

---

# **Lecture 22: Data Level Parallelism -- Graphical Processing Unit (GPU) and Loop- Level Parallelism**

**CSCE 513 Computer Architecture**

**Department of Computer Science and Engineering**

**Yonghong Yan**

**[yanyh@cse.sc.edu](mailto:yanyh@cse.sc.edu)**

**<https://passlab.github.io/CSCE513>**

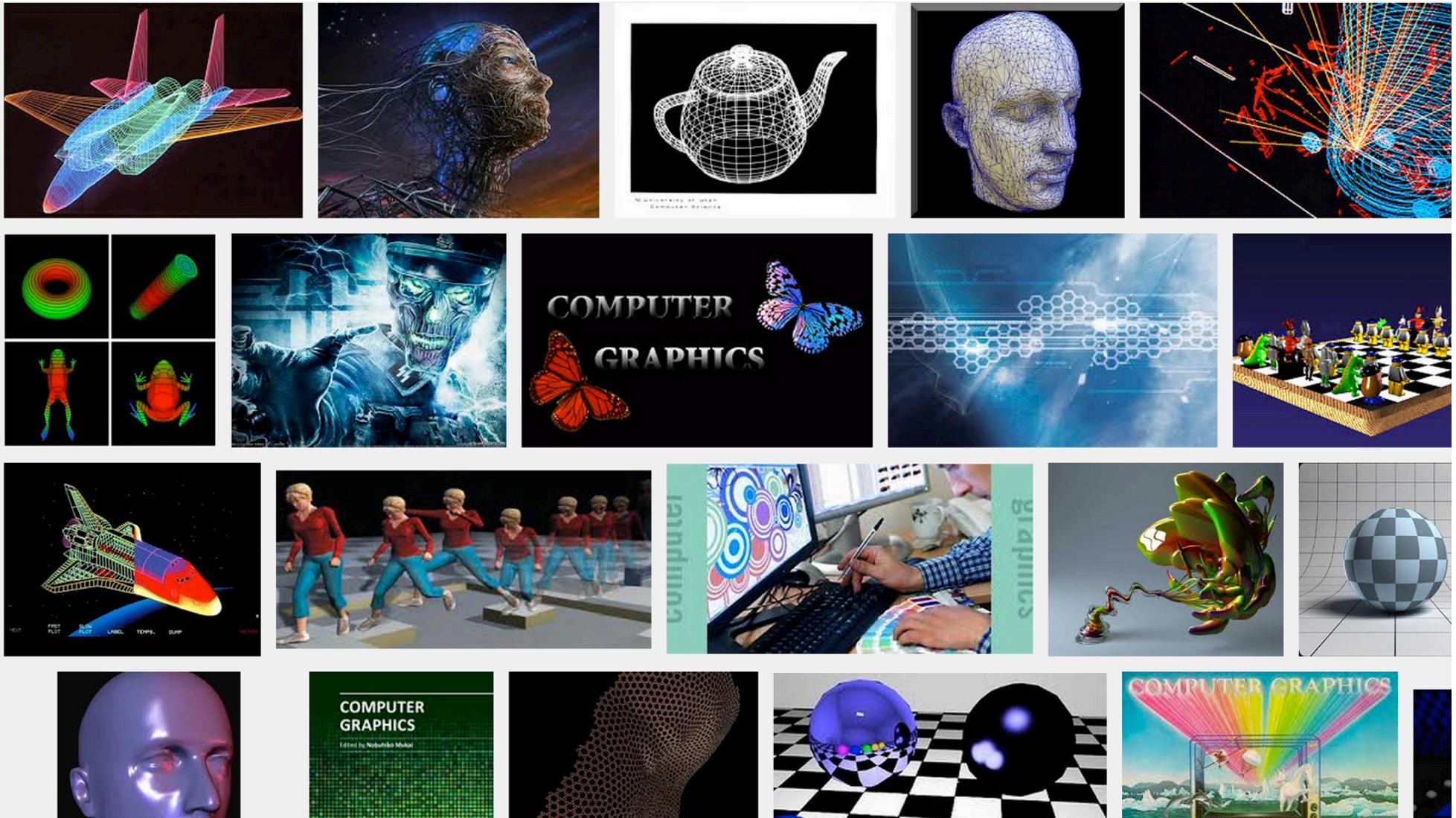
---

# Topics for Data Level Parallelism (DLP)

---

- Parallelism (centered around ... )
  - Instruction Level Parallelism
  - Data Level Parallelism
  - Thread Level Parallelism
- DLP Introduction and Vector Architecture
  - 4.1, 4.2
- SIMD Instruction Set Extensions for Multimedia
  - 4.3
- Graphical Processing Units (GPU)
  - 4.4
- GPU and Loop-Level Parallelism and Others
  - 4.4, 4.5

# Computer Graphics



# Graphics Processing Unit (GPU)

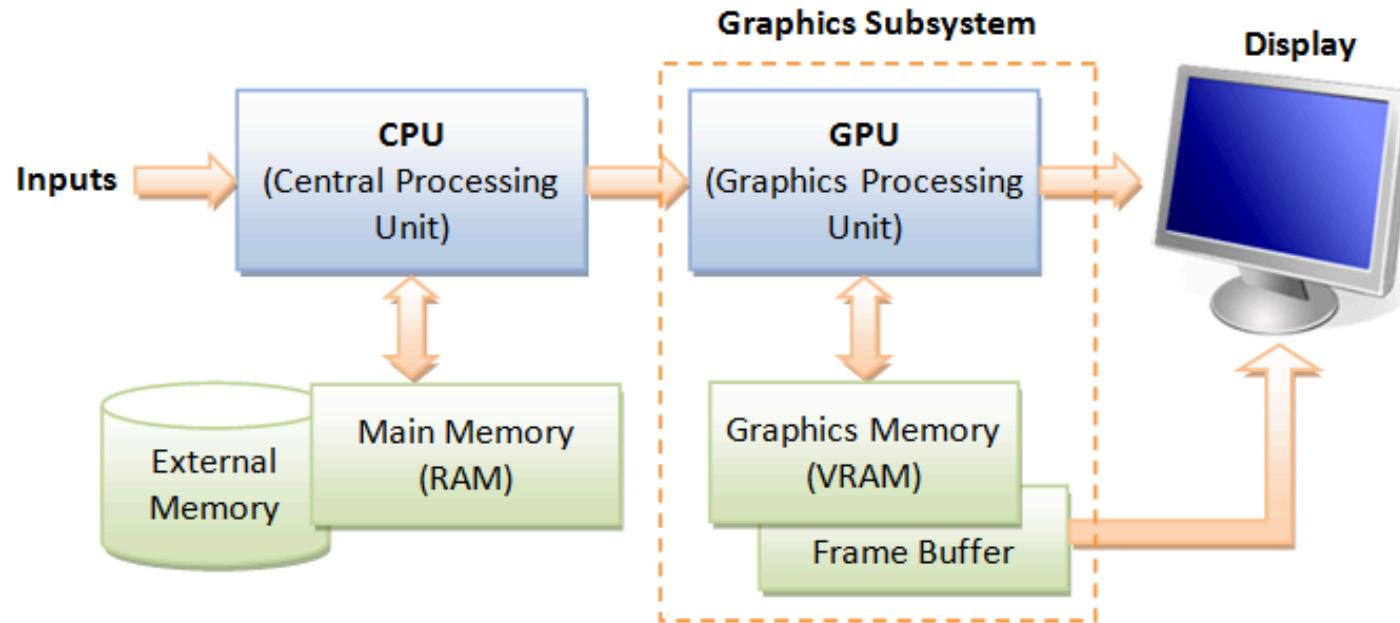
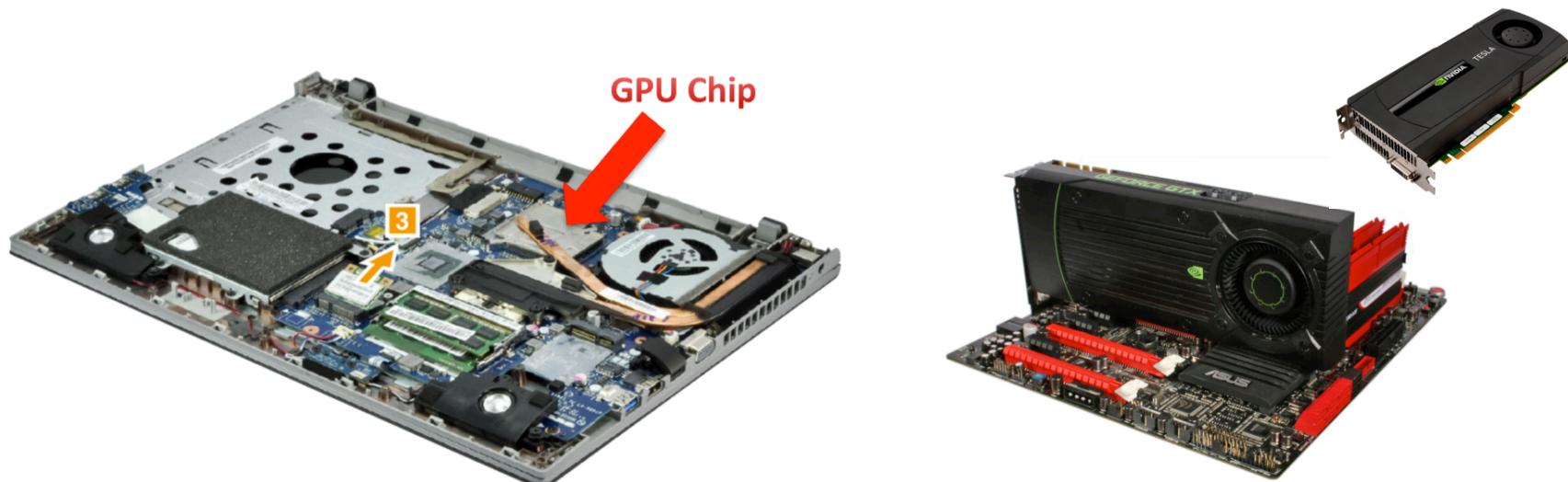
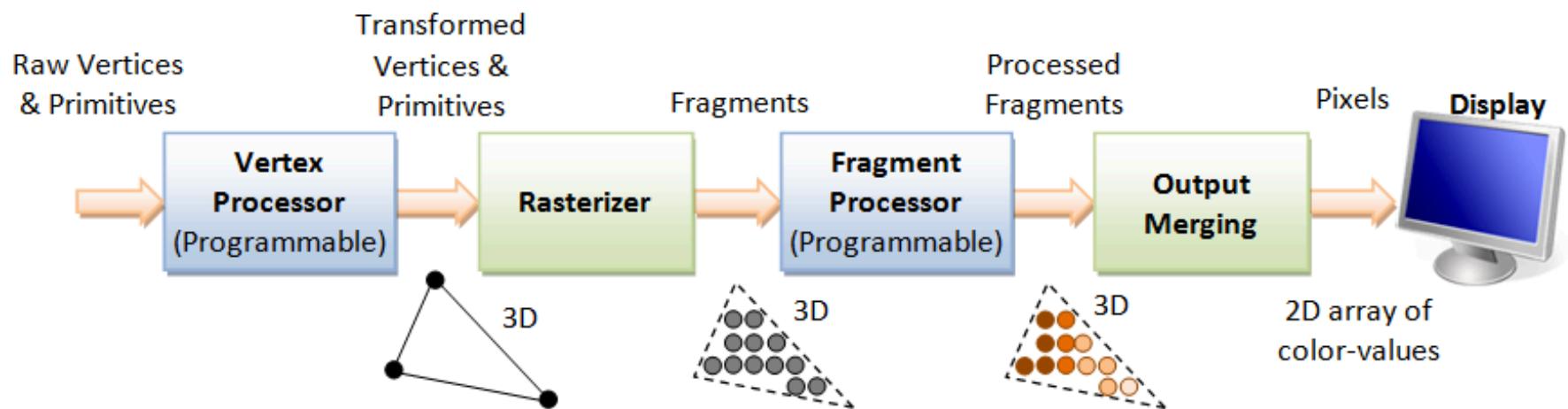


Image: [http://www.ntu.edu.sg/home/ehchua/programming/opengl/CG\\_BasicsTheory.html](http://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html)



# Recent GPU Architecture

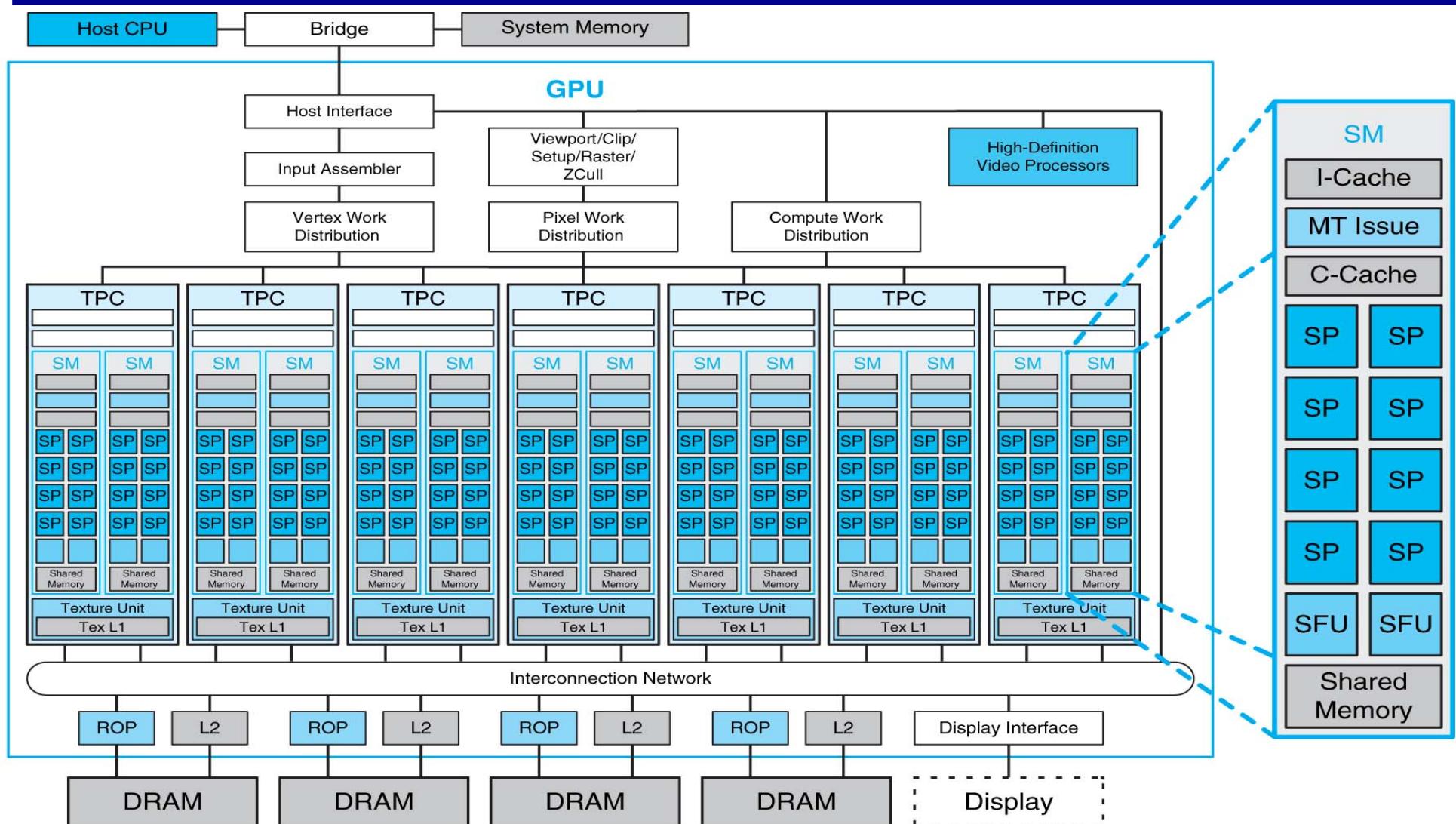
- Unified Scalar Shader Architecture
- Highly Data Parallel Stream Processing



**3D Graphics Rendering Pipeline:** Output of one stage is fed as input of the next stage. A vertex has attributes such as  $(x, y, z)$  position, color (RGB or RGBA), vertex-normal  $(n_x, n_y, n_z)$ , and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Image: [http://www.ntu.edu.sg/home/ehchua/programming/opengl/CG\\_BasicsTheory.html](http://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html)

# Unified Shader Architecture



**FIGURE A.2.5 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 6 shared memory. Copyright © 2009 Elsevier, Inc. All rights reserved.

# GPU Today

---

- It is a processor optimized for 2D/3D graphics, video, visual computing, and display.
- It is highly parallel, highly multithreaded multiprocessor optimized for visual computing.
- It provides real-time visual interaction with computed objects via graphics images, and video.
- It serves as both a programmable graphics processor and a scalable parallel computing platform.
  - Heterogeneous systems: combine a GPU with a CPU
- It is called as **Many-core**

# Latest NVIDIA Volta GV100 GPU

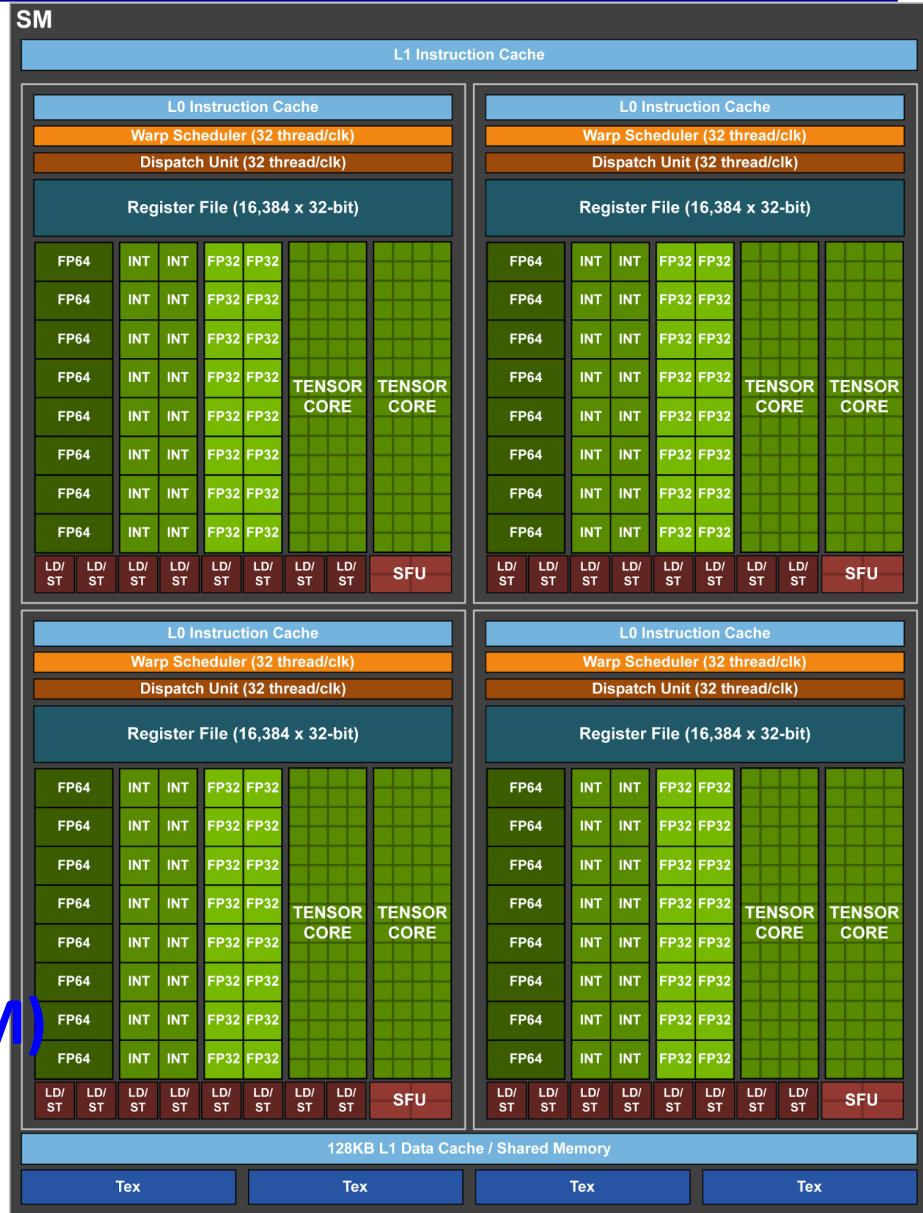
- Released May 2017
  - Total 84 Stream Multiprocessors (SM)
- Cores
  - 5120 FP32 cores
    - Can do FP16 also
  - 2560 FP64 cores
  - 640 Tensor cores
- Memory
  - 16G HBM2
  - L2: 6144 KB
  - Shared memory: 96KB \* 80 (SM)
  - Register File: 20,480 KB (Huge)



<https://devblogs.nvidia.com/parallelforall/inside-volta/>

# SM of Volta GPU

- Released May 2017
  - Total 84 SM
- Cores
  - 5120 FP32 cores
    - Can do FP16 also
  - 2560 FP64 cores
  - 640 Tensor cores
- Memory
  - 16G HBM2
  - L2: 6144 KB
  - Shared memory: 96KB \* 80 (SM)
  - Register File: 20,480 KB (Huge)



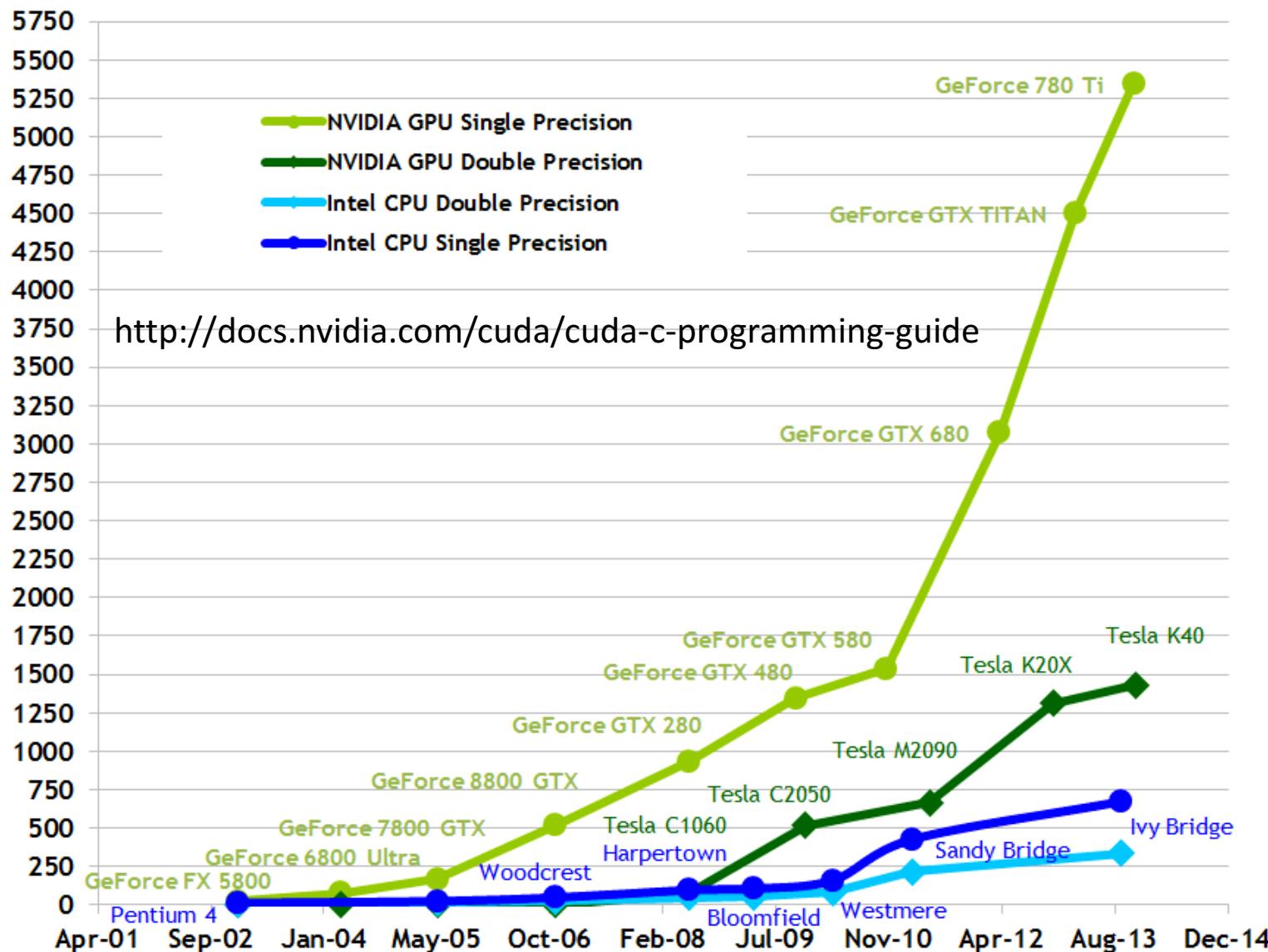
# SM of Volta GPU

- Released May 2017
  - Total 84 SM
- Cores
  - 5120 FP32 cores
    - Can do FP16 also
  - 2560 FP64 cores
  - 640 Tensor cores
- Memory
  - 16G HBM2
  - L2: 6144 KB
  - Shared memory: 96KB \* 80 (SM)
  - Register File: 20,480 KB (Huge)



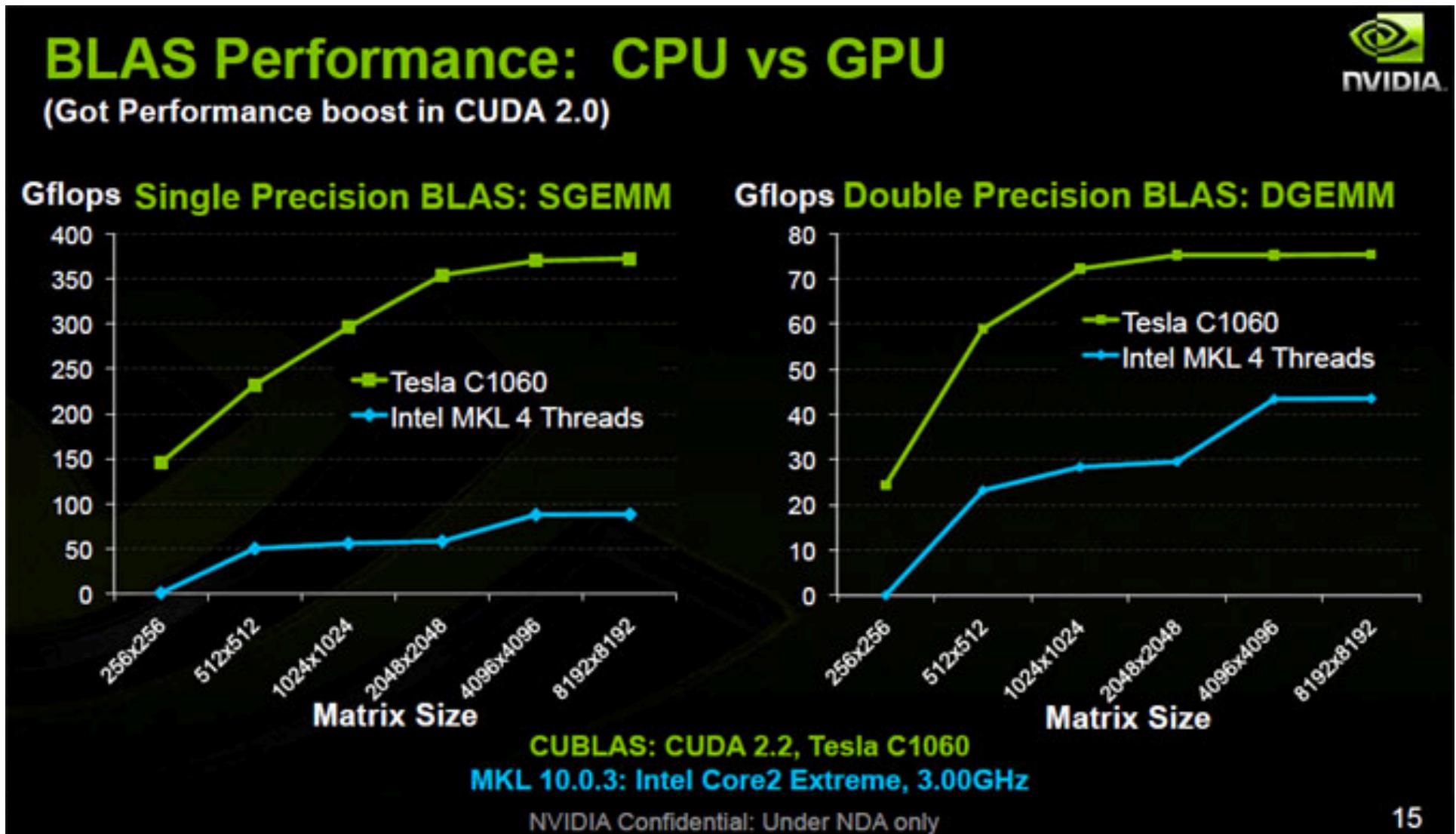
# GPU Performance Gains Over CPU

Theoretical GFLOP/s



<http://docs.nvidia.com/cuda/cuda-c-programming-guide>

# GPU Performance Gains Over CPU



# Programming for NVIDIA GPUs

GPU Computing Applications						
Libraries and Middleware						
CUFFT CUBLAS CURAND CUSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX	iray	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
CUDA-Enabled NVIDIA GPUs						
Kepler Architecture (compute capabilities 3.x)	GeForce 600 Series	Quadro Kepler Series	Tesla K20 Tesla K10			
Fermi Architecture (compute capabilities 2.x)	GeForce 500 Series GeForce 400 Series	Quadro Fermi Series	Tesla 20 Series			
Tesla Architecture (compute capabilities 1.x)	GeForce 200 Series GeForce 9 Series GeForce 8 Series	Quadro FX Series Quadro Plex Series Quadro NVS Series	Tesla 10 Series			
<a href="http://docs.nvidia.com/cuda/cuda-c-programming-guide/">http://docs.nvidia.com/cuda/cuda-c-programming-guide/</a>		 Entertainment	 Professional Graphics	 High Performance Computing		

# CUDA(Compute Unified Device Architecture)

---

Both an *architecture* and *programming model*

- Architecture and execution model
  - Introduced in NVIDIA in 2007
  - Get highest possible execution performance requires understanding of hardware architecture
- Programming model
  - Small set of extensions to C
  - Enables GPUs to execute programs written in C
  - Within C programs, call SIMD “kernel” routines that are executed on GPU.

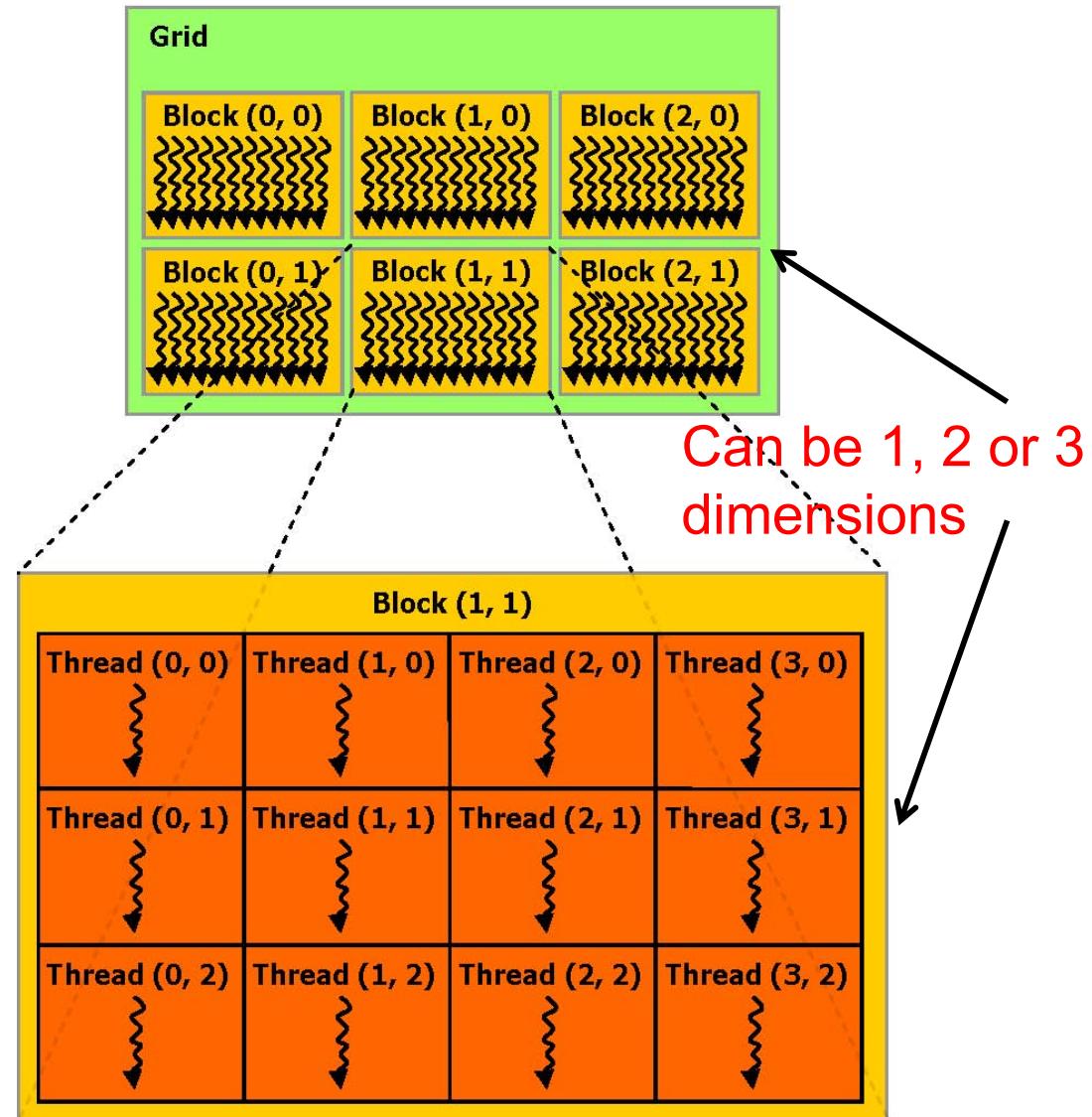
# CUDA Thread

---

- Parallelism in Vector/SIMD is the combination of lanes (# PUs) and vector length
- CUDA thread is a unified term that abstract the parallelism for both programmers and GPU execution model
  - **Programmer: A CUDA thread performs operations for one data element (think of this way as of now)**
    - **There could be thousands or millions of threads**
  - **A CUDA thread represents a hardware FU**
    - **GPU calls it a core (much simpler than a conventional CPU core)**
- **Hardware-level parallelism is more explicit**

# CUDA Thread Hierarchy:

- Allows **flexibility** and **efficiency** in **processing 1D, 2-D, and 3-D data on GPU.**
- Linked to **internal organization**
- **Threads in one block execute together.**



# DAXPY

```
// DAXPY in CUDA
```

```
__global__
```

```
void daxpy(int n, double a, double *x, double *y) {  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if (i < n) y[i] = a*x[i] + y[i];  
}
```

*Each thread finds its element to compute and do the work.*

```
// Invoke DAXPY with 256 threads per Thread Block
```

```
int nblocks = (n + 255) / 256;
```

```
daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
```

*Creating a number of threads which is (or slightly greater) the number of elements to be processed, and each thread launches the same daxpy function.*

```
// Invoke DAXPY  
daxpy(n, 2.0, x, y);  
// DAXPY in C  
void daxpy(int n, double a, double *x, double *y)  
{  
    for (int i = 0; i < n; ++i)  
        y[i] = a*x[i] + y[i];  
}
```

# DAXPY with Device Code

---

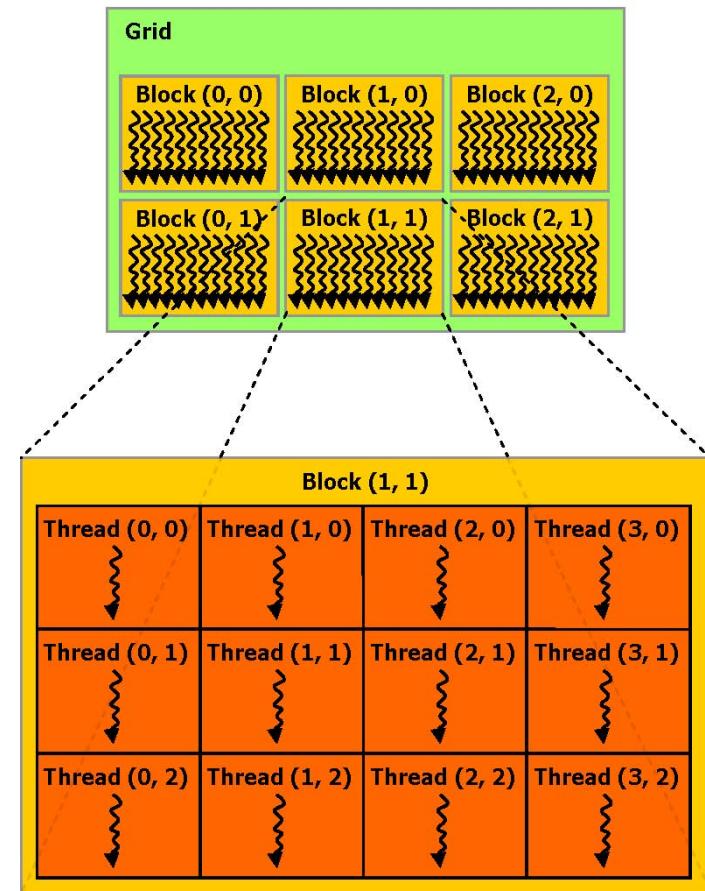
```
__global__ void daxpy( ... )
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- `nvcc` compiler separates source code into host and device components
  - Device functions (e.g. `axpy()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc, cl.exe`

# DAXPY with Device Code

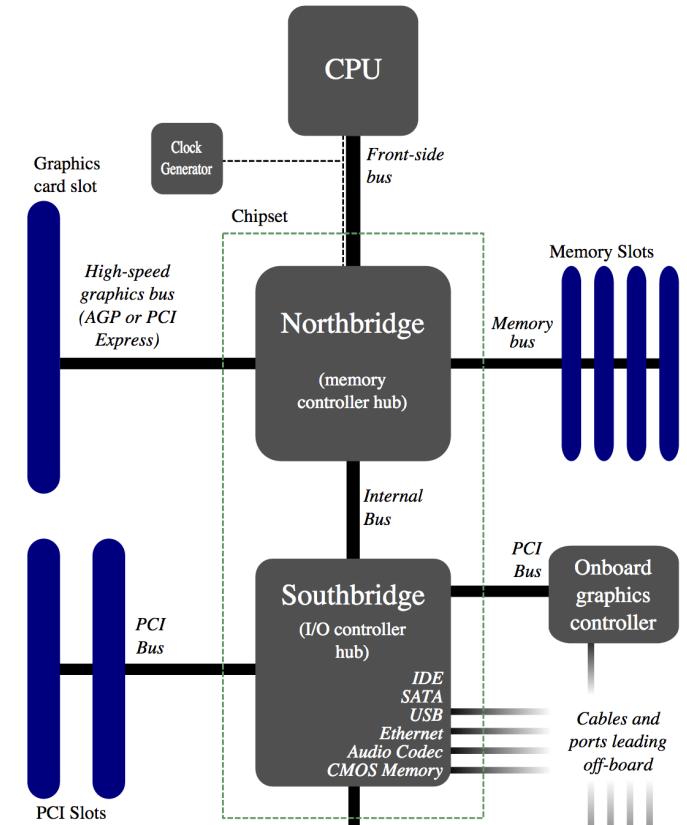
```
axpy<<<num_blocks, num_threads>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - <<< ... >>> parameters are for thread dimensionality
- That’s all that is required to execute a function on the GPU!



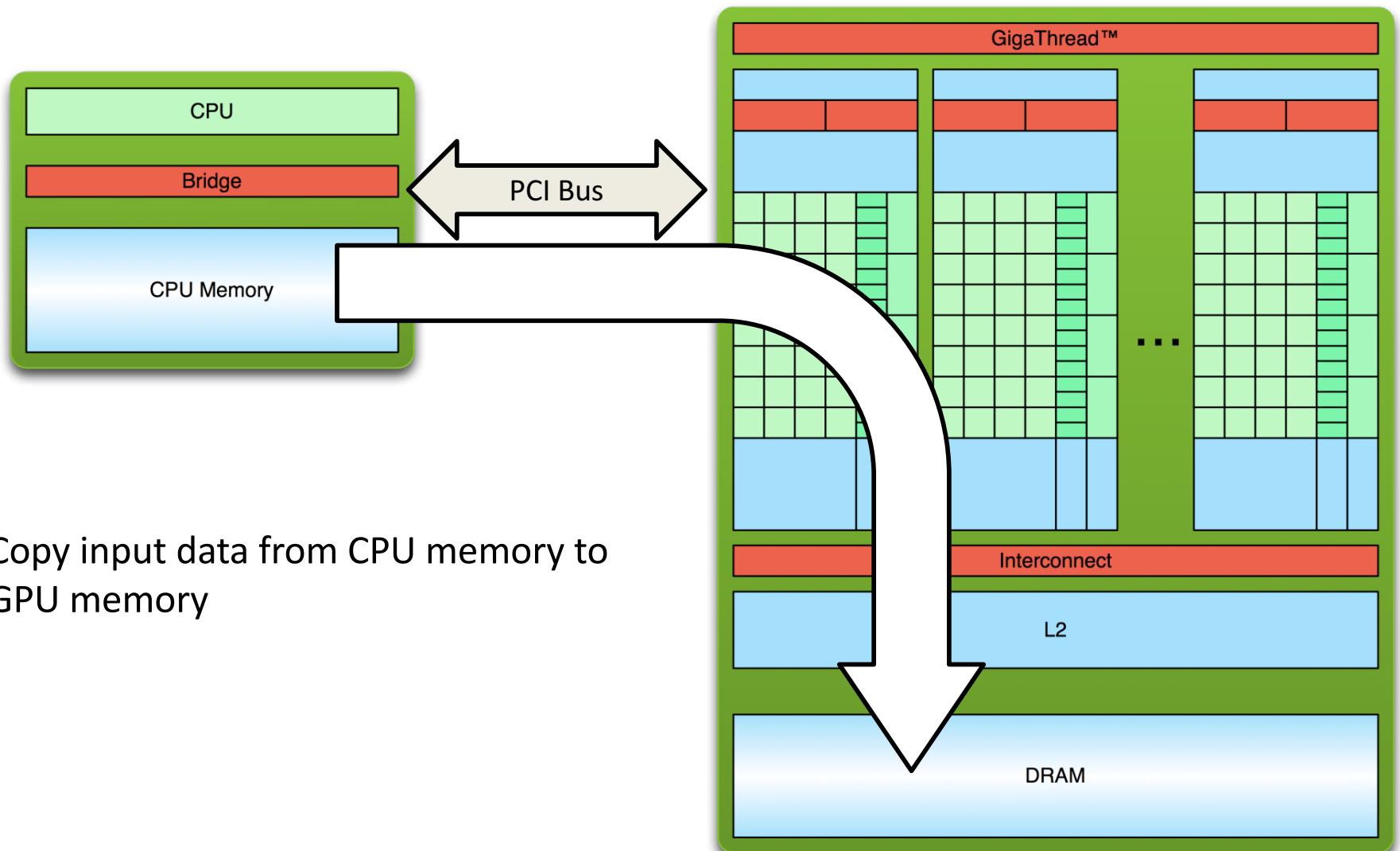
# GPU Computing – Offloading Computation

- The GPU is connected to the CPU by a reasonable fast bus (8 GB/s is typical today): PCIe

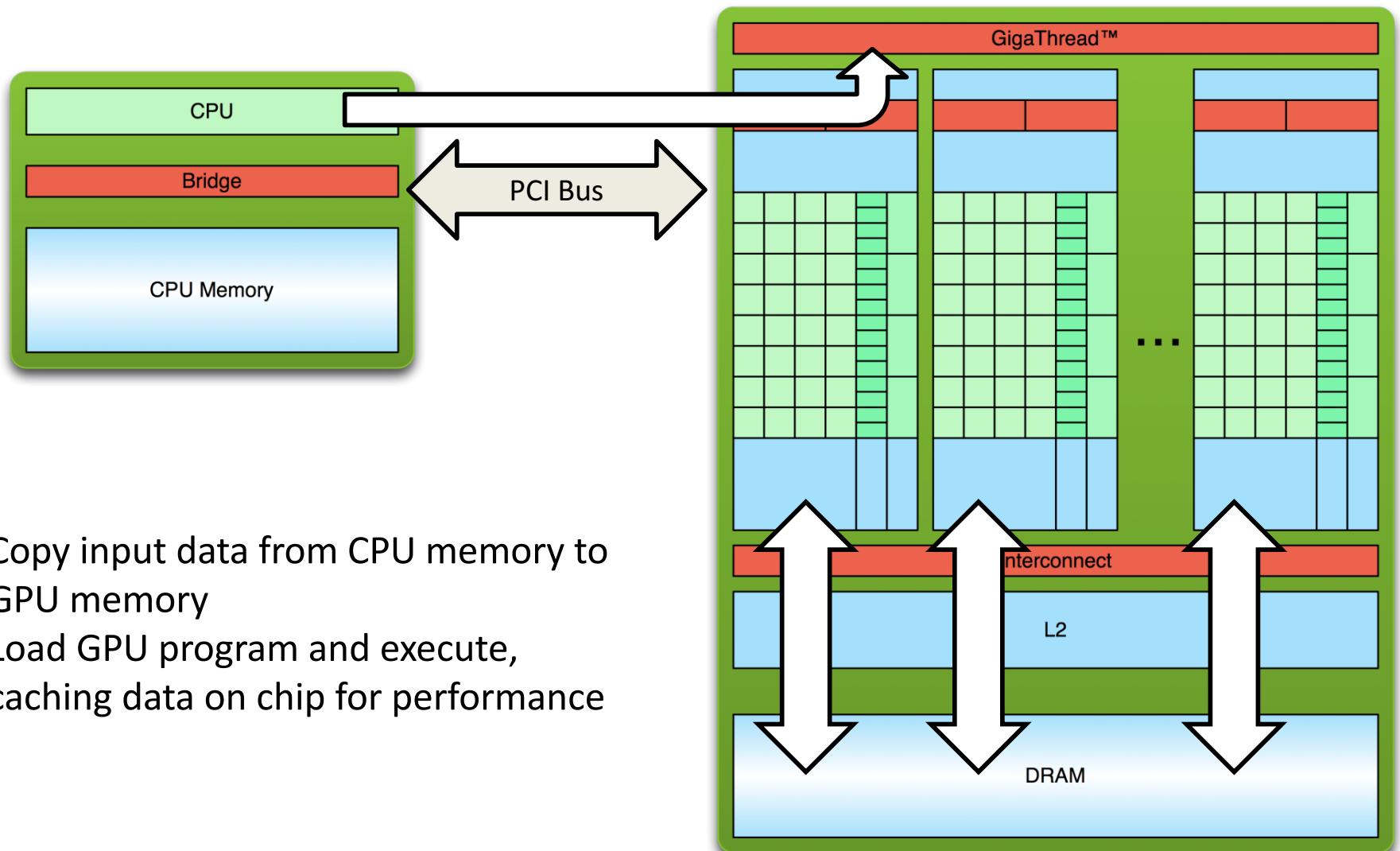


- **Terminology**
  - **Host:** The CPU and its memory (**host memory**)
  - **Device:** The GPU and its memory (**device memory**)

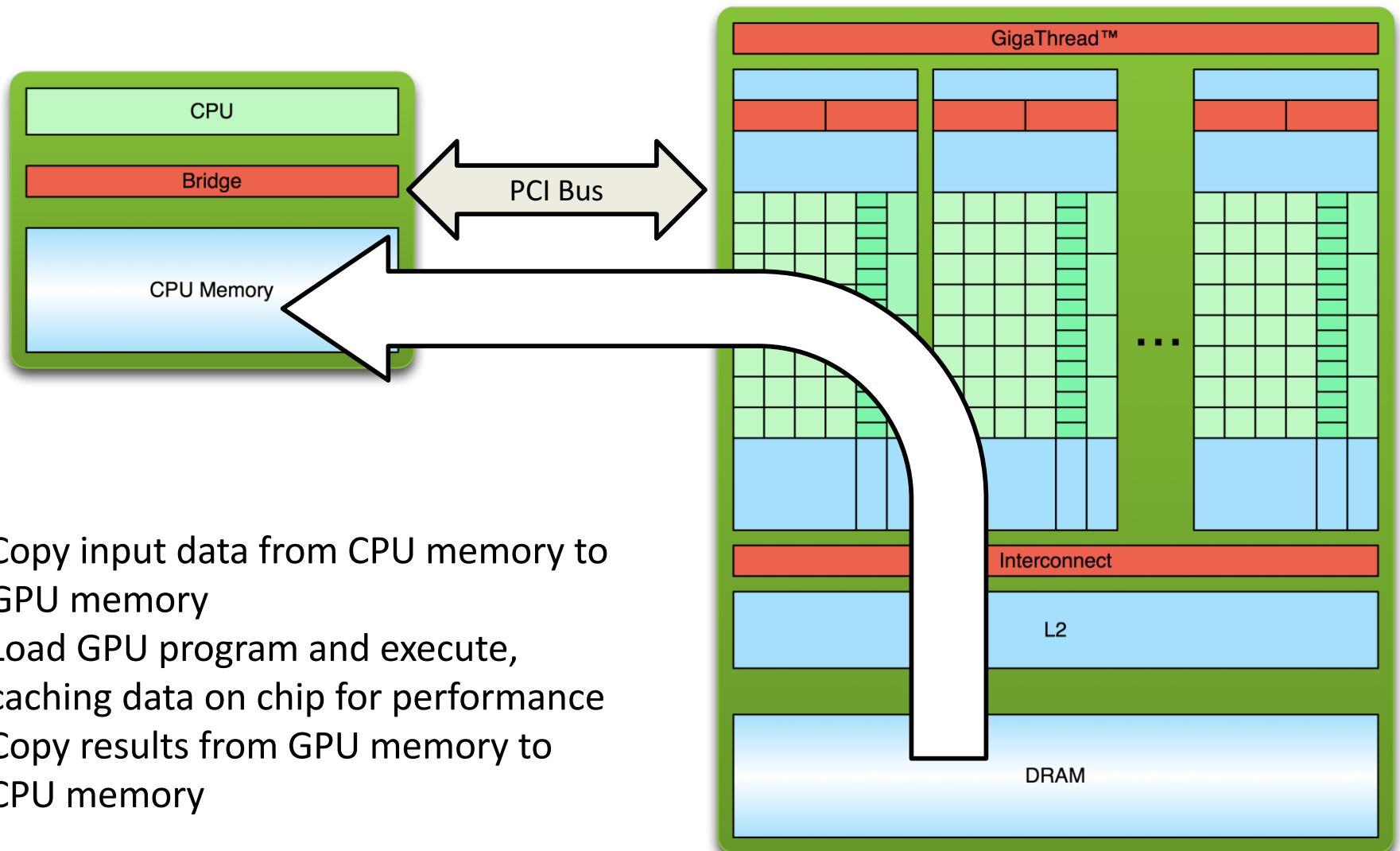
# Simple Processing Flow



# Simple Processing Flow



# Simple Processing Flow



# Offloading Computation

```
// DAXPY in CUDA
__global__
void daxpy(int n, double a, double *x, double *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void) {
    int n = 1024;
    double a;
    double *x, *y; /* host copy of x and y */
    double *x_d, *y_d; /* device copy of x and y */
    int size = n * sizeof(double)
    // Alloc space for host copies and setup values
    x = (double *)malloc(size); fill_doubles(x, n);
    y = (double *)malloc(size); fill_doubles(y, n);

    // Alloc space for device copies
    cudaMalloc((void **)&d_x, size);
    cudaMalloc((void **)&d_y, size);

    // Copy to device
    cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);

    // Invoke DAXPY with 256 threads per Block
    int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x_d, y_d);

    // Copy result back to host
    cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(x); free(y);
    cudaFree(d_x); cudaFree(d_y);
    return 0;
}
```

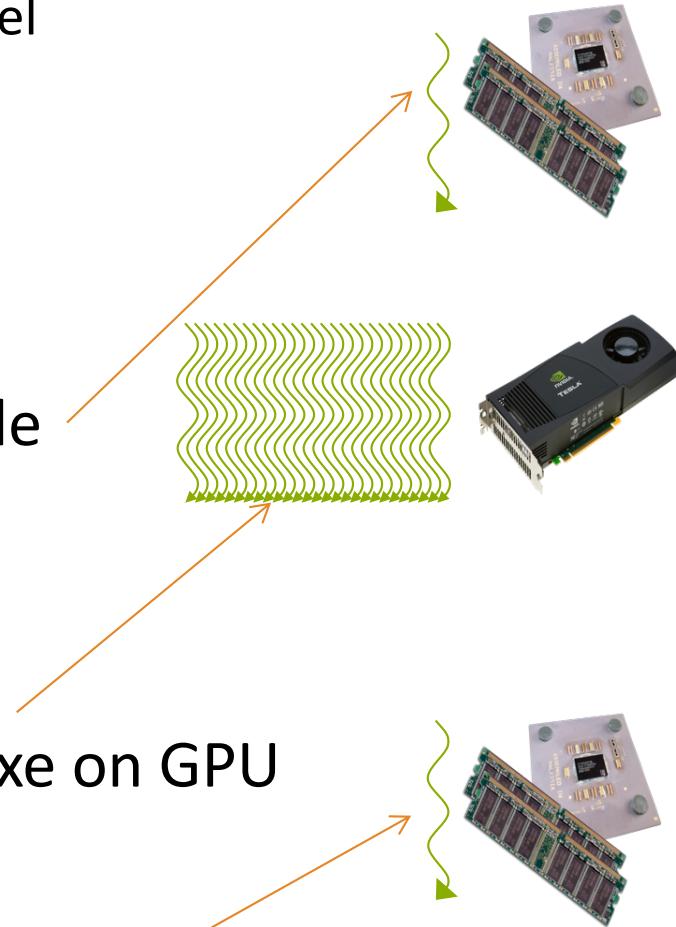
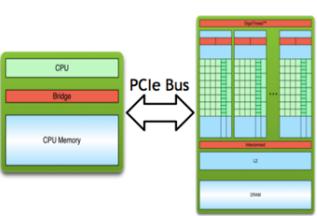
Memory allocation and data copy-in  
Offloading computation  
Data copy-out and deallocation

CUDA kernel

serial code

parallel exe on GPU

serial code



# CUDA Programming Model for NVIDIA GPUs

---

- The CUDA API is split into:
  - The CUDA Management API
  - The CUDA Kernel API
- The CUDA Management API is for a variety of operations
  - GPU memory allocation, data transfer, execution, resource creation
  - Mostly regular C function and calls
- The CUDA Kernel API is used to define the computation to be performed by the GPU
  - C extensions

# CUDA Kernel, i.e. Thread Functions

---

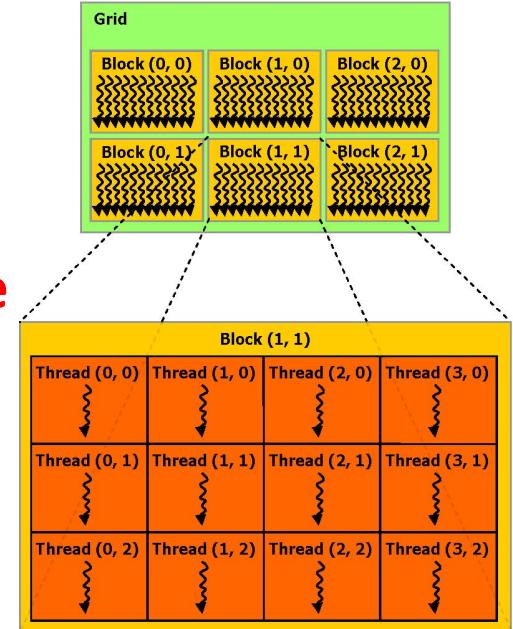
- A CUDA kernel:
  - Defines the operations to be performed by a single thread on the GPU
  - Just as a C/C++ function defines work to be done on the CPU
  - Syntactically, a kernel looks like C/C++ with some extensions

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

- Every CUDA thread executes the same kernel logic (SIMT)
- Initially, the only difference between threads are their *thread coordinates*

# Programming View: How are CUDA threads organized?

- CUDA thread hierarchy
  - Thread Block = SIMT Groups that run concurrently on an SM
    - Can barrier sync and have shared access to the SM shared memory
  - Grid = All Thread Blocks created by the same kernel launch
    - Shared access to GPU global memory
- Launching a kernel is simple and similar to a function call.
  - kernel name and arguments
  - # of thread blocks/grid and # of threads/block to create:  
`kernel<<<nblocks ,  
threads_per_block>>>(arg1 , arg2 , . . . ) ;`



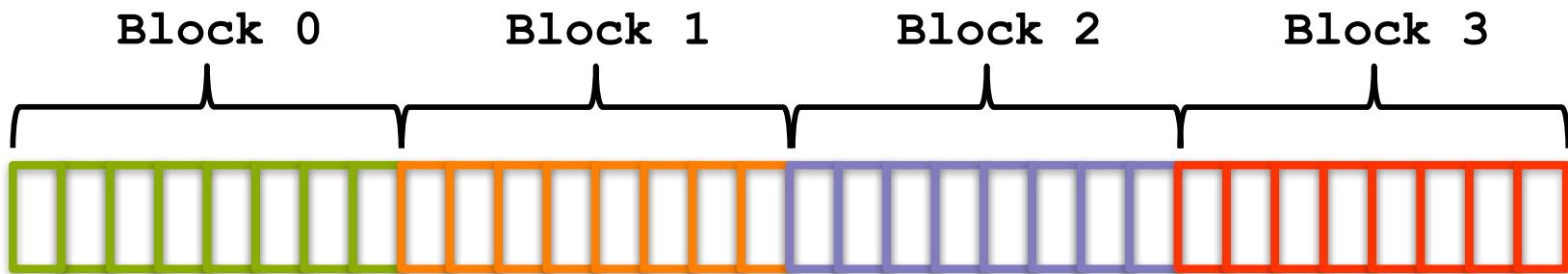
# How are CUDA threads organized?

---

- Threads can be configured in one-, two-, or three-dimensional layouts

- One-dimensional blocks and grids:

```
int nblocks = 4;  
int threads_per_block = 8;  
kernel<<<nblocks, threads_per_block>>>(...);
```



# How are CUDA threads organized?

---

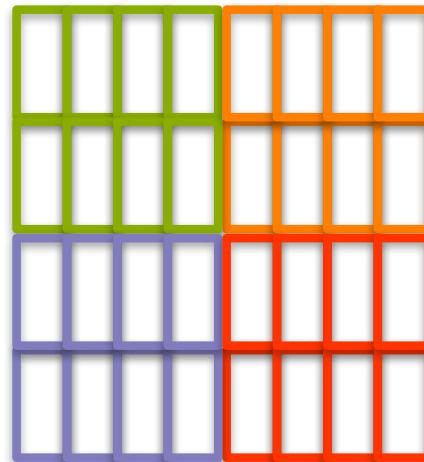
- Threads can be configured in one-, two-, or three-dimensional layouts

- Two-dimensional blocks and grids:

```
dim3 nblocks(2,2)
```

```
dim3 threads_per_block(4,2);
```

```
kernel<<<nblocks, threads_per_block>>>(...);
```



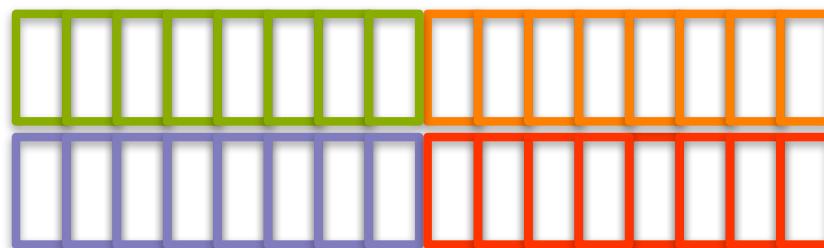
# How are CUDA threads organized?

---

- Threads can be configured in one-, two-, or three-dimensional layouts

- Two-dimensional grid and one-dimensional blocks:

```
dim3 nblocks(2,2);  
int threads_per_block = 8;  
kernel<<<nblocks, threads_per_block>>>(...);
```



# How are CUDA threads organized?

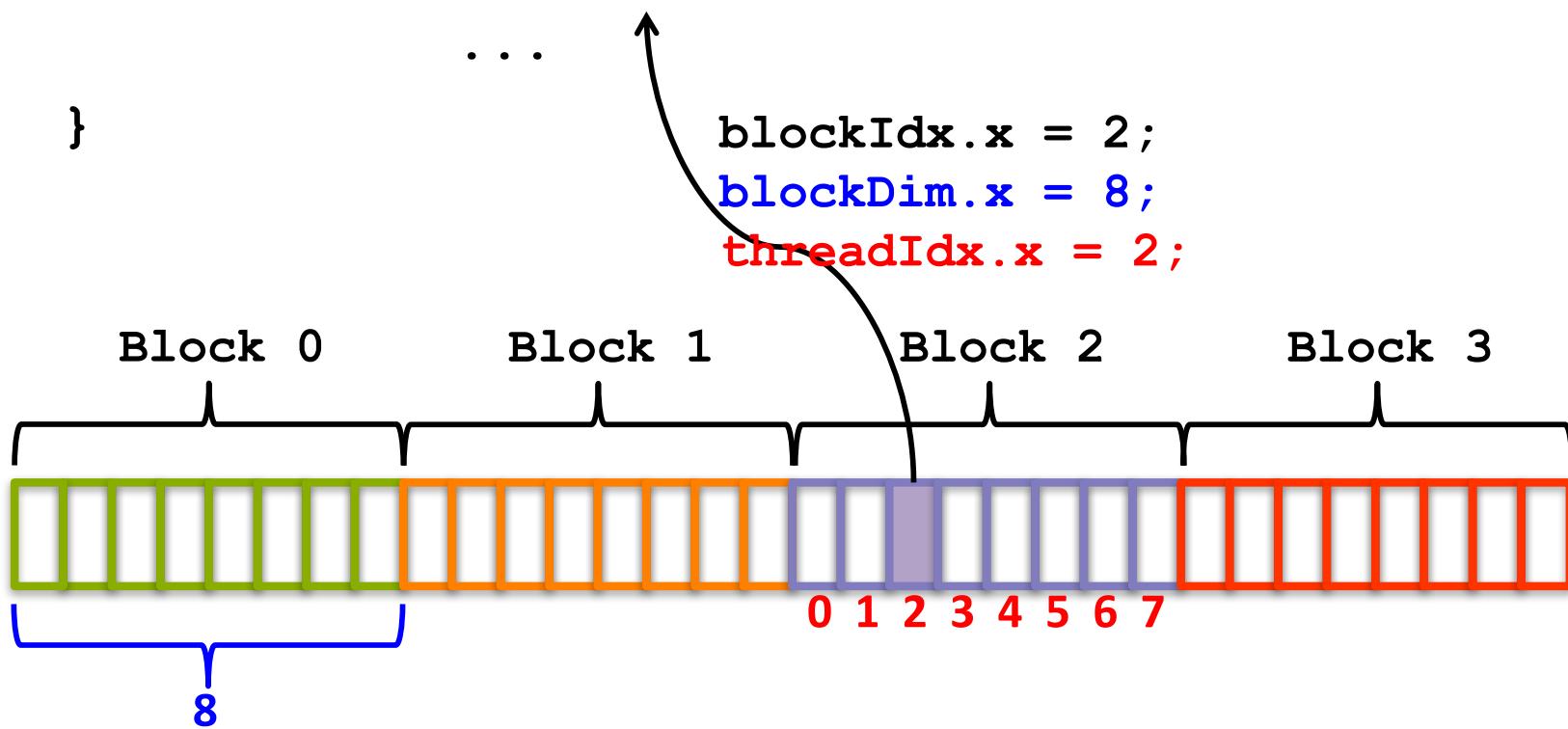
- The number of blocks and threads per block is exposed through *intrinsic thread coordinate variables*:
  - Dimensions
  - IDs

Variable	Meaning
<code>gridDim.x</code> , <code>gridDim.y</code> , <code>gridDim.z</code>	Number of blocks in a kernel launch.
<code>blockIdx.x</code> , <code>blockIdx.y</code> , <code>blockIdx.z</code>	Unique ID of the block that contains the current thread.
<code>blockDim.x</code> , <code>blockDim.y</code> , <code>blockDim.z</code>	Number of threads in each block.
<code>threadIdx.x</code> , <code>threadIdx.y</code> , <code>threadIdx.z</code>	Unique ID of the current thread within its block.

# How are CUDA threads organized?

to calculate a globally unique ID for a thread inside a one-dimensional grid and one-dimensional block:

