

Introduction to GPU computing

Outline

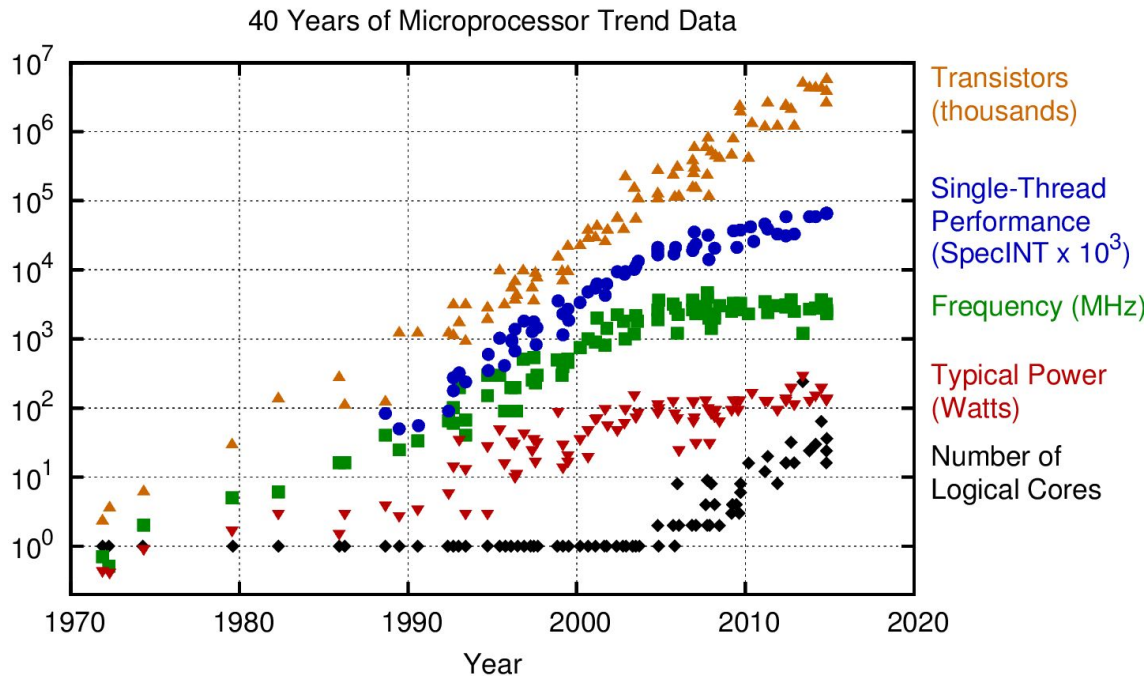
- Motivation: Trends in HPC
- Hardware Architecture
- Software Environment
- How to program on GPU

Motivation: Why running on GPUs?

- Massively parallel
- Hundreds of cores
- Many threads approach
- Programmable: CUDA
- Several software available:
 - Machine Learning: Caffe, Tensorflow, Theano
 - MD: Lammmps, Amber
 - Weather prediction: Cosmo

Motivation: Why running on GPUs?

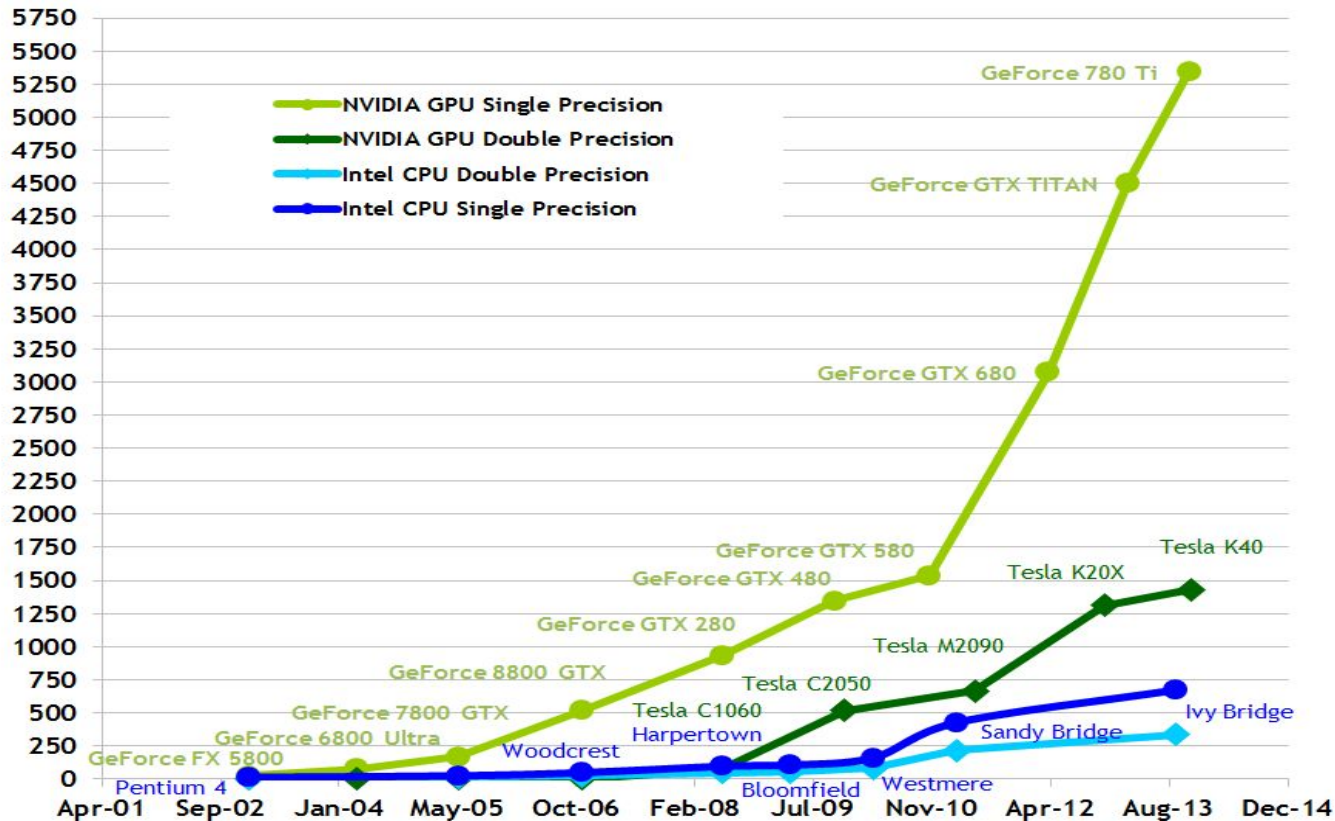
- Single-thread performance increased \sim linearly
- Number of logical cores is increasing \sim power law
- The frequency \sim constant



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Why GPU? Performance!

