COMP 633 - Parallel Computing

Lecture 13
September 30, 2021

Computational Accelerators

Evolution of high-performance computing

- Long-standing market forces have shaped modern HPC systems
  - constructed using *commodity* CPUs (mostly)

- Recent market forces
  - Server farms
    - large memory, more cores, more I/O
  - Gaming
    - GPUs for real-time graphics
  - Cell phones
    - Signal processing hardware:
      - compression, computational photography

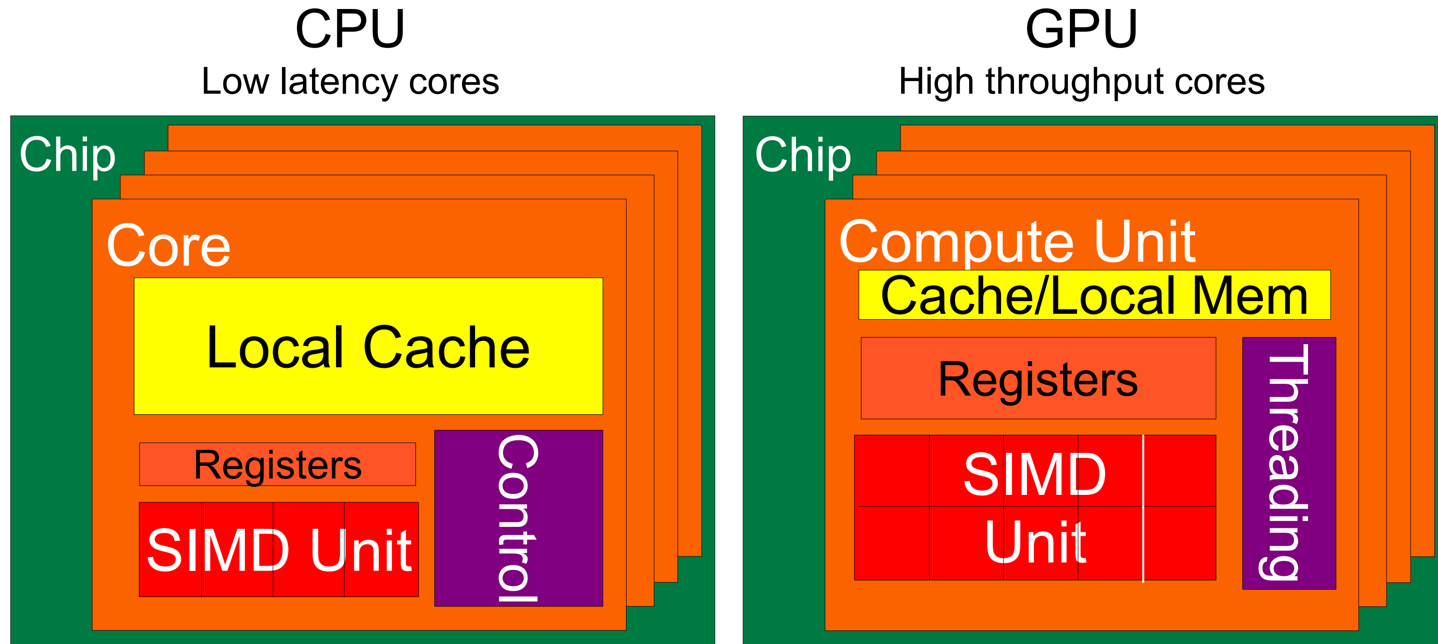
- Computational accelerators emerge from GPUs
  - 2007: Nvidia Compute Unified Device Architecture GPU (CUDA)
  - 2009: IBM/Toshiba/Sony Cell Broadband Engine (Cell BE) PlayStation 3
  - 2010: Intel Larrabee (DOA) → Many Integrated Cores (MIC) → Xeon Phi

