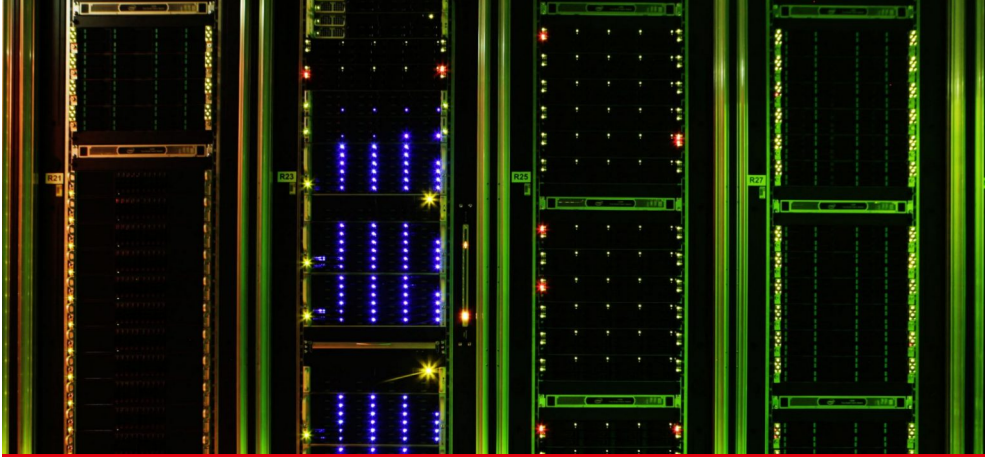


MATH-454 Parallel and High Performance Computing

Lecture 7: Introduction to GPU computing

Pablo Antolin

Slides of N. Varini's lecture notes

April 10 2025

What will we learn today?

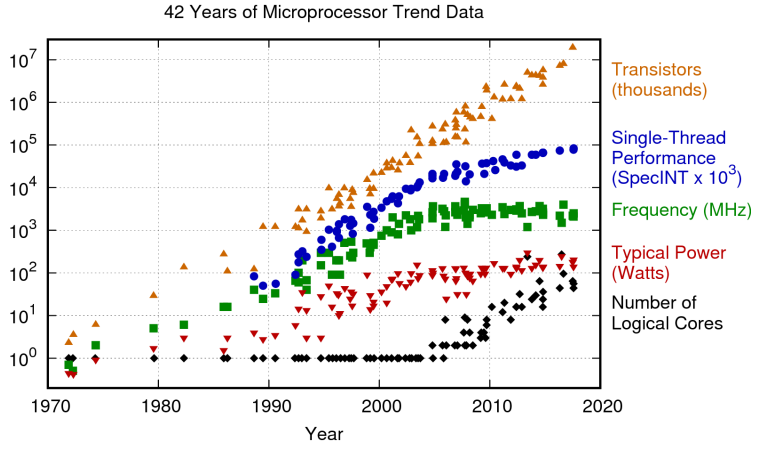
- Motivation: Trends in HPC
- Hardware architecture
- Software environment
- How to program on GPU using CUDA

Motivation: Why running on GPUs?

- Massively parallel
- Thousands of cores
- Many threads approach
- Programmable: CUDA
- Many softwares available:
  - Machine Learning: Caffe, PyTorch, Tensorflow, Theano
  - Molecular Dynamics: Lammps, Amber
  - Weather prediction: Cosmo
  - Dozens of general purpose libraries from NVIDIA