Theoretical GFLOP/s

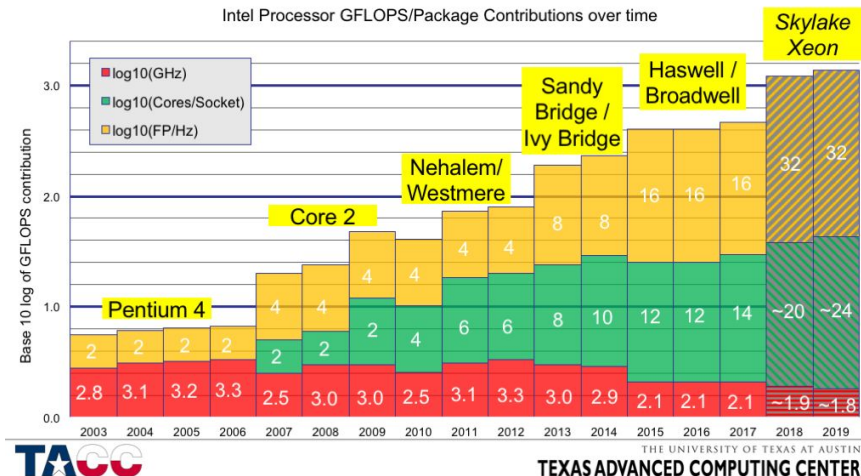


From Nvidia
programming
guide

V100 vs Intel procs

NVIDIA TESLA V100 SPECIFICATIONS

	Tesla V100 for NVLink	Tesla V100 for PCIe
PERFORMANCE with NVIDIA GPU Boost*		
DOUBLE-PRECISION	7.8 teraFLOPS	7 teraFLOPS
SINGLE-PRECISION	15.7 teraFLOPS	14 teraFLOPS
DEEP LEARNING	125 teraFLOPS	112 teraFLOPS
INTERCONNECT BANDWIDTH Bi-Directional	NVLINK 300 GB/s	PCIe 32 GB/s
MEMORY CoWoS Stacked HBM2		CAPACITY 32/16 GB HBM2
		BANDWIDTH 900 GB/s
POWER Max Consumption	300 WATTS	250 WATTS



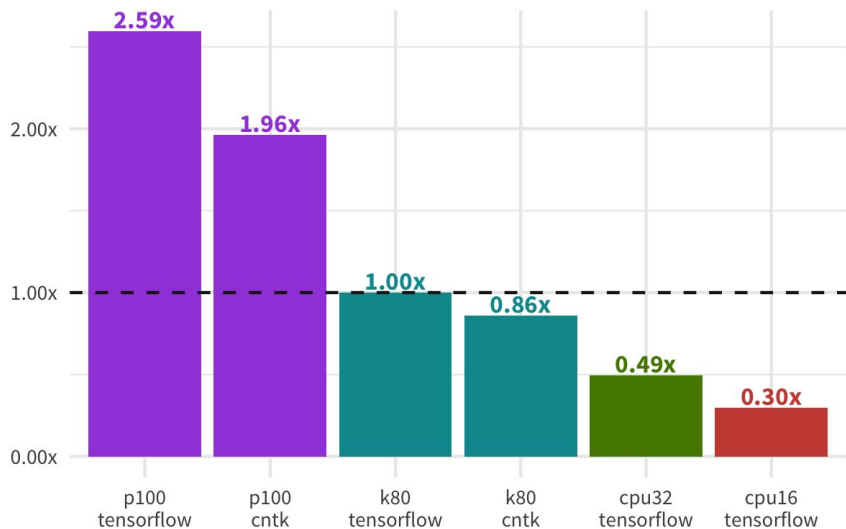


**GPUs are everywhere the Web goes.
Making full use of GPUs is essential for any modern computing platform.
But.. Traditionally the Web has not made effective use of GPUs.
That is changing...**

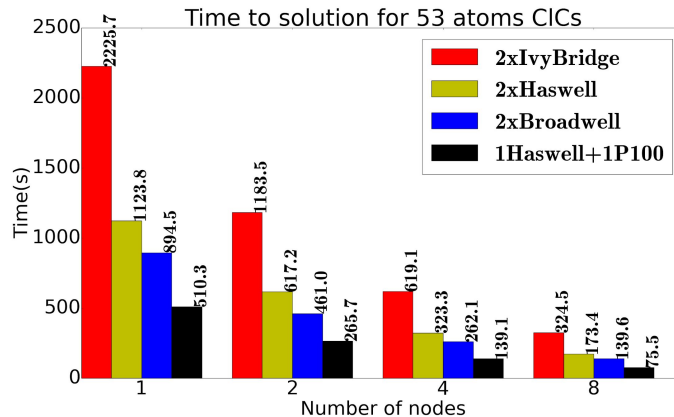
GPU advantage: Performance

Benchmarking Cost of Training MLP for MNIST

Total Model Training Cost, Relative to TensorFlow on K80 GPU



Max Woolf — minimaxir.com



Quantum-ESPRESSO +
SIRIUS

TAKE HOME MESSAGE

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation - **exactly what graphics rendering is about** - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.

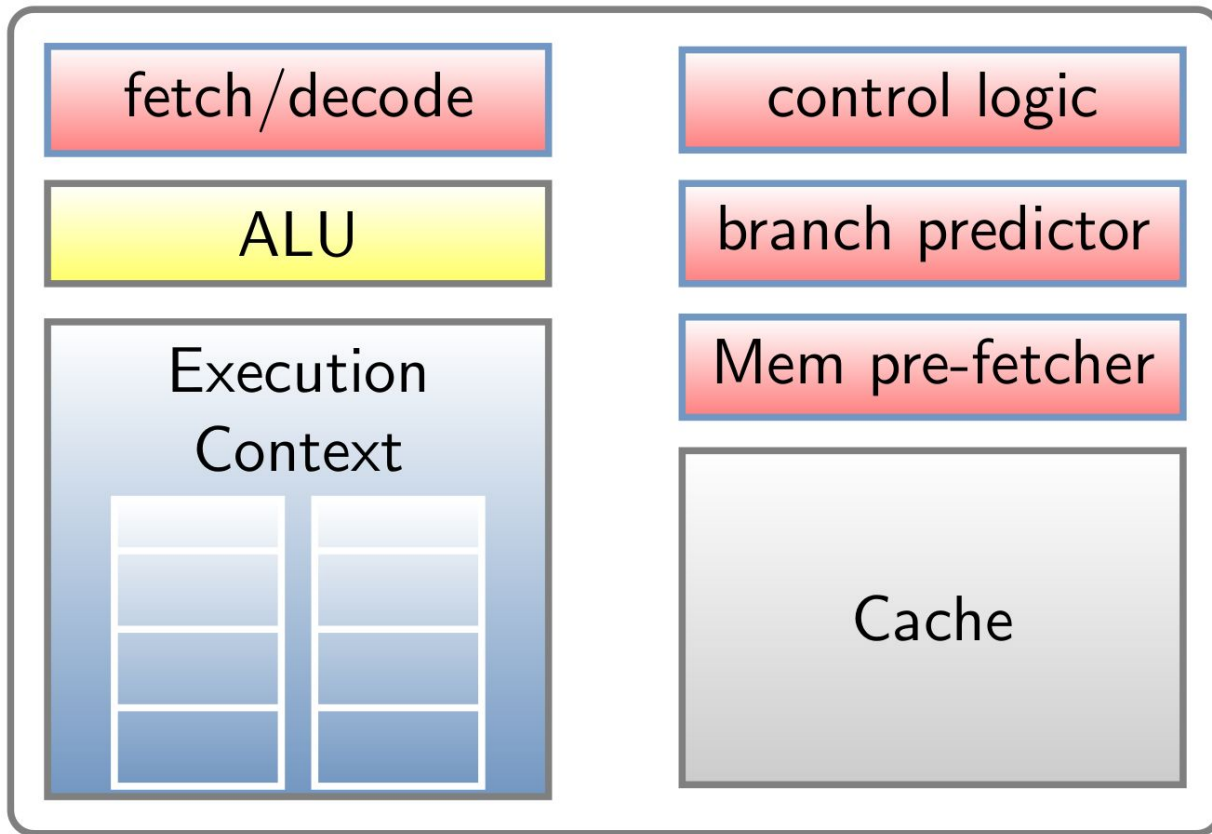
From NVidia programming guide:
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

GPU disadvantage?

- Architecture not as flexible as CPU.(But easier to get performance).
- Must rewrite algorithms and maintain software in GPU languages.
- Discrete GPUs attached to CPU via relatively slow PCIe
 - 16/32 GB/s bi-directional via PCIe
 - 300 GB/s via NVlink, but not very much available
- Limited memory (16-32GB).