HPC retrospective

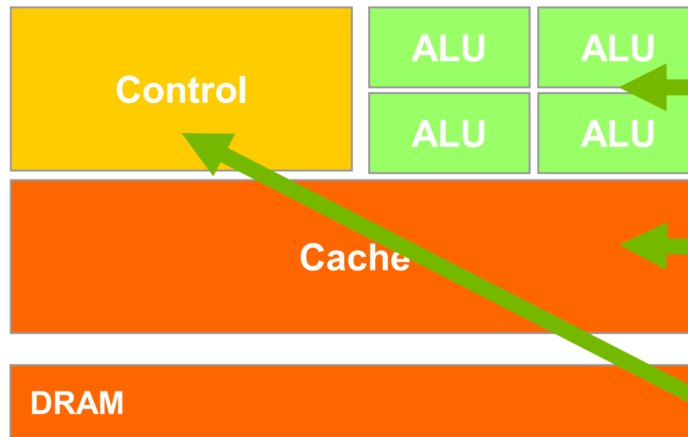
- ASCI white
  - 2001 top supercomputer in the world
  - 4.9TF/s using 8192 processors and 6 TB of memory, occupying the space of 2 basketball courts and weighing over 100 tons.

- Nvidia Tesla V100
  - released 2017
  - 7 TF/s with 5120 ALUs and 32GB of memory on a single die

CPU and GPU are designed very differently



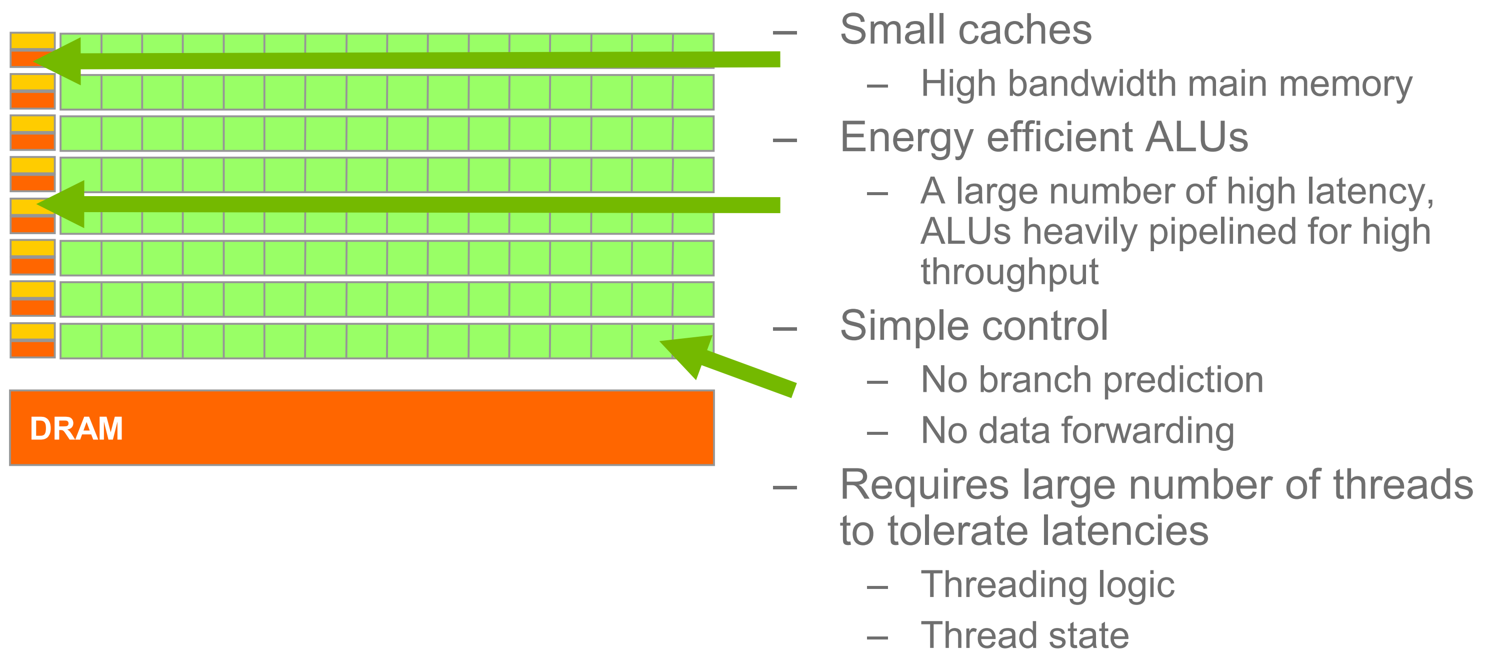
CPUs: Latency-minimizing design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Instruction dependency analysis and superscalar operation
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

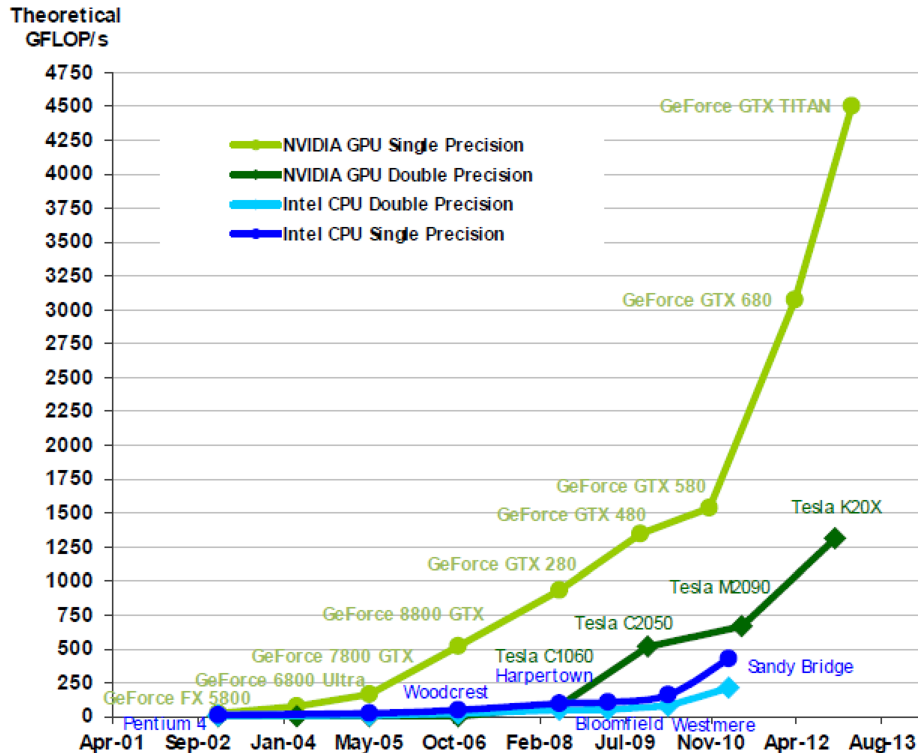
NVIDIA ILLINOIS

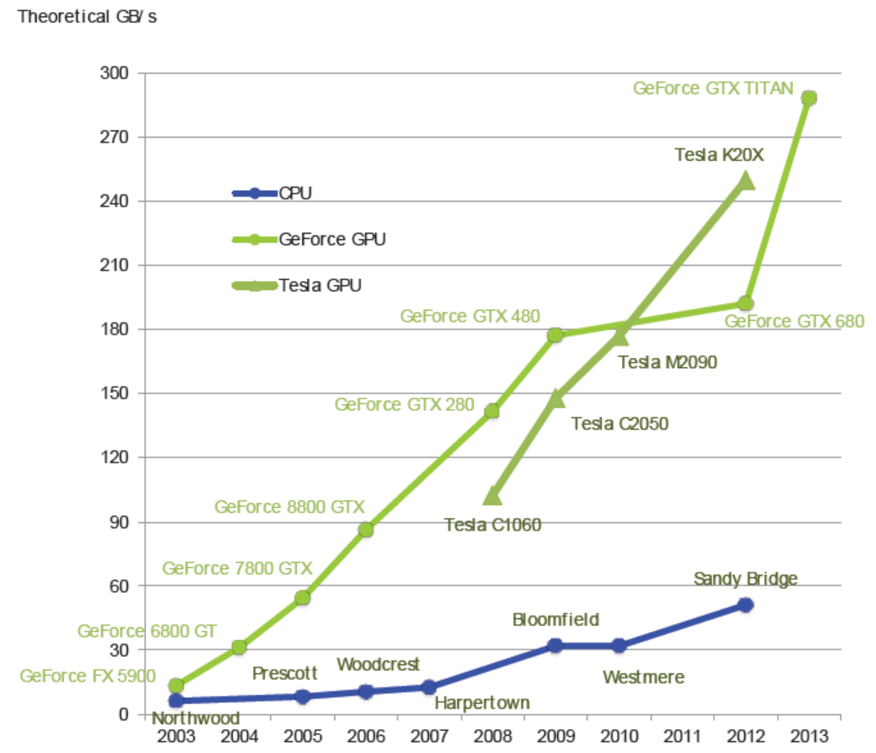
GPUs: Throughput-maximizing design



- Small caches
  - High bandwidth main memory
- Energy efficient ALUs
  - A large number of high latency, ALUs heavily pipelined for high throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Requires large number of threads to tolerate latencies
  - Threading logic
  - Thread state

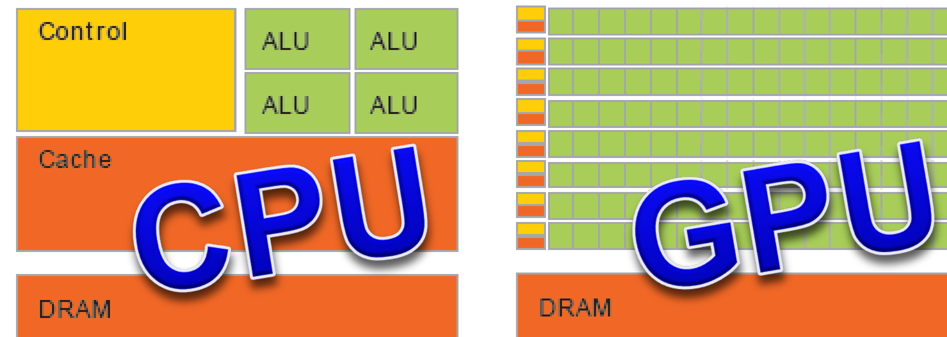
Performance Growth: GPU vs. CPU





Performance scaling has encountered major limitations

- cannot increase clock frequency
- cannot increase power
- can increase transistor count



Using accelerators in HPC systems

- Accelerators
  - generic term for compute-intensive attached devices
- Barriers
  - not general purpose, only good for some problems
  - difficult to program
  - interface to host system can be a bottleneck
  - low precision arithmetic (this is now a feature!)
- Incentives
  - cheap
  - increasingly general-purpose and simpler to program
  - improving host interfaces and performance
  - IEEE double precision
  - very high compute and local memory performance
- They are being used!
  - NSC China Tianhe-2: 48,000 Intel Xeon Phi
  - ORNL USA Summit: 27,600 Nvidia Tesla V100
- Current trends
  - Simplified access from host
  - Improved integration of multiple GPUs
  - Low- and mixed-precision FP arithmetic

Host and accelerator interface

(dual socket Intel Xeon E5 v3)