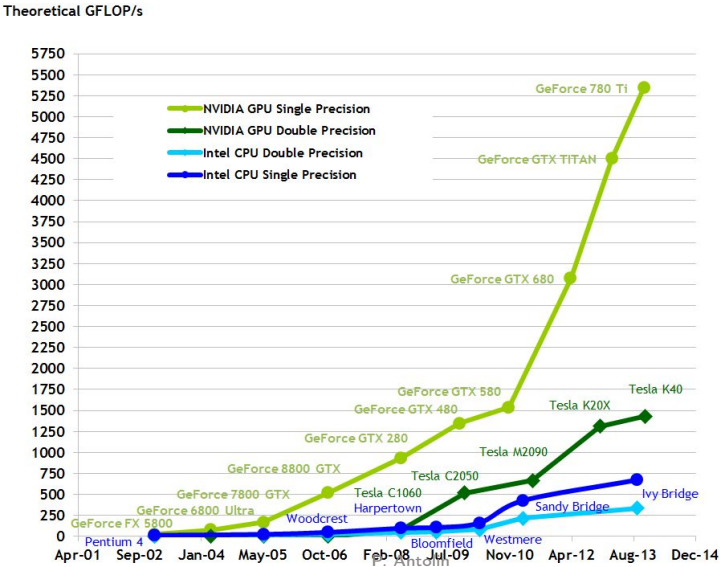
Motivation: Why running on GPUs

- Single-thread performance increased $\sim$ linearly
- Number of logical cores is increasing $\sim$ power law
- Frequence $\sim$ 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 2010-2017 by K. Rupp

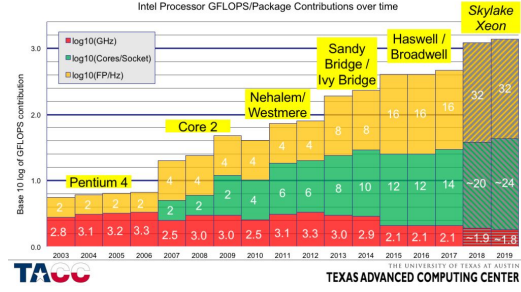
Why GPU? Performance!



Nvidia V100 vs Intel processors

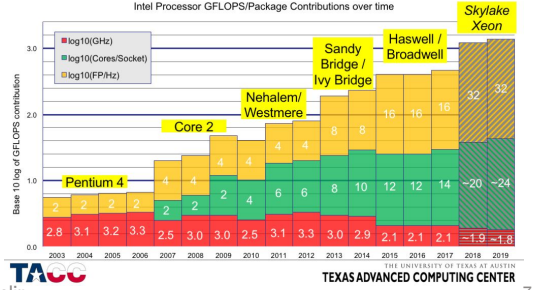
NVIDIA TESLA V100 SPECIFICATIONS

	Tesla V100 for NVLink	Tesla V100 for PCIe
PERFORMANCE with NVIDIA GPU Boost™	DOUBLE-PRECISION 7.8 teraFLOPS SINGLE-PRECISION 15.7 teraFLOPS DEEP LEARNING 125 teraFLOPS	DOUBLE-PRECISION 7 teraFLOPS SINGLE-PRECISION 14 teraFLOPS DEEP LEARNING 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



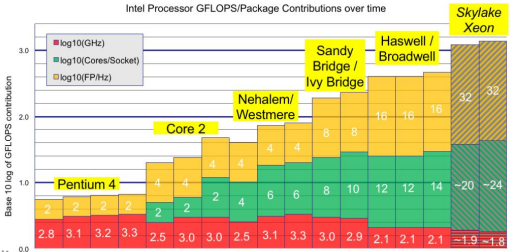
Nvidia A100 vs Intel processors

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm^2
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS | 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS | 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS | 624 teraFLOPS*
INT8 Tensor Core	624 TOPS | 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS | 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



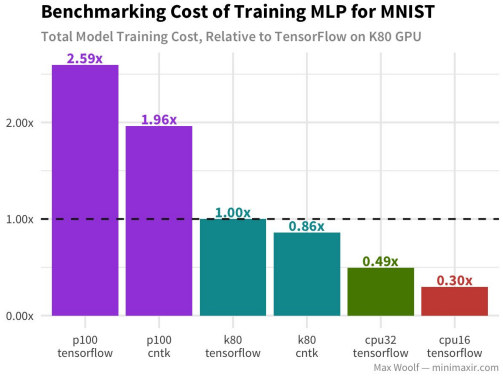
Nvidia H100 vs Intel processors

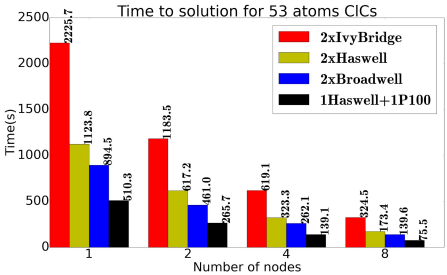
Form Factor	H100 SXM	H100 PCIe
FP64	34 teraFLOPS	26 teraFLOPS
FP64 Tensor Core	67 teraFLOPS	51 teraFLOPS
FP32	67 teraFLOPS	51 teraFLOPS
TF32 Tensor Core	989 teraFLOPS*	756teraFLOPS*
BFLOAT16 Tensor Core	1,979 teraFLOPS*	1,513 teraFLOPS*
FP16 Tensor Core	1,979 teraFLOPS*	1,513 teraFLOPS*
FP8 Tensor Core	3,958 teraFLOPS*	3,026 teraFLOPS*
INT8 Tensor Core	3,958 TOPS*	3,026 TOPS*
GPU memory	80GB	80GB
GPU memory bandwidth	3.35TB/s	2TB/s
Decoders	7 NVDEC 7 JPEG	7 NVDEC 7 JPEG
Max thermal design power (TDP)	Up to 700W (configurable)	300-350W (configurable)
Multi-Instance GPUs	Up to 7 MIGS @ 10GB each	
Form factor	SXM	PCIe Dual-slot air-cooled
Interconnect	NVLink: 900GB/s PCIe Gen5: 128GB/s	NVLINK: 600GB/s PCIe Gen5: 128GB/s
Server options	NVIDIA HGX™ H100 Partner and NVIDIA-Certified Systems™ with 4 or 8 GPUs NVIDIA DGX™ H100 with 8 GPUs	Partner and NVIDIA-Certified Systems with 1–8 GPUs
NVIDIA AI Enterprise	Add-on	Included



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





Quantum-ESPRESSO + SIRIUS

GPU advantage: Performance - TOP 500 (November 2024)

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
5	**HPC6** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, HPE Eni S.p.A. Italy	3,143,520	477.90	606.97	8,461

GPU advantage: Performance - TOP 500 (November 2024)

Not only NVIDIA ...

TAKE HOME MESSAGE

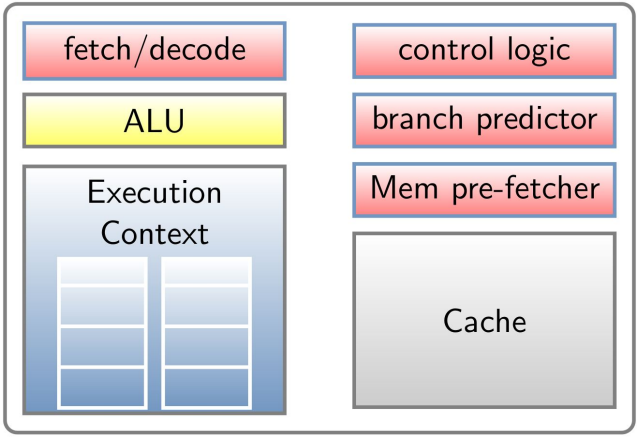
The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation - **exactly what graphics rendering is about** - and therefore designed such that more transistors are devoted to data processing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

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

GPU disadvantages?

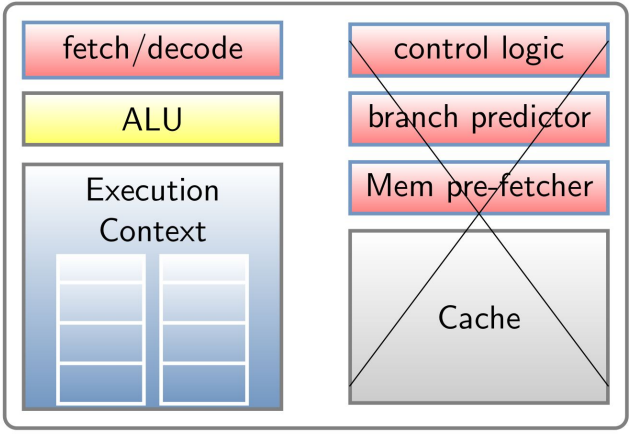
- Architecture not as flexible as CPU (but easier to get performance).
- Must rewrite algorithms and maintain software in GPU languages.
- Discrete GPUs attached to CPU via relatively slow PCIe
  - 128 GB/s bi-directional via PCIE (H100)
  - 900 GB/s via NVlink (H100)
- (Not so) limited memory: 80 GB (H100). Increasing.

