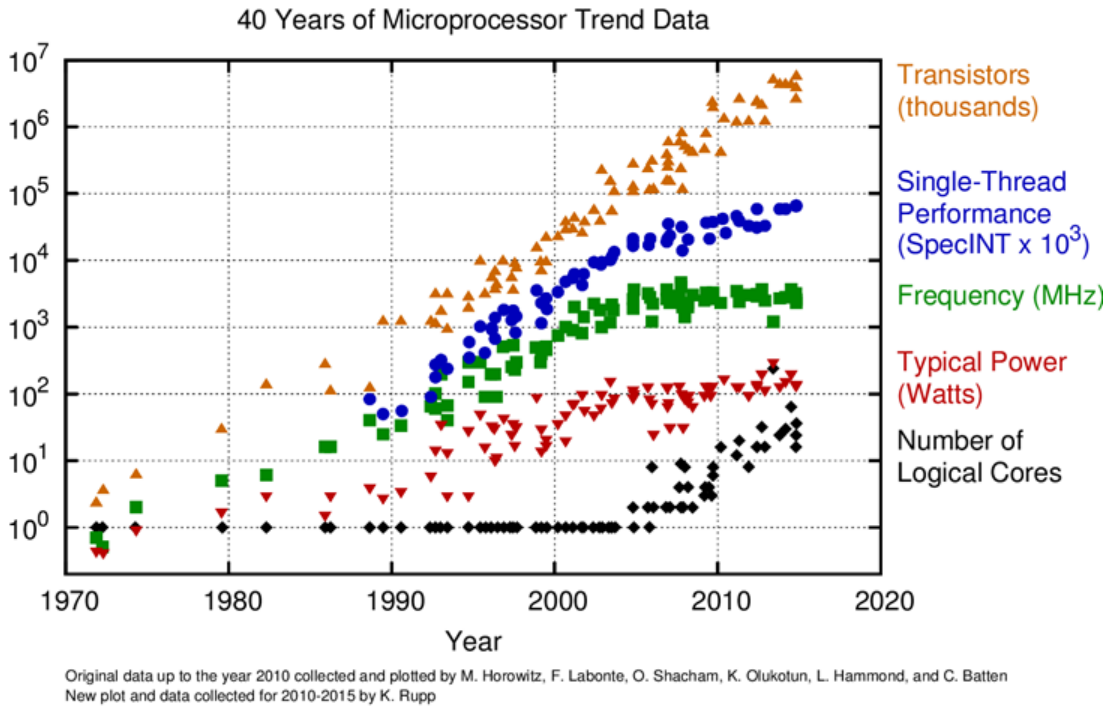
Introduction to GPU computing (1)

Computing Methods for Experimental Physics and Data Analysis
Lecture 3

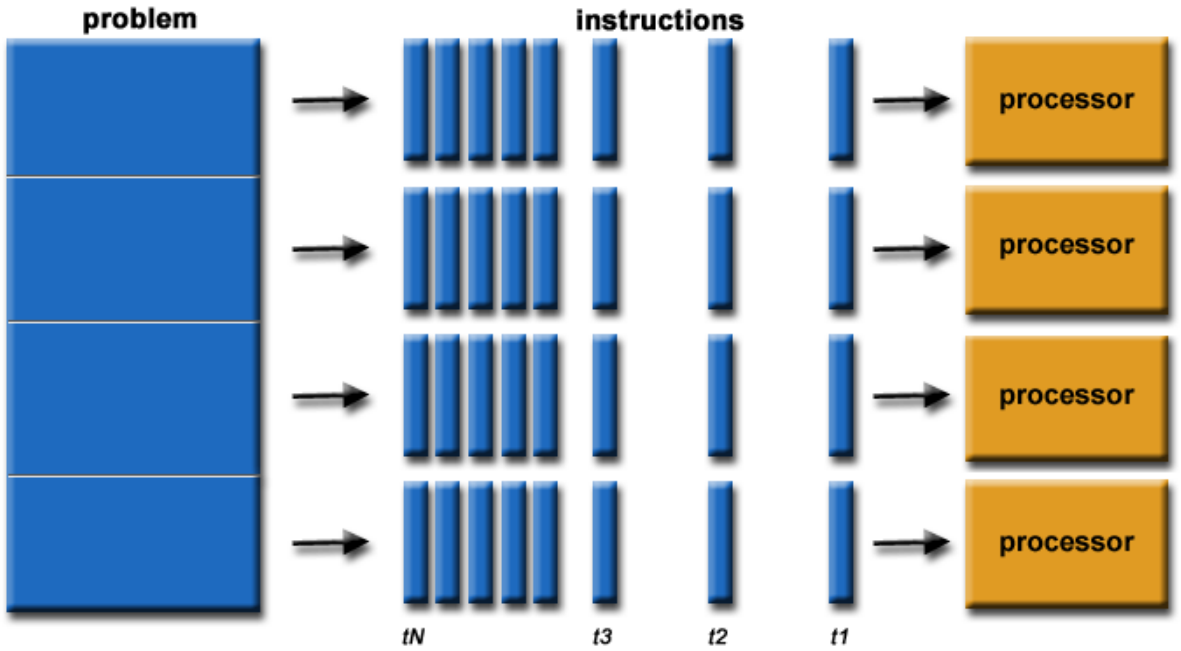
gianluca.lamanna@unipi.it

Moore's Law

- Moore's law: "The performance of microprocessors and the number of their transistors will double every 18 months"
- The increasing of performance is related to the clock
- Faster clock means higher dissipation → power wall



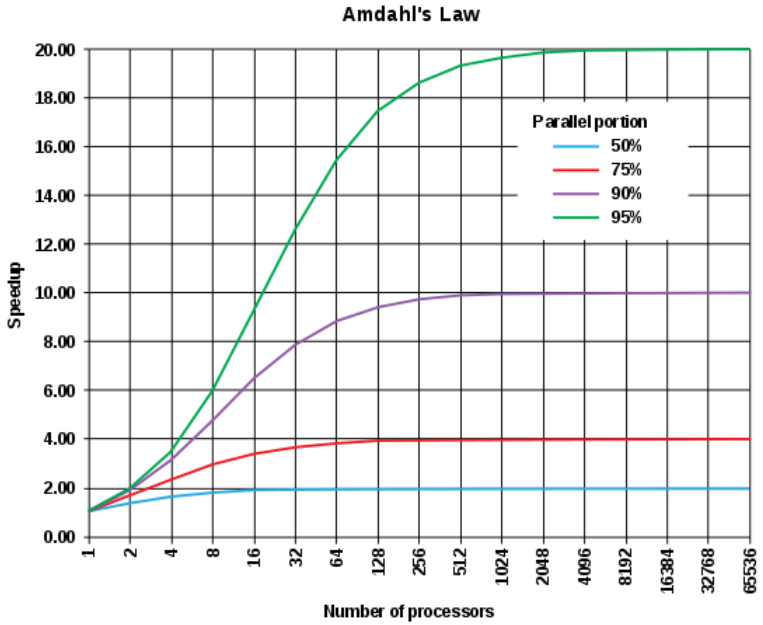
Parallel programming



- Parallel computing is no longer something for SuperComputers
  - All the processors nowadays are multicores
- The use of parallel architectures is mainly due to the physical constraints to frequency scaling

Limits of parallel programming

- Several problems can be split in smaller problems to be solved concurrently
- In any case the maximum speed-up is not linear , but it depends on the serial part of the code
  - ➡Amdahls's law
- The situation can improve if the amount of parallelizable part depends on the resources
  - ➡Gustafson's Law

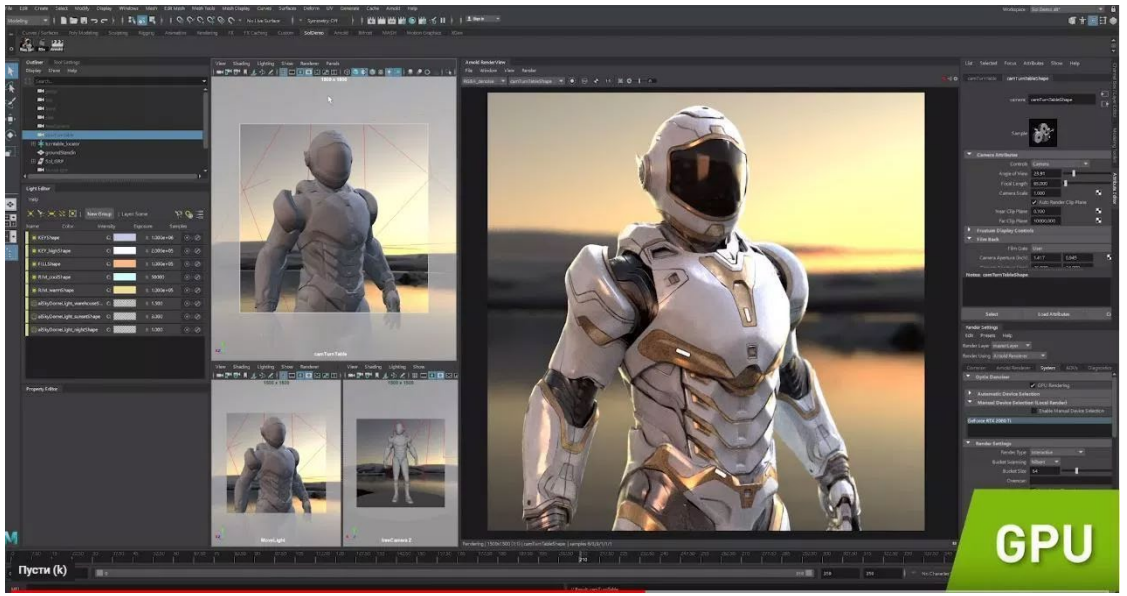


$$S_{latency} = \frac{1}{1 - p + \frac{p}{s}}$$

$$S_{latency} = 1 - p + sp$$

What are the GPUs?

- The GPUs are processors dedicated to parallel programming for graphical application.
- Rendering, Image transformation, ray tracing, etc. are typical application where parallelization can helps a lot.



Standard GPU pipeline

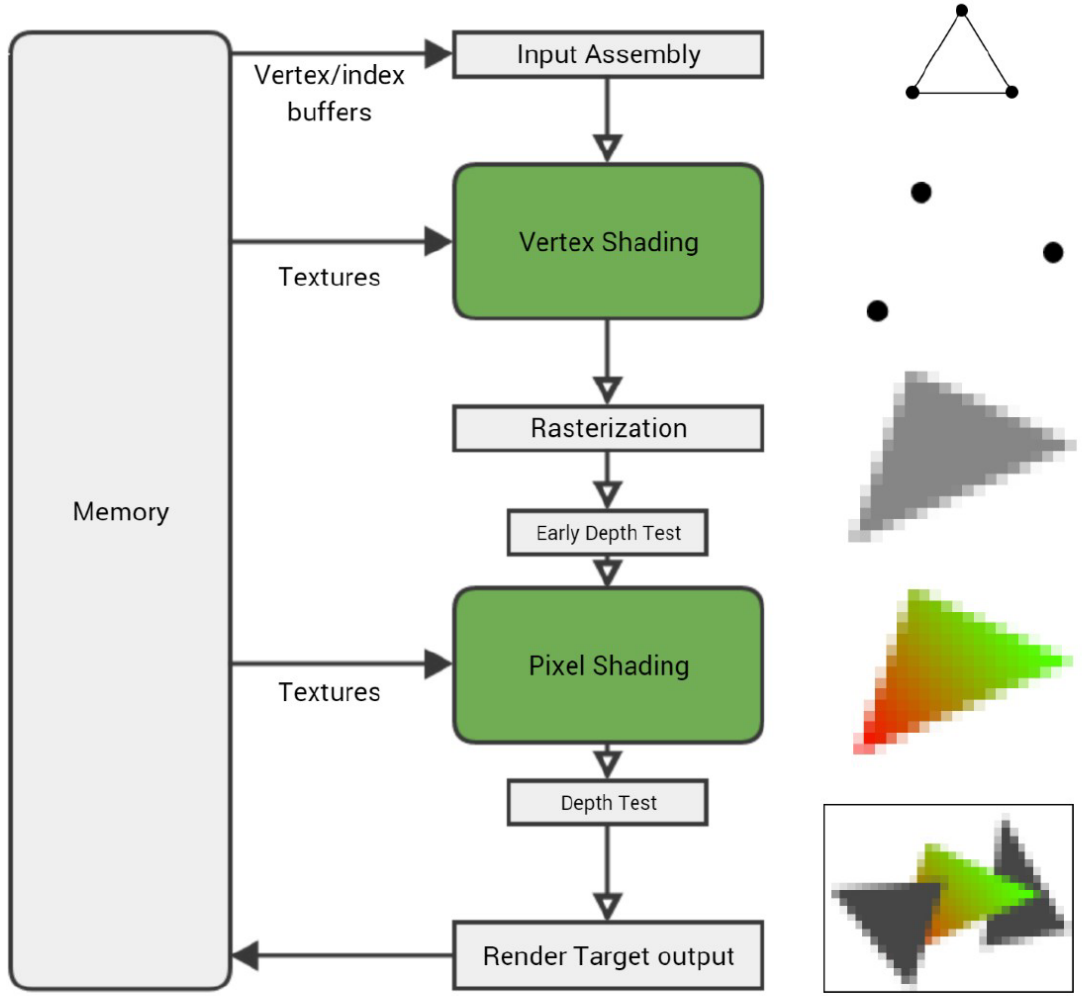
Vertex/index buffers:
Description of image with vertices and their connection to triangles