Intel Xeon Phi 5110P

16 GB/s
bidirectional

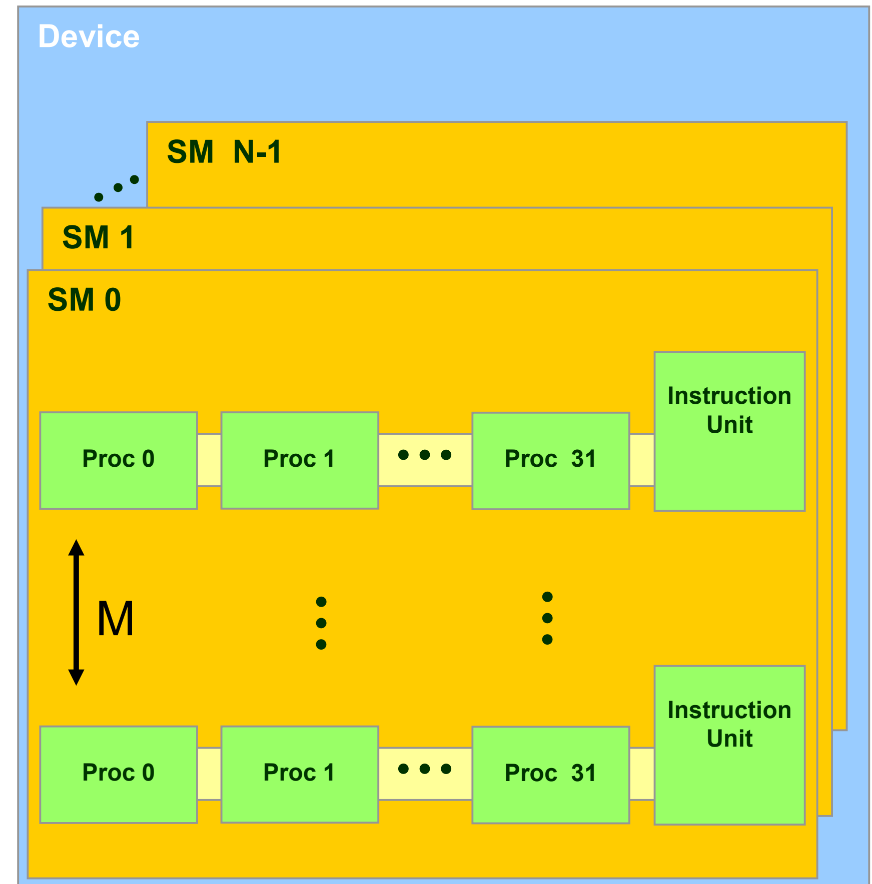
Nvidia Titan V100

Host system diagram
(Phaedra)

accelerators

Nvidia GPU organization

- GPU
  - device is a set of N (1 - 84) streaming multiprocessors (SM)
  - each SM executes one or more blocks of threads
  - each SM has M (1 - 4) sets of 32 SIMD processors
  - at each clock cycle, a SIMD processor executes a single instruction on a group of 32 threads called a warp
  - total of N * M * 32 arithmetic operations per clock
    - Volta V100 N=80, M=2
      up to 5120 SP floating point operations per clock



Volta V100 chip organization

- up to 84 SMs
- shared L2 cache (6MB)
- interfaces: 8 memory controllers, 6 NVLink intfcs, PCIe host intfc

SM

PCI Express 3.0 Host Interface

GigaThread Engine

GPC GPC GPC

Memory Controller

HBM2

L2 Cache

High-Speed Hub

NVLink NVLink NVLink NVLink NVLink NVLink

Volta V100 SM organization

- 64 single-precision FP32 arithmetic units
- 32 double-precision FP64 arithmetic units
- 64 integer arithmetic units
- 16 special function units
- 8 tensor cores (4 x 4 matrix multiply)
- 32 load/store units

- 64K registers
  - allocated across threads

- 128KB data cache / shared memory
  - L1 cache
  - user-allocated shared memory

- 4 warps can be running concurrently
  - up to 2 instructions per warp concurrently

