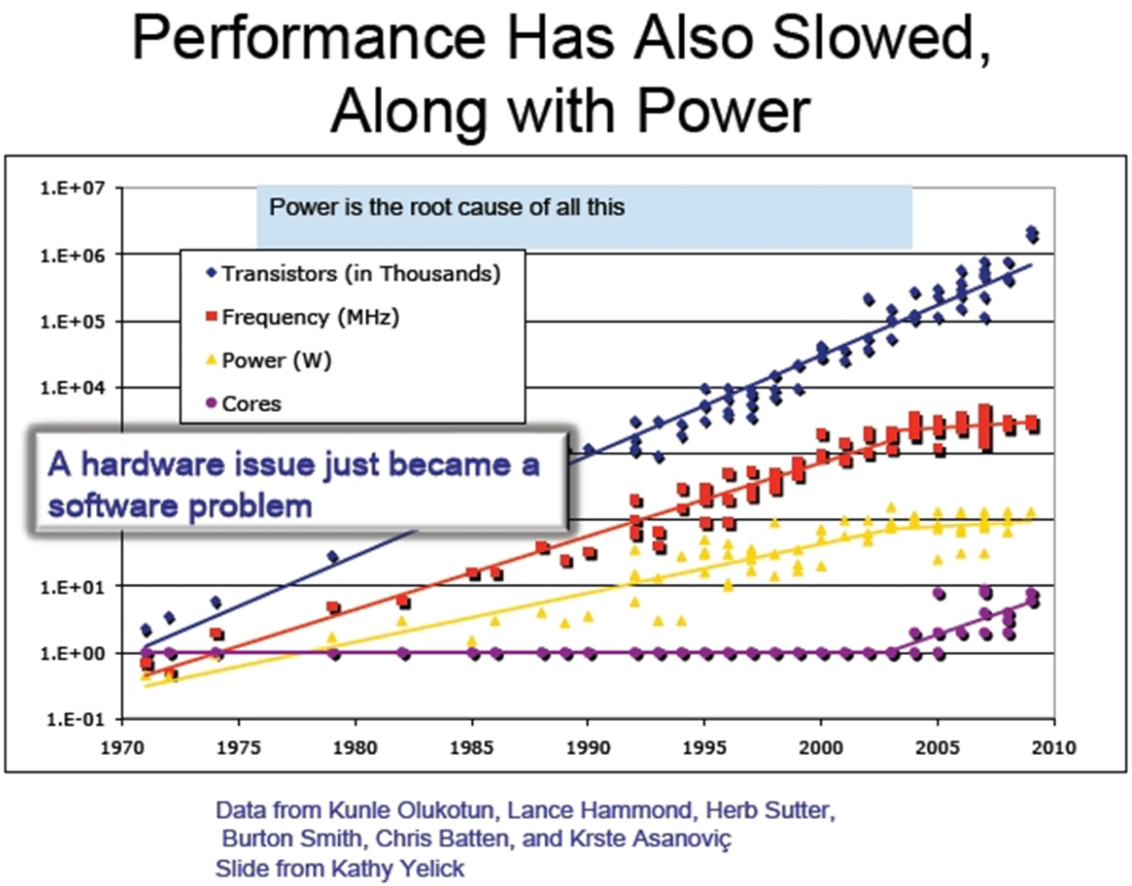
- Tradic (1954): The first transistor machine
- Intel 4004 (1971): Commercialization of the first microprocessor
- Amortizing the production costs by selling to the large public
- RAM became affordable (70s-80s) and used in microcomputers
- The memory hierarchy (Registers, cache, RAM, Hard Disc) is essential in computers
- The Moore's Law was satisfied on one core: Doubling the operating frequency each 18 months period

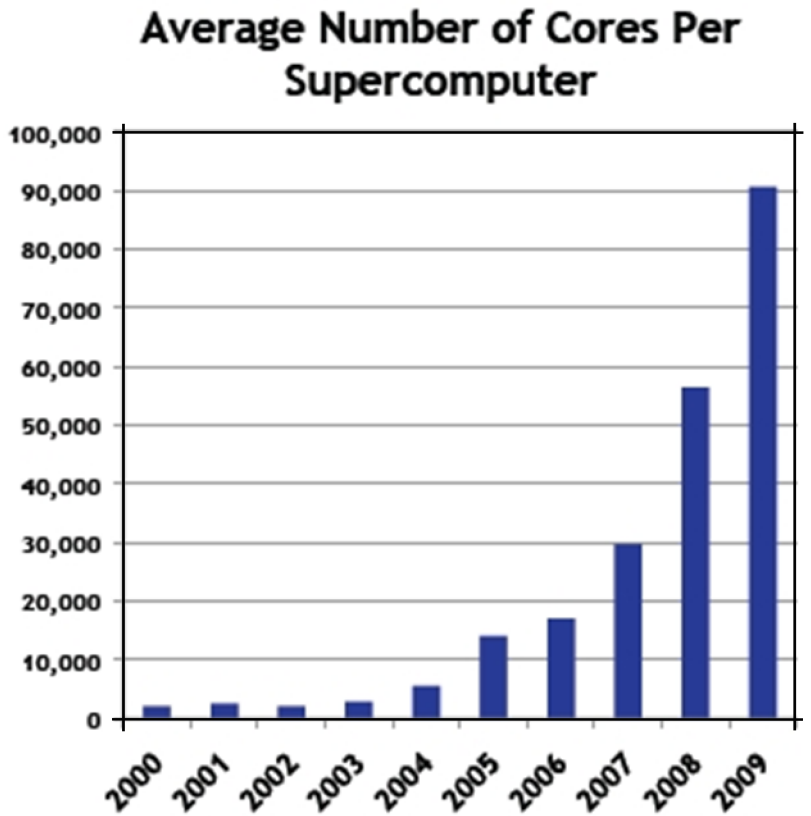


Scientific simulation

- Caltech Cosmic Cube (1981): Proposing a parallel computer at a reasonable cost
- From 1980 to 2000: Connection of serial machines to increase performance
- Difficulties due to multiplicities of platforms, inefficient inter-machine communication and insufficient documentation
- Applied mathematics expanded in the world of serial resolution of PDEs

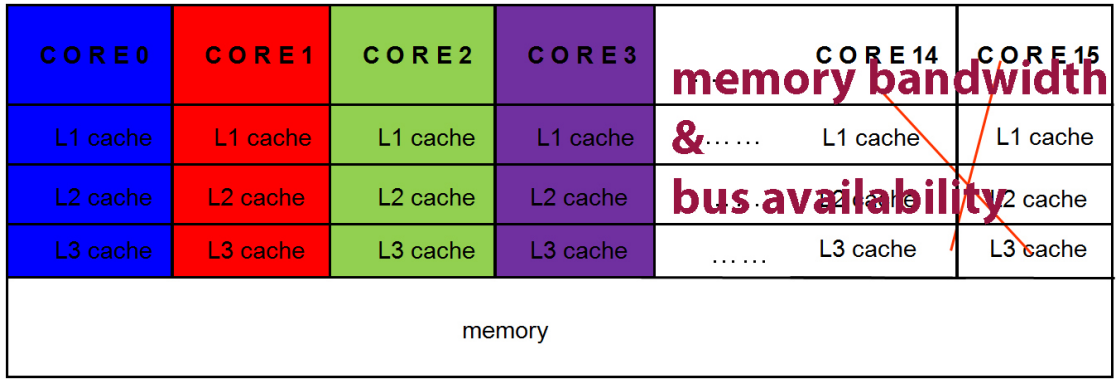
Reaching the limit



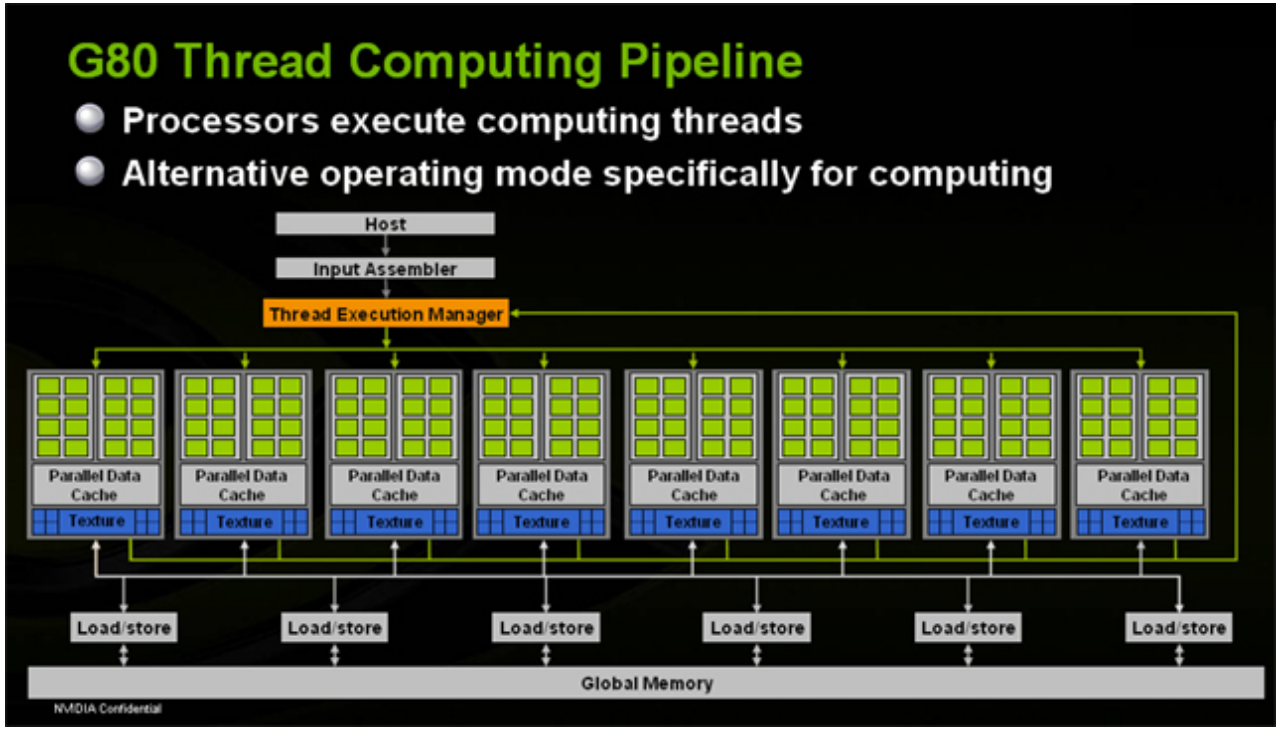


Architecture overview

Sandia National Laboratories
16 cores =? 2 cores



The limit architecture!
GPU (Graphic Processing Unit)