Vertex shading
For every vertex: calculate position on screen based on original position and camera view point

Rasterization
Get per-pixel color values

Pixel shading
For every pixel: get color based on texture properties (material, light, ...)

Rendering
Write output to render target



http://fragmentbuffer.com/gpu-performance-for-game-artists/

Standard GPU requirements

- Graphics pipeline: huge amount of arithmetic on independent data:
  - Transforming positions
  - Generating pixel colors
  - Applying material properties and light situation to every pixel
- Hardware needs
  - Access memory simultaneously and contiguously
  - Bandwidth more important than latency
  - Floating point and fixed-function logic

What are the GPUs?

(1997)

(2019)

- The technical definition of a GPU is "a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.“
- The possibility to use the GPU for generic computing (GPGPU) has been introduced by NVIDIA in 2007 (CUDA)
- In 2008 OpenCL: consortium of different firms to introduce a multi-platform language for manycores computing.

Why the GPUs?

- GPU is a way to cheat the Moore's law
- SIMD/SIMT parallel architecture
- The PC no longer get faster, just wider.
- Very high computing power for «vectorizable» problems
- Impressive derivative almost a factor of 2 in each generation
- Continuous development
- Easy to have a desktop PC with teraflops of computing power, with thousand of cores.
- Several applications in HPC, simulation, scientific computing…

A lot of cores…

	"Fermi"	"Fermi"	"Kepler"	"Kepler"	"Maxwell"	"Pascal"
Tesla GPU	**GF100**	**GF104**	**GK104**	**GK110**	**GM200**	**GP100**
Compute Capability	2.0	2.1	3.0	3.5	5.3	6.0
Streaming Multiprocessors (SMs)	16	16	8	15	24	56
FP32 CUDA Cores / SM	32	32	192	192	128	64
FP32 CUDA Cores	512	512	1536	2880	3072	3584
FP64 Units	-	-	512	960	96	1792
Threads / Warp	32	32	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16	32	32
32-bit Registers / Multiprocessor	32768	32768	65536	65536	65536	65536
Max Registers / Thread	63	63	63	255	255	255
Max Threads / Thread Block	1024	1024	1024	1024	1024	1024
Shared Memory Size Configurations	16 KB	16 KB	16 KB	16 KB	96 KB	64 KB
	48 KB	48 KB	32 KB	32 KB		
			48 KB	48 KB		
Hyper-Q	No	No	No	Yes	Yes	Yes
Dynamic Parallelism	No	No	No	Yes	Yes	Yes
Unified Memory	No	No	No	No	No	Yes
Pre-Emption	No	No	No	No	No	Yes

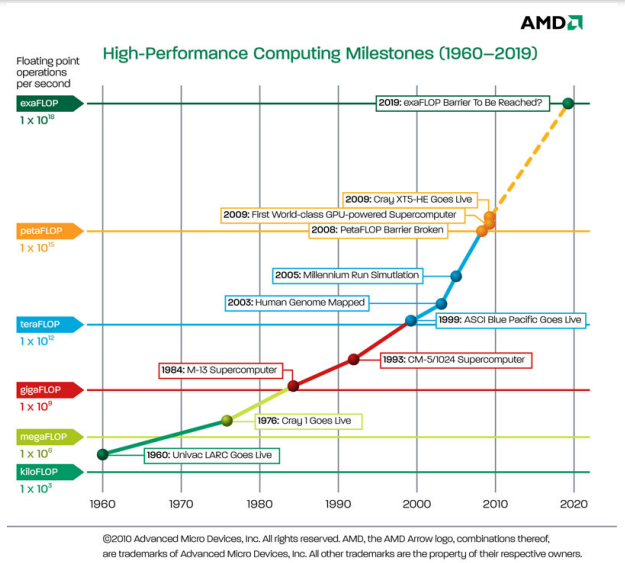
GPU Features	GTX 1080Ti	RTX 2080 Ti	Quadro P6000	Quadro RTX 6000
Architecture	Pascal	Turing	Pascal	Turing
GPCs	6	6	6	6
TPCs	28	34	30	36
SMs	28	68	30	72
CUDA Cores / SM	128	64	128	64
CUDA Cores / GPU	3584	4352	3840	4608
Tensor Cores / SM	NA	8	NA	8
Tensor Cores / GPU	NA	544	NA	576
RT Cores	NA	68	NA	72
GPU Base Clock MHz (Reference / Founders Edition)	1480 / 1480	1350 / 1350	1506	1455

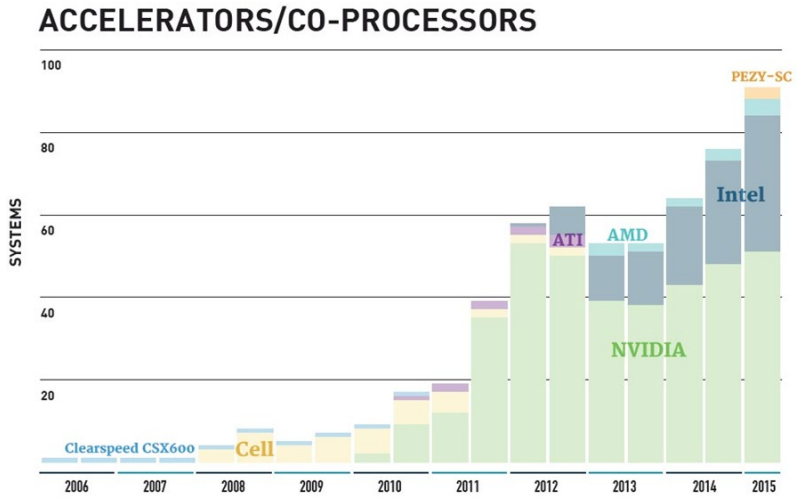
Metrics

- FLOPS (Floating Point operation per second)
  - ➡ It is a measurement of the computing power of a processor
  - ➡ Theoretically is defined as

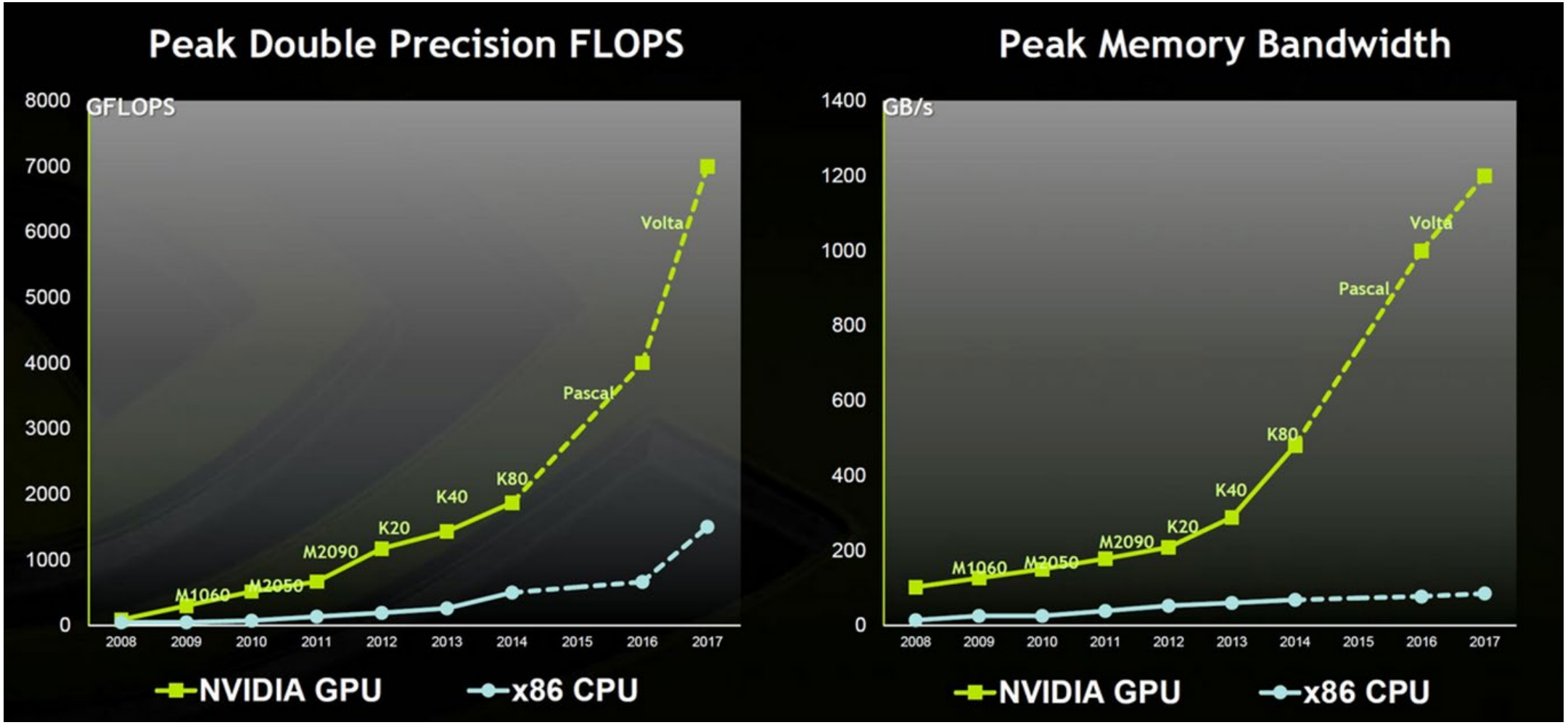
$$FLOPS = clock * cores * Operation/cycle$$

