COEN-4730/EECE-5730 Computer Architecture

Lecture 9
Introduction to Graphics Processing Units (GPUs) and CUDA

Cristinel Ababei
Dept. of Electrical and Computer Engineering
MARQUETTE UNIVERSITY
BE THE DIFFERENCE.

Credits: Slides adapted from presentations of Sudeep Pasricha and others: Kubiatowicz, Patterson, Mutlu, Elsevier

1

Flynn's Classification (1966)

Broad classification of parallel computing systems

- SISD: Single Instruction, Single Data
  - conventional uniprocessor
- SIMD: Single Instruction, Multiple Data
  - one instruction stream, multiple data paths
  - distributed memory SIMD
  - shared memory SIMD
- MIMD: Multiple Instruction, Multiple Data
  - conventional multiprocessors
  - message passing machines
  - non-cache-coherent shared memory machines
  - cache-coherent shared memory machines
- MISD: Multiple Instruction, Single Data
  - Not a practical configuration

2

Types of Parallelism

- Instruction-Level Parallelism (ILP)
  - Execute independent instructions from one instruction stream in parallel (pipelining, superscalar, VLIW)
- Thread-Level Parallelism (TLP)
  - Execute independent instruction streams in parallel (multithreading, multiple cores)
- Data-Level Parallelism (DLP)
  - Execute multiple operations of the same type in parallel (vector/SIMD execution)
- Which is easiest to program?
- Which is most flexible form of parallelism?
  - i.e., can be used in more situations
- Which is most efficient?
  - i.e., greatest tasks/second/area, lowest energy/task

3

Resurgence of DLP

- Convergence of application demands and technology constraints drives architecture choice
- New applications, such as graphics, machine vision, speech recognition, machine learning, etc. all require large numerical computations that are often trivially data parallel
- SIMD-based architectures (vector-SIMD, subword-SIMD, SIMT/GPUs) are most efficient way to execute these algorithms

4

SIMD

- Single Instruction Multiple Data (SIMD) architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

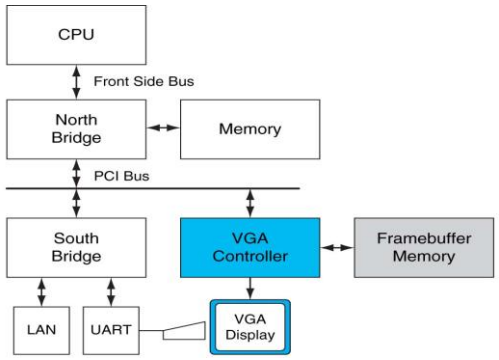
5

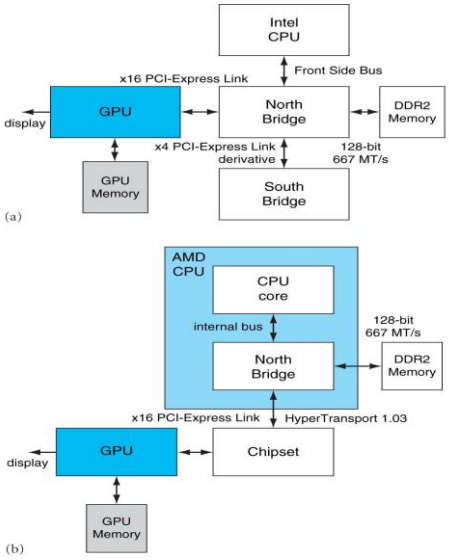
Graphics Processing Units (GPUs)

- Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  - Provide workstation-like graphics for PCs
  - Programmability was an afterthought
- Over time, more programmability added (2001-2005)
  - E.g., New language Cg ("C for graphics" from Nvidia) for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model

6

Historical PC vs. Contemporary: Intel, AMD





7

A Shift in the GPU Landscape

- Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations
- Referred to as general-purpose computing on graphics processing units (GP-GPU)
- Incredibly difficult programming model as had to use graphics pipeline model for general computation
  - A programming revolution was needed!

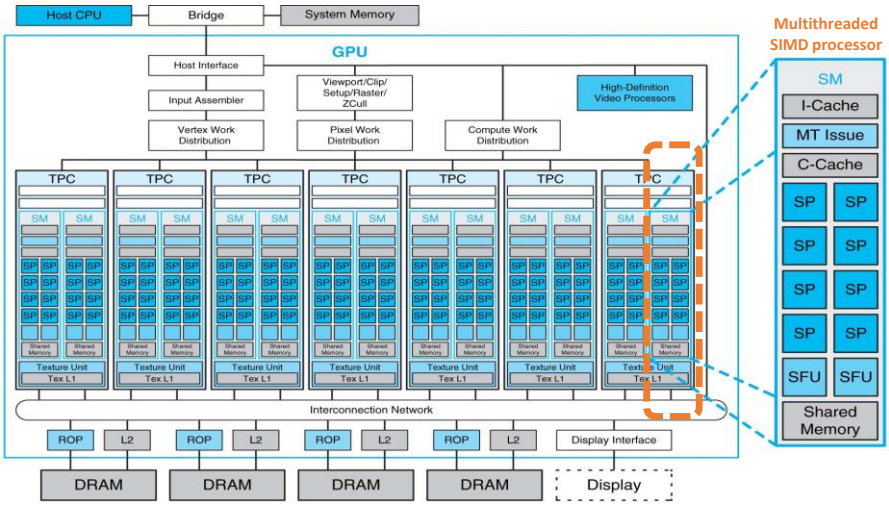
8

General-Purpose GPUs (GP-GPUs)

- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language:
  - **CUDA "Compute Unified Device Architecture"**
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution

9

Basic unified GPU architecture



Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

10

CUDA Revolution!

- CUDA Community Showcase
  - http://www.nvidia.com/object/gpu-applications.html
  - Computational fluid dynamics, EDA, finance, life sciences, signal processing, …
  - Speed-up's of >300x for some applications
- GPU Technology Conference
  - http://www.gputechconf.com/page/home.html
  - Include archive of previous editions
- Download CUDA
  - https://developer.nvidia.com/cuda-downloads
  - And start using it!
- NVIDIA YouTube Videos:
  - https://www.youtube.com/user/nvidia/videos
- Many universities have already courses dedicated to teaching and using CUDA for research

11

CUDA Programming Model

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
    - » Hardware switching between threads (in 1 cycle) on long-latency memory reference
    - » Overprovision (1000s of threads) → hide latencies
- Data-parallel portions of an application are executed on the device as Kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - » Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - » Multi-core CPU needs only a few

12

Basic Steps

- **Device allocation, CPU-GPU transfer, and GPU-CPU transfer**
  - `cudaMalloc();`
  - `cudaMemcpy();`

```
// Allocate input
malloc(input, ...);
cudaMalloc(d_input, ...);
cudaMemcpy(d_input, input, ..., HostToDevice); // Copy to device memory

// Allocate output
malloc(output, ...);
cudaMalloc(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();

// Copy output to host memory
cudaMemcpy(output, d_output, ..., DeviceToHost);
```

13

Example 1: Vector Addition kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

14

Example 2: Changing an Array

- The code has been divided into two files:
  - simple.c
  - simple.cu
- simple.c is ordinary code in C
- It allocates an array of integers, initializes it to values corresponding to the indices in the array and prints the array
- It calls a function that modifies the array
- The array is printed again

15

simple.c

```
#include <stdio.h>

#define SIZEOFARRAY 64
extern void fillArray(int *a,int size);

/* The main program */
int main(int argc,char *argv[])
{
  /* Declare the array that will be modified by the GPU */
  int a[SIZEOFARRAY];
  int i;
  /* Initialize the array */
  for(i=0; i < SIZEOFARRAY; i++) {
    a[i]=i;
  }
  /* Print the initial array */
  printf("Initial state of the array:\n");
  for(i = 0; i < SIZEOFARRAY; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");

  /* Call the function that will in turn call the function in
     the GPU that will fill the array */
  fillArray(a,SIZEOFARRAY);

  /* Now print the array after calling fillArray */
  printf("Final state of the array:\n");
  for(i = 0; i < SIZEOFARRAY; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");
  return 0;
}
```

16

simple.cu

- simple.cu contains two functions
  1. fillArray(): A function that will be executed on the host and which takes care of:
     - Allocating variables in the global GPU memory
     - Copying the array from the host to the GPU memory
     - Setting the grid and block sizes
     - Invoking the kernel that is executed on the GPU
     - Copying the values back to the host memory
     - Freeing the GPU memory

17

fillArray (part 1)

```
#define BLOCK_SIZE 32

extern "C" void fillArray(int *array, int arraySize)
{
  /* array_d is the GPU counterpart of the array that
     exists on the host memory */
  int *array_d;
  cudaError_t result;

  /* allocate memory on device */
  /* cudaMalloc allocates space in memory of GPU card */
  result = cudaMalloc((void**)&array_d, sizeof(int)*arraySize);

  /* copy array into the variable array_d in the device */
  /* The memory from the host is being copied
     to corresponding variable in the GPU global memory */
  result = cudaMemcpy(array_d,array, sizeof(int)*arraySize,
           cudaMemcpyHostToDevice);
```

18

fillArray (part 2)

```
/* execution configuration... */
/* Indicate the dimension of the block */
dim3 dimblock(BLOCK_SIZE);

/* Indicate the dimension of the grid in blocks */
dim3 dimgrid(arraySize/BLOCK_SIZE);

/* actual computation: Call the kernel, the
   function that is executed by each and every
   processing element on the GPU card */
cu_fillArray<<<dimgrid,dimblock>>>(array_d);

/* read results back: */
/* Copy results from GPU back to memory on the host */
result = cudaMemcpy(array, array_d, sizeof(int)*arraySize,
          cudaMemcpyDeviceToHost);

/* Release the memory on the GPU card */
cudaFree(array_d);
}
```

19

simple.cu (cont.)