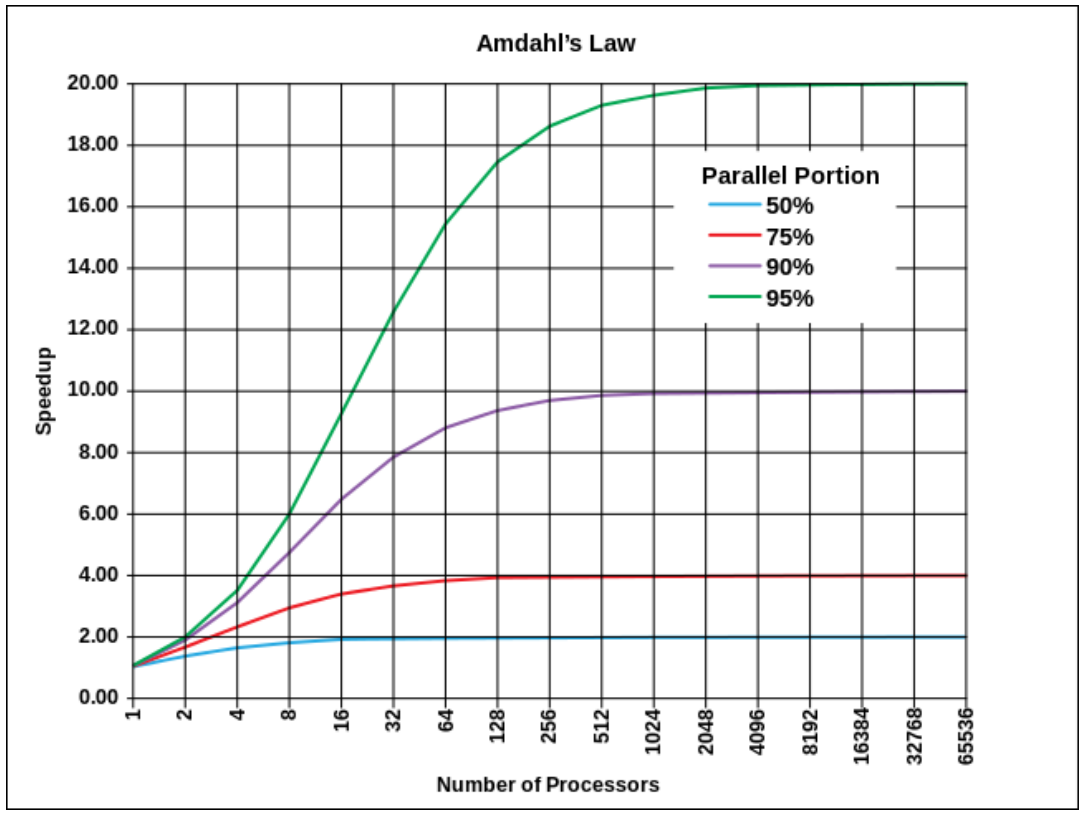
Amdahl's law

For a fixed problem

$$T(P) = T(1)\left(\alpha + \frac{1-\alpha}{P}\right), \quad S(P) = \frac{T(1)}{T(P)} = \frac{1}{\alpha + \frac{1-\alpha}{P}}, \tag{1}$$

α: the fraction of the algorithm that is purely serial.

From Wikipedia



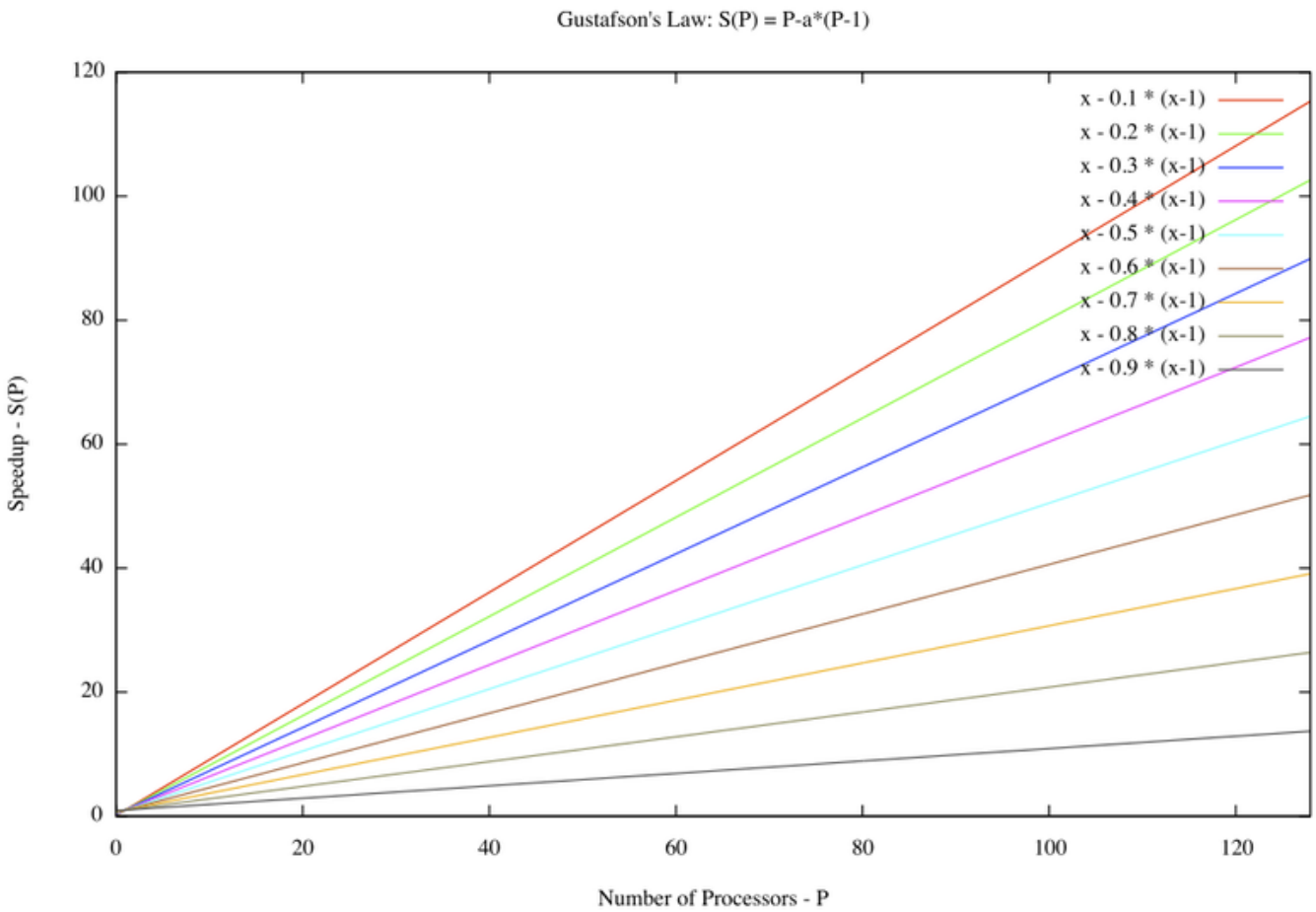
Gustafson's law

Making it bigger

$$T(1) = (\alpha + [1 - \alpha]P)\,T(P), \quad S(P) = \frac{T(1)}{T(P)} = P - \alpha(P - 1), \qquad (2)$$

α: the fraction of the algorithm that is purely serial.

From Wikipedia



GPUs

GPU = Graphical Processing Unit = specialized microcircuit to accelerate the creation and manipulation of images in video frame for display devices.

GPUs are used in game consoles, embedded systems (like systems on cars for automatic driving), computers and **supercomputers**.

- Since 2012, GPUs are the main workforce for training deep-learning networks

Some important GPU vendors: **NVIDIA**, **AMD**, ...

Slide courtesy of the PDC Summer School by S. Markidis et al.

The Rise of GPUs in HPC

GPUs are a core technology in many world's fastest and most energy-efficient supercomputers

GPUs compete well in terms of **FLOPS/Watt**

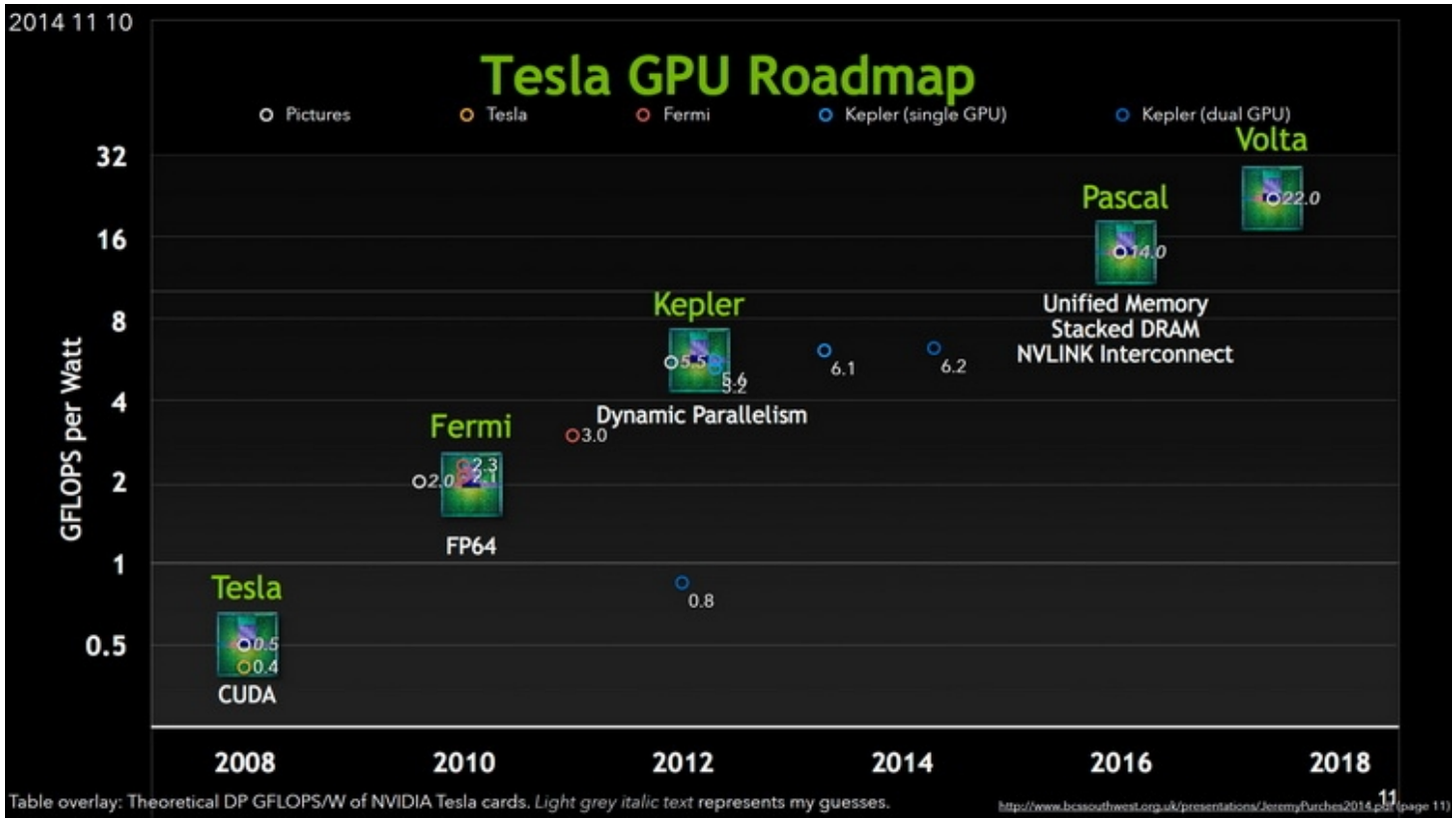
- In the current Green500, the top 6 most energy-efficient supercomputers use NVIDIA P100 GPU

Rank	TOP500 Rank	System	Cores	Rmax (TFlop/s)	Power (kW)	Power Efficiency (GFlops/watts)
1	61	**TSUBAME3.0** - SGI ICE XA, IP139-SXM2, Xeon E5-2680v4 14C 2.4GHz, Intel Omni-Path, NVIDIA Tesla P100 SXM2 , HPE GSIC Center, Tokyo Institute of Technology Japan	36,288	1,998.0	142	14.110
2	465	**kukai** - ZettaScaler-1.6 GPGPU system, Xeon E5-2650Lv4 14C 1.70Hz, Infiniband FDR, NVIDIA Tesla P100 , ExaScalar Yahoo Japan Corporation Japan	10,080	460.7	33	14.046
3	148	**AIST AI Cloud** - NEC 4U-8GPU Server, Xeon E5-2630Lv4 10C 1.8GHz, Infiniband EDR, NVIDIA Tesla P100 SXM2 , NEC National Institute of Advanced Industrial Science and Technology Japan	23,400	961.0	76	12.681
4	305	**RAIDEN GPU subsystem** - NVIDIA DGX-1, Xeon E5-2698v4 20C 2.2GHz, Infiniband EDR, NVIDIA Tesla P100 , Fujitsu Center for Advanced Intelligence Project, RIKEN Japan	11,712	635.1	60	10.603
5	100	**Wilkes-2** - Dell C4130, Xeon E5-2650v4 12C 2.2GHz, Infiniband EDR, NVIDIA Tesla P100 , Dell University of Cambridge United Kingdom	21,240	1,193.0	114	10.428

Slide courtesy of the PDC Summer School by S. Markidis et al.