From CPU to GPU



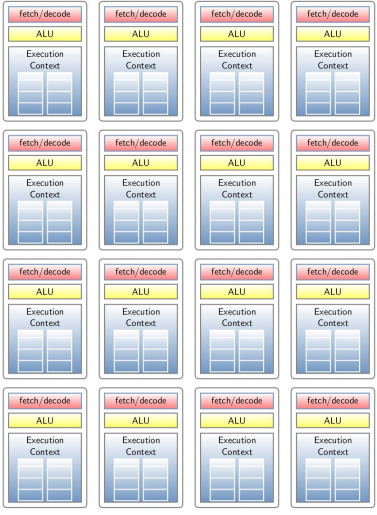
From CPU to GPU

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

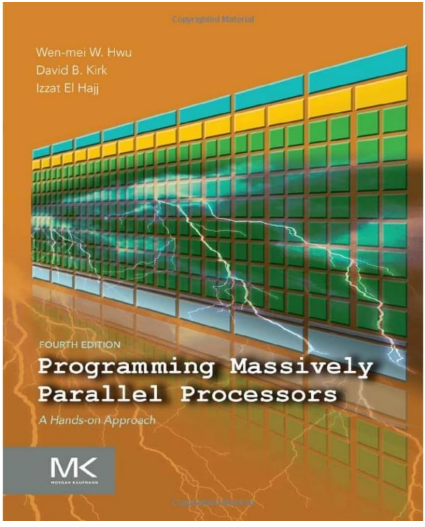


From CPU to GPU

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



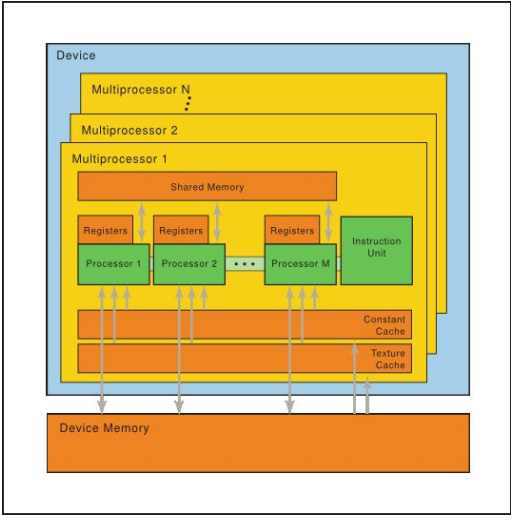
"CPUs are designed to minimize the latency of instruction execution and the GPUs are designed to maximize the throughput of executing instructions"



Streaming Multiprocessors Architecture simplified

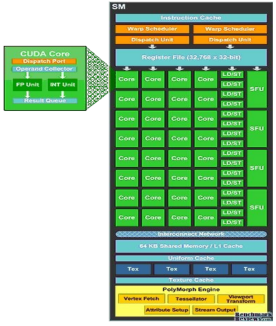
In a single **Streaming Multiprocessor** (SM):

- Many memory regions available, each with different performance characteristics.
- Must map the dataset to the right memory type.



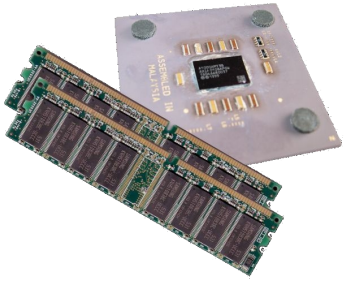
Streaming Multiprocessors

- Each GPU is comprised of one or more **Streaming Multiprocessors (SM)** (e.g., NVidia V100 has 80 SMs, H100 has 114 SMs).
- Each SM has a **multiples of 32 cores** (e.g., one V100's SM has 64 single-precision cores, and 32 for double-precision. I.e., they have 5120/2560 single/double-precision CUDA cores + 640 tensor cores).
- Instructions are executed in multiples of **32 threads (warp)**.
- Each SM has a collection of cores, registers, memory.