- The other function in simple.cu is
  2. cu_fillArray():
     - This is the kernel that will be executed in every stream processor in the GPU
     - It is identified as a kernel by the use of the keyword: __global__
     - This function uses the built-in variables
       - blockIdx.x
       - threadIdx.x

     to identify a particular position in the array

20

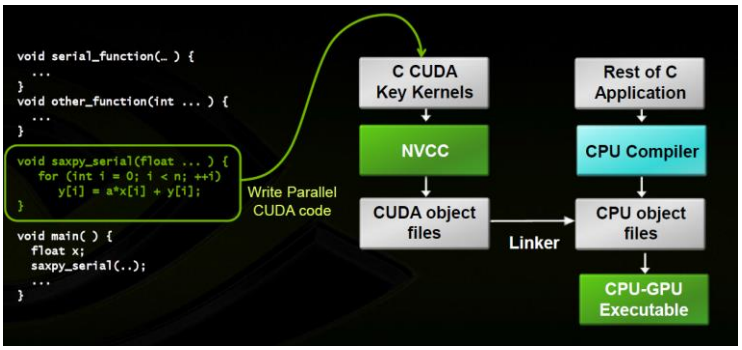
cu_fillArray

```
__global__ void cu_fillArray(int *array_d)
{
  int x;

  /* blockIdx.x is a built-in variable in CUDA
     that returns the blockId in the x axis
     of the block that is executing this block of code
     threadIdx.x is another built-in variable in CUDA
     that returns the threadId in the x axis
     of the thread that is being executed by this
     stream processor in this particular block
  */

  x = blockIdx.x*BLOCK_SIZE + threadIdx.x;
  array_d[x] += array_d[x];
}
```

21

CUDA Compilation

- % nvcc simple.c simple.cu -o simple
- CPU code is compiled by the host C compiler and the GPU code (kernel) is compiled by the CUDA compiler
- Separate binaries are produced



22

OpenCL – Open Compute Language

- CUDA alternative
- Developed by Khronos
  - Industry Consortium that includes: AMD, ARM, Intel, and NVIDIA
- Designed as an open standard for cross-platform parallel programming
- Allows for more general programming across multiple GPUs/CPUs
- Problems: Not as well developed/specialized as CUDA

	OpenCL	CUDA
Programming Language	C	C/C++
Supported GPUs	AMD, NVIDIA	NVIDIA
Supported CPUs	AMD, Intel, ARM	None
Method of Creating GPU Work	Kernel	Kernel
Run-time compilation of kernels	Yes	No
Multiple Kernel Execution	Yes (in certain hardware)	Yes (in certain hardware)
Execution Across Multiple Components	Yes	Yes – only GPUs
Need to Optimize for Best Performance	High	High
Coding Complexity	High	Medium

23

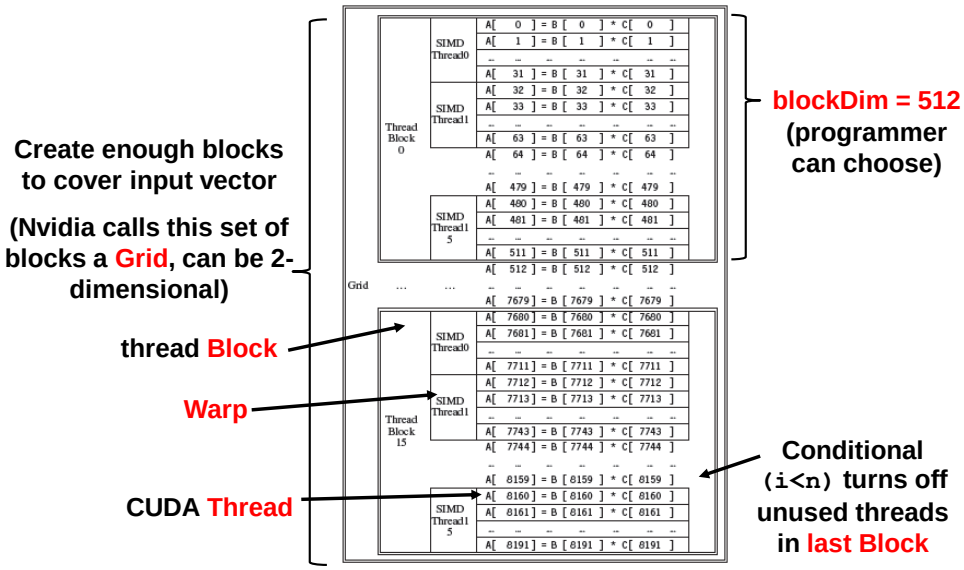
Quick guide to GPU terms

A major obstacle to understanding GPUs has been the jargon, with some terms even having misleading names. This obstacle has been surprisingly difficult to overcome.

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

24

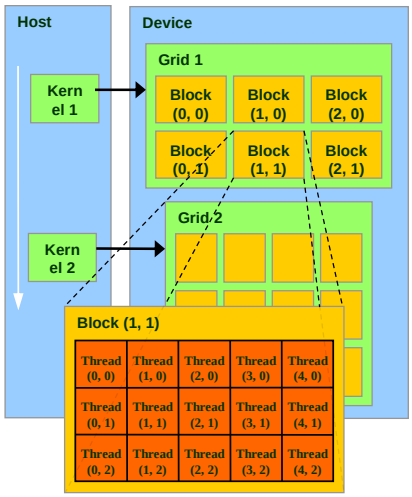
Programmer's View of Execution



25

Thread Batching: Grids and Blocks

- Kernel executed as a grid of thread blocks
  - All threads share data memory space
- Thread block is a batch of threads, can cooperate with each other by:
  - Synchronizing their execution: For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
  - (Unless thru slow global memory)
- Threads and blocks have IDs

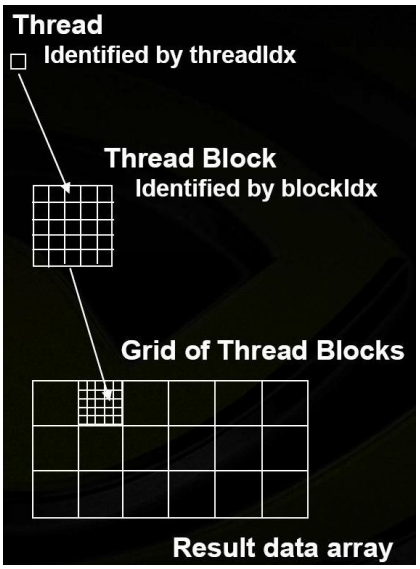


26

Execution Model

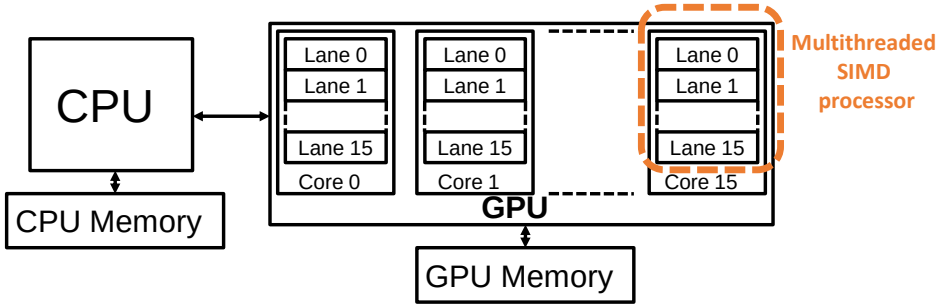
Multiple levels of parallelism

- **Thread block**
  - Max. 1024 threads/block
  - Communication through shared memory (fast)
  - Thread guaranteed to be resident
  - threadIdx, blockIdx
- **Grid of thread blocks**
  - F<<<nblocks, nthreads>>>(a, b, c)



27

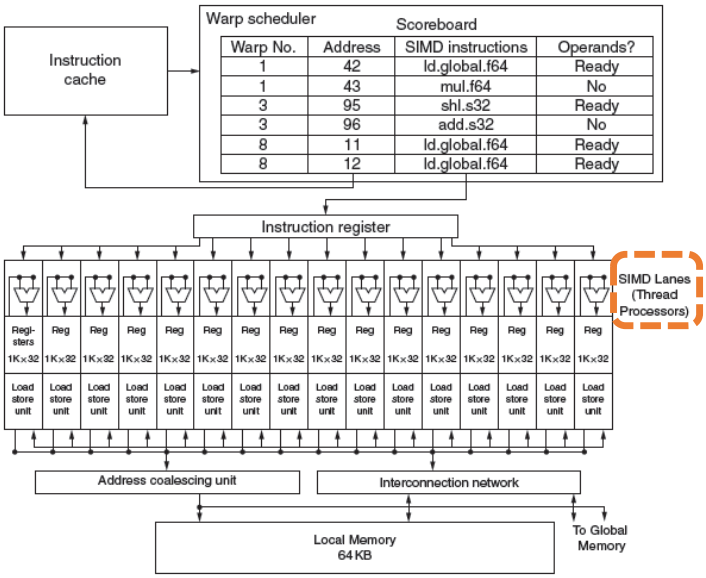
Hardware Execution Model



- GPU is built from multiple parallel **cores**, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
- CPU sends whole "grid" (i.e., vectorizable loop) over to GPU, which distributes thread blocks among cores (each thread block executes on one core)
  - Programmer unaware of number of cores

28

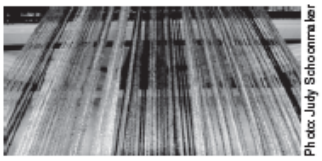
Simplified Diagram of Multithreaded SIMD Processor

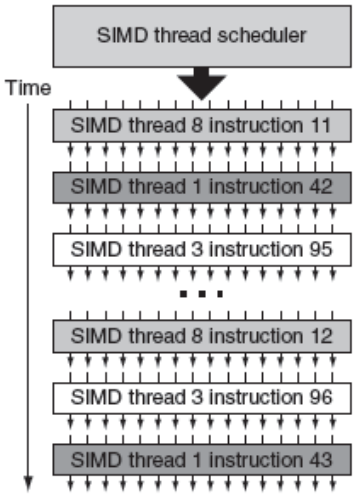


29

CUDA Thread Scheduling

- GPU hardware has two levels of hardware schedulers:
  1) **Thread Block Scheduler (top level)** that assigns Thread Blocks to multithreaded SIMD processors, which ensures that thread blocks are assigned to the processors whose local memories have corresponding data
  2) **SIMD Thread Scheduler (lower level)** (warp scheduler) within a SIMD Processor, which schedules when threads of SIMD instructions should run