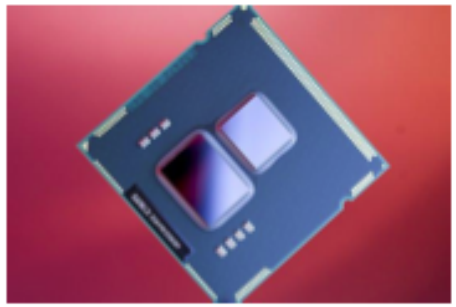
Efficiency and programmability

Computational capabilities



Where do you find GPUs?

- **Integrated:** Every laptop has an integrated GPU built into its processor, i.e. Intel HD or Iris Graphics.
- **Dedicated:** A standalone GPU uses its own processor and memory. Most dedicated GPUs are removable. They require more power but also provide higher performance
  - **In HPC, we use dedicated GPUs**



Source: PC Authority

Source: bit-tech.net

Question: What is the main difference between the two?

Slide courtesy of the PDC Summer School by S. Markidis et al.

GPU Design Motivation: Process Pixels in Parallel

Data parallel

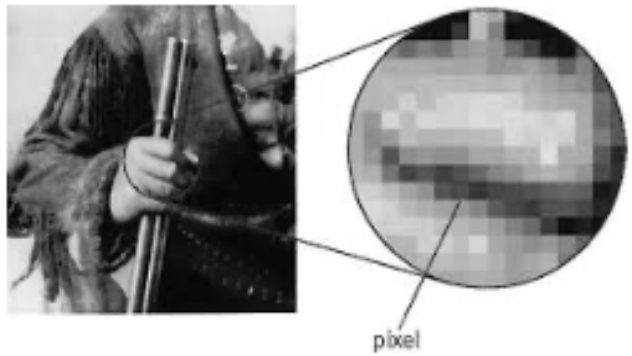
- In 1080i and 1080p videos, 1920 x 1080 pixels = 2M pixels per video frame → compute intensive
- Lots of parallelism at low clock speed → power efficient

Computation on each pixel is independent from computation on other pixels.

- No need for synchronization

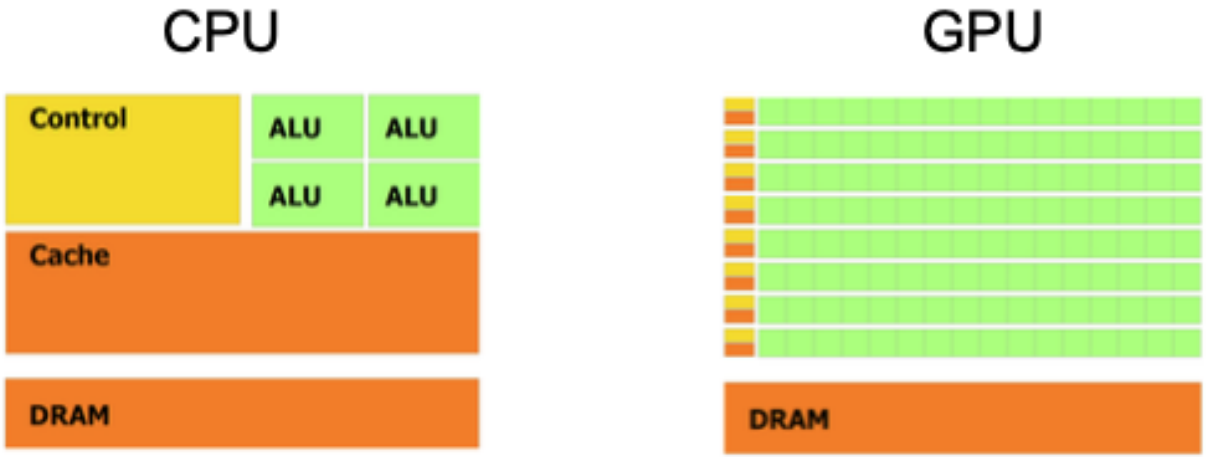
Large data-locality = access to data is regular

- No need for large caches



Slide courtesy of the PDC Summer School by S. Markidis et al.

What are the differences?



CPU has tens of massive cores, CPU excels at irregular control-intensive work

- Lots of hardware for control, fewer ALUs

GPU has thousands of small cores, GPU excels at regular math-intensive work

- Lots of ALUs, little hardware for control

Slide courtesy of the PDC Summer School by S. Markidis et al.

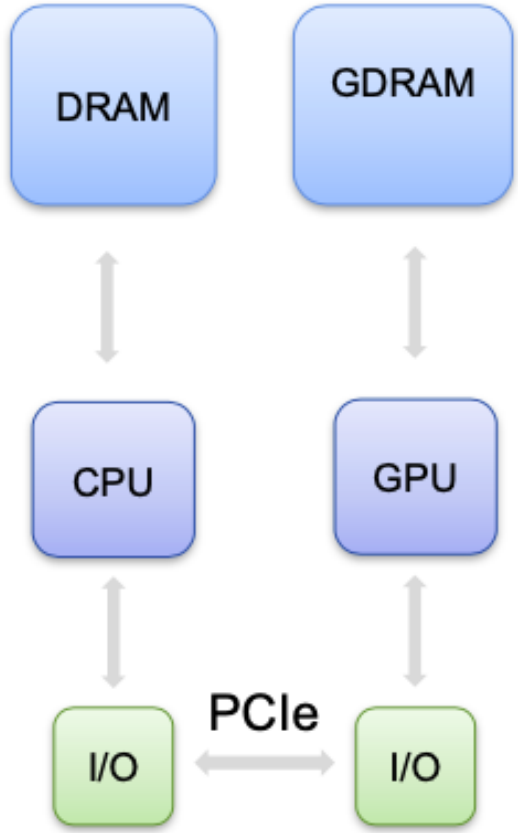
GPUs as Accelerators

GPU are simple, lower power and highly parallel

Problem: Still require OS, IO and scheduling

Solution: "Hybrid System"

- CPU provides management
- "Accelerators" (or co-processors) such as GPUs provide compute power



Slide courtesy of the PDC Summer School by S. Markidis et al.

GPU Hardware Model

In order to program a GPU program, it is important to understand the Hardware Model.

The fundamental computing entity is

- **Streaming Processor** (**SP**) or **CUDA core**

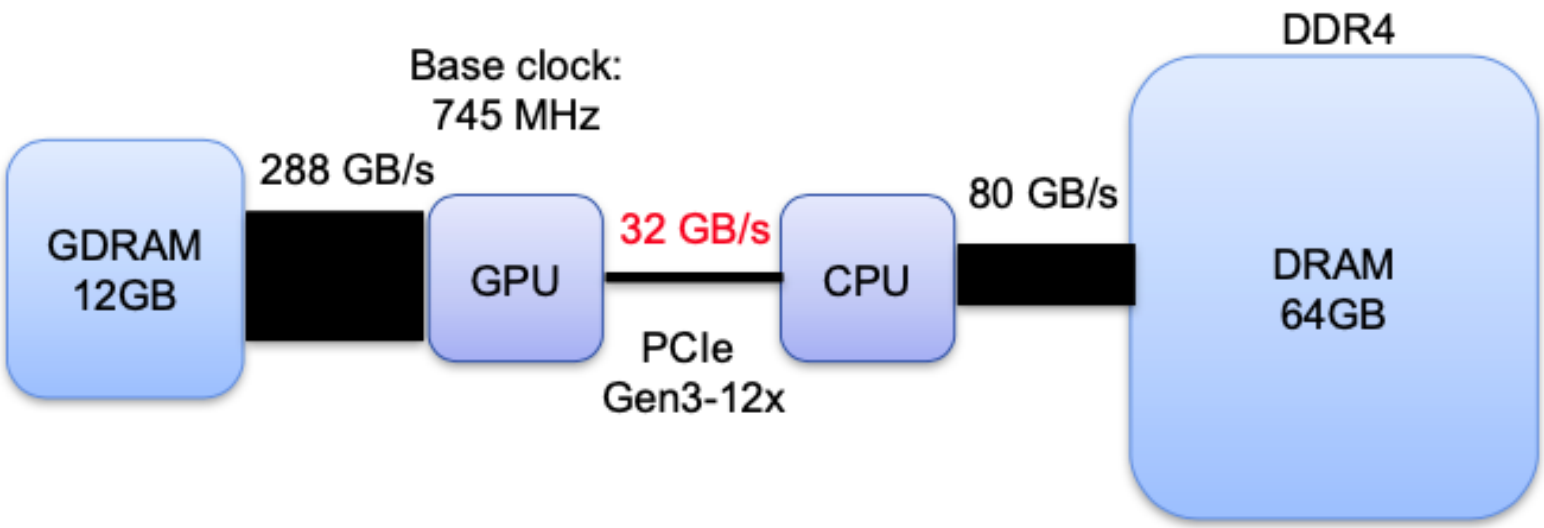
A **Streaming Multiprocessor** (**SM**):

- A collection of 8/32/192 CUDA Cores (depends on SM architecture)
- All CUDA cores in SM run the same instructions
- Has some fast cache shared memory
- Can synchronize

Slide courtesy of the PDC Summer School by S. Markidis et al.

Weakness of GPU (but not for too long ... **NVLink**)

GPU is very fast (huge parallelism) but getting data from/to GPU is slow



NVIDIA TESLA K40 = the most common GPU on supercomputers in Nov. 2016 Top500 list

Slide courtesy of the PDC Summer School by S. Markidis et al.

Is GPU good for my non-graphics application?

It depends on the application:

- **Compute-intensive applications** with little synchronization benefit the most from GPU:
  - Deep-learning network training 8×-10×, GROMACS 2×-3×, LAMMPS 2×-8×, QMCPack 3×.
- Irregular applications, such as sorting and constraint solvers, are faster on CPU.

General strategy when you work on your code: **take the computational-heavy part of your code and run it on GPU**

Slide courtesy of the PDC Summer School by S. Markidis et al.

Low-Level Programming GPUs

- **OpenCL (Open Computing Language)**: based on C, not only for GPUs but also for other "accelerators" (DSP, FPGA, ...)
- **CUDA (compute unified device architecture)**: extension to C language. Only for **NVIDIA GPUs**.