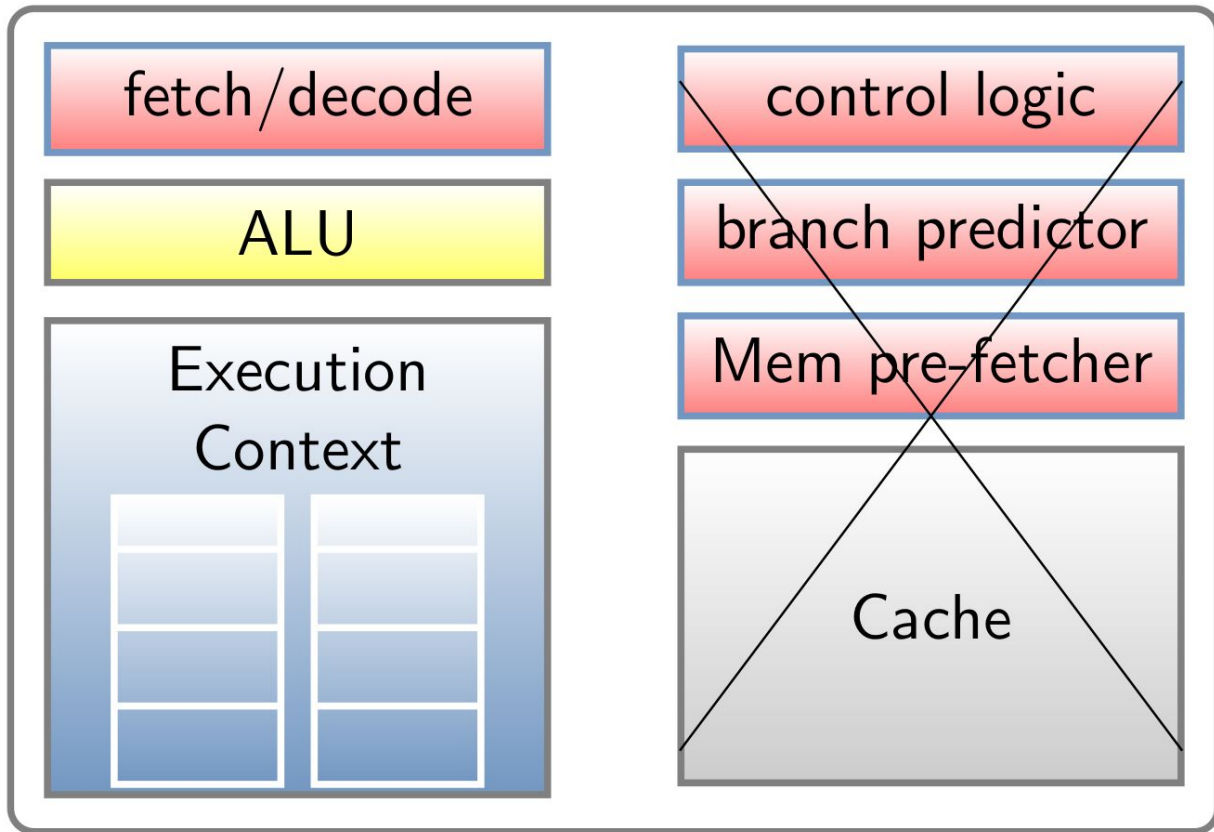
GPUs Hardware Architecture

From CPU to GPU



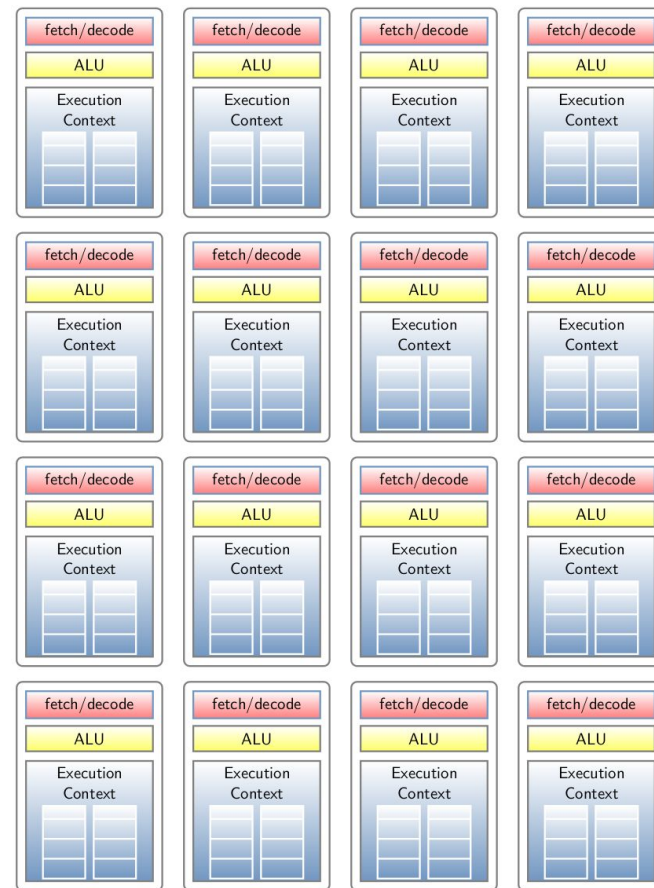
From CPU to GPU

- Remove the components that help single instruction stream to run faster.
- Maximize the chip area dedicated to computation



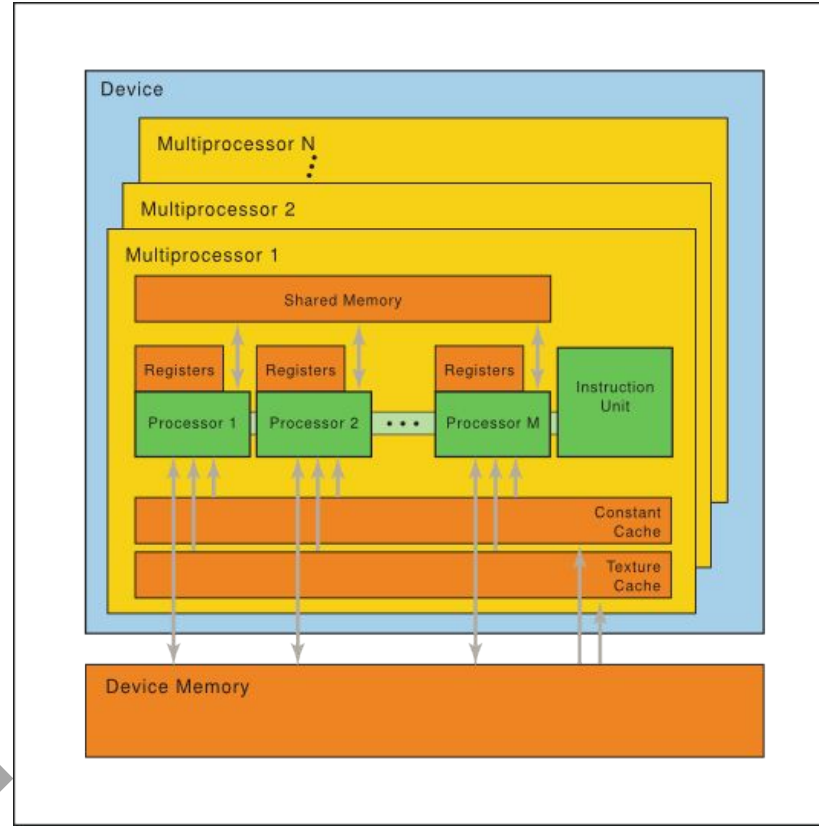
From CPU to GPU

- GPUs: many core devices.
- Designed for gaming industry: optimize the execution throughput of massive number of threads(1000+).
- Mask memory latency by interleaving threads execution.