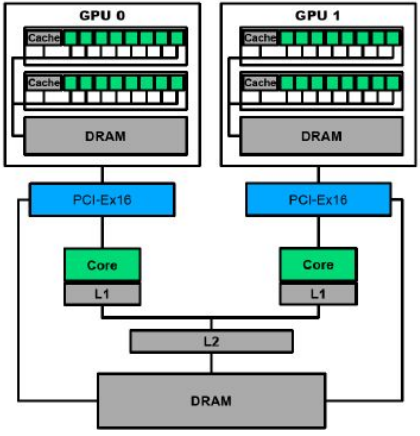


CUDA jargon

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

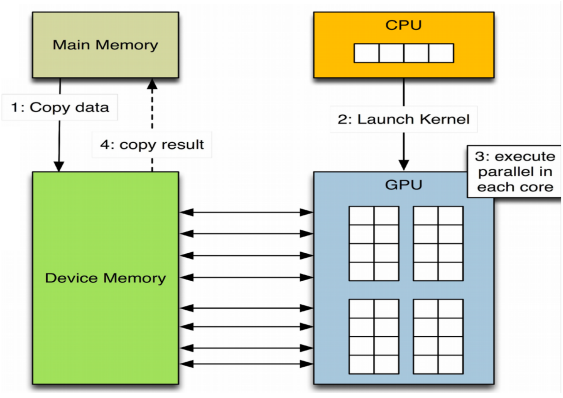


Accelerated node: Overview



Accelerated node: Overview

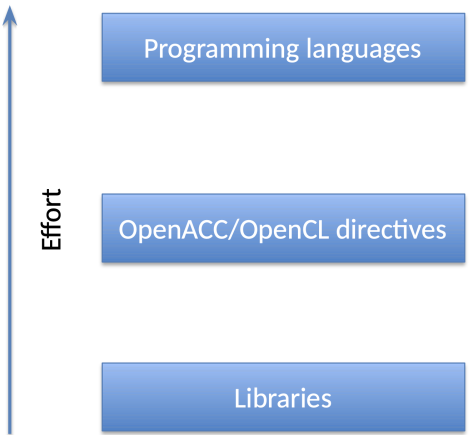
- Separated memory space.
- CPU and GPU have to manage the memory separately.
- The CPU is responsible for allocating the memory in both the host and and the device.
- Kernels must be launched from the CPU.



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?

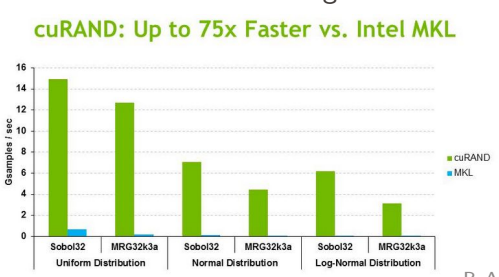


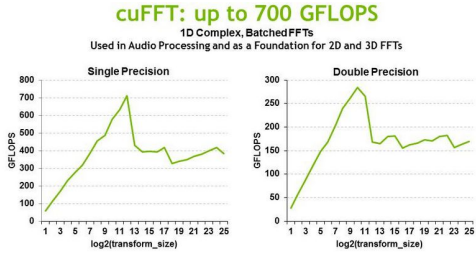
- Maximum flexibility
  - CUDA
    C/C++/Fortran/Python.
- Simple programming directives:
  - Simple compiler pragma.
  - Compiler parallelization code.
  - Target a variety of platforms.
- Drop-in Acceleration:
  - Highly optimized by GPU experts.
  - FFT, BLAS, LAPACK, magma, RNG.
  - OpenCV, Pytorch, Tensorflow,