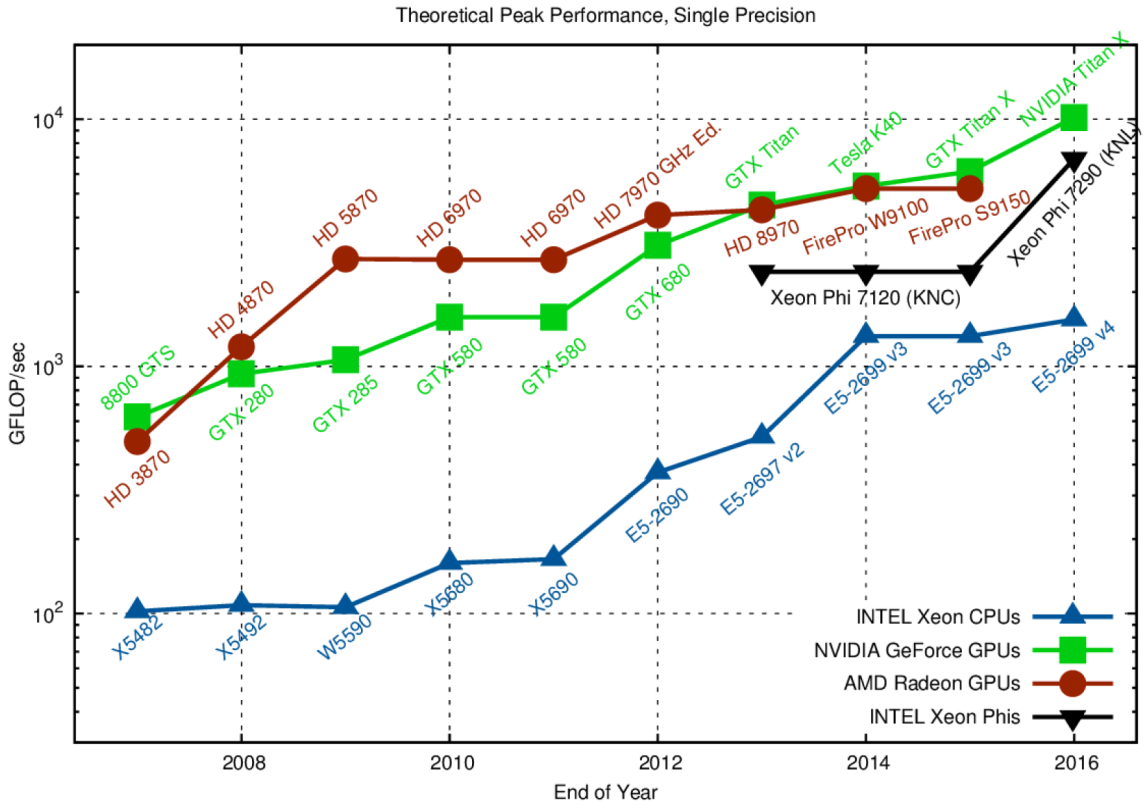
  - ➡ Actually this formula doesn't take into account several things
  - ➡ The real extimation is made experimentally by using standard packages (LINPACK, LAPACK)
  - ➡ The FLOPS is only a first indication of the computing power
  - ➡ Other metrics has been invented to avoid the limitations of the FLOPS
- SPECint and SPECf
  - ➡ They are suites of 12 benchmarks of diffent type (for integer and floating point)
  - ➡ The extimation is relative to a particular machine

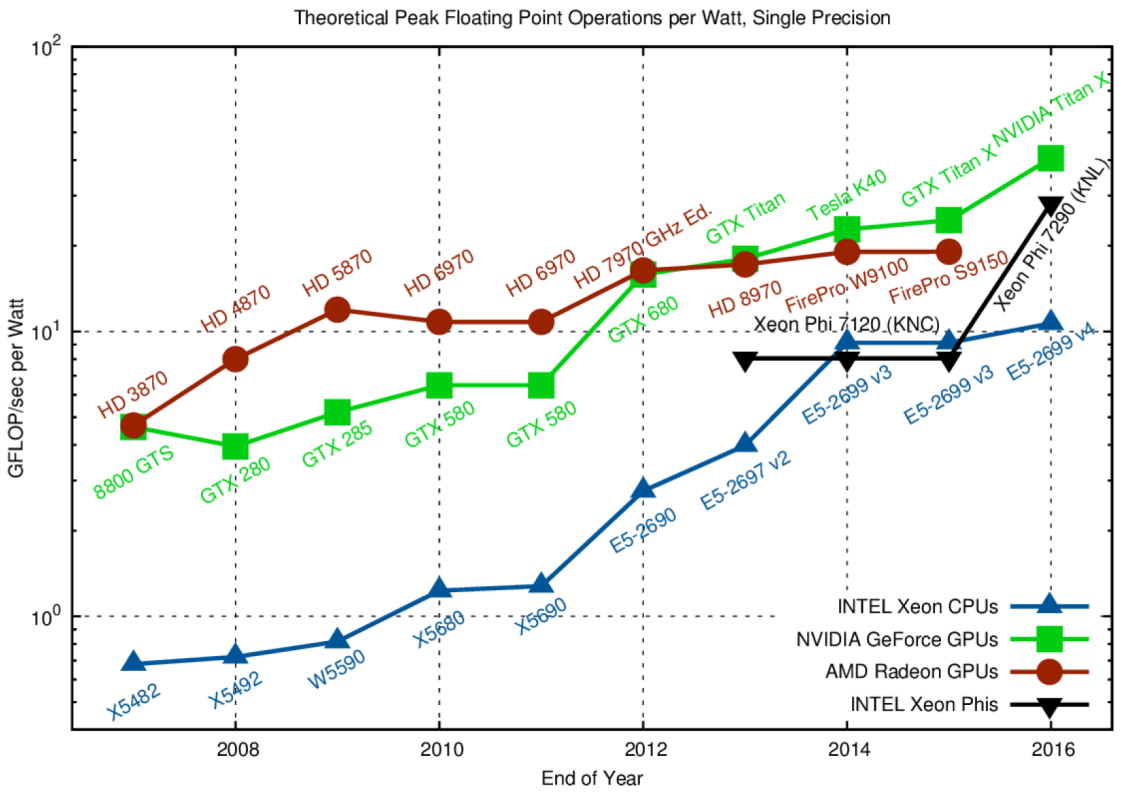




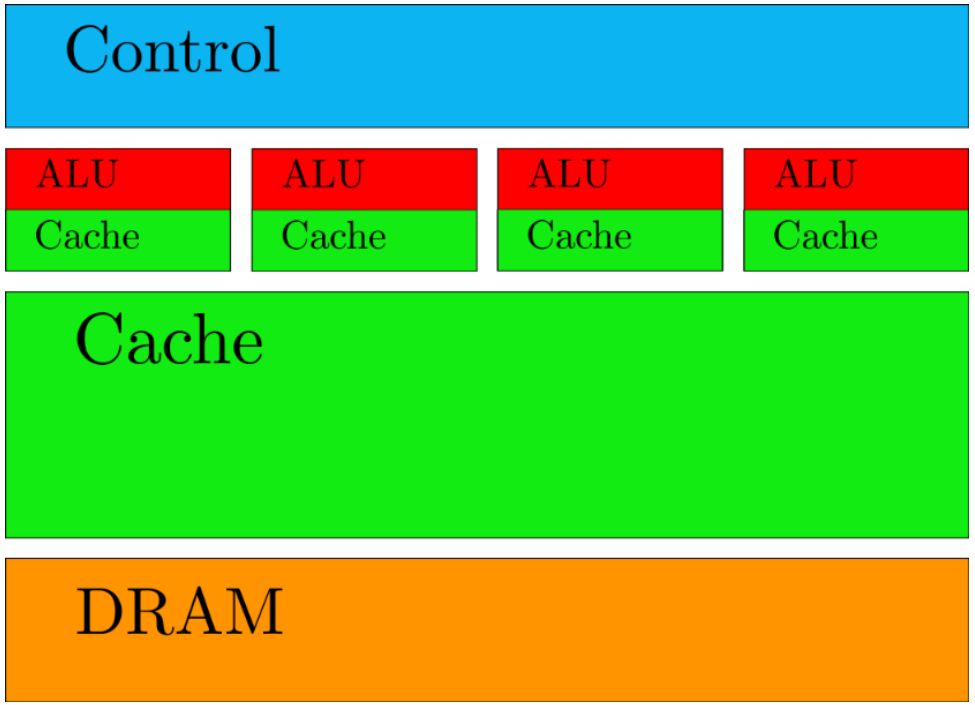
Computing power comparison




Theoretical Peak Performance, Single Precision
GFLOP/sec
End of Year
2008
2010
2012
2014
2016
8800 GTS
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
GTX Titan
Tesla K40
GTX Titan X
NVIDIA Titan X
HD 3870
GTX 280
GTX 285
GTX 580
GTX 580
GTX 680
HD 8970
FirePro W9100
FirePro S9150
Xeon Phi 7290 (KNL)
Xeon Phi 7120 (KNC)
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis


Theoretical Peak Floating Point Operations per Watt, Single Precision
GFLOP/sec per Watt
End of Year
2008
2010
2012
2014
2016
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
GTX Titan
Tesla K40
GTX Titan X
NVIDIA Titan X
HD 3870
8800 GTS
GTX 280
GTX 285
GTX 580
GTX 580
GTX 680
HD 8970
FirePro W9100
FirePro S9150
Xeon Phi 7290 (KNL)
Xeon Phi 7120 (KNC)
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis

CPU

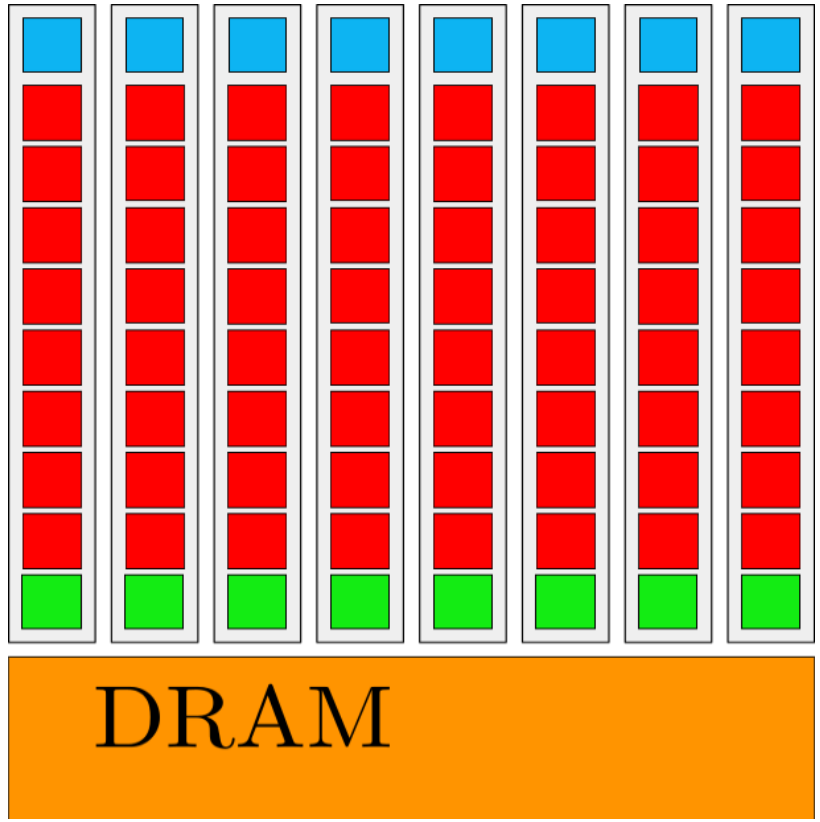


CPU: latency oriented design

- Multilevel and Large Caches
  - ➡Convert long latency memory access to short latency cache latency
- Branch prediction
  - ➡To reduce latency in branching
- Instruction level parallelism (ILP)
- Powerful ALU
  - ➡Reduced operation latency
- Memory management
- Large control part

GPU

- SIMT/SIMD (Single instruction Multiple Thread/Data) architecture
- SMX (Streaming Multi Processors) to execute kernels
- Thread level parallelism
  - ➡Massive threading to hide the latency
- Limited caching
  - ➡To boost memory throughput
- Limited control
- No branch prediction, but branch predication



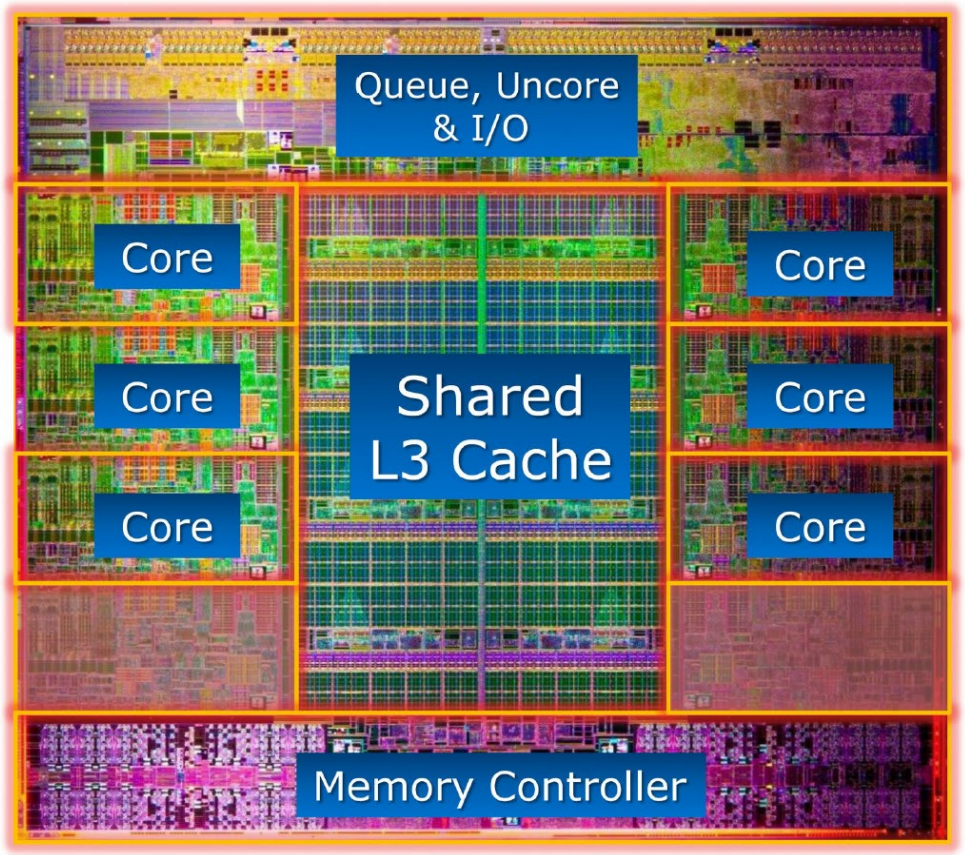
GPU: throughput oriented design

CPU vs GPU

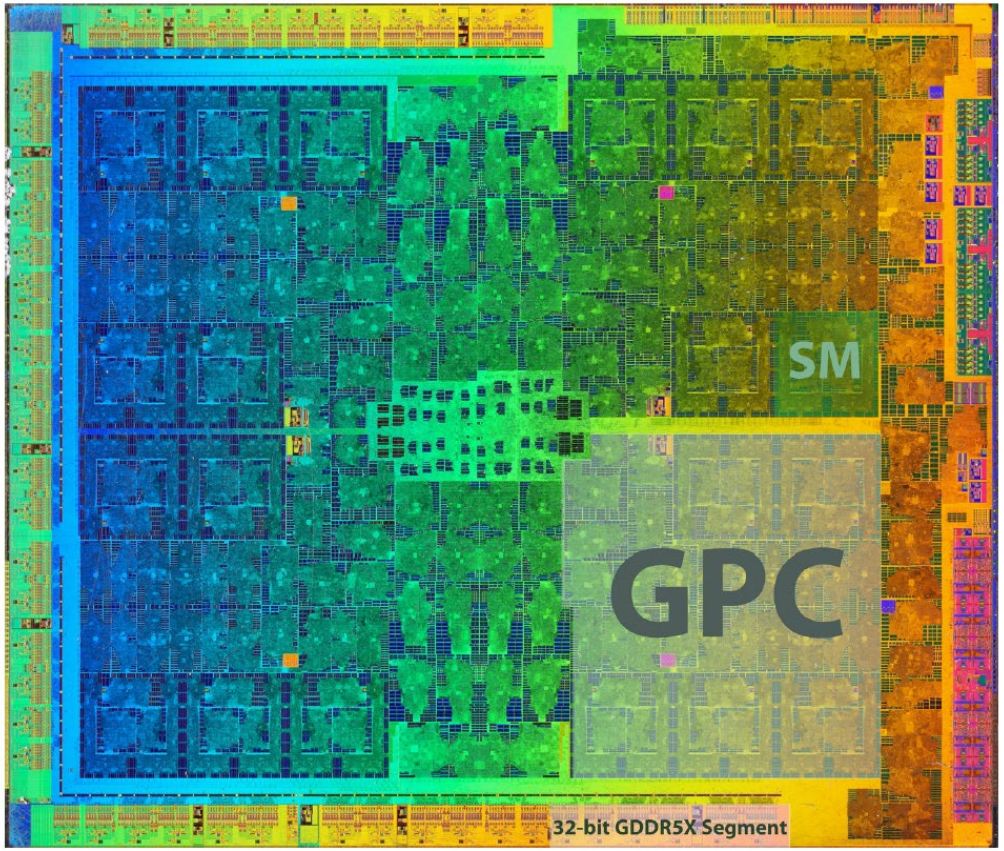
- + Large main memory
- + Fast clock rate
- + Large caches
- + Branch prediction
- + Powerful ALU
- Relatively low memory bandwidth
- Cache misses costly
- Low performance per watt

- + High bandwidth main memory
- + Latency tolerant (parallelism)
- + More compute resources
- + High performance per watt
- Limited memory capacity
- Low per-thread performance
- Extension card

CPU vs GPU



http://images.anandtech.com/reviews/cpu/intel/SNBE/Core_I7_LGA_2011_Die.jpg



http://wccftech.com/nvidia-gtx-1080-gp104-die-shot/

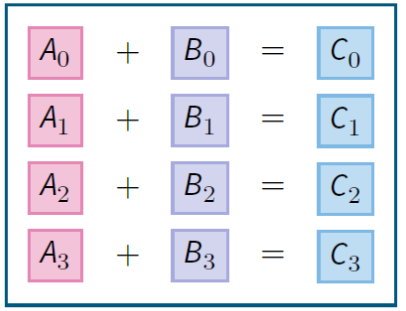
CPU vs GPU

	Intel Core E7-8890 v3	**GeForce GTX 1080**
Core count	18 cores / 36 threads	20 SMs / 2560 cores
Frequency	2.5 GHz	1.6 GHz
Peak Compute Performance	1.8 GFLOPs	8873 GFLOPs
Memory bandwidth	Max. 102 GB/s	320 GB/s
Memory capacity	Max. 1.54 TB	8 GB
Technology	22 nm	16 nm
Die size	662 mm^2	314 mm^2
Transistor count	5.6 billion	7.2 billion
Model	Minimize latency	Hide latency through parallelism

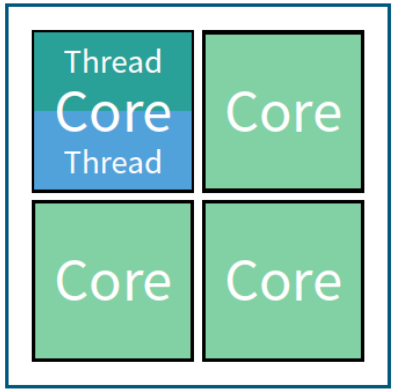
SIMT

- Standard CPU : Scalar processors
- SIMD CPU: vector processors
- Simultaneous threads in multicore processors
- SIMT (Single Instruction Multiple Threads)
  - ➡ CPU core ~ GPU multiprocessor (SMX)
  - ➡ Working unit: a set of threads (32, a warp)
  - ➡ Fast switching of threads

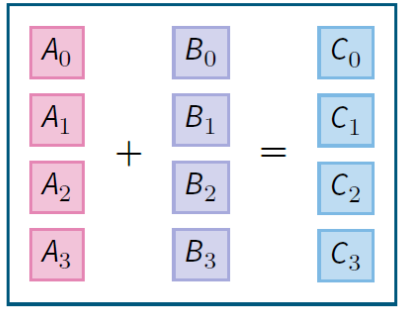
Scalar



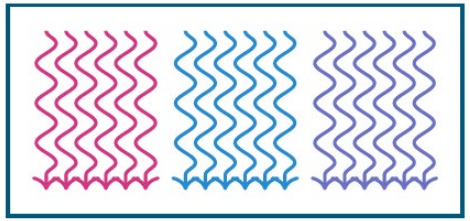
SMT



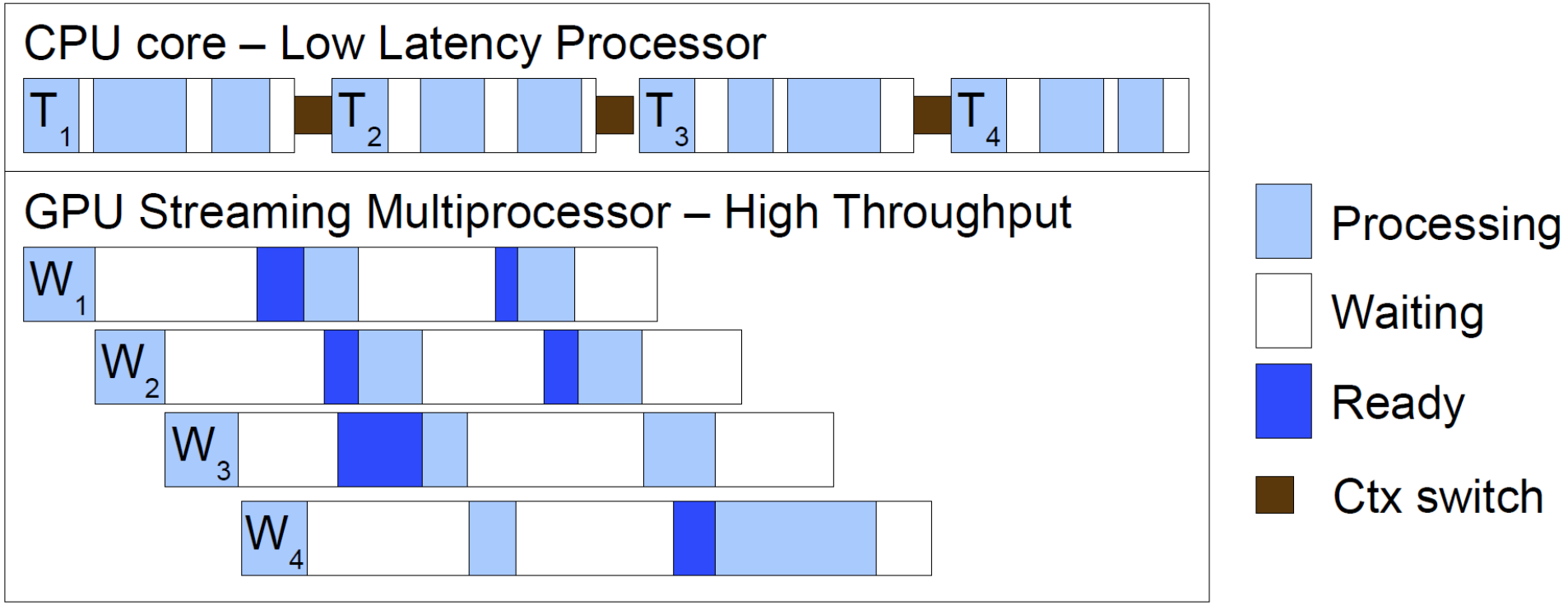
Vector



SIMT



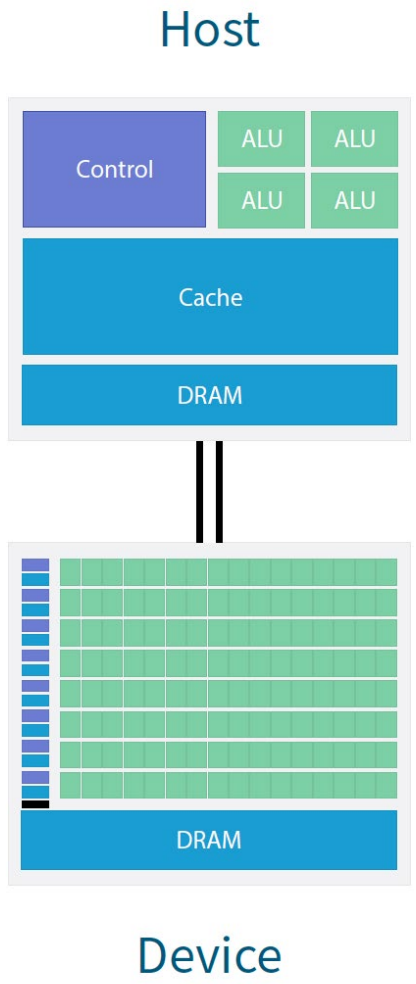
CPU core vs GPU SMX



- The latency in a SMX is hidden thanks to very deep pipelines
- The multithreading in a single CPU core is based on context switching

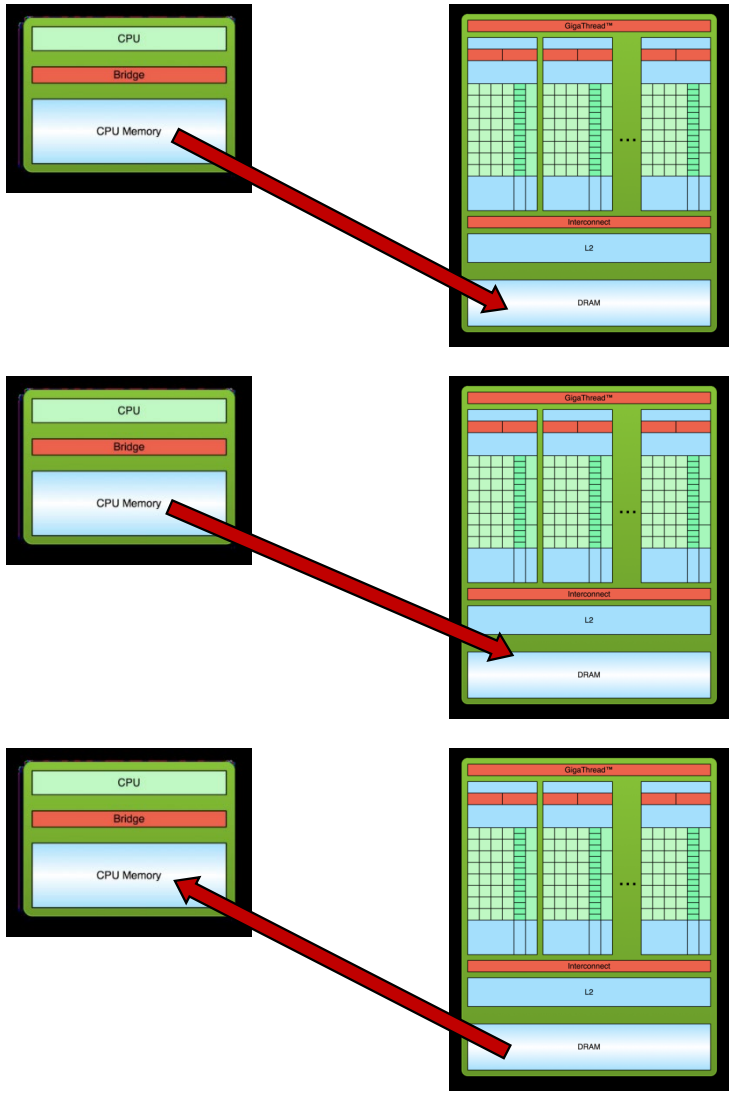
GPU+CPU

- The winning application uses both CPU and GPU
- CPUs for sequential parts
  - ➡ can be 10X faster than GPU for sequential code
- GPUs for parallel part where throughput wins
  - ➡ can be 100X faster than CPU for parallel code
- The Host-Device connection is done with PCIe-gen3 (16 GB/s) or NVLINK (80 GB/s)
  - ➡Relativelly slow
  - ➡Do as little as possible
- The bandwidth between GPU and video memory is HBM2 (720 GB/s in P100, 900 GB/s in V100)



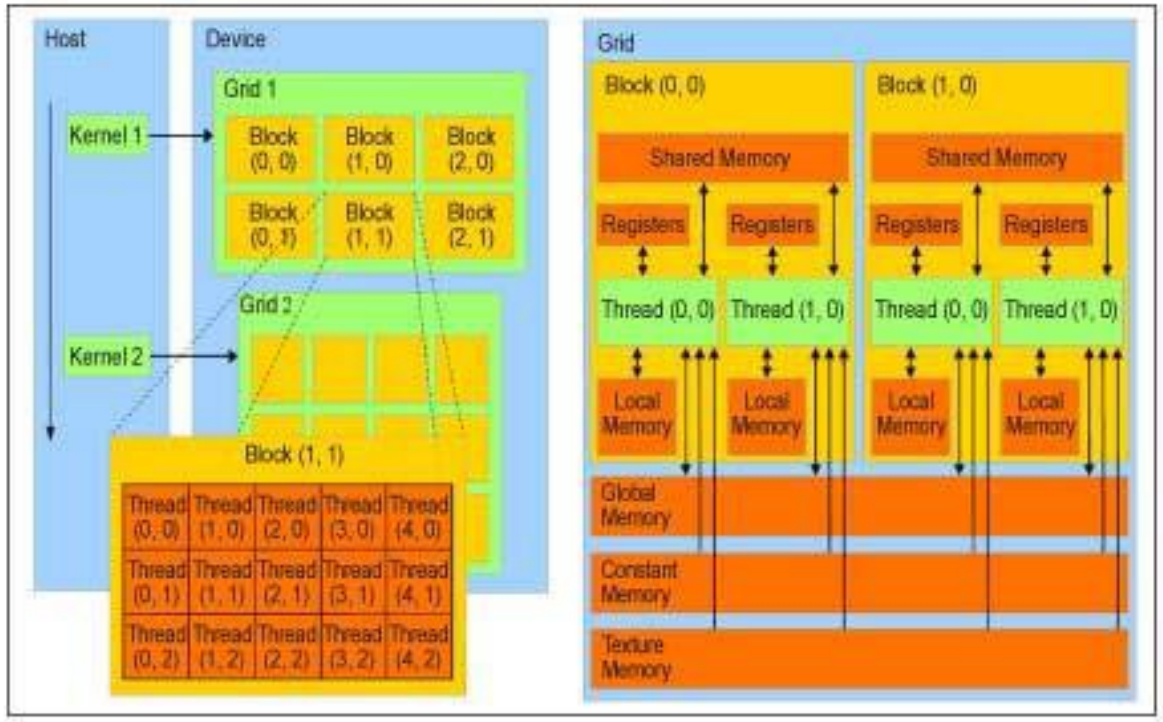
CUDA model

- What is CUDA?
- It is a set of C/C++ extensions to enable the GPGPU computing on NVIDIA GPUs
- Dedicated APIs allow to control almost all the functions of the graphics processor
- Three steps:
  - 1) copy data from Host to Device
  - 2) copy Kernel and execute
  - 3) copy back results
- We will come back in few slides…

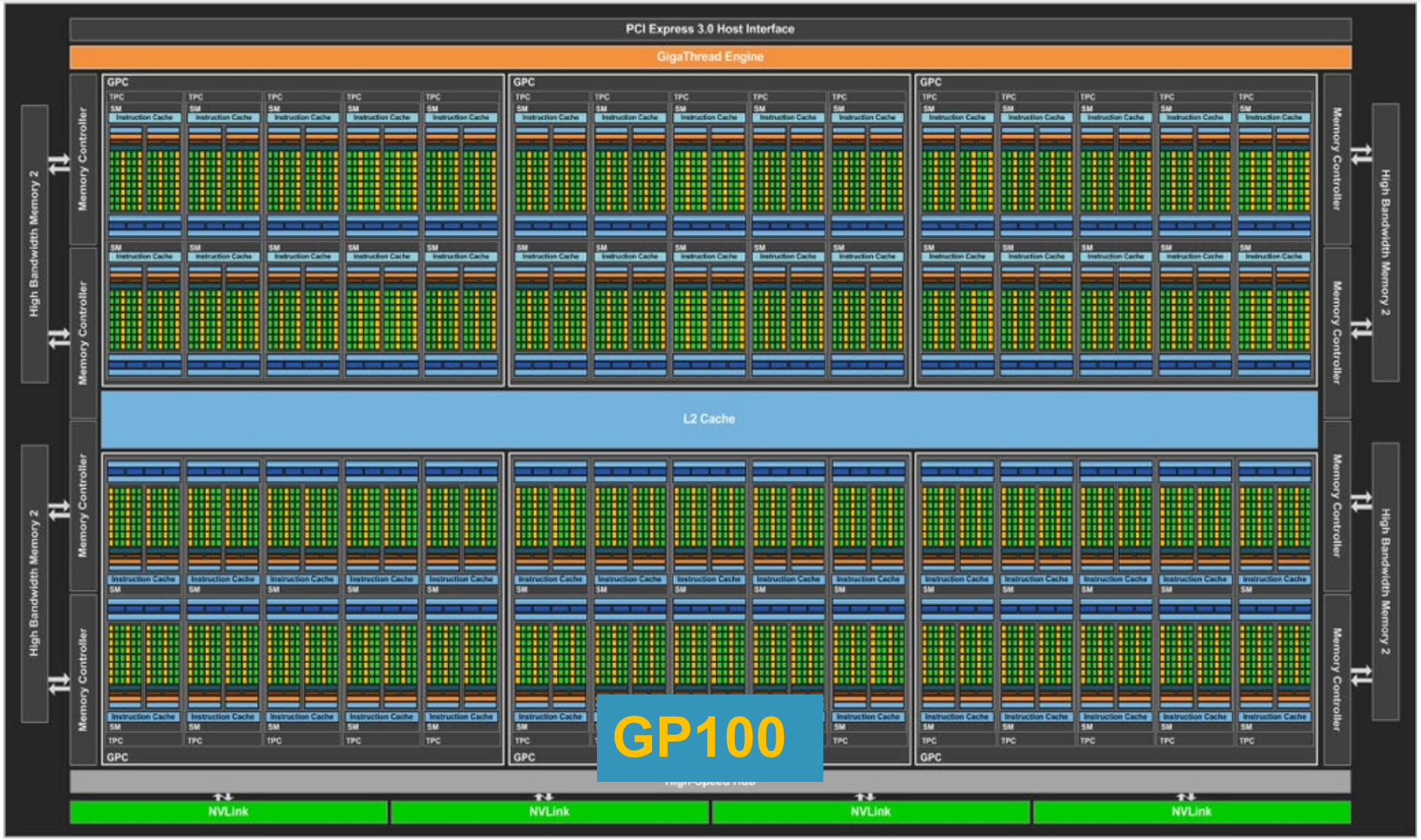


Grid, blocks and threads

- The computing resources are logically (and physically) grouped in a flexible parallel model of computation:
  - ➡1D,2D and 3D grid
  - ➡With 1D, 2D and 3D blocks
  - ➡With 1D, 2D and 3D threads
- Only threads can communicate and synchronize in a block
- Threads in different blocks do not interact, threads in same block execute same instruction at the same time
- The “shape” of the system is decided at kernel launch time

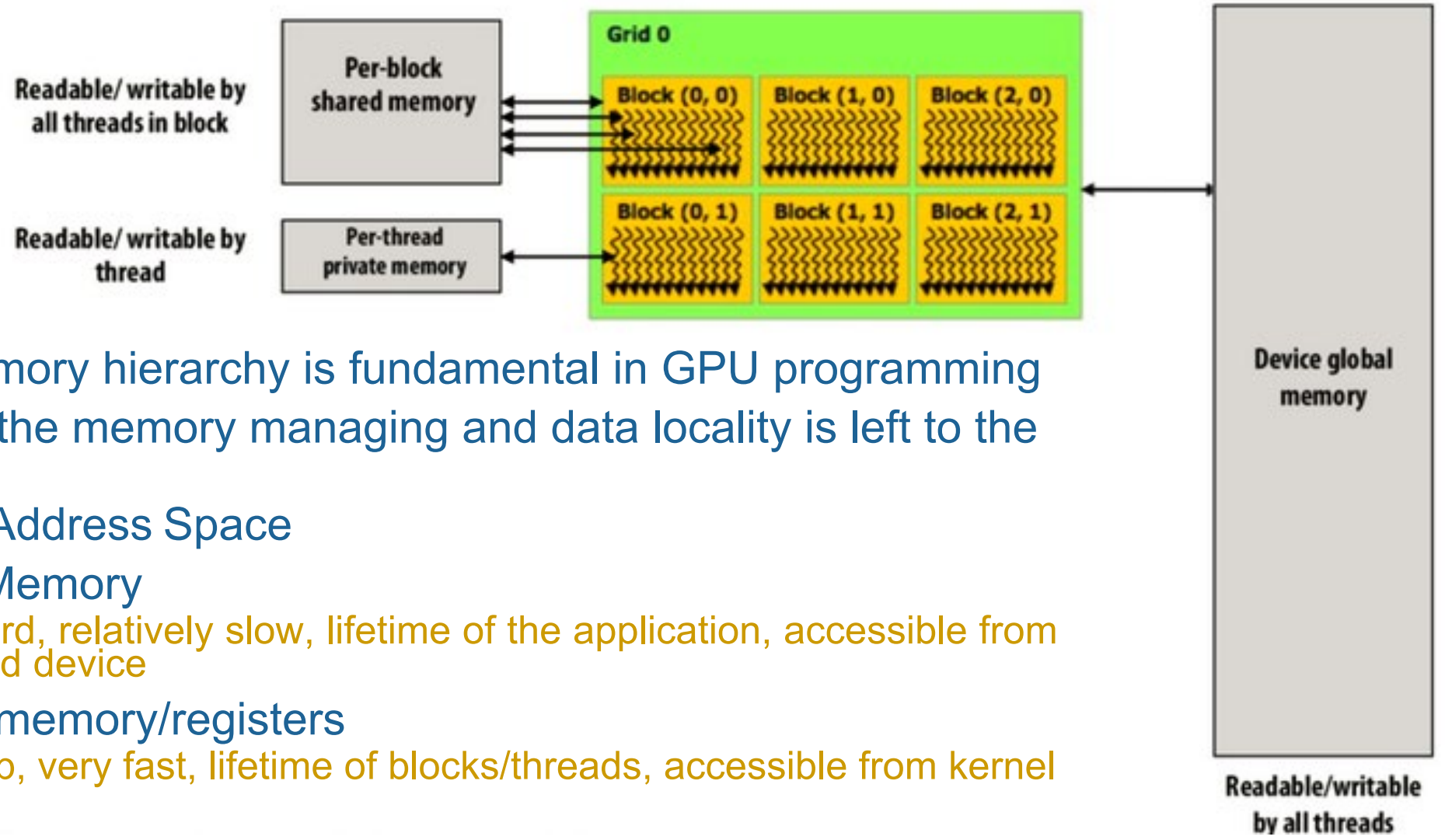


GPU structure



Multiprocessor

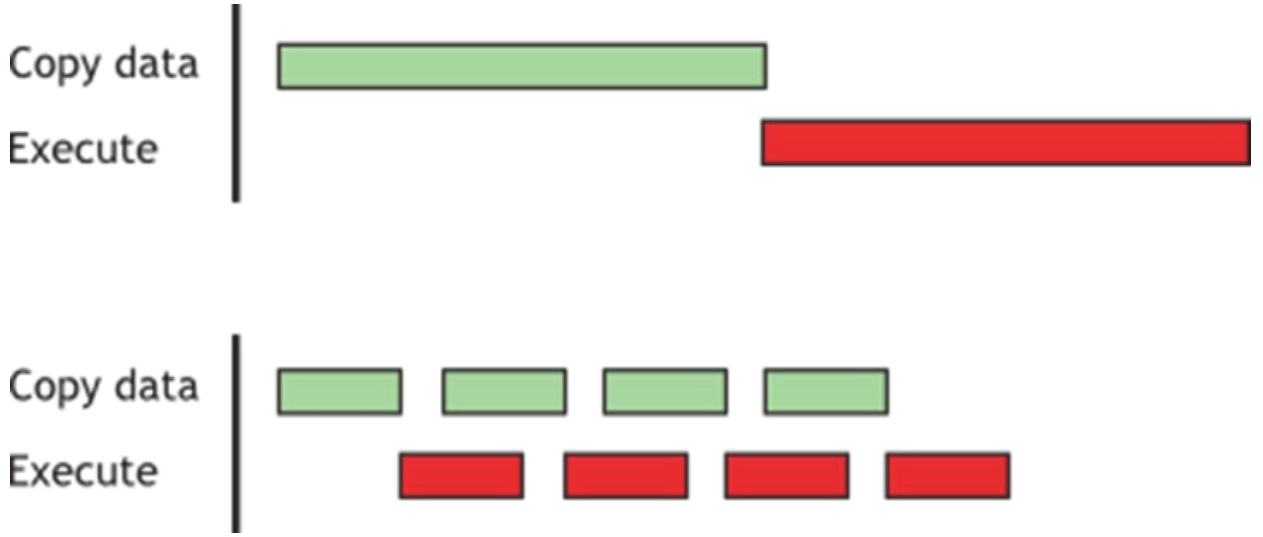
Memory



- The memory hierarchy is fundamental in GPU programming
- Most of the memory managing and data locality is left to the user
- Unified Address Space
- Global Memory
  - ➞ On board, relatively slow, lifetime of the application, accessible from host and device
- Shared memory/registers
  - ➞ On Chip, very fast, lifetime of blocks/threads, accessible from kernel only

Asynchronicity

- Problem: Memory transfer is comparably slow
- Solution: Do something else in meantime (computation)!
- Overlap tasks
  - ➞ Copy and compute engines run separately (streams)
  - ➞ GPU needs to be fed: Schedule many computations
  - ➞ CPU can do other work while GPU computes; synchronization



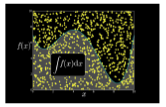
How to program GPU?

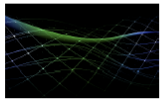
- CUDA is the “best” way to program NVIDIA GPU at “low level”
- If your code is almost CPU or if you need to accelerate dedicated functions, you could consider to use
  - ➡ Directives
    - OpenMP, OpenACC, …
  - ➡ Libraries
    - Thrust, ArrayFire,…
- OpenCL is a framework equivalent to CUDA to program multiplatforms
  - ➡ GPU, CPU, DSP, FPGA,…
- C/C++ and Fortran are the “official” languages for CUDA.
  - ➡ Python and other languages are supported through wrapping and libraries

Numba

theano


cuFFT


cuRAND


CUDA Math


cuSPARSE


cuBLAS

Libraries: cuBLAS

- GPU-parallel linear algebra routines (152 routines)
- Single, double, complex data types
- Possibility to use multiple GPUs
- Example (among 152 routines):
  - Saxpy: given two vectors x[10] and y[10] compute y[i]=a*x[i]+y[i]

```
int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y
cublasInit();
float * d_x, * d_y;
cudaMalloc((void **)&d_x, n * sizeof(x[0]);
cudaMalloc((void **)&d_y, n * sizeof(y[0]);
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);
cublasSaxpy(n, a, d_x, 1, d_y, 1);
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
cublasShutdown();
```

https://docs.nvidia.com/cuda/cublas/index.html
https://developer.nvidia.com/cublas

Libraries: Thrust

- Template library
- Data parallel primitives (scan(), sort(), reduce(), …)
- Comes when you install CUDA for free

```
int a = 42;
int n = 10;
thrust::host_vector<float> x(n), y(n);
// fill x, y
thrust::device_vector d_x = x, d_y = y;
using namespace thrust::placeholders;
thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_y.begin(), a * _1 + _2);
x = d_x;
```

Directives: OpenMP, OpenACC

- The directive is the best transparent way to use GPU
- You must only «annotate» the part of the code you want to parallelize

```
#pragma acc loop
for (int i = 0; i < 100; i++) {};
```

- Pro
  - ➡ Portability
  - ➡ Easy to program
- Cons
  - ➡ Not all the raw GPU power available
  - ➡ Harder to debug
  - ➡ Easy to program wrong
- OpenACC is more focused on GPU, while OpenMP is for multi-computers (but still usable with GPU)

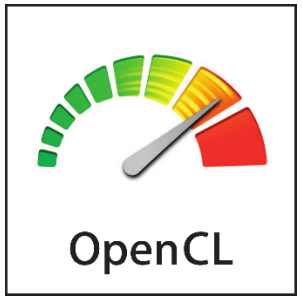
```
void saxpy_acc(int n, float a, float * x, float * y) {
#pragma acc kernels
for (int i = 0; i < n; i++) y[i] = a * x[i] + y[i];
}
…
int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y
saxpy_acc(n, a, x, y);
```

Direct Programming: CUDA vs OpenCL

- Two alternatives way to direct program GPU
  - CUDA (2007)
  - OpenCL (2009)

- CUDA
  - NVIDIA GPU's Platform
  - Platform: Drivers, programming language (CUDA C/C++), API, compiler, debuggers, profilers, …
  - Only NVIDIA GPUs
  - Compilation with dedicated compiler (nvcc)
  - CUDA fortran

- OpenCL
  - Consortium: Open Computing Language by Khronos Group (Apple, IBM, AMD, NVIDIA, …)
  - Programming language (OpenCL C/C++), API, and compiler
  - Targets CPUs, GPUs, FPGAs, and other many-core machines
  - Fully open source
  - Different compilers available

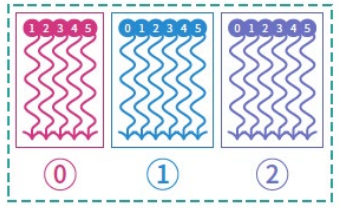


CUDA C/C++

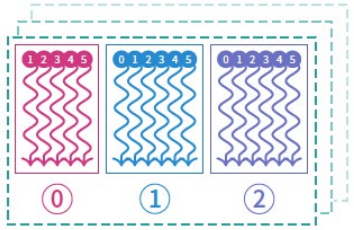
- Threads

- Blocks

- Grid



- In 3D



- The function running on GPU is called Kernel
  - ➡ Access own ID by global variables threadIdx.x, blockIdx.y, …
  - ➡ Execution order non-deterministic!
  - ➡ Only threads in one warp (32 threads of block) can communicate quickly
  - ➡ A kernel can call other kernels to run on the same GPU (more than one kernel can be executed in the GPU at the same time)
  - ➡ The kernels exploit the SIMD/SIMT structure of the GPU