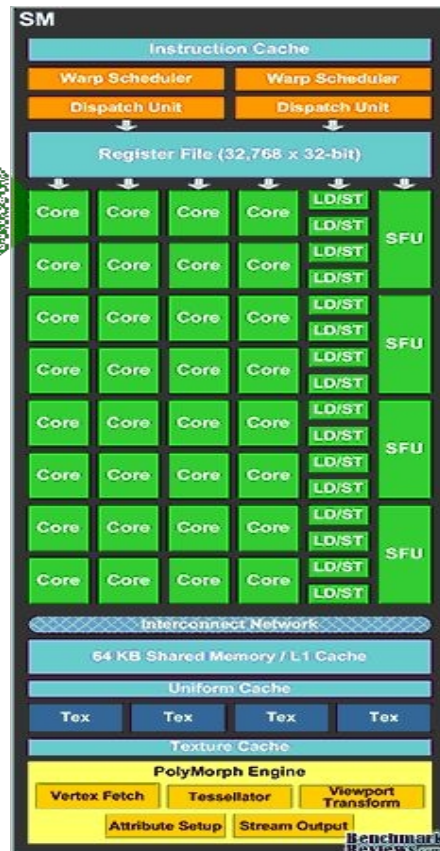
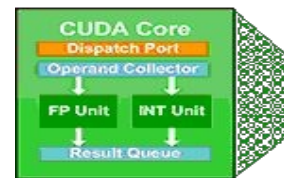
Streaming Multiprocessors Architecture simplified

- Many memory regions available each with different performance characteristics
- Must map the data set to the right memory type



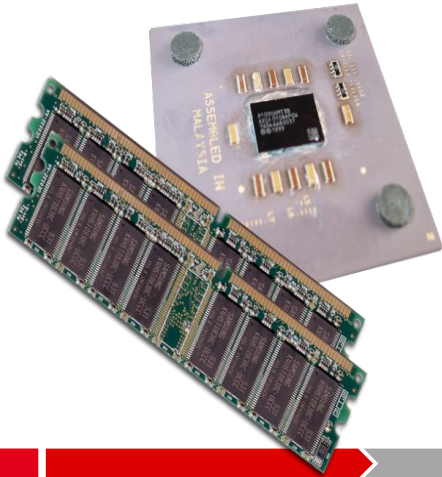
Streaming Multiprocessors

- Each GPU is comprised of one or more Streaming Multiprocessors(SM)
- Instructions are executed in multiple of 32 threads(warp).
- Each SM has a collection of cores, register, memory

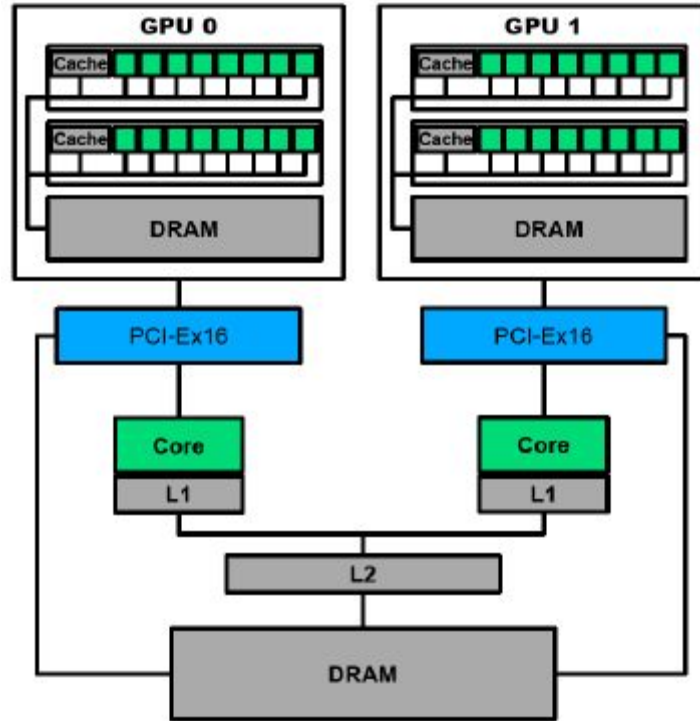


CUDA Jargon

- **Host** – The CPU and its memory (host memory)
- **Device** – The GPU and its memory (device memory)

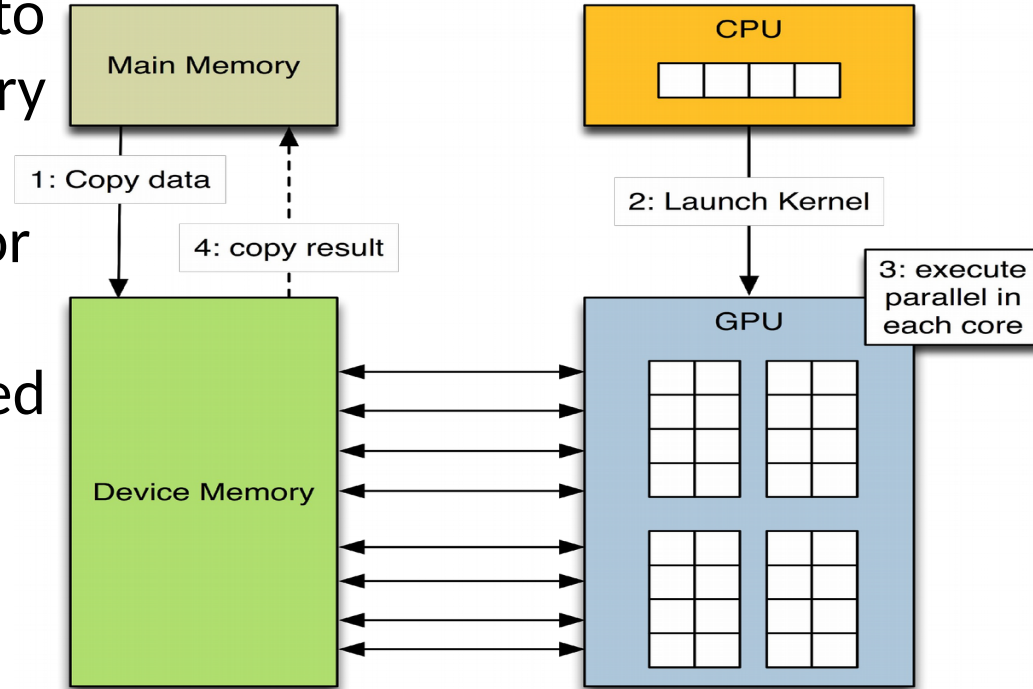


Accelerated node: Overview



Accelerated node:

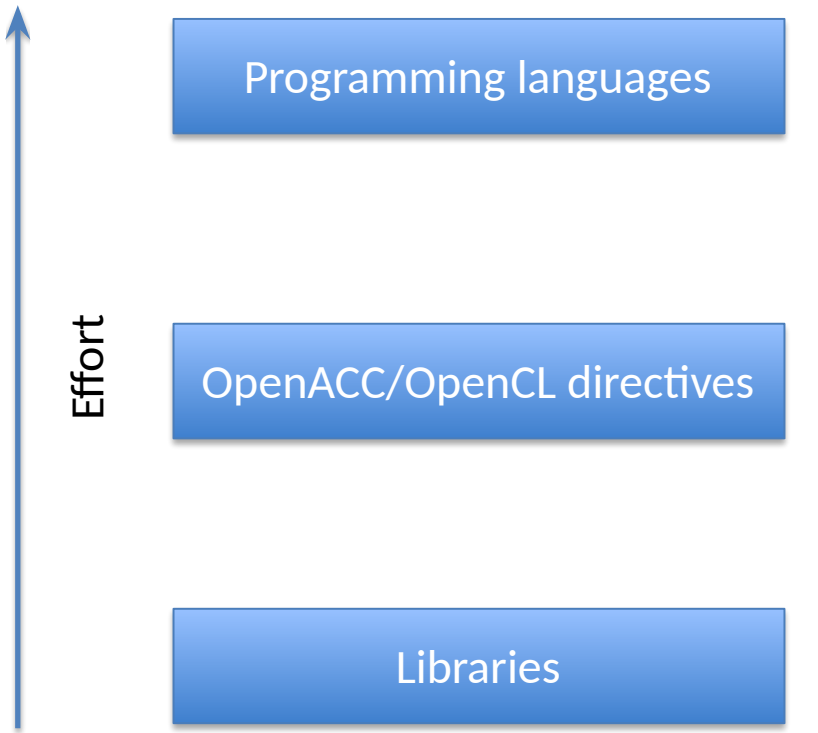
- Separated memory space.
- CPU and GPU have to manage the memory separately.
- The CPU is responsible for allocate the memory.
- Kernels must be launched from the CPU.



TAKE HOME MESSAGE

CPU and GPUs are independent from each other.
As such, it is the programmer's responsibility to manage the resources.

How can you program on GPU?



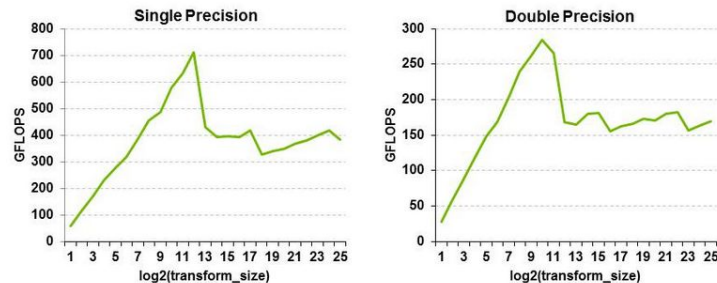
- Maximum flexibility
 - Cuda C/C++/Fortran
- Simple programming directives
 - Simple compiler pragma
 - Compiler parallelization code
 - Target a variety of platform
- Drop-in Acceleration
 - Highly optimized by GPU experts
 - FFT, BLAS, LAPACK, magma, RNG

GPU-accelerated libraries(NVIDIA)

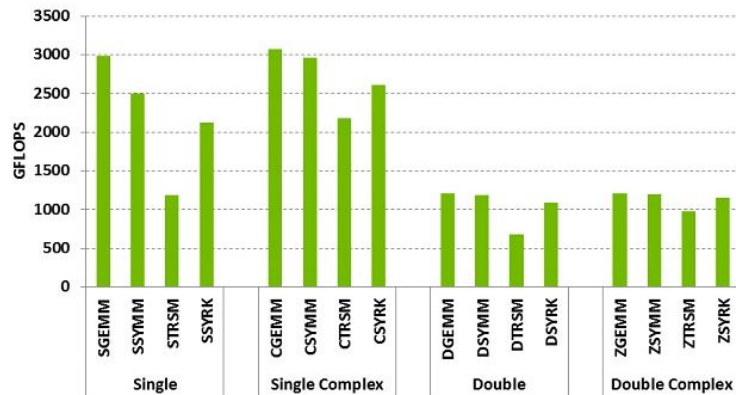
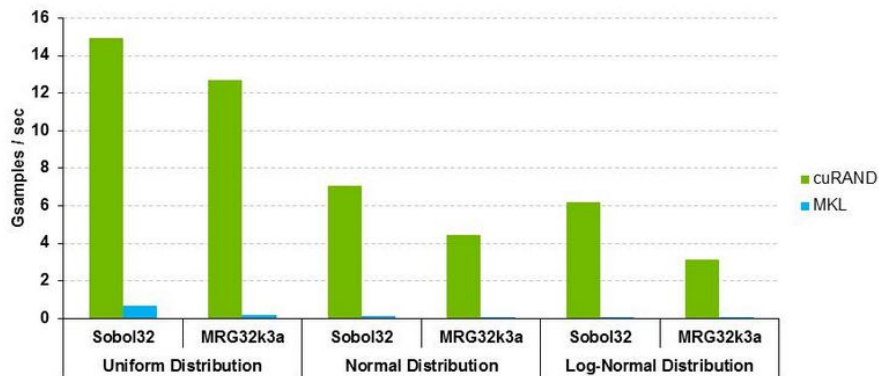
- cuFFT - FFT transform
- cuBLAS - Basic Linear Algebra
- cuSPARSE - Sparse Matrix Routines
- cuSOLVER - Dense and Sparse Direct Solver
- cuDNN - deep learning
- Magma - Matrix Algebra on GPU and multicore architectures

cuFFT: up to 700 GFLOPS

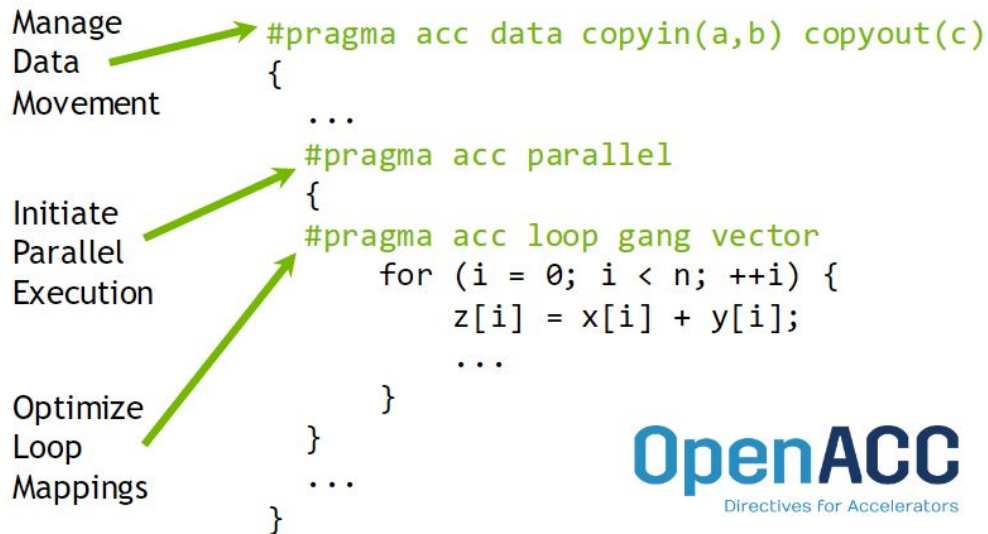
1D Complex, Batched FFTs
Used in Audio Processing and as a Foundation for 2D and 3D FFTs



cuRAND: Up to 75x Faster vs. Intel MKL



OpenACC Directives



- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, MIC

CUDA Overview

Parallel computing architecture developed by NVIDIA

CUDA programming interface consist of:

- C language extensions to target portions of source code on the compute device
- A runtime library split into:
 - *Host(CPU)*: component – executes on host, provides functions to control and access one or more compute devices
 - *Device(GPU)*: *component* – executes on device, provides device-specific functions
 - *Common component* – provide built-in vector types and subset of C standard library supported both on host and device









CUDA: Tools

- Compilers
 - Need to compile separately host code and device code
 - Host code: gcc, intel Device code: nvcc (module load cuda on Izar)
- Debugger
 - Cuda-gdb: extension of gdb debugger
- Nvprof
 - Cuda profiler to help with cuda optimization
- <https://developer.nvidia.com/cuda-zone>

Home > Accelerated Computing > CUDA Zone

CUDA Zone

CUDA® is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are using GPU-accelerated computing for broad-ranging applications.

 CUDA Toolkit All about the NVIDIA CUDA parallel computing platform Learn more >	 Training First steps for getting started in parallel computing Learn more >
 Tools & Ecosystem From accelerated cloud appliances to profiling tools, a gold mine of information Learn more >	 More Accelerated Computing Learn more about accelerated computing Learn more >
 Downloads Get the latest accelerated computing downloads Learn more >	 Resources Materials and links especially for GPU Computing professionals and developers Learn more >

Compute Capability: which GPU?

Specification and features of a compute device depends on its compute capability

Table 14. Feature Support per Compute Capability

Feature Support	Compute Capability				
	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x	8.x
(Unlisted features are supported for all compute capabilities)					
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)			Yes		
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)			Yes		
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)			Yes		
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)			Yes		
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())			Yes		
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No			Yes	
Warp vote functions (Warp Vote Functions)					
Memory fence functions (Memory Fence Functions)					
Synchronization functions (Synchronization Functions)					
Surface functions (Surface Functions)					
Unified Memory Programming (Unified Memory Programming)					
Dynamic Parallelism (CUDA Dynamic Parallelism)					
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No		Yes		
Bfloat16-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion		No			Yes
Tensor Cores		No		Yes	
Mixed Precision Warp-Matrix Functions (Warp matrix functions)		No		Yes	
Hardware-accelerated <code>memcpy_async</code> (Asynchronous Data Copies)		No			Yes
Hardware-accelerated Split Arrive/Wait Barrier (Asynchronous Barrier)		No			Yes
L2 Cache Residency Management (Device Memory L2 Access Management)		No			Yes

Data-Parallel Computing

- Performs operations on a data set organized into a common structure (eg. an array).
- A set of tasks work collectively and **simultaneously** on the same structure with each task operating on its own portion of the structure.
- Tasks perform identical operations on their portions of the structure. Operations on each portion must be **independent**.

Data Dependence

- Data dependence occurs when a program statement refers to the data of a preceding statement

```
a = 2 * x;
```

```
b = 2 * y;
```

```
c = 3 * x;
```

These 3 statements are independent

```
a = 2 * x;
```

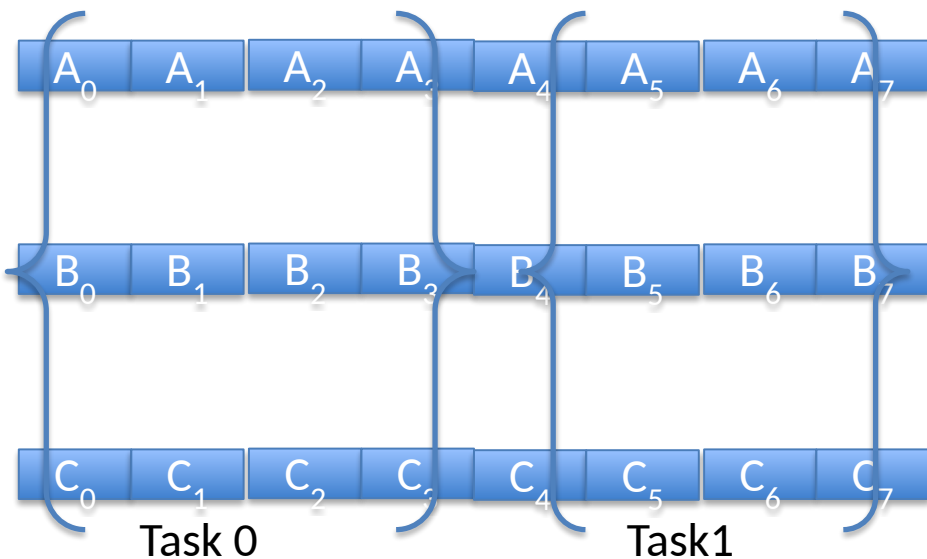
```
b = 2 * a * a;
```

```
c = b * 9;
```

b depends on a, c depends on a and b

- Data dependence limits the parallelism

Data-Parallel Computing Example



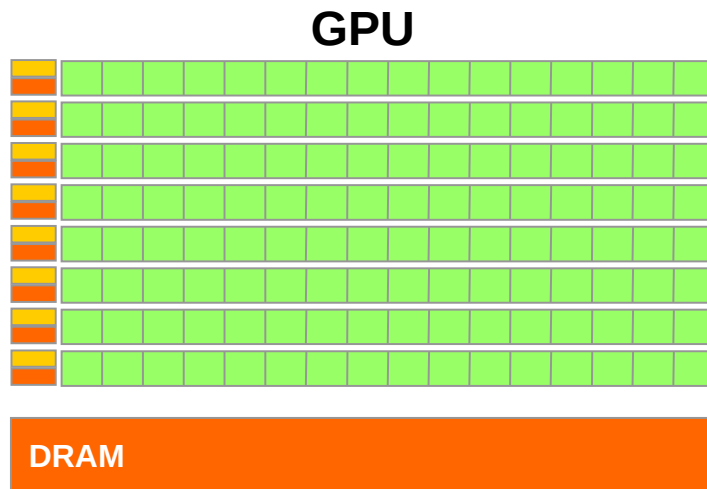
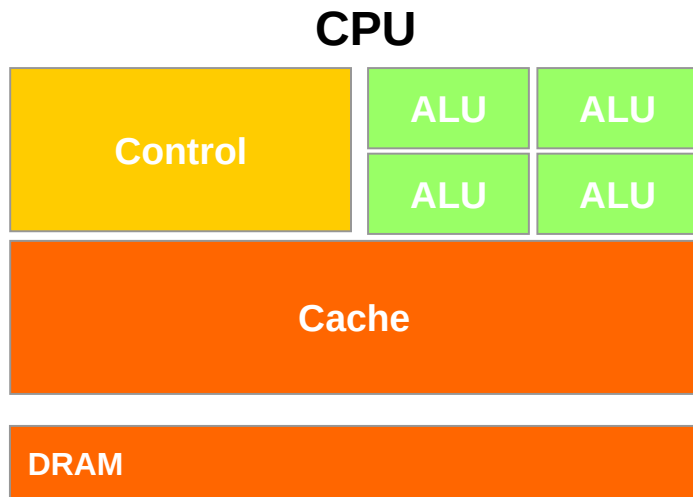
$$C_x = A_x + B_x$$

- Data set consisting of arrays A, B and C
- Same operations performed on each element
$$C_x = A_x + B_x$$
- Two tasks operating on a subset of the arrays. Tasks 0 and 1 are independent. Could have more tasks.

Data-Parallel Computing on GPUs

Data-parallel computing maps well to GPUs:

- Identical operations executed on many data elements in parallel
- Simplified logic allows increased ratio of computation



The CUDA Programming Model

- The GPU is a compute **device** that
 - Has its own RAM(**device memory**)
 - Runs data-parallel portions of an application as **kernels** by using many threads
- Kernels are:
 - C/C++ functions with some restrictions, and a few language extension
 - Executed by many threads
- GPU vs. CPU threads
 - GPU threads are **extremely lightweight**
 - GPU needs **1000s of threads for full efficiency**
 - A multi-core CPU needs only a few

Many-threads approach

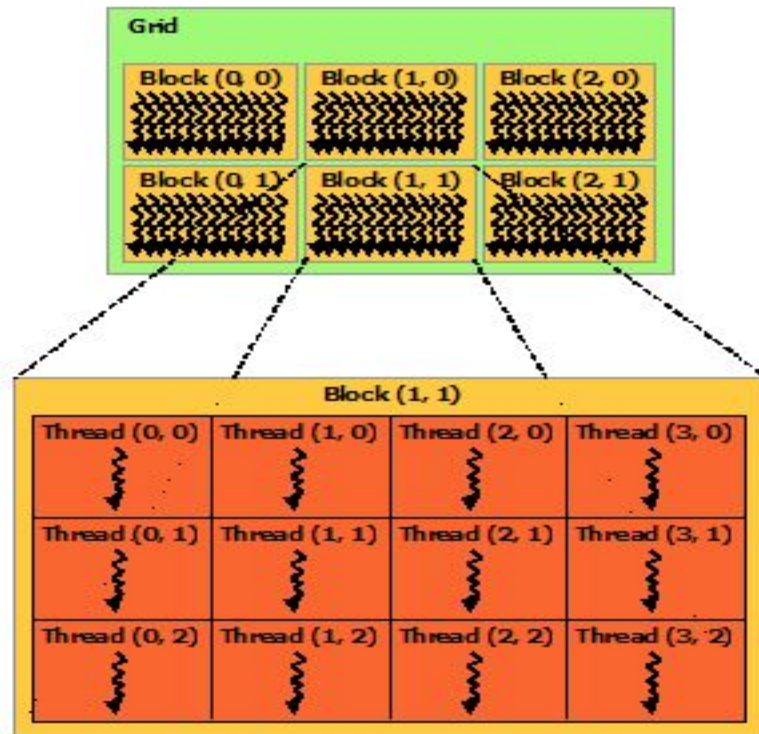
- The cores in the streaming multiprocessors are SIMT(Single Instruction Multiple Threads)
- All the cores execute the same instruction on different data
- Vector computing
- minimum of 32 threads doing the same thing at the same time (warp)
- Lots of active threads = the key to performance
- Execution alternates between active warps which become inactive when they wait for data.
- Threads are organized in **grids of blocks**.

CUDA Threads Hierarchy

- CUDA is designed to execute 1000s of threads
- Threads are grouped together into **thread blocks**
- Threads blocks are grouped together into a **grid**

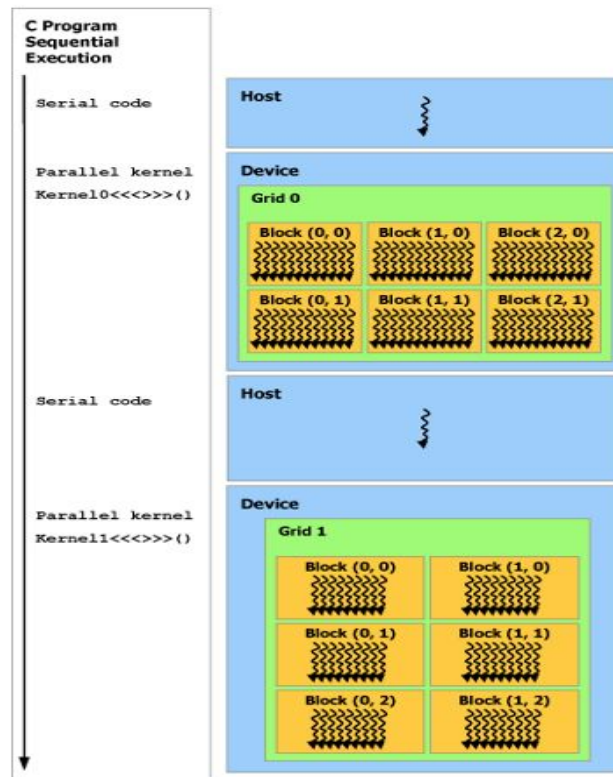
CUDA Thread Hierarchy

- Thread blocks and Grids can be 1D, 2D or 3D
- Dimensions set at launch time
- Thread blocks and grids do not need to have the same dimensionality, e.g. 1D grid of 2D blocks
- Thread blocks must execute independently



CUDA Programming Model

- The host launches kernels
- The host is responsible for:
 - Managing the allocated memory on host and device
 - Data exchange between host and device
 - Error handling



CUDA APIs

Can use CUDA through CUDA C (Runtime API), or Driver API

- This course will focus on CUDA C
- Driver API has a much more verbose syntax
- Don't confuse the two when referring to CUDA Documentation
 - `cuFunctionName` – Driver API
 - `cudaFunctionName()` – Runtime API

What the programmer expresses in CUDA

- Computation partitioning (where does the computation occur?)
 - Declarations on functions `__host__`, `__global__`, `__device__`
 - Mapping of thread programs to device: `compute<<<gs,bs>>>(<args>)`
- Data partitioning (where does data reside, who may access it and how?)
 - Declaration on data `__shared__`, `__device__`, `__constant`
- Data management and orchestration
 - Copying to/from host:
 - e.g., `cudaMemcpy(h_obj, d_obj, size, cudaMemcpyDeviceToHost)`
or **Unified memory** !
- Concurrency management
 - e.g. `__syncthreads()`

Unified memory

Unified memory is a single memory address space accessible from the GPU and the CPU.

- Allocating memory in the unified memory means replacing all **malloc()** or **new()** by **cudaMallocManaged()**
- A **free()** is replaced by a **cudaFree()**

Warning : memory must be synchronized if either CPU or GPU tries to access it before/after completion.

Use **cudaDeviceSynchronize()** to synchronize host and device.

The allocated memory must fit entirely within the GPU's memory !

CUDA Kernel Launch Syntax

- CUDA Kernels are launched by the host using a modified C function call syntax

myKernel<<<*dim3 dGrid, dim3dBlock*>>>(...)

dim3 is vector type x,y and z component (dG.x)

Table 15. Technical Specifications per Compute Capability

	Compute Capability												
Technical Specifications	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Maximum number of resident grids per device (Concurrent Kernel Execution)	32				16	128	32	16	128	16	128		
Maximum dimensionality of grid of thread blocks	3												
Maximum x-dimension of a grid of thread blocks	2 ³¹ -1												
Maximum y- or z-dimension of a grid of thread blocks	65535												
Maximum dimensionality of a thread block	3												
Maximum x- or y-dimension of a block	1024												
Maximum z-dimension of a block	64												
Maximum number of threads per block	1024												
Warp size	32												
Maximum number of resident blocks per SM	16		32							16	32	16	
Maximum number of resident warps per SM	64										32	64	48
Maximum number of resident threads per SM	2048										1024	2048	1536

CUDA Kernels

- Denoted by `__global__` function qualifier
 - `__global__ void mykernel(*float a)`
- Called from host, executed on device(on the SM)
- Some restrictions:
 - Must return void
 - No static variable
 - No access to host functions

CUDA Syntax – Kernels

Kernels can take arguments just like any C function

- Pointer
- Parameters passed by value

```
__global__ void SimpleKernel(float *a, float b)
{
    a[0] = b;
}
```

CUDA Syntax - Kernels

Kernels must be declared in
the source/header files
before they are called

```
// Kernel declaration
__global__ void kernel(*float a)

Int main()
{
    dim3 gridSize, blockSize;
    float *a;

    kernel<<<gridSize,blockSize>>>(a);
}

__global__ void kernel(float* a)
{
    -----
}
```

CUDA Syntax - Kernels

Kernels have read-only built-in variables:

- `gridDim`: dimensions of the grid.
- `blockIdx`: unique index of a block within a grid.
- `blockDim`: dimensions of the block.
- `threadIdx`: unique index of the thread within a block.
- Cannot vary the size of the blocks or grids during kernel call.
- The code inside the kernel is written from single thread point of view.

CUDA Syntax - Kernels

- All C operators are supported
 - eg. +, *, /, ^, >, >>
- All functions from the standard math library
 - eg. sinf(), cosf(), ceilf(), fabsf()
- Control flow statements too
 - eg. If(), while(), for()

Hello, World!

```
int main( void ) {  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

- To compile: `nvcc -o hello_world hello_world.cu`
- To execute: `./hello_world`
- This basic program is just standard C that runs on the *host*
- NVIDIA's compiler (**nvcc**) will not complain about CUDA programs with no *device* code

Hello, World! with Device Code

```
__global__ void kernel( void ) {  
}
```

- CUDA C keyword **__global__** indicates that a function
 - Runs on the device
 - Called from host code
- **nvcc** splits source file into host and device components
 - NVIDIA's compiler handles device functions like **kernel()**
 - Standard host compiler handles host functions like **main()**
 - **gcc, icc, ...**
 - **Microsoft Visual C**

Hello, World! with Device Code

```
int main( void ) {  
    kernel<<< 1, 1 >>>();  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

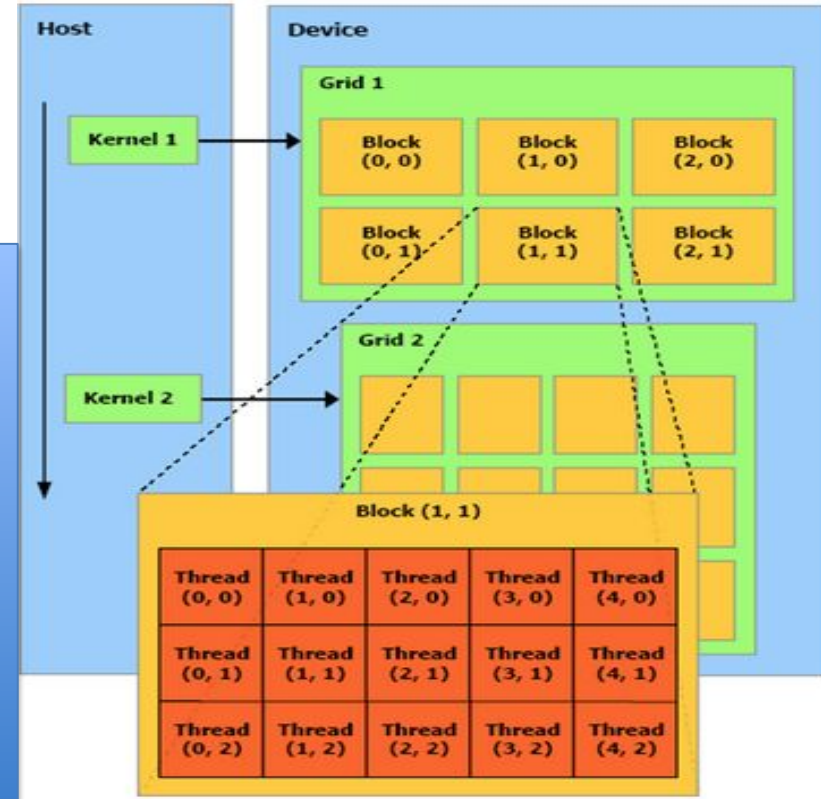
- Triple angle brackets mark a call from *host* code to *device* code
 - A “kernel launch” in CUDA jargon
 - We’ll discuss the parameters inside the angle brackets later
- This is all that’s required to execute a function on the GPU!
- The function **kernel**() does nothing, so let us run something a bit more useful:

CUDA Syntax - Kernels

- Through built-in variable is possible to index your arrays
- Map local thread ID to a global array index

```
// create two dimensional 5x3 thread blocks
dim3 block_size;
block_size.x = 5;
block_size.y = 3;

// configure a two dimensional grid as well
dim3 grid_size;
grid_size.x = 3;
grid_size.y = 2;
// grid_size & block_size are passed as arguments to the triple
chevrons as usual
kernel<<<grid_size,block_size>>>(device_array);
```



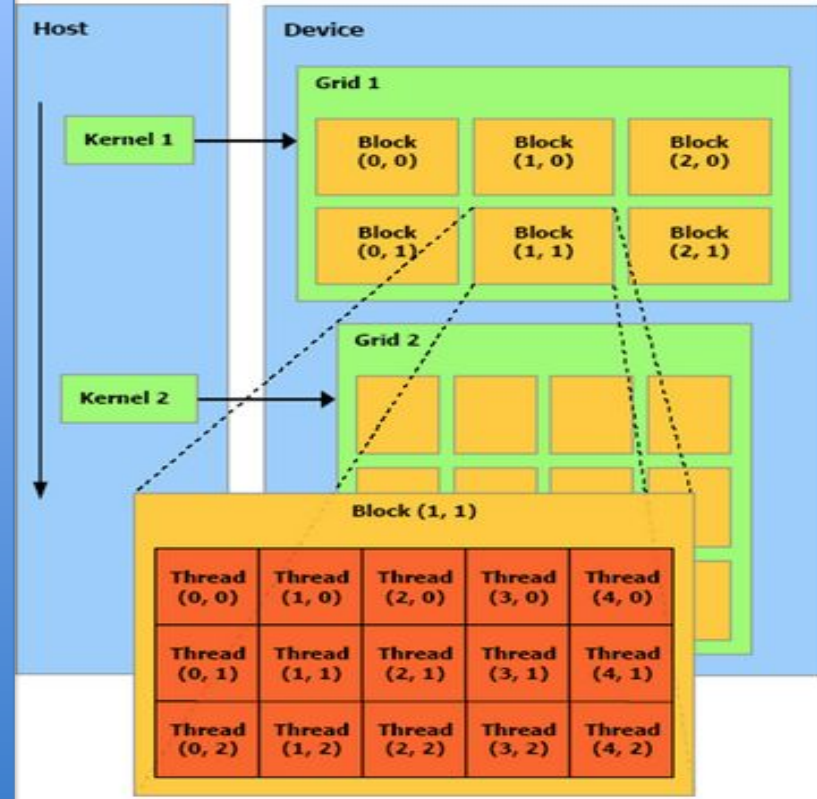
CUDA Syntax – Inside the Kernel

```
__global__ void kernel(int *array)
{
    int index_x = blockIdx.x * blockDim.x + threadIdx.x;
    int index_y = blockIdx.y * blockDim.y + threadIdx.y;

    // map the two 2D indices to a single linear, 1D index
    int grid_width = gridDim.x * blockDim.x;
    int index = index_y * grid_width + index_x;

    // map the two 2D block indices to a single linear, 1D block index
    int result = blockIdx.y * gridDim.x + blockIdx.x;

    // write out the result
    array[index] = result;
}
```



CUDA Syntax – Index & Size Calculation

- Global index calculation in 1D
 - $Idx = blockIdx.x * blockDim.x + threadIdx.x$
- Grid size calculation

$$GridSize = (Size + BlkDim - 1) / BlkDim$$

(integer division)

- Size: Total size of the array
- BlkDim: Size of the block (max 1024)
- GridSize: Number of blocks in the grid

CUDA Syntax – Thread Identifiers

Result for each kernel launched with the following execution configuration:

MyKernel<<<3,4>>>(a);

```
__global__ void Mykernel(int* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = 5;
}

__global__ void Mykernel(int* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = blockIdx.x;
}

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

a = 5 5 5 5 5 5 5 5 5 5

a = 0 0 0 1 1 1 1 2 2 2 2

a = 0 1 2 3 0 1 2 3 0 1 2 3

User-defined device functions

- Can write/call your own device functions
 - `__device__ float myDeviceFunctions()`
 - Device functions cannot be called by the host

```
__device__ float myDeviceFunction()
{
    .....
}

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

CUDA Syntax - Synchronization

- Kernels launches are asynchronous,
 - They return to CPU immediately
 - Kernel starts executing once all outstanding CUDA calls are complete
- **cudaDeviceSynchronize()** block until all outstanding CUDA calls are completed

CUDA Syntax – Error Management

- Host code manages errors
- Most CUDA functions return `cudaError_t`
 - Enumeration type
 - `cudaSuccess` (value 0) indicates no error
 - Use `cudaGetErrorString()`
- `char* cudaGetErrorString(cudaError_t err)`
 - Return a string describing the error condition

```
cudaError_t err;  
e = cudaMemcpy(...);  
If(e) printf("Error: %s\n", cudaGetErrorString(err));
```

CUDA Syntax – Error Management

- Kernels launches have no return value!
- `cudaError_t cudaGetLastError()`
 - Return error code for last CUDA runtime function (including kernel launches)
 - At exit, clears global error state; subsequent calls will return “success”
 - In case of multiple errors, only the last one is reported
 - If asynchronous `cudaDeviceSynchronize()` must be called before