Slide courtesy of the PDC Summer School by S. Markidis et al.

High-Level Programming GPUs

- **OpenMP**: compiler directives and library for accelerators
- **OpenACC**: compiler directives and library for NVIDIA GPUs

Compiler + runtime library

- **Thrust**: C++ template library resembling C++ STL.
- **OpenCV**: Computer vision library using GPU
- **CUDA-based libraries for math**: cuBLAS, cuFFT, cuDNN, …

Libraries atop CUDA

Slide courtesy of the PDC Summer School by S. Markidis et al.

Plan

PuTTY as a command console

- You should know your account name `hpcrise` and password
- `ssh ghome.metz.supelec.fr` to access to CentraleSupélec network
- `ssh term2.grid` to access to the GPU cluster
- `oarsub -p "cluster='cameron'" -q day -l nodes=1,walltime=4:00:00 -I` to get one GPU node during 4 hours
- Once finished, log out with successive `exit` or `Ctrl+d`

FileZilla Client to transfer files

Write your code on your local machine then transfer it on `ghome.metz.supelec.fr` using FileZilla Client.

Important!

Very often use the documentation provided by NVIDIA, in particular:

- **CUDA_C_Programming_Guide**: Necessary document for the CUDA language handling and global understanding of the hardware architecture of the GPU
- **CUDA_Runtime_API**: Document describing the CUDA functions that allow to program the GPU

Install CUDA on Linux machines

- gcc/g++ should be already available on your machine
- Install CUDA: `https://developer.nvidia.com/cuda-downloads`
- Disabling Secure Boot on UEFI (BIOS)
- Add `/usr/local/cuda/bin` to `PATH` and `/usr/local/cuda/lib64` to `LD_LIBRARY_PATH`

Install CUDA on Windows machines

- Install Visual Studio 2017 Community with C/C++ tools: `https://visualstudio.microsoft.com/fr/downloads/`
- Install CUDA: `https://developer.nvidia.com/cuda-downloads`
- Disabling Secure Boot on UEFI (BIOS)
- Add the address of `cl` compiler to `Path`
- Perform register changes explained at 7:40 in the video `https://www.youtube.com/watch?v=8NtHDkUoN98`

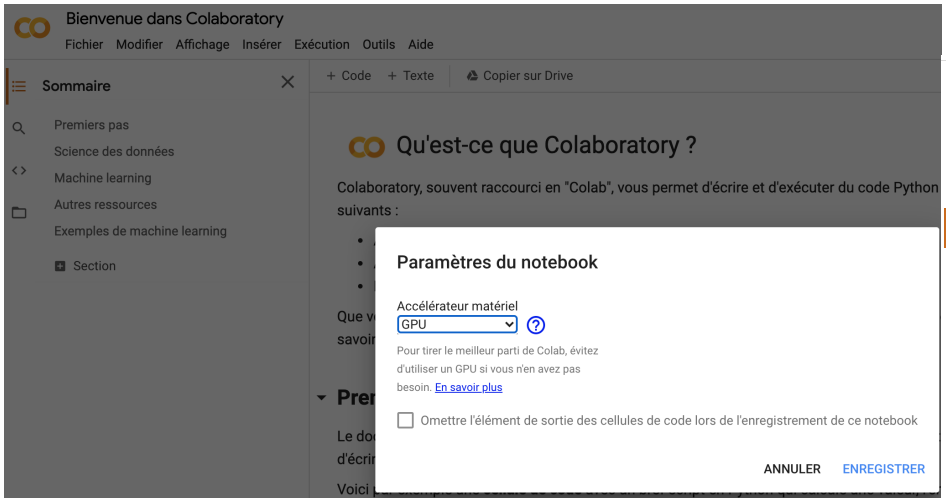
Important!

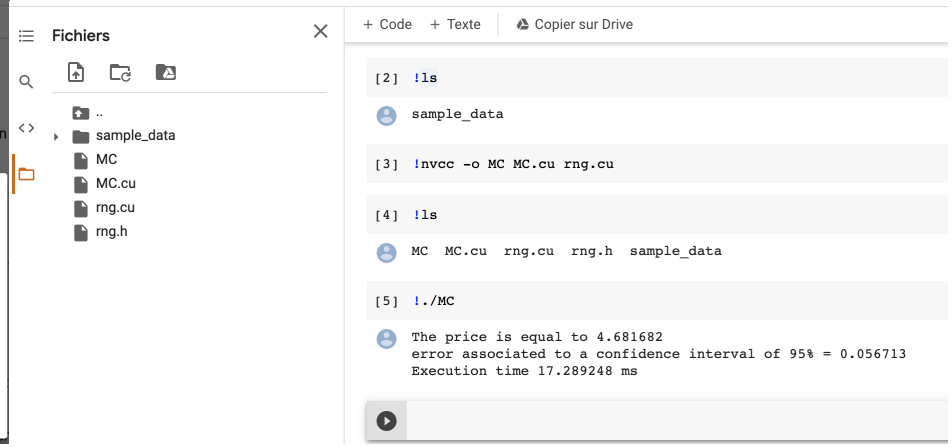
Very often use the documentation provided by NVIDIA, in particular:

- **CUDA_C_Programming_Guide**: Necessary document for the CUDA language handling and global understanding of the hardware architecture of the GPU
- **CUDA_Runtime_API**: Document describing the CUDA functions that allow to program the GPU

Google colab

- In case you do not have GPU on you laptop or cannot use them at SU
- NB: you need a google account for `colab.research.google.com`
- In console, all commands start with '!', e.g. '!ls'
- In console, commands are executed by pressing 'Shift + Enter'
- To begin with, select 'Execution' –> 'Modify type of execution' –> 'GPU'
- Create a new 'Section'
- It is better to code on your laptop and upload the code
- Compile: '!nvcc -o out Mycode.cu'
- Run: '!./out'





CUDA Componets

Installing CUDA on a system, there are 3 components:

1. **Driver** low-level software that controls the graphics card
2. **Toolkit**
   - **`nvcc` CUDA compiler**
   - Nsight IDE plugin for Eclipse or Visual Studio
   - profiling and debugging tools
   - several libraries
3. **SDK**
   - lots of demonstration examples
   - some error-checking utilities

Slide courtesy of the PDC Summer School by S. Markidis et al.

CUDA Programming

CUDA terminology:

- **host** = CPU and its memory
- **device** = GPU and its memory

At the host level, there is a choice of 2 APIs:

- **runtime** simpler, more convenient
- **driver** much more verbose, more flexible (e.g. allows run-time compilation), closer to OpenCL

We will only use the **runtime API**

Slide courtesy of the PDC Summer School by S. Markidis et al.

CUDA Parallel Model

CUDA employs the **Single Instruction Multiple Thread** (**SIMT**) model of parallelization.

- Each thread executes the same code but operates different data (**Data parallelism**)
- Each tread has it own context (it can be treated, restarted and executed independently)

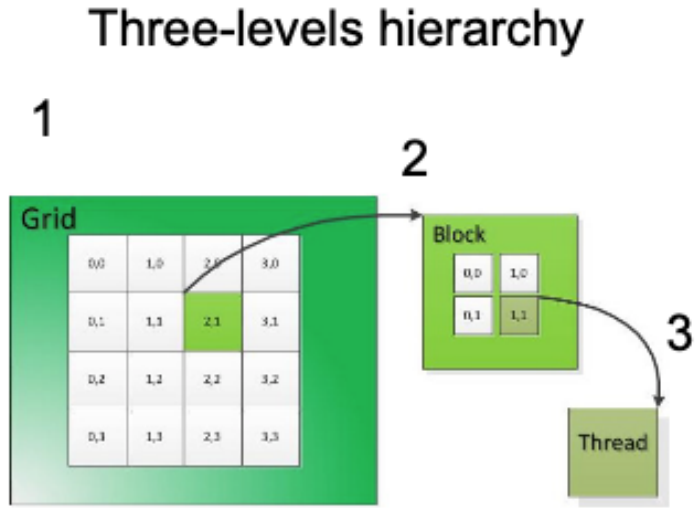
A set of threads executing the same instructions are dynamically group into **warp** by the hardware

- A warp is essentially a SIMD operation formed by the hardware

Slide courtesy of the PDC Summer School by S. Markidis et al.

Parallelization with CUDA

- As in OpenMP, we parallelize with **threads** - but now organized into a computational **grid** (**1D**, 2D or 3D) of **blocks of threads** (or **threadblocks**)
- The essential software construct is launching **kernel (function that runs on the GPU)**, that spawns a large collection of threads on the GPU



Slide courtesy of the PDC Summer School by S. Markidis et al.

Launch a Kernel in CUDA

Kernel is a kind of **special function**
Kernel **launch** ≅ regular function **call**

```
aKernel<<<Dg, Db>>>(arg1, arg2, ...)
```

To specify a kernel launch, we start with kernel name (`aKernel`) and end with argument list between `()`

Now for the CUDA extension: we specify the dimensional of the computational grid, the **grid dimensions** and **block dimension** between triple angle brackets (`<<<Dg, Db>>>`).

Slide courtesy of the PDC Summer School by S. Markidis et al.

Execution Configuration

`Dg` = number of blocks in the grid

`Db` =number of threads in the block

Together they constitute the **execution configuration** and specify the **dimensions of the kernel launch**

Slide courtesy of the PDC Summer School by S. Markidis et al.

Question: What is the total number of threads?

If we operate on a vector of length N, we set `DB` to a number that is some multiple 32 and `DG = N/DB`.

Question: What is the total number of threads?

Slide courtesy of the PDC Summer School by S. Markidis et al.

How to declare a function called by host but executed on device?

CUDA makes this distinction by prepending one of the following function type qualifiers:

- `__global__` is the **qualifier for kernels** (which can be called by the host and executed on device)
- `__host__` functions called from the host and executed on the host (default qualifier, often omitted)
- `__device__` functions are called from **the device and execute on the device** (a function that is called from a kernel needs the `__device__` qualifier)

Slide courtesy of the PDC Summer School by S. Markidis et al.

Question: which qualifier do you have before the function you call **from the GPU** and you want to run **on GPU**:

- __global__
- __host__
- __device__

?

Slide courtesy of the PDC Summer School by S. Markidis et al.

Question: which qualifier do you have before the function you call **from the CPU** and you want to run on **GPU**:

- __global__
- __host__
- __device__

?

Slide courtesy of the PDC Summer School by S. Markidis et al.

Question: which qualifier do you have before the function you call **from the GPU** and you want to run **on CPU**:

- __global__
- __host__
- __device__

?

Slide courtesy of the PDC Summer School by S. Markidis et al.

Kernel Launching is Asynchronous

As soon as the kernel is launched, **the CPU returns from the call of kernel without waiting for the completion of the kernel**.

In practice, the CPU launches the kernel and right away executes what is after the kernel launch without waiting for the kernel to finish

Slide courtesy of the PDC Summer School by S. Markidis et al.

In the kernel we have access to built-in variables

Kernel provides dimension and index variables

- **Dimension variables**
  - `gridDim` = number of blocks in the grid
  - `blockDim` = number of threads in each block
- **Index variables**
  - `blockIdx` = index of the block in the grid
  - `threadIdx` = index of the thread within the block

Slide courtesy of the PDC Summer School by S. Markidis et al.

Question: How do I calculate my global thread ID (1D grid)?

Using `threadIdx`, `blockIdx`, and what do I need also?

Slide courtesy of the PDC Summer School by S. Markidis et al.

Why kernels are special functions?