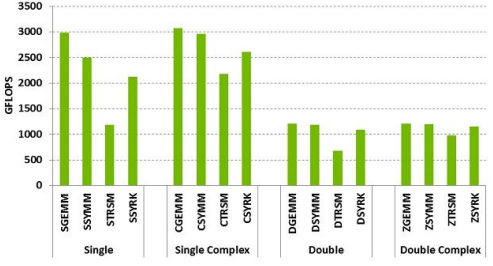
NVIDIA CUDA-X accelerated libraries

NVIDIA CUDA-X accelerated libraries

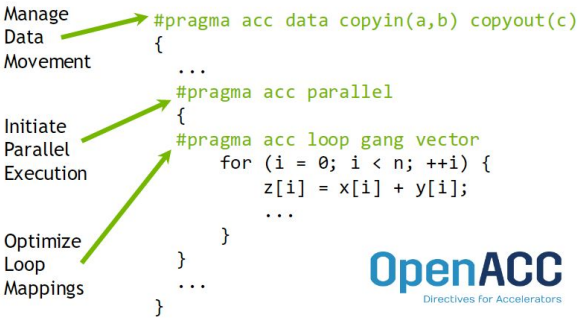
- cuFFT - FFT transform
- cuBLAS - Basic Linear Algebra
- cuSPARSE - Sparse Matrix Routines
- cuSOLVER - Dense and Sparse Direct Solver
- cuDNN - deep learning
- cuML - machine learning







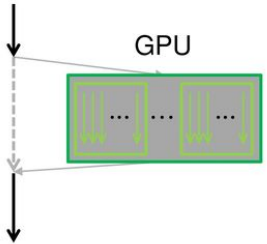
OpenACC Directives



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

OpenMP: GPU Offloading

- A productive, high-level, directive based API to parallelize an application
- Portable across multiple architectures and platforms




```
template <class T> void OMP45Stream<T>::triad()
{
  unsigned int len = this->array_size;
  T *a = this->a;  T *b = this->b;  T *c = this->c;

  #pragma omp target teams distribute parallel for\
   map(tofrom: a[:len]) map(to: b[:len], c[:len])
  for (int i = 0; i < array_size; i++)
  {
    a[i] = b[i] + startScalar * c[i];
  }
}
```

- GPU offloading with single line!

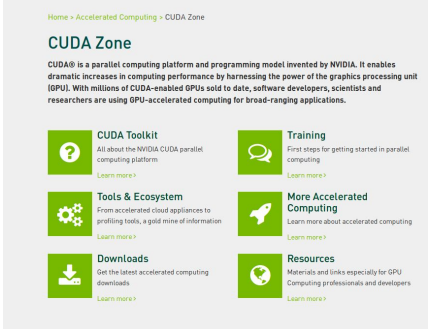
Parallel computing architecture developed by NVIDIA

CUDA programming interface consists of:

- C++ language extensions to target parts 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



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**				
(Unlisted features are supported for all compute capabilities)	**3.5, 3.7, 5.0, 5.2**	**5.3**	**6.x**	**7.x**	**8.x**
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)	Yes				
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 `memcpy_async` (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

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**			V100	
(Unlisted features are supported for all compute capabilities)	**3.5, 3.7, 5.0, 5.2**	**5.3**	**6.x**	**7.x**	**8.x**
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)	Yes				
Memory fence functions (Memory Fence Functions)	Yes				
Synchronization functions (Synchronization Functions)	Yes				
Surface functions (Surface Functions)	Yes				
Unified Memory Programming (Unified Memory Programming)	Yes				
Dynamic Parallelism (CUDA Dynamic Parallelism)	Yes				
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 `memcpy_async` (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 dataset organized into a common structure (e.g., 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 parts 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 preceeding statement.

These 3 statement are independent

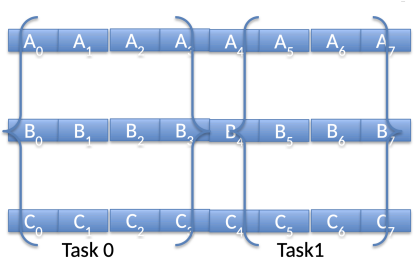
```
1 a = 2 * x;
2 b = 2 * y;
3 c = 3 * x;
```

b depends on a, c depends on a and b

```
1 a = 2 * x;
2 b = 2 * a * a;
3 c = b * 9;
```

- Data dependence limits the parallelism.

Data-parallel computing example

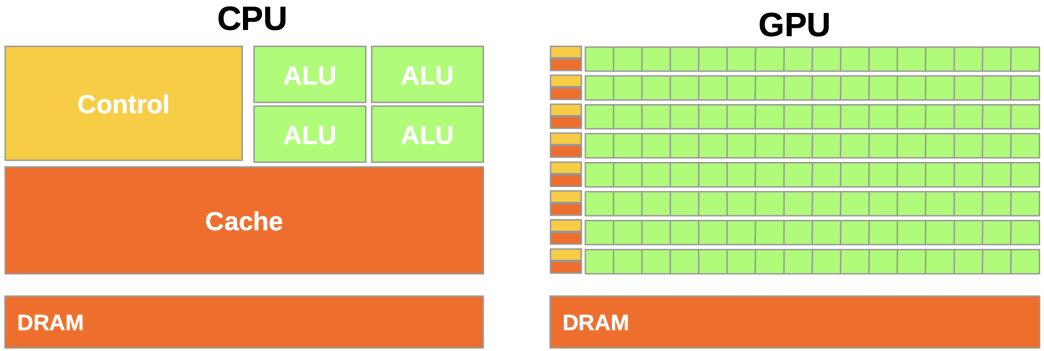


$C_x = A_x + B_x$

- Dataset 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 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 parts of an application as **kernels** by using many threads.
- Kernels are:
  - C/C++ functions with some restrictions, and a few language extensions.
  - 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

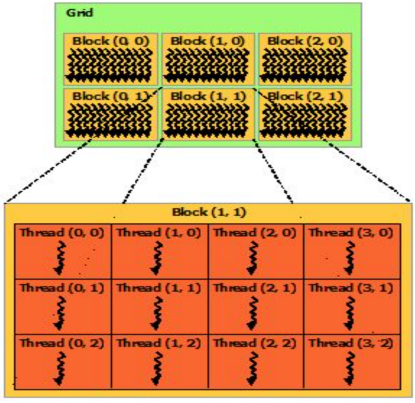
- The cores in the streaming multiprocessors are SIMT (Single Instruction Multiple Threads).
- All the cores execute the same instruction on different data (vector computing).
- Minimum of 32 threads doing the same thing at the same time (warp).
- Lots of active threads = the key to performance (occupancy).
- 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**.

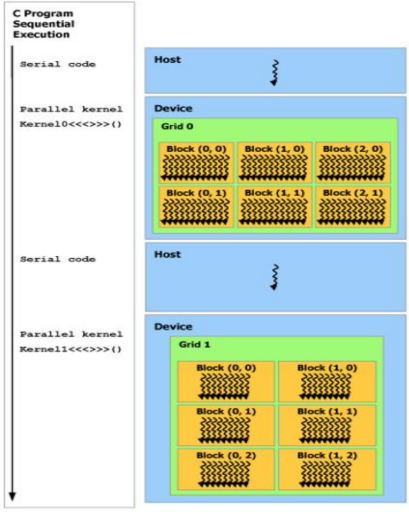
CUDA Threads Hierarchy

- 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 is lower level and has a much more verbose syntax.
- Don't confuse the two when referring to CUDA Documentation:
  - cuFunctionName $\rightarrow$ Driver API
  - cudaFunctionName() $\rightarrow$ Runtime API (the one we will use)

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., `__synchthreads()`

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, dim3 dBlock>>>(...)`
  `dim3` is vector type `x`, `y`, and `z` components (`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 variables.
  - No access to host functions.

CUDA Syntax - Kernels

Kernels can take arguments just like any C++ function

- Pointer
- Parameters passed by value

```
1 __global__ void SimpleKernel(float *a, float b) {
2     a[0] = b
3 }
```

CUDA Syntax - Kernels

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

```
1  // Kernel forward declaration
2  __global__ void kernel(float *a);
3
4  int main() {
5      dim3 gridSize, blockSize;
6      float *a;
7  }
8
9  __global__ void kernel(float* a) {
10     // Kernel implementation
11 }
```

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 Kernels - Syntax

- All C operators are supported:
  - eg, `+`,`*`,`/`,`^`,`>`,`>>`
- Many functions from the C/C++ standard math library:
  - e.g., `sin`, `cos`, `sqrt`, `ceil`, `exp`
  - See this link for a complete list of supported functions.
- Control flow statements too:
  - eg, `if()`, `while()`, `for()`

Hello, World!

```
1 int main( void ) {
2     printf( "Hello, World!\n" );
3     return 0;
4   }
```

- To compile: `nvcc -oo hello_world hello_world.cu`
- To execute: `./hello_world`
- This basic program is just standard C/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

```
1 __global__
2 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 Studio C

Hello, World! with Device Code

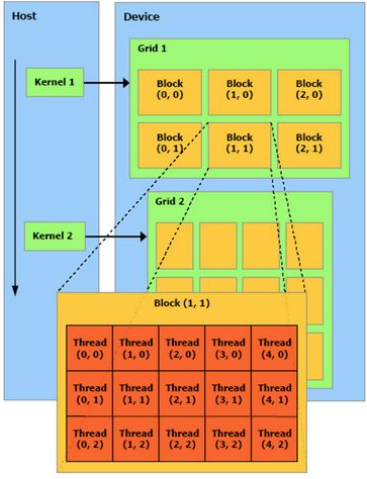
```
int main( void ) {
    kernel<<< 1, 1 >>>();
    printf( "Hello, World!\n" );
    return 0;
  }
```

- 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 Kernels - Syntax

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

```
1  // create 2D 5x3 thread bloks
2  dim3 block_size;
3  block_size.x = 5;
4  block_size.y = 3;
5  // configure 2D grid as well
6  dim3 grid_size;
7  grid_size.x = 3;
8  grid_size.y = 2;
9
10 // grid_size & block_size are passed as
   ↪  arguments to <<< >>>
11 kernel<<<grid_size,block_size>>>(dev_arr)
```



CUDA Syntax - Inside the kernel

```cpp
__global__ void kernel(int *arr) {
    // Map the two 2D block indices to a single linear, 1D block
    ↪  index
    int id_in_block = threadIdx.y * blockDim.x + threadIdx.x;

    // Number of blocks before the current one.
    int offset = blockIdx.y * gridDim.x + blockIdx.x;

    // 1D index
    const int block_size = blockDim.x * blockDim.y;
    const int id = offset * block_size + id_in_block;

    // Write out the result
    arr[id] = id_in_block;
}
```

CUDA Syntax - Index & Size calculation

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

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

  (integer division: round down)

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

```
1  __global__ void Mykernel(int* a) {
2    int idx = blockIdx.x*blockDim.x + threadIdx.x;
3    a[idx] = 5;
4  }
5
6  __global__ void Mykernel(int* a) {
7    int idx = blockIdx.x*blockDim.x + threadIdx.x;
8    a[idx] = blockIdx.x;
9  }
10
11 __global__ void Mykernel(int* a) {
12   int idx = blockIdx.x*blockDim.x + threadIdx.x;
13   a[idx] = threadIdx.x;
14 }
```

```
a = 5 5 5 5 5 5
↪   5 5 5 5 5 5

a = 0 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.

```
1 __device__ float myDeviceFunction() {
2   // .....
3 }
4
5 __global__ void myKernel(float* a) {
6   int idx = blockIdx.x*blockDim.x + threadIdx.x;
7   a[idx] = myDeviceFunction(idx);
8 }
```

CUDA Syntax - Synchronization

- Kernel's launches are asynchronous.
- They return to CPU immediately.
- Kernel starts executing once all outstanding CUDA calls are complete.
- **cudaDeviceSynchronize()** blocks 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.

```
1 cudaError_t err;
2 e = cudaMemcpy(...);
3 if(e) {
4     printf("Error: %s\n", cudaGetErrorString(err));
5   }
```

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()` is needed, `cudaGetLastError()` must be called before.