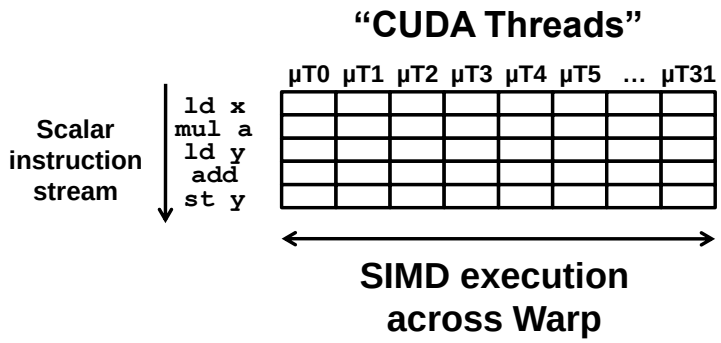




30

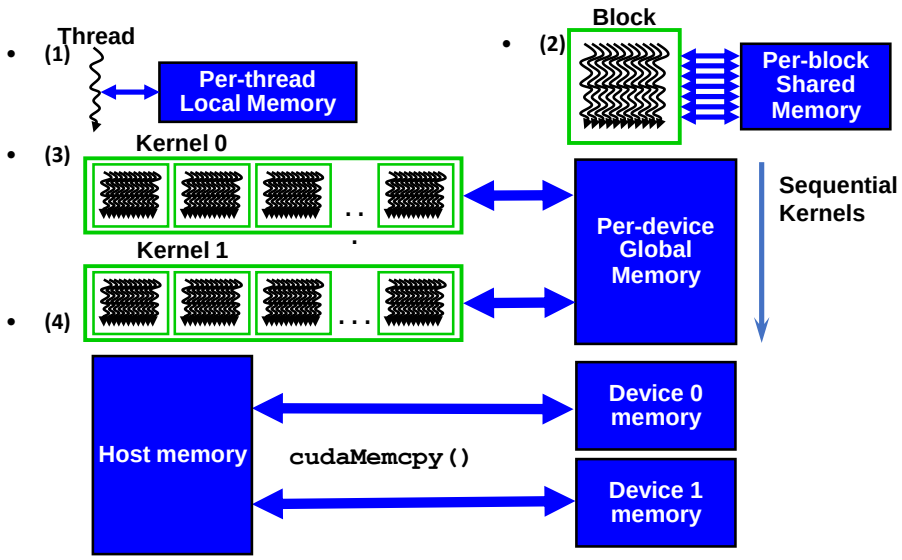
"Single Instruction, Multiple Thread"

- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (NVIDIA groups 32 CUDA threads into a Warp)



31

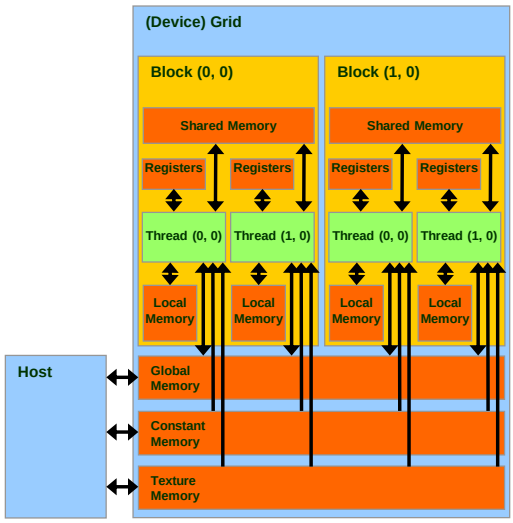
GPU Memory Hierarchy



32

CUDA Device Memory Space Overview

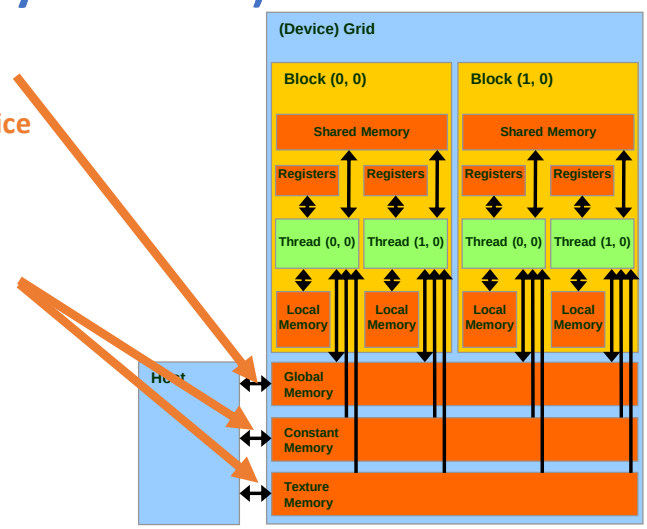
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



33

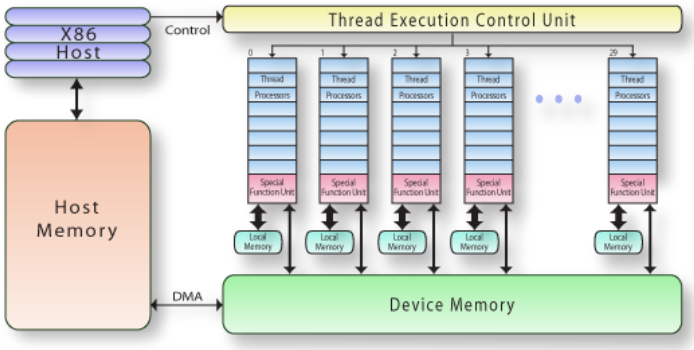
Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



34

Example: Tesla Architecture



- Used for Technical and Scientific Computing
- L1/L2 Data Cache
  - Allows for caching of global and local data
  - Same on-chip memory used for Shared and L1
  - Configurable at kernel invocation

35

Example: Nvidia Tesla K20

TECHNICAL SPECIFICATIONS	TESLA K10[a]	TESLA K20	TESLA K20X
Peak double precision floating point performance (board)	0.19 teraflops	1.17 teraflops	1.31 teraflops
Peak single precision floating point performance (board)	4.58 teraflops	3.52 teraflops	3.95 teraflops
Number of GPUs	2 x GK104s	1 x GK110	
Number of CUDA cores	2 x 1536	2496	2688
Memory size per board (GDDR5)	8 GB	5 GB	6 GB
Memory bandwidth for board (ECC off)[b]	320 GBytes/sec	208 GBytes/sec	250 GBytes/sec
GPU computing applications	Seismic, image, signal processing, video analytics	CFD, CAE, financial computing, computational chemistry and physics, data analytics, satellite imaging, weather modeling	
Architecture features	SMX	SMX, Dynamic Parallelism, Hyper-Q	
System	Servers only	Servers and Workstations	Servers on

https://www.nvidia.com/content/PDF/kepler/tesla-k20-active-bd-06499-001-v03.pdf

Graphics Processor	
GPU Name:	GK110
Process Size:	28 nm
Transistors:	7,080 million
Die Size:	561 mm²

Render Config	
Shading Units:	2496
TMUs:	208
ROPs:	40
SMX Count:	13
Pixel Rate:	36.7 GPixel/s
Texture Rate:	147 GTexel/s
Floating-point performance:	3,524 GFLOPS

Graphics Card	
Released:	Nov 12th, 2012
Production Status:	Active
Launch Price:	3,199 USD
Bus Interface:	PCIe 2.0 x16

Reference Board	
Slot Width:	Dual-slot
Length:	10.5 inches 267 mm
TDP:	225 W

VGA BIOS

Find graphics card BIOS for this card.

GPU-Z Validation

Find GPU-Z validations for this card.

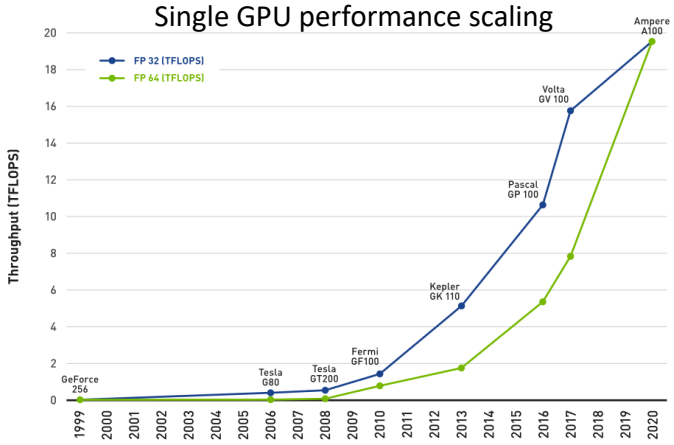
Clock Speeds	
GPU Clock:	706 MHz
Memory Clock:	1300 MHz 5200 MHz effective

Memory	
Memory Size:	5120 MB
Memory Type:	GDDR5
Memory Bus:	320 bit
Bandwidth:	208 GB/s

Graphics Features	
DirectX:	11.0
OpenGL:	4.4
OpenCL:	1.1
Shader Model:	5.0

36

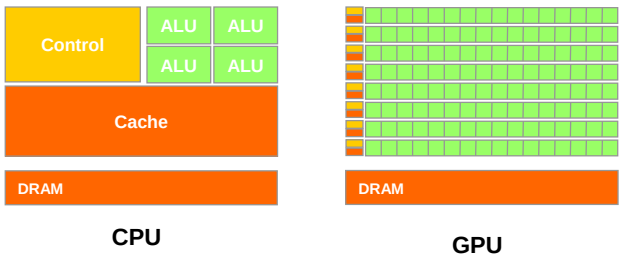
Evolution of NVIDIA GPUs



Source: W. J. Dally, S. W. Keckler and D. B. Kirk, "Evolution of the Graphics Processing Unit (GPU)," in IEEE Micro, vol. 41, no. 6, pp. 42-51, 1 Nov.-Dec. 2021, doi: 10.1109/MM.2021.3113475.

37

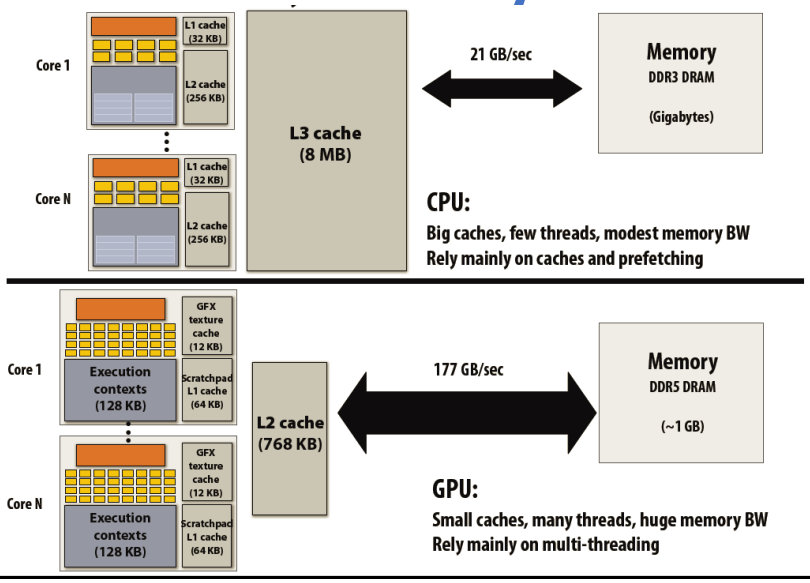
CPU vs. GPU



- GPU
  - More transistors devoted to computation, instead of caching or flow control
  - Suitable for data-intensive computation
    - » High arithmetic/memory operation ratio

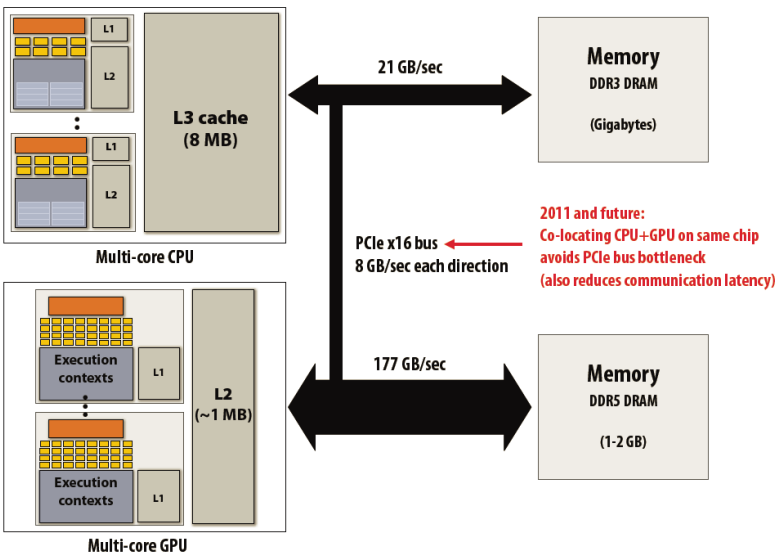
38

CPU vs. GPU memory hierarchies



39

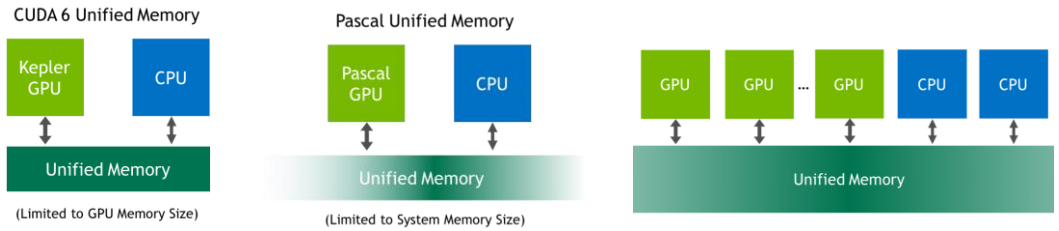
Entire system view: CPU + discrete GPU



40

Unified Memory

- **Unified Virtual Address**
- **Since CUDA 6.0: Unified memory**
- **Since CUDA 8.0 + Pascal: GPU page faults**



More information:
https://developer.nvidia.com/blog/unified-memory-in-cuda-6/
https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf
https://www.olcf.ornl.gov/wp-content/uploads/2019/06/06_Managed_Memory.pdf

41

Unified Memory

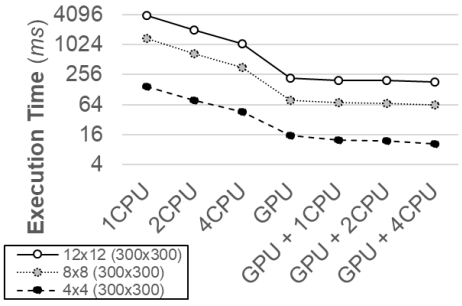
- **Simpler Programming and Memory Model**
- **Performance Through Data Locality**
  - `cudaMallocManaged();`

```
// Allocate input
malloc(input, ...);
cudaMallocManaged(d_input, ...);
memcpy(d_input, input, ...); // Copy to managed memory

// Allocate output
cudaMallocManaged(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();
```

42

Collaborative Computing Algorithms

- Case studies using CPU and GPU
- Kernel launches are asynchronous
  - CPU can work while waits for GPU to finish
  - Traditionally, this is the most efficient way to exploit heterogeneity
- Fine-grain heterogeneity becomes possible with Pascal/Volta architecture
- Pascal/Volta Unified Memory
  - CPU-GPU memory coherence
  - System-wide atomic operations
- Benefits of Collaboration - Example: Bézier Surfaces
  - [1] J. Gomez-Luna et. al, Chai: Collaborative Heterogeneous Applications for Integrated-architectures, ISPASS 2017.
  - Data partitioning improves performance
    - AMD Kaveri (4 CPU cores + 8 GPU CUs)



Bézier Surfaces
(up to 47% improvement over GPU only)

43

GPUs for mobile clients and servers

Goal is for the graphics quality of a movie such as *Avatar* to be achieved in real time on a server GPU in 2015 and on your mobile GPU in 2020

	NVIDIA Tegra 2	NVIDIA Fermi GTX 480
Market	Mobile client	Desktop, server
System processor	Dual-Core ARM Cortex-A9	Not applicable
System interface	Not applicable	PCI Express 2.0 × 16
System interface bandwidth	Not applicable	6 GBytes/sec (each direction), 12 GBytes/sec (total)
Clock rate	Up to 1 GHz	1.4 GHz
SIMD multiprocessors	Unavailable	15
SIMD lanes/SIMD multiprocessor	Unavailable	32
Memory interface	32-bit LP-DDR2/DDR2	384-bit GDDR5
Memory bandwidth	2.7 GBytes/sec	177 GBytes/sec
Memory capacity	1 GByte	1.5 GBytes
Transistors	242 M	3030 M
Process	40 nm TSMC process G	40 nm TSMC process G
Die area	57 mm^2	520 mm^2
Power	1.5 watts	167 watts

44

Comparison of a GPU and a MIMD with Multimedia SIMD

Purpose is *not* to determine how much faster one product is than another, but to try to understand the relative value of features of these two contrasting architecture styles

	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TSMC 65 nm	TSMC 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3030 M	2.0	4.4
Memory bandwidth (GBytes/sec)	32	141	177	4.4	5.5
Single-precision SIMD width	4	8	32	2.0	8.0
Double-precision SIMD width	2	1	16	0.5	8.0
Peak single-precision scalar FLOPS (GFLOP/Sec)	26	117	63	4.6	2.5
Peak single-precision SIMD FLOPS (GFLOP/Sec)	102	311 to 933	515 or 1344	3.0–9.1	6.6–13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused multiply-adds)	N.A.	(622)	(1344)	(6.1)	(13.1)
(Rare SP dual issue fused multiply-add and multiply)	N.A.	(933)	N.A.	(9.1)	--
Peak double-precision SIMD FLOPS (GFLOP/sec)	51	78	515	1.5	10.1

45

Relative Performance

Kernel	Units	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOP/sec	94	364	3.9
MC	Billion paths/sec	0.8	1.4	1.8
Conv	Million pixels/sec	1250	3500	2.8
FFT	GFLOP/sec	71.4	213	3.0
SAXPY	GBytes/sec	16.8	88.8	5.3
LBM	Million lookups/sec	85	426	5.0
Solv	Frames/sec	103	52	0.5
SpMV	GFLOP/sec	4.9	9.1	1.9
GJK	Frames/sec	67	1020	15.2
Sort	Million elements/sec	250	198	0.8
RC	Frames/sec	5	8.1	1.6
Search	Million queries/sec	50	90	1.8
Hist	Million pixels/sec	1517	2583	1.7
Bilat	Million pixels/sec	83	475	5.7

46

Reasons for differences from Intel

- GPU has 4.4× the memory bandwidth
  - Explains why LBM and SAXPY run 5.0 and 5.3× faster; their working sets are hundreds of megabytes and hence don't fit into the Core i7 cache
- Five of the remaining kernels are compute bound:
  - SGEMM, Conv, FFT, MC, and Bilat
  - GTX 280 single precision is 3 to 6× faster; DP performance is only 1.5× faster; has direct support for transcendental functions lacking in i7
- Cache blocking optimizations benefit i7
  - Convert RC, Search, Sort, SGEMM, FFT, and SpMV from memory bound to compute bound
- Multimedia SIMD extensions are of little help if the data are scattered throughout main memory
  - Reinforces the importance of gather-scatter to vector and GPU architectures that is missing from SIMD extensions

47

Conclusion

- GPU: A type of Vector Processor originally optimized for graphics processing
  - Has become general purpose with introduction of CUDA
  - "CUDA Threads" grouped into "Warps" automatically (32 threads)
  - "Thread-Blocks" (with up to 512 CUDA Threads) dynamically assigned to processors (since number of processors/system varies)
- High-end desktops have separate GPU chip, but trend towards integrating GPU on same die as CPU
  - Advantage is shared memory with CPU, no need to transfer data
  - Disadvantage is reduced memory bandwidth compared to dedicated smaller-capacity specialized memory system
    - Graphics DRAM (GDDR) versus regular DRAM (DDR3)
- Unified Memory
  - Collaborative computing

48