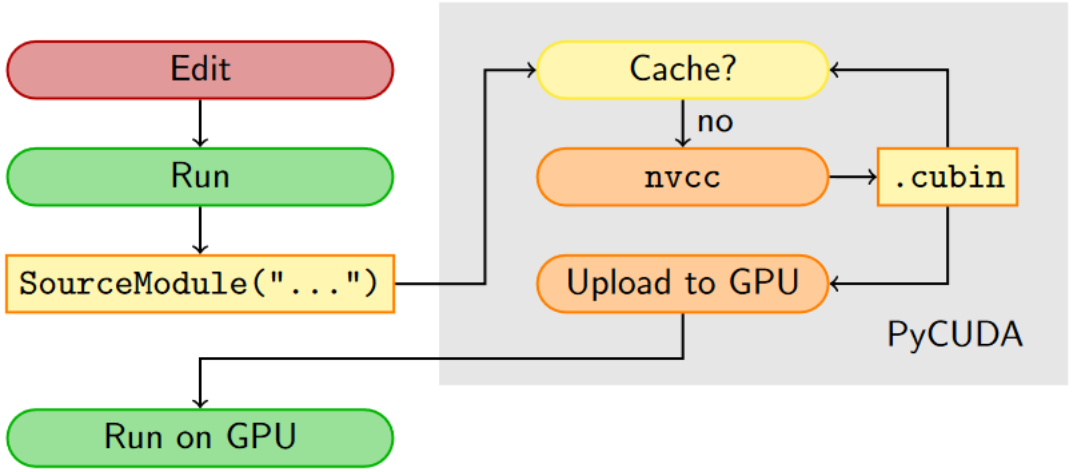
Example

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = a * x[i] + y[i];
}
int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y
cudaMallocManaged(&x, n * sizeof(float));
cudaMallocManaged(&y, n * sizeof(float));
saxpy_cuda<<<2, 5>>>(n, a, x, y);
cudaDeviceSynchronize();
```

- First the data must be copied on the device from the host
- Then the kernel is launched
- The architecture of threads and blocks is decided at run time

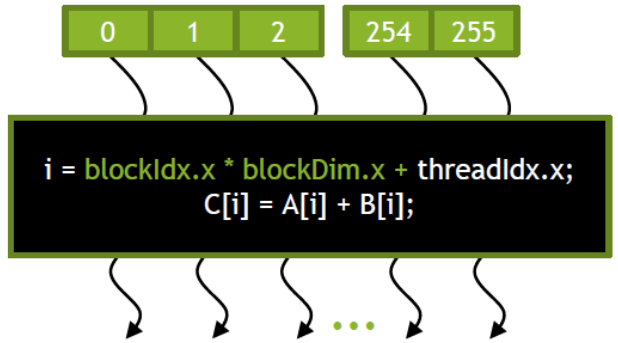
PyCUDA

- GPUs are everything that scripting languages are not.
  - ➡Highly parallel
  - ➡Very architecture-sensitive
  - ➡Built for maximum throughput
- In this sense GPU and Python can complement each other
- “Alternative” to write the code
  - ➡Scripting for ‘brains’
  - ➡GPUs for ‘inner loops

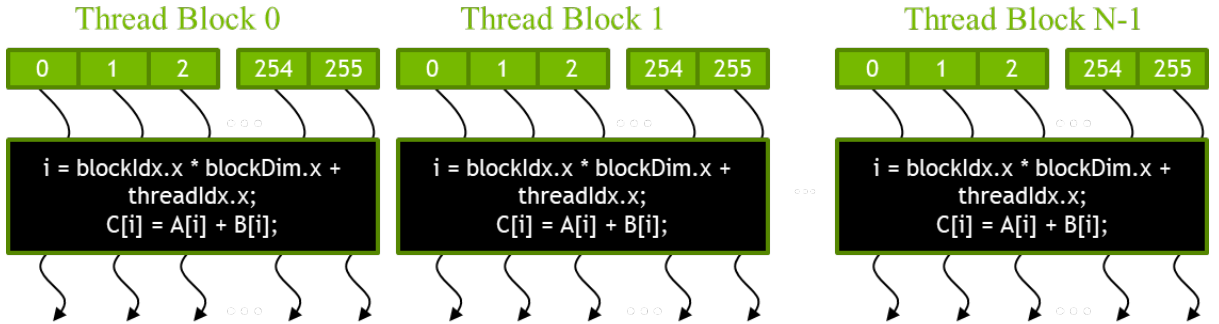


CUDA threads and blocks

- A CUDA kernel is executed by a grid of threads
  - All threads in a grid run the same code (SIMD or better SPMD (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions

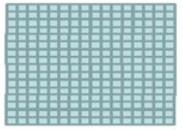


- Organize threads in blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

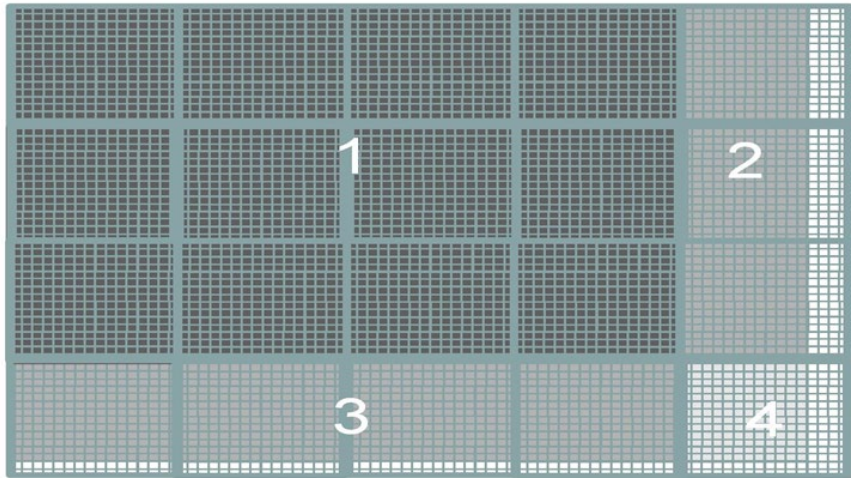


GPU for images

- Assume to have a picture of 62x76 pixels
- You want to increase the «luminosity» of each pixel by a factor of 2



16×16 block



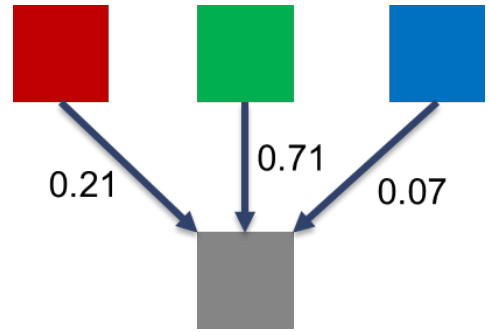

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                              int height, int width)
{

  // Calculate the row # of the d_Pin and d_Pout element
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < height) && (Col < width)) {
    d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
  }
```

```
// assume that the picture is m×n,
// m pixels in y dimension and n pixels in x dimension
// input d_Pin has been allocated on and copied to device
// output d_Pout has been allocated on device
…
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
…
```

RGB to Grayscale conversion

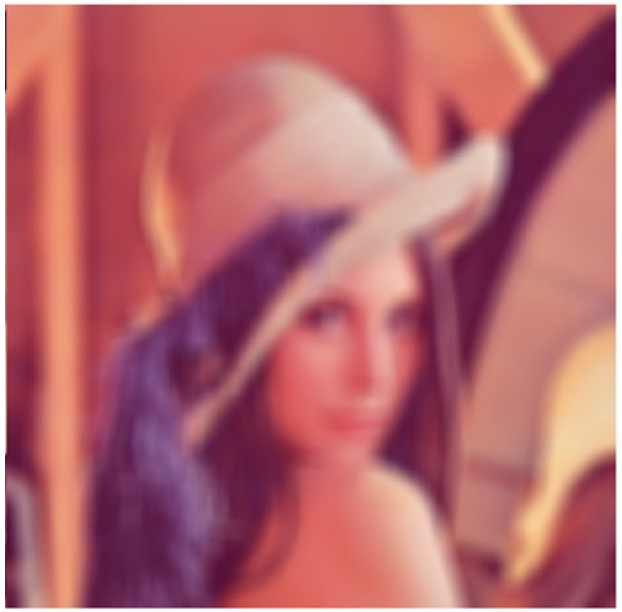
- Assume you want to convert an image in which you have the rgb code for each pixel in greyscale
  - Rgb is a standard to define the quantity of red, green and blue in each pixel
  - A greyscale image is an image in which the value of each pixel carries only intensity information.

- Conversion formula: For each pixel (I, J) do: grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b



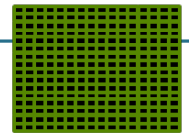

```
                    // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                                        unsigned char * rgbImage,
                                int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {
    // get 1D coordinate for the grayscale image
    int grayOffset = y*width + x;
    // one can think of the RGB image having
    // CHANNEL times columns than the gray scale image
    int rgbOffset = grayOffset*CHANNELS;
    unsigned char r =  rgbImage[rgbOffset      ]; // red value for pixel
    unsigned char g = rgbImage[rgbOffset + 2]; // green value for pixel
    unsigned char b = rgbImage[rgbOffset + 3]; // blue value for pixel
    // perform the rescaling and store it
    // We multiply by floating point constants
    grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

- Assume you want to Blur an image

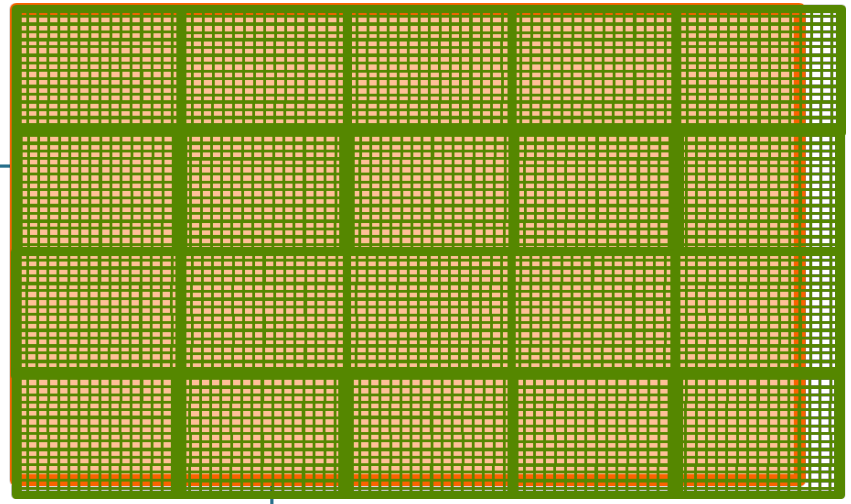


- Defines a Blur box
  - ➡ The blurring is a kind of «average» of the pixel in the blurring box

```
__global__
 void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
     int Col  = blockIdx.x * blockDim.x + threadIdx.x;
     int Row  = blockIdx.y * blockDim.y + threadIdx.y;

     if (Col < w && Row < h) {
         int pixVal = 0;
         int pixels = 0;

         // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
         for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
             for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

                 int curRow = Row + blurRow;
                 int curCol = Col + blurCol;
                 // Verify we have a valid image pixel
                 if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                     pixVal += in[curRow * w + curCol];
                     pixels++; // Keep track of number of pixels in the accumulated total
                 }
             }
         }