- Kernels execute on the GPU and **do not**, *in general*, **have access to data stored on the host side**
- **Kernels cannot return a value**, so the return type is always void, and kernel declarations starts as

```
__global__ void aKernel(arg1, arg2, ...)
```

  - **How do I get the results from my kernel ??**

Slide courtesy of the PDC Summer School by S. Markidis et al.

Transferring data from/ to device

The CUDA runtime API provides these functions for transferring inp
data to the device and transferring results back to the host:

- `cudaMalloc()` allocates device memory
- `cudaMemcpy()` transfers data to or from a device
  - `cudaMemcpy(void* dest, void* src, size_t size, cudaMemcpyHostToDevice)` **host mem → GPU mem**
  - `cudaMemcpy(void* dest, void* src, size_t size, cudaMemcpyDeviceToHost)` **GPU mem → host mem**
- `cudaFree()` frees device memory that is no longer in use

Slide courtesy of the PDC Summer School by S. Markidis et al.

Question: how I get my result from the kernel?

- **Kernels cannot return a value**, so the return type is always void, and kernel declarations starts as

```
__global__ void aKernel(arg1, arg2, ...)
```

  - **How do I get the results from my kernel ??**

Slide courtesy of the PDC Summer School by S. Markidis et al.

Data Transfers are Synchronous

By default, **data transfers are synchronous (the function does not return until the data transfer is complete)**, so `cudaMemcpy()` finishes execution before the GPU can move to other operations.

Slide courtesy of the PDC Summer School by S. Markidis et al.

Thread Synchronization

Kernels enable multiple computations in parallel but **they don't ensure order of execution** (asynchronous). CUDA provides functions to synchronize :

- **`cudaDeviceSynchronize()`** effectively synchronizes **all threads** in a grid → waits for all the threads in the kernel to complete before proceed.
- `__synchThreads()` synchronizes **threads within a block**

Slide courtesy of the PDC Summer School by S. Markidis et al.

CUDA Vector Types

Vector types CUDA extends the standard C data types of length up to 4.

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
```

Individual components are accessed with the **suffixes** **`.x`**, **`.y`**, **`.z`**, **and** **`.w`**. Accessing components beyond those declared for the vector type is an error.

```
float3 pos;
pos.z = 1.0f; // is legal
pos.w = 1.0f; // is illegal
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

Data Types for Index and Dimension Variables

CUDA uses the vector type `uint3` for the index variables, `blockIdx` and `threadIdx`. A `uint3` variable is a vector with three unsigned integer components.

CUDA uses the vector type `dim3` for the dimension variables, `gridDim` and `blockDim`. The dim3 type is equivalent to uint3 with unspecified entries set to 1. **We will use `dim3` variables for specifying execution configuration.**

Question: How do I get component of `threadIdx` in a 1D grid in the `x` direction?

Slide courtesy of the PDC Summer School by S. Markidis et al.

Compilation + Execution

- Compile `DevQuery.cu` using `nvcc DevQuery.cu -arch=sm_50 -o DQ`
- Execute `DQ` using `./DQ` on Linux machines and using `DQ` on Windows machines

Use documentation

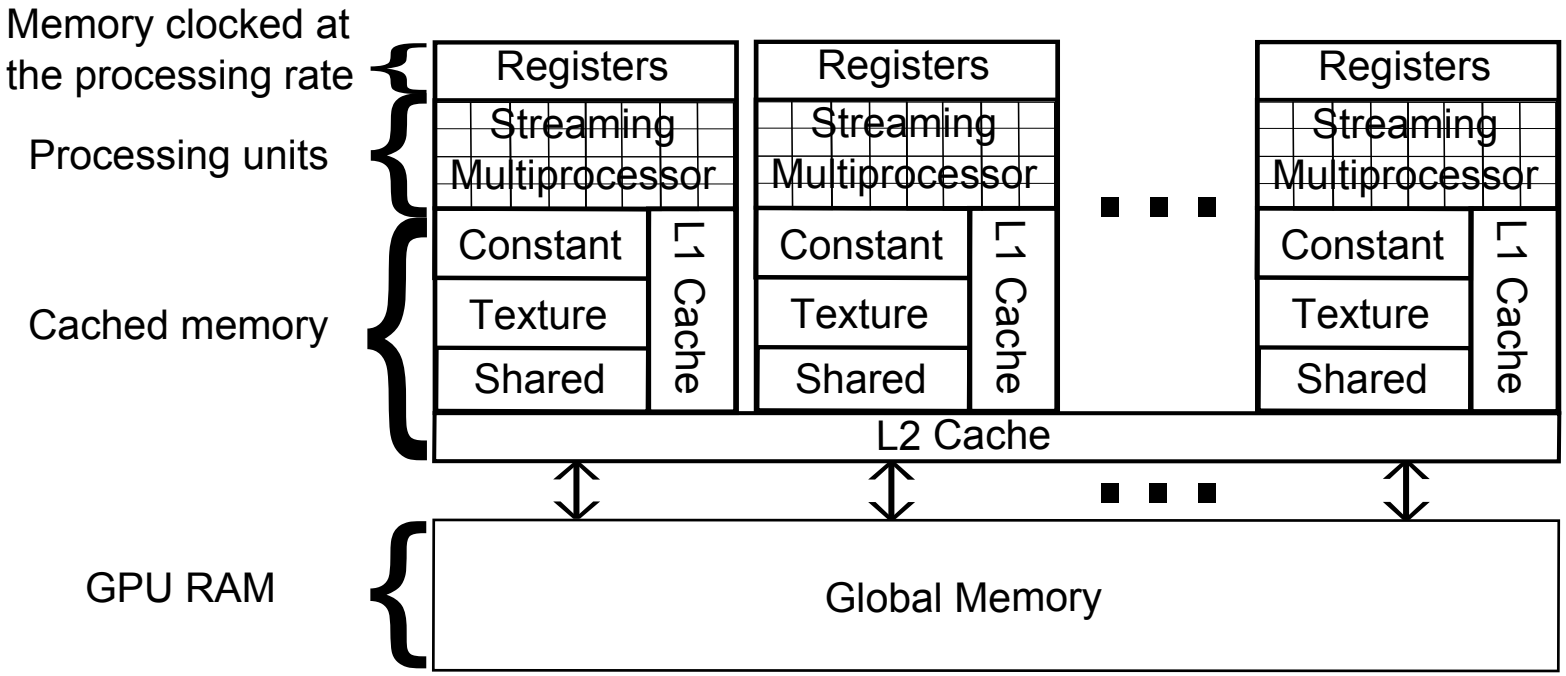
- Get the specifications of `cudaGetDeviceCount` and of `cudaGetDeviceProperties` in **CUDA_Runtime_API**
- See how they can be used from the examples of **CUDA_C_Programming_Guide**

First code

Write in the `main` function of `DevQuery.cu` the appropriate code that

- Displays the number of available GPUs
- Give the properties of GPUs
- How could you catch execution errors using testCUDA?

GPU architecture



Hardware software equivalence

- Streaming processor: Excutes threads
- Streaming multiprocessor: Executes blocks

Built-in variables

Known within functions excuted on GPU: threadIdx.x, blockDim.x, blockIdx.x, gridDim.x

Motivational Example: *dist_v1*

Compute an array of distances from a reference point to each of **`N` points uniformly spaced** along a line segment.

In large applications, there is always one or more functions that take most of the execution time → candidate for running on GPU

Take the *calculate distance* function as the expensive function in your large application. No need to port all your application to GPU!

Slide courtesy of the PDC Summer School by S. Markidis et al.

```
#include <math.h> //Include standard math library containing sqrt.
#define N 64 // Specify a constant value for array length.

// A scaling function to convert integers 0,1,...,N-1 to evenly spaced floats float
scale(int i, int n)
{
  return ((float)i) / (n - 1);
}

// Compute the distance between 2 points on a line.
float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}
// main function
int main()
{
  float out[N] = {0.0};
  // Choose a reference value from which distances are measured.
  const float ref = 0.5;
    for (int i = 0; i < N; ++i)
  {
    float x = scale(i, N);
    out[i] = distance(x, ref);
  }
  return 0;
}
```

`dist_v1` has a **single loop that scales** the loop index to create an input location and **the computes/stores the distance from the reference location**

Slide courtesy of the PDC Summer School by S. Markidis et al.

Create the CUDE source file

- Create the file `kernel.cu` where you will have CUDA source code → cuda codes have extension .cu
- Copy and paste the content of `main.cpp` into kernel.cu

Question: Is this a CUDA code?

```
#include <math.h>
#define N 64

float scale(int i, int n)
{
  return ((float)i) / (n - 1);
}

float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}

int main()
{
    float out[N] = {0.0};
    const float ref = 0.5;
    for (int i = 0; i < N; ++i)
    {
       float x = scale(i, N);
       out[i] = distance(x, ref);
    }
   return 0;
}
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

Modify `kernel.cu` 1/2

- Delete `#include <math.h>` because CUDA internal files already include `math.h`, and insert `<stdio.h>` to enable printing the output

- Add `#define TPB 32`, **to indicate the number of threads per block** that will be used in your kernel launch

```
#include <math.h>
#include <stdio.h>
#define N 64
#define TPB 32

float scale(int i, int n){
  return ((float)i) / (n - 1);
}

float distance(float x1, float x2){
  return sqrt((x2 - x1)*(x2 - x1));
}
…
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

Modify `kernel.cu` 2/2

```
__xxx__ void distanceKernel(float *d_out,
float ref, int len)
{
…

}
```

Question: `__global__`, `__device__`, or `__host__` ?

Hint: We call this function from the host and want to run on GPU

Slide courtesy of the PDC Summer School by S. Markidis et al.

Define the kernel 1/3

```
__xxx__ void distanceKernel(float *d_out,
float ref, int len)
{
…

}
```

Question: `__global__`, `__device__`, or `__host__` ?

Hint: We call this function from the host and want to run on GPU

Slide courtesy of the PDC Summer School by S. Markidis et al.

Define the kernel 2/3

```
__xxx__ float scale(int i, int n)
{
  return ((float)i)/(n - 1);
}
```

Question: `__global__`, `__device__`, or `__host__` ?

Hint: We call this function from the GPU and want to run on GPU

Slide courtesy of the PDC Summer School by S. Markidis et al.

Define the kernel 3/3

```
__xxx__ float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}
```

Question: `__global__`, `__device__`, or `__host__` ?

Hint: We call this function from the GPU and want to run on GPU

Slide courtesy of the PDC Summer School by S. Markidis et al.

Get the global thread ID using index variables

```
__global__ void distanceKernel(float *d_out, float ref, int len)
{
  const int i = blockIdx.x*blockDim.x + threadIdx.x;
  const float x = scale(i, len);
  d_out[i] = distance(x, ref);
  printf("i = %2d: dist from %f to %f is %f.\n", i, ref, x, d_out[i]);
}
```

Inside the kernel add the formula for computing index `i` (**to replace the loop index of the same name that is now removed**) using built-in index and dimension variables that CUDA provides with every kernel launch:

```
const int i = blockIdx.x*blockDim.x + threadIdx.x
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

Create results array (d_out) on the GPU

Question: Which CUDA function do we use?

```
…
int main()
{
  …
  // Declare a pointer for an array of floats
  float *d_out = 0;
  // Allocate device memory for d_out
  cudaMalloc(&d_out, N*sizeof(float));
  // Launch kernel to compute
  distanceKernel<<<N/TPB, TPB>>>(d_out, ref, N);
  return(0);
}
```

Did we forget anything?

Slide courtesy of the PDC Summer School by S. Markidis et al.

Putting everything together: our first CUDA code

```
#include <stdio.h>
#define N 64
#define TPB 32


__device__ float scale(int i, int n)
{
  return ((float)i)/(n - 1);
}


__device__ float distance(float x1, float x2)
{
  return sqrt((x2 - x1)*(x2 - x1));
}


__global__ void distanceKernel(float *d_out, float ref, int len)
{
  const int i = blockIdx.x*blockDim.x + threadIdx.x;
  const float x = scale(i, len);
  d_out[i] = distance(x, ref);
  printf("i = %2d: dist from %f to %f is %f.\n", i, ref, x,
d_out[i]);
}
```

```
int main()
{
  const float ref = 0.5f;

  // Declare a pointer for an array of floats
  float *d_out = 0;

  // Allocate device memory to store the output array
  cudaMalloc(&d_out, N*sizeof(float));

  // Launch kernel to compute and store distance values
  distanceKernel<<<N/TPB, TPB>>>(d_out, ref, N);

  cudaFree(d_out); // Free the memory
  return 0;
}
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

Questions

- Does it work?
- Does it print anything ?

Use `cudaDeviceSynchronize()` !

Slide courtesy of the PDC Summer School by S. Markidis et al.

Where is my data: host or device memory?

- Remember that the kernel (`distanceKernel()`) executes on the device, so it cannot return a value to the host.
- The kernel generally has access to device memory, **not to the host memory**, so we allocate device memory for the output array using `cudaMalloc()`

Question: In `kernel.cu`, how would you move `d_out` from the device to host memory?

Slide courtesy of the PDC Summer School by S. Markidis et al.

Careful with Integer Arithmetic!

The kernel execution configuration is specified so that each block has `TPB` threads, and there are `N/TPB` blocks.

Problem: What happens if `N = 65` ?

We get 65/32 = **2 blocks of 32 threads**. In this case, **the last entry in the array would not get computed** because there is no thread with the corresponding index.

The simple trick is to change the number of blocks as `(N+TPB-1) /TPB` to **ensure that the number of blocks is rounded up**.

Slide courtesy of the PDC Summer School by S. Markidis et al.

How to choose TPB or execution configuration?

To choose the specific execution configuration that will produce the best performance involve both art and science.

- To choose **some multiple of 32 is reasonable** since it matches up somehow with the number of **CUDA cores in an SM**
- There are limits: a single block **cannot contain more than 1,024 threads**
- For large problems, reasonable to test are 128, 256 and 512.

Slide courtesy of the PDC Summer School by S. Markidis et al.

Always with `DevQuery.cu`

Hello World!

- See in CUDA documentation how `cudaDeviceSynchronize` and `printf` work
- Printf Hello World! in `empty_k`
- Use `cudaDeviceSynchronize` just after `empty_k` call in the `main` function
- Call `empty_k` with <<<1, 1>>>, then with <<<1, 8>>> and with <<<8, 1>>>

Built-in variables

- Instead of Hello World!, display the built-in variables
- Execute with <<<1, 1>>>, <<<1, 32>>>, <<<1, 33>>>, <<<32, 1>>> and with <<<8, 4>>>
- Propose a linear combination of threadIdx.x and blockIdx.x that provides successive different values

Function declaration and calling

Standard C functions

The same as for C or C++ programming

Kernel functions

- Called by the CPU and executed on the GPU
- Declared as `__global__ void myKernel (...) { ...; }`
- Called standardly by
  `myKernel<<<numBlocks, threadsPerBlock>>>(...);`
  where
  - `numBlocks` should take into account the number of multiprocessors
  - `threadsPerBlock` should take into account the warp size
- Dynamic parallelism: kernels can be called within kernels by the GPU and executed on the GPU

device functions

- Called by the GPU and executed on the GPU
- Declared as

```
__device__ void myDivFun (...) { ...; }
__device__ float myDivFun (...) { ...; }
```

- Called simply by `myDivFun(...)` but only within other device functions or kernels

On CPU We want to add two large arrays of integers and put the result in a third one.

- Create a new file .cu, include `stdio.h` and `timer.h` in the top
- In the main function, allocate three arrays `a, b, c` using `malloc`
- Assign some values to `a and b`
- Using functions defined in `timer.h`, compute the execution time of adding `a and b`
- Free the CPU memory using `free`

On GPU We keep the same CPU code. For given values of *numBlocks* and *threadsPerBlock*, we want to perform an addition of *numBlocks* $\times$ *threadsPerBlock* integers

- Allocate `aGPU, bGPU, cGPU` on the GPU using `cudaMalloc`
- Transfer the values of `a, b` to `aGPU, bGPU` using `cudaMemcpy`
- Write the kernel that adds `aGPU` to `bGPU` and return the result in `cGPU`
- Copy `cGPU` to `c`
- Compute the execution time
- Propose a trick to deal with sizes different from *numBlocks* $\times$ *threadsPerBlock*

Exam – 60 %

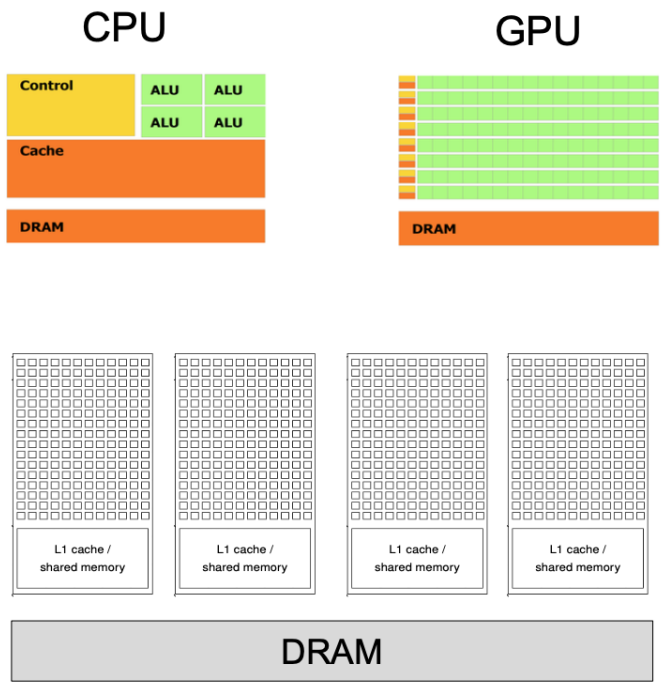
- CUDA + Laura Grigori's part
- In February 2021
- M2 will have an extra exercise on 19/11

Project – 40 %

- Only on CUDA
- Project presentations on 10/12
- (1/2) Code with runs on computer
- (2/2) Oral presentation of results or a report

Recap - What is a GPU?

- A specialized processor initially designed for graphics-like workload (videogames, video processing and CAD)
  - **Lots of cores**, **fewer control units**: very good in compute-heavy applications with little synchronization
- Now present in several supercomputers
  - **Power efficiency**: lot of parallelism but lower clock frequency
- GPU consists of one or more **SMs**, each one comprising **several cores (K80 almost 5k cores!)**



Slide courtesy of the PDC Summer School by S. Markidis et al.

Recap - What is CUDA?

It is an extension of the C language that provide basic mechanisms to:

- Create allocate variable on GPU memory

Question: Which CUDA function?

- Move data from CPU to GPU memory and vice-versa

Question: Which CUDA function?

- Define kernel and launch a kernel

Question: Which qualifier I have to use? What is the difference between a kernel and a function.

- Synchronize threads

Question: Which CUDA function?

Slide courtesy of the PDC Summer School by S. Markidis et al.

Recap - Lab

- `HelloWorld` in CUDA

Problem: not printing because of the asynchronous nature of the kernel launch

- `saxpy` in CUDA

Problem: `ARRAY_SIZE` was not a multiple of `BLOCK_SIZE`

Problem: create variable on GPU and move data to/GPU.

Easy to get it wrong:

In C, **the size of the data** to be created or moved is **in byte** (Fortran the size is the number of array elements)

Slide courtesy of the PDC Summer School by S. Markidis et al.

Back to CUDA – CUDA Vector Types

CUDA extends the standard C data types, like `int` and `float`, to be vector with 2, 3 and 4 components, like `int2`, `int3`, `int4`, `float2`, `float3` and `float4`. Other vector types are also supported.

For example, you can declare an integer vector `d` with three components and initialize with 128, 1 and 1 element in the x, y and z direction:

```
int3 d = int3(128, 1, 1);
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

Access Vector Type Components

CUDA vector types are structures (Fortran: modules) and the 1st, 2nd, 3rd, and 4th components are accessible through the fields .x, .y, .z, and .w (Fortran: %x, %y, %z and %w), respectively.

```
float3 part_pos;
part_pos.z = 1.0f; // is legal
part_pos.w = 1.0f; // is illegal: Why?
```

Question: What do the `.x` remind you of?

Slide courtesy of the PDC Summer School by S. Markidis et al.

Type of `blockIdx` and `threadIdx`

CUDA uses the vector type `uint3` for the index variables, `blockIdx` and `threadIdx`. A `uint3` variable is a vector with three unsigned integer components.

We used `threadIdx.x` and `blockIdx.x` to retrieve indices in 1D grid.

Slide courtesy of the PDC Summer School by S. Markidis et al.

CUDA Type dim3

CUDA uses the vector type `dim3` for the dimension variables, `gridDim` and `blockDim`.

The `dim3` type is equivalent to `uint3` with unspecified entries set to 1.

As you probably noticed in the Lab1 for the lab, we could use either:

```
dim3 grid(1,1,1); // 1 block in the grid
dim3 block(32,1,1); // 32 threads per block
```

Or set block and thread per block as scalar quantity in the <<< >>> (execution configuration)

Slide courtesy of the PDC Summer School by S. Markidis et al.

Why do we need higher dimension CUDA grids?

Several applications points regularly distributed on a **2D plane**. A first example can be a matrix. A second example involves digital image processing.

A digital raster imagine consists of a collection of **picture elements** (**pixel**) arranged in a uniform 2D rectangular grid with each pixel having an **intensity value**.

Example of 3x3 .bmp image file (see lab today)

Header								
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	(0,8)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)

Slide courtesy of the PDC Summer School by S. Markidis et al.

2D Grid Kernel – Thread per block in x and y

Computing data for an image involves `W` columns and `H` rows, and we can organize the computation into 2D blocks with `TX` threads in the x-direction and `TY` threads in the y-direction.

```
dim3 blockSize(TX, TY); // Equivalent to dim3 blockSize(TX, TY, 1);
```

Question: can we use 1D block in a 2D grid?

Slide courtesy of the PDC Summer School by S. Markidis et al.

2D Grid Kernel – Number of blocks in x and y

Questions: how do we choose the number of blocks in x and y ? If we follow the 1D example, what would be `N` or the `ARRAY_SIZE` equivalent?

We compute the number of blocks (`bx` and `by`) needed in each direction exactly as in the 1D case:

```
int bx = (W + blockSize.x - 1)/blockSize.x ;
int by = (H + blockSize.y - 1)/blockSize.y ;
```

The syntax for specifying the grid size (in blocks) is

```
dim3 gridSize = dim3 (bx, by);
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

2D Grid Kernel Launch

We are ready now to launch (no difference with 1D grid):

```
kernelName<<<gridSize, blockSize>>>(args)
```

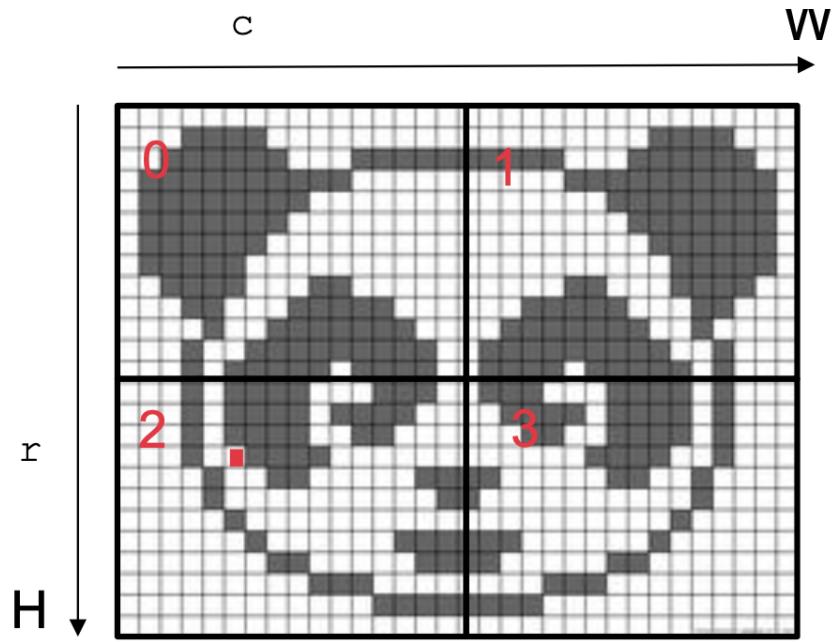
Slide courtesy of the PDC Summer School by S. Markidis et al.

Determine global indices

To identify our pixel in the image we will use to global indices `c` and `r`.

Question: How you calculate `c` and `r` for the red pixel?

```
int c = blockIdx.x*blockDim.x + threadIdx.x;
Int r = blockIdx.y*blockDim.y + threadIdx.y;
```

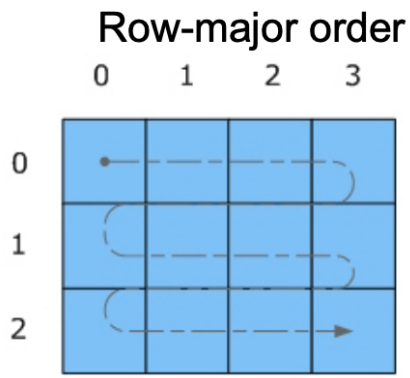


Slide courtesy of the PDC Summer School by S. Markidis et al.

Flattening global indices to 1D global indices

In several cases, it is convenient to express our 2D data as 1D data (flattening): use simply a 1D array of length `W*H`

We place values in the 1D array in **row-major order:** we store the data from row 0, followed by data from row 1 and so on.



Question: Why row-major order and not column-major order ?

Question: How do you calculate `i`, 1D index? `int i = r*w + c;`

Slide courtesy of the PDC Summer School by S. Markidis et al.

CUDA code for distance between points in 2D

```
#define W 32
#define H 32
#define TX 8 // number of threads per block along x-axis
#define TY 8 // number of threads per block along y-axis

int divUp(int a, int b) { return (a + b - 1) / b; }
…
int main() {
  float *out = (float*)calloc(W*H, sizeof(float)); // set all the points to 0
  float *d_out = NULL;
  cudaMalloc(&d_out, W*H*sizeof(float));
  float2 pos = { 1.0, 0.0};   // ref. point
  dim3 blockSize(TX, TY);
  dim3 gridSize(divUp(W, TX), divUp(H, TY));
  distanceKernel<<<gridSize, blockSize>>>(d_out, W, H, pos);
  cudaMemcpy(out, d_out, W*H*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(d_out);
  free(out);
  return 0;
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

CUDA kernel and device code

```
__global__ void distanceKernel(float *d_out, int w, int h, float2 pos)
{
  const int c = blockIdx.x * blockDim.x + threadIdx.x; // column
  const int r = blockIdx.y * blockDim.y + threadIdx.y; // row
  const int i = c + r*w;
  if ((c >= w) || (r >= h)) return;
      d_out[i] = distance(c, r, pos); // compute and store result
}


__device__ float distance(int c, int r, float2 pos)
{
  return sqrtf((c - pos.x)*(c - pos.x) + (r - pos.y)*(r - pos.y));
}
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

3D Grids

3D data set can be thought as image stack composed of **3D voxels** is a volume `W*H*D` (`D` = Depth)

An execution configuration in 3D will require to define the number of threads in the x, y and z direction, i.e TX, TY and TZ

```
dim3 blockSize(TX, TY, TZ);
```

As usual, the block grid size is then calculate depending on the input size:

```
int bx = (W + blockSize.x - 1)/blockSize.x;
int by = (H + blockSize.y - 1)/blockSize.y;
int bz = (D + blockSize.z - 1)/blockSize.z;
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

3D Grids – Indices

In addition to row (`r`) and column (`c`) global indices, we need a new integer variable to have a global index in the stack (`s` for *stack* or *stratum*):

```
int s = blockIdx.z*blockDim.z + threadIdx.z;
```

The flattened 1D index becomes:

```
int i = c + r*w + s*w*h;
```

Slide courtesy of the PDC Summer School by S. Markidis et al.

Pricing European

$F_t = e^{-r(T-t)} E(f(S_s, t \leq s < T) | \mathfrak{F}_t),\ t \in [0, T)$ where

- f is the contract's payoff
- T is the contract's maturity
- r is the risk-free interest rate

$X = f(S_{s, t \leq s < T})$

We want to simulate $F(0, y) = E(X)$ using a family $\{X_i\}_{i \leq n}$ of i.i.d $\sim X$

- **Strong law of large numbers**:

$$P\left(\lim_{n \to +\infty} \frac{X_1 + X_2 + ... + X_n}{n} = E(X)\right) = 1$$

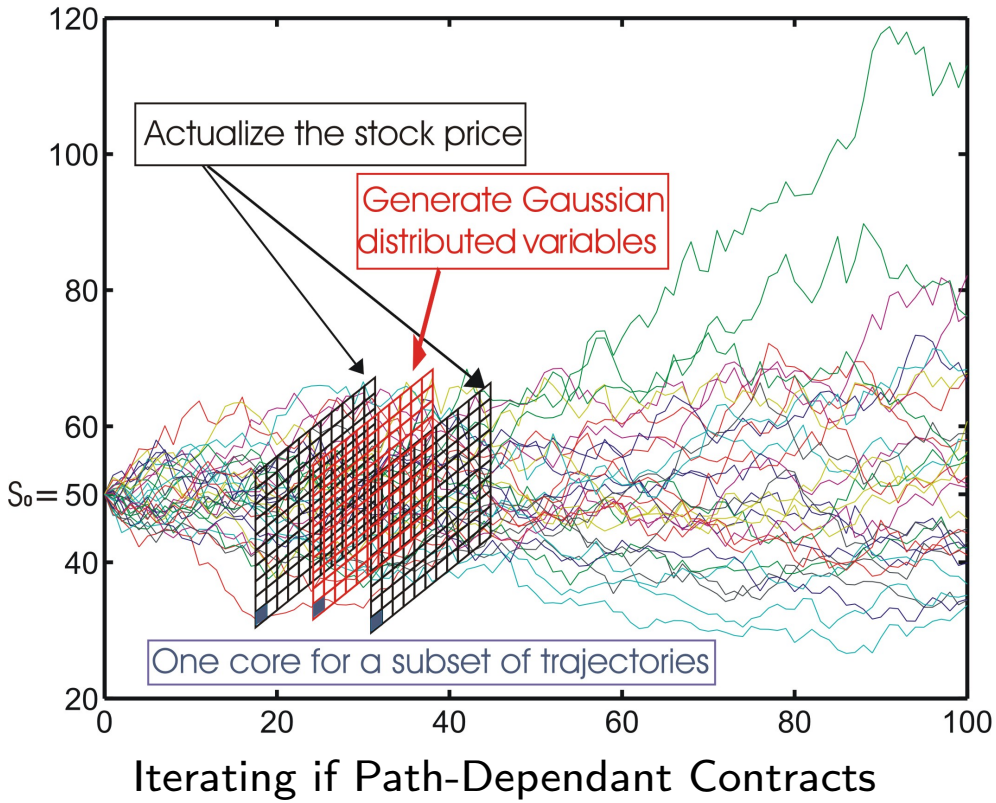
- Denoting $\epsilon_n = E(X) - \frac{X_1 + X_2 + ... + X_n}{n}$
- **Central limit theorem**:

$$\frac{\sqrt{n}}{\sigma} \epsilon_n \to G \sim \mathcal{N}(0, 1)$$

- There is a 95% chance of having:

$$|\epsilon_n| \leq 1.96 \frac{\sigma}{\sqrt{n}}$$

European path-dependant pricing



For each time step:

1. Random number generation (if parallelized)
2. Stock price actualization
3. Compute the payoff

General Form of linear RNGs

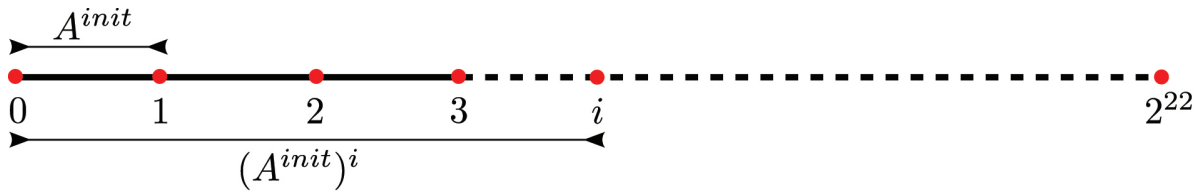
Without loss of generality:

$$X_n = (AX_{n-1} + C)\ mod(m) = (A\ :\ C) \begin{pmatrix} X_{n-1} \\ \dots \\ 1 \end{pmatrix}\ mod(m) \qquad (3)$$

Parallel-RNG from Period Splitting of One RNG

* Pierre L'Ecuyer proposed a very efficient RNG (1996) which is a CMRG on 32 bits: Combination of two MRG with $lag = 3$ for each MRG.
* Very long period $\sim 2^{185}$

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) mod(m)$$



Parallel-RNG from Parameterization of RNGs

* Same parallelization as SPRNG Prime Modulus LCG.
* The same RNG with different parameters "a":

$$x_n = ax_{n-1} + c \quad mod(m)$$

Bullet option in Black & Scholes model

Price at $t = 0$

$F_0 = e^{-r_0 T} E\left((S_T - K)_+ 1_{\{I_T \in [P_1, P_2]\}}\right)$ with $I_t = \sum_{T_i \leq t} 1_{\{S_{T_i} < B\}}$

- K, T are respectively the contract's strike and maturity
- $0 < T_1 < ... < T_M = T$ is a predetermined schedule
- The barrier B should be above S I_T times $\in \{P_1, ..., P_2\} \subset \{0, ..., M\}$
- r_0 risk-free rate

Black & Scholes model

For $0 \leq s < t \leq T$, $S_t \equiv S_s \exp\left((r_0 - \sigma^2/2)(t-s) + \sigma\sqrt{t-s}G\right)$ and $S_0 = x_0$

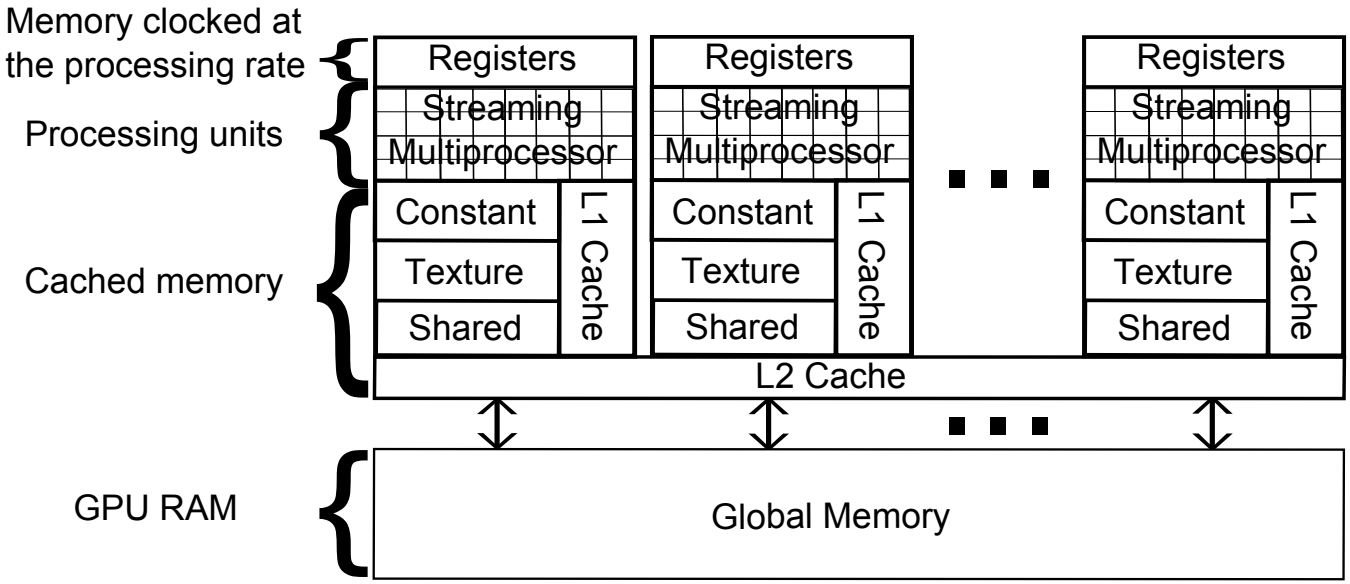
- σ is the volatility
- G is independent from S_s and has a standard random distribution
- x_0 is the initial spot price of S at time 0

Complete `MC.cu`

After memory allocation and free, uncomment the kernel call then fill it with the appropriate call of `BoxMuller_d` and of `BS_d`.

Plan

Shared: Second fastest memory



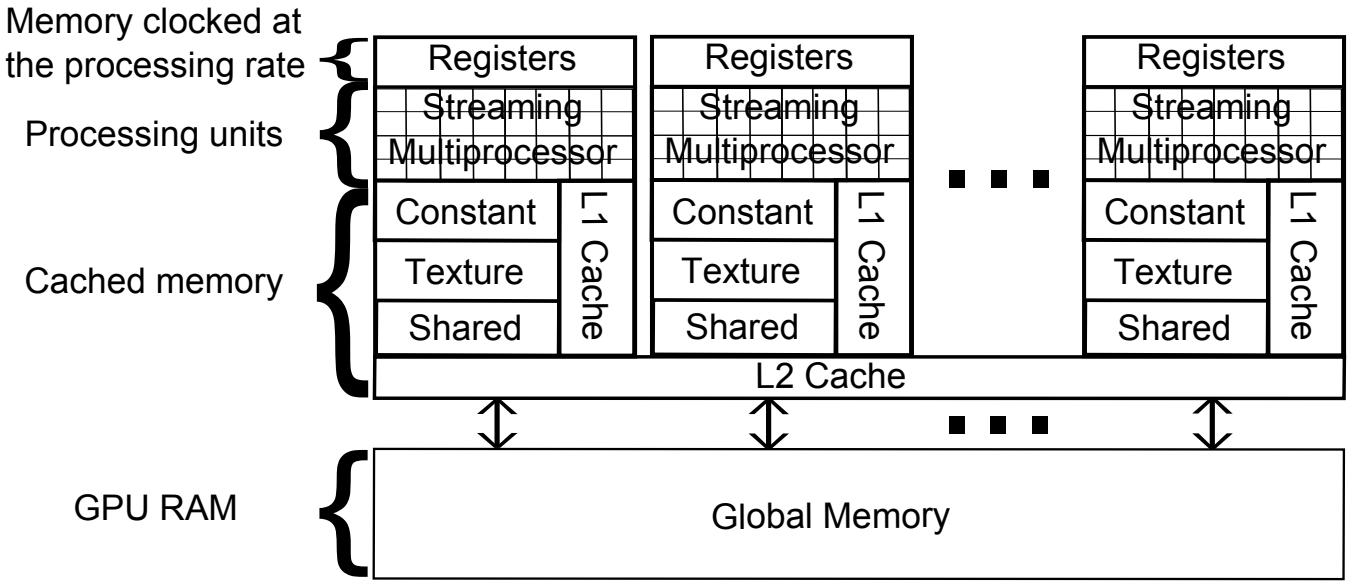
Shared memory

- Cached memory visible to all threads of the same block
- Has a lifetime of a kernel
- Static allocation of arrays: `__shared__ float A[100];`
- Dynamic allocation of arrays: `extern __shared__ float A[];`
  kernel call: `myKernel<<<..., ..., 100*sizeof(float)>>>(...);`

In `MemComp` Replace the use of `aGlob` by a static shared array

In `MC_k` For the Black & Scholes process, replace the global array by static shared array

Registers: Fastest memory



Registers

- Divided homogeneously between threads of the same block
- Have a lifetime of a kernel
- Cannot be used for arrays

In `MemComp` Replace the use of `aGlob` by a register variable

In `MC_k`

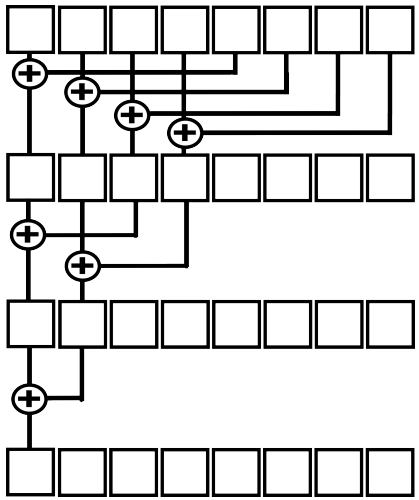
- Except for the Black & Scholes process, replace as much as possible the use of global arrays by registers
- For the Black & Scholes process, replace the static allocation of the shared memory by a dynamic one

Some facts

- Threads can access to any memory space of the shared memory of their block
- Threads of different blocks cannot exchange values within the same kernel
- The synchronization barrier `__syncthreads();` ensures that threads of the same block wait for all other threads of the same block

Dot product exercise

- Store the product result in the shared memory then perform a reduction using the following scheme with a `__syncthreads();` at each step



- How `atomicAdd` operates? Use it for the reduction through blocks
- What happens when `atomicAdd` is used for the whole sum?

Exercise

Using

```
sum_k<<<NB, NTPB>>>(float *In, float *Out, int d)
```

we compute the sum `Out` of elements of a floating point array `In` of size d. In the following, when needed, we assume only static shared memory allocation, for instance:

```
__shared__ float InSh[64];
```

1. Define the kernel `sum_k` using the maximum number of threads to sum on `In` when we execute
   a) `sum_k<<<1, 1>>>(In, Out, 32);`
   b) `sum_k<<<1, 2>>>(In, Out, 32);`
   c) `sum_k<<<1, 4>>>(In, Out, 32);`
   d) `sum_k<<<1, 64>>>(In, Out, 128);`
2. Using question 1.d, define `sum_k` that sums on `In` when we execute `sum_k<<<2, 64>>>(In, Out, 256);`
3. Compare solution 2. to a sum that involves only atomicAdd.
4. From now on, we want to compute various sums stored in the same array pointed by `Out`. We sum on various arrays of the same size d stored in a concatenated way in `In`. Explain how the indices `int tidx = threadIdx.x%d; int Qt = (threadIdx.x-tidx)/d; int gbx = Qt + blockIdx.x*(blockDim.x/d);` can be used in `sum_k`.
5. Write `sum_k<<<40, 512>>>(In, Out, 256);` that batch computes various sums.

Some facts

- Threads in the same warp, called lanes, can access to any register associated to their warp
- Functions prefixed with `__shfl` are needed for this communication
- Lanes do not need synchronization

Syntax example

```
float __shfl_down(float var, unsigned int delta, int width);
```

- `var` is the value to be communicated
- `delta` is a lane translation index
- `width` is the number of involved threads $\leq$`warpSize=32`

Test this

```
__global__ void kernel(int *A){
    int idx = threadIdx.x + blockIdx.x*blockDim.x;
    int lane = threadIdx.x%warpSize;
    int loc;
    loc = A[idx];
    if(blockIdx.x<1 && lane==idx){
        printf("%d , %i, %i \n", lane, loc,
                __shfl_down(loc, 1, warpSize));
    }
}
```

Now, make the appropriate changes to `MC_k`

Benefits

- Makes the shared memory available for other tasks
- Registers are faster
- Does not have to synchronize between threads

Plan

Standard host allocation vs. locked memory vs. mapped memory

Standard host allocation

- Memory space virtually limited by the machine's RAM
- The OS controls the memory space
- The data transfer is the slowest it can get

Locked allocation

- Memory space much smaller then the machine's RAM
- The OS loses some control on the memory space
- The data transfer is faster then standard allocation

Mapped allocation

- Memory space much smaller then the machine's RAM
- The OS loses some control on the memory space
- Double pointers: One used by CPU and the other one used by GPU.
- The **implicit** data transfer is the fastest it can get

How to implement?

Locked memory

- Make sure that the memory space to be locked is not too big
- Replace `malloc` by `cudaHostAlloc` and `free` by `cudaFreeHost`

Mapped allocation

- Make sure that the memory space to be locked is not too big
- Before calling any other function (at the beginning of main), one has to execute `cudaSetDeviceFlags(cudaDeviceMapHost)`
- The CPU allocation has to be done using `cudaHostAlloc` with `cudaHostAllocMapped` as an option.
- The GPU gets a pointer with `cudaHostGetDevicePointer`
- Make sure that the GPU had finished working on mapped memory thanks to `cudaDeviceSynchronize`.

Do this

- Write a code that compares the standard transfer to the locked memory transfer
- Compare the GPU use of mapped memory to the standard solution and to the locked memory solution

Using different streams of type cudaStream_t

For concurrency

- Create streams with `cudaStreamCreate`
- Execute concurrent kernels with different streams. For example:

```
myKernel1<<<..., ..., ..., stream1>>>(...);
myKernel2<<<..., ..., ..., stream2>>>(...);
```

- Make sure that streams finished processing with `cudaStreamSynchronize`
- Destroy streams with `cudaStreamDestroy`

For asynchronous execution

- Use `cudaHostAlloc` for allocation on the host memory
- Create streams with `cudaStreamCreate`
- The transfer has to be done using `cudaMemcpyAsync` involving a different stream from the one used in the kernel call
- Destroy streams with `cudaStreamDestroy`

In `StreamT.cu`

- Function `withoutStream` allows comparison between using a big array or small frames
- Define a function `withStream` that operates on frames using streams