CUDA memory hierarchy

- Host memory — Host memory
- Device memory — Device memory
  - shared between N multiprocessors
  - global, constant, and texture memory (4-32 GB total)
  - can be accessed by host
- Shared Memory — Shared Memory
  - shared by SIMD processors
  - R/W shared memory and L1 cache
  - R/O constant/texture cache
- SIMD register memory — Registers
  - set of 32-bit registers

Device

SM N-1

SM 1

SM 0

Shared memory and L1 cache

Registers | Registers | Registers

Proc 0 | Proc 1 | • • • | Proc 31

Instruction Unit

Constant Cache

Texture Cache

Host memory

Device memory

Global, constant, texture data

CUDA Control Hierarchy

- A CUDA context consists of streams
  - A stream is a sequence of kernels
    - kernels execute in sequence
    - kernels share device memory
    - different streams may run concurrently
  - A kernel is a grid of blocks
    - blocks share device memory
    - blocks are scheduled across SMs and run concurrently
  - A block is a collection of threads that
    - may access shared memory
    - can synchronize execution
    - are executed as a set of warps
  - A warp is 32 SIMD threads
    - Multiple warps may be active concurrently

Host

Stream

Kernel 1

Kernel 2

Device

Grid 1

Block (0, 0)	Block (1, 0)	Block (2, 0)
Block (0, 1)	Block (1, 1)	Block (2, 1)

Grid 2

Block (1, 1)

Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)

Execution Model

- A *grid* consists of multiple *blocks*
  - each block has a 1D, 2D, or 3D Block ID
  - a block is assigned to an SM
  - multiple blocks are required to fully utilize all SMs
    - more blocks per grid are better

- Each *block* consists of multiple *threads*
  - each thread has a 1D, 2D, or 3D Thread ID
  - threads are executed concurrently SIMD style one warp at a time
  - hardware switches between warps on any stall (e.g. load)
  - multiple threads are required to keep hardware busy
    - 64 - 1024 threads can be used to hide latency

- Each *warp* consists of 32 threads
  - execution of a warp is like the synchronous CRCW PRAM model

Compute capability

Feature	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Regs to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16/32/48 KB	96KB	64KB	config 96KB

Comparison of Nvidia Tesla GPUs

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm^2	601 mm^2	610 mm^2	815 mm^2
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

[1] Peak TFLOPS rates are based on GPU Boost Clock

CUDA Application Programming Interface

- The cuda API is an extension to the C programming language
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library split into:
    - A common component for host and device codes providing
      - built-in vector types and a
      - subset of the C runtime library
    - A host component to control and access CUDA devices
    - A device component providing device-specific functions

- Tools for cuda
  - nvcc compiler
    - runs cuda compiler on .cu files, and gcc on other files
  - nvprof profiler
    - reports on device performance including host-device transfers

CUDA C Language Extensions: Type Qualifiers

	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1

Language Extensions: Built-in Variables

- `dim3 gridDim;`
  - Dimensions of the grid in blocks
- `dim3 blockDim;`
  - Dimensions of the block in # threads
- `dim3 blockIdx;`
  - Block index within the grid
- `dim3 threadIdx;`
  - Thread index within the block

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1

Calling a Kernel Function

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);     // 5000 thread blocks
dim3   DimBlock(4, 8, 8);    // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous in the host from CUDA 1.0 on, explicit synchronization needed to await completion

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1

Host and device memory

- Separate address spaces (compute capability <6.0)
  - cudaMemCopy to move data back and forth
- Unified address space (compute capability >= 6.0)
  - host and device “page” out of a single address space
- Tesla V100 has compute capability 7.0

A simple example

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

- single block, with N threads
  - also need to allocate and initialize A and B, return C
  - easiest with unified memory model
- How large can the vectors be?
- What kind of performance could we expect?