         // Write our new pixel value out
         out[Row * w + Col] = (unsigned char)(pixVal / pixels);
     }
 }
```



Pixels processed by a thread block

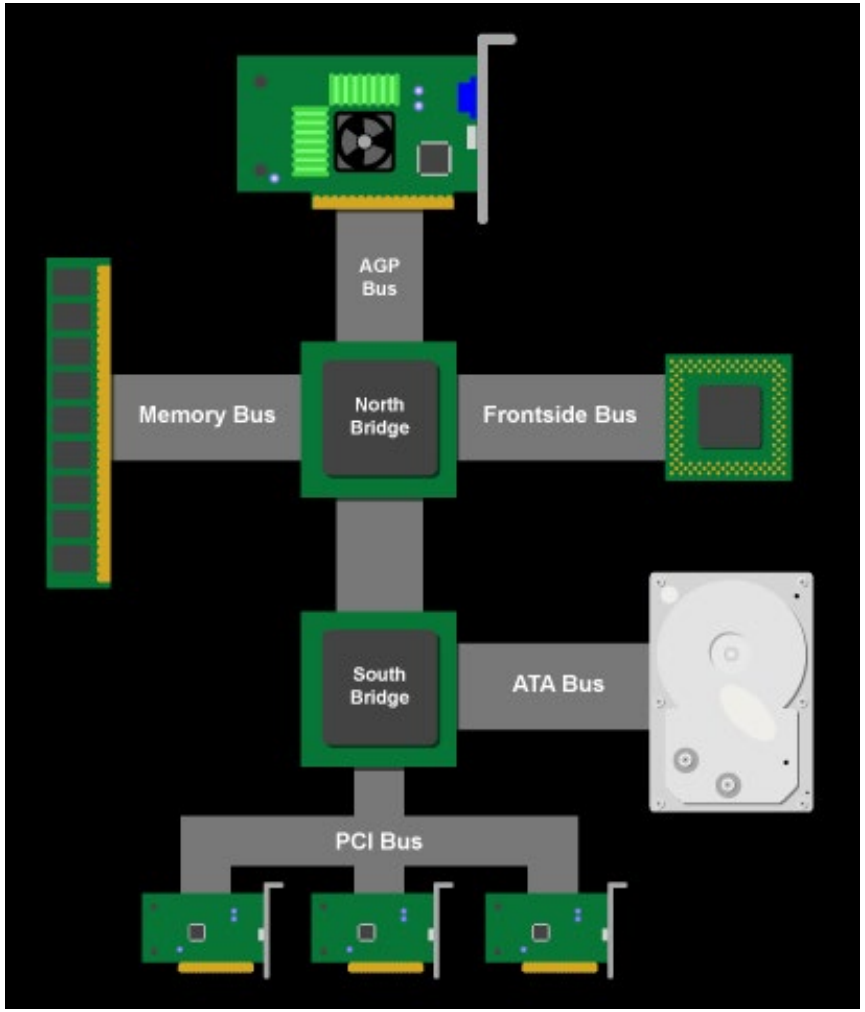


Recap: CUDA program structure

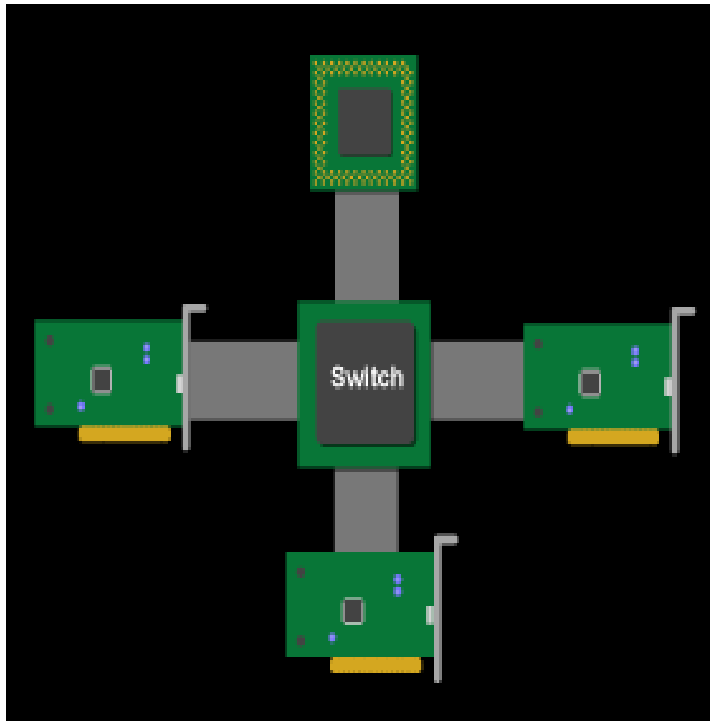
- Global variables declaration
- Function prototypes
  - `__global__ void kernelOne(…)`
- Main ()
  - allocate memory space on the device transfer data from host to device
  - execution configuration setup
  - kernel call – `kernelOne<<<execution configuration>>>( args… );`
  - transfer results from device to host
  - optional: compare against golden (host computed) solution
- Kernel – `void kernelOne(type args,…)`
  - variables declaration - `__local__, __shared__`
    - automatic variables transparently assigned to registers or local memory
  - `syncthreads()...`

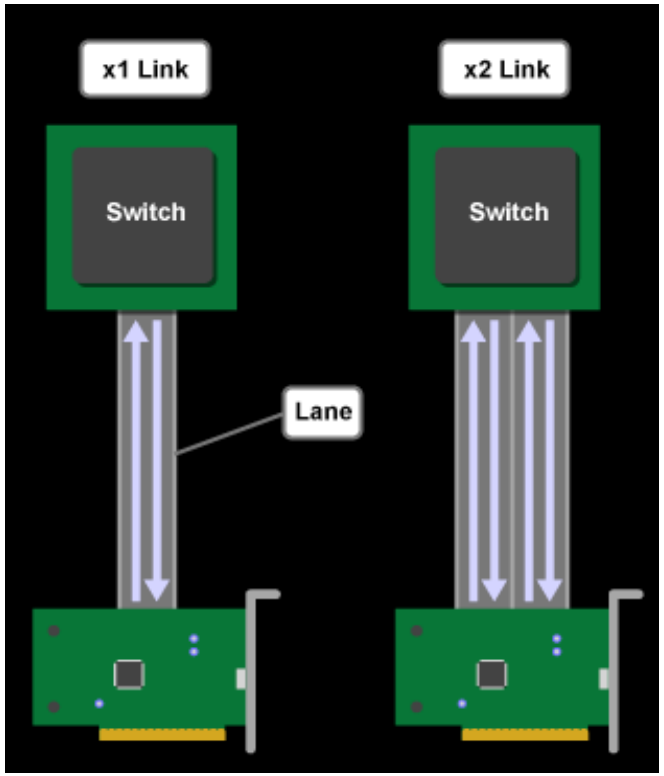
Old PC architecture

- Northbridge connects 3 components that must communicate at high speed
  - CPU, DRAM, video
  - Video also needs to have 1st-class access to DRAM
  - Previous NVIDIA cards are connected to AGP, up to 2 GB/s transfers
- Southbridge serves as a concentrator for slower I/O devices
- The PCI bus was connected to the Southbridge
  - Originally 33 MHz, 32-bit wide, 132 MB/second peak transfer rate
  - More recently 66 MHz, 64-bit, 528 MB/second peak
  - Upstream bandwidth remain slow for device (~256 MB/s peak)
- Shared bus with arbitration



PCI express

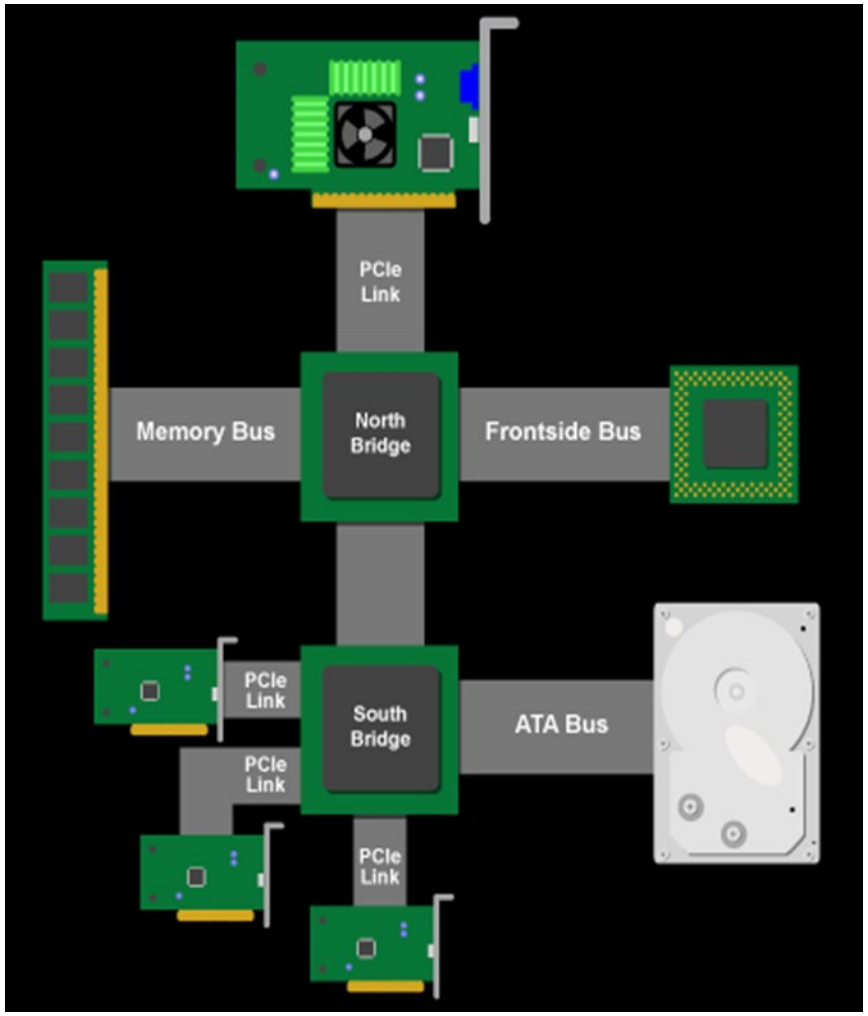




- PCIe is a serial link on multiple lanes
  - Switched point-to-point connection
  - Each card has a dedicated “link” to the central switch, no bus arbitration
  - Each lane is 1-bit wide (4 wires, each 2-wire pair can transmit 8Gb/s in one direction) (PCIe gen3)
    - Upstream and downstream now simultaneous and symmetric
  - Each Link can combine 1, 2, 4, 8, 12, 16 lanes- x1, x2, etc.
  - Each byte data is 8b/10b encoded into 10 bits with equal number of 1’s and 0’s; net data rate 2 Gb/s per lane each way
  - Thus, the net data rates are 985 MB/s (x1) 2 GB/s (x2), 4GB/s (x4),.., each way (PCIe gen3)

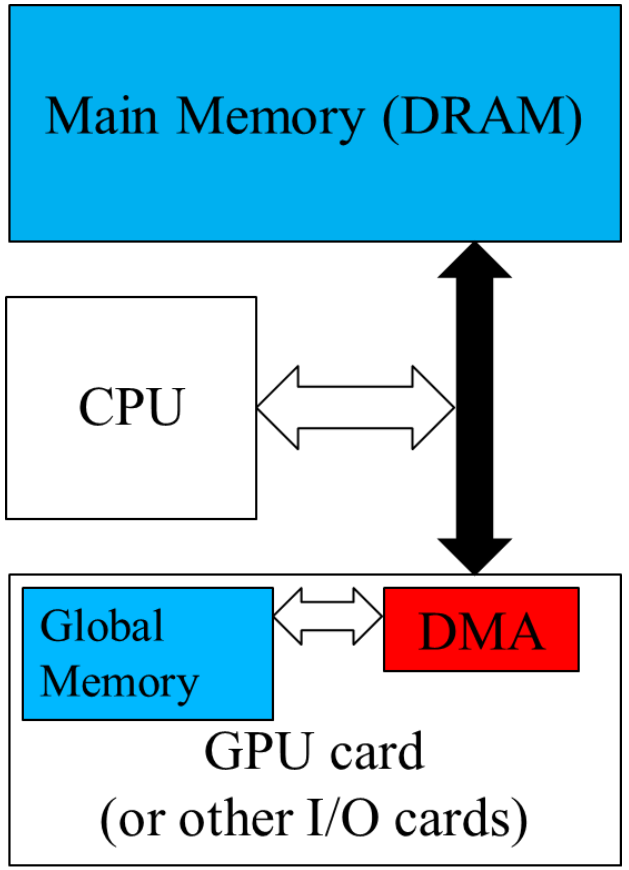
PCI express PC architecture

- Both North Bridge and South Bridge are PCIe switch (until PCIe gen2)
- In present architecture both north and south bridge are integrated in the main processor
- Other fast bus are alteranative to PCIe (i.e. NVLINK)



Bandwidth problem

- In any case the bandwidth (and the latency) of the data transfer between Host and Device is the main bottleneck of the GPU computing
  - Especially true for massively parallel systems processing massive amount of data
  - Tricks like buffering, reordering and caching can temporarily defy the rules in some cases
  - Streaming data transfer and processing is a winning strategy to hide latency
  - Some hardware strategy to mitigate the data transfer problem (DMA)
- But remain still valid the concept that the GPU is done mainly for computation



License

- Part of the material used for this presentation comes from the «GPU Teaching KIT» licensed by NVIDIA and University of Illinois, and has been used and modified according to Creative Commons attribution not-commercial 4.0 (http://creativecommons.org/licenses/by-nc/4.0/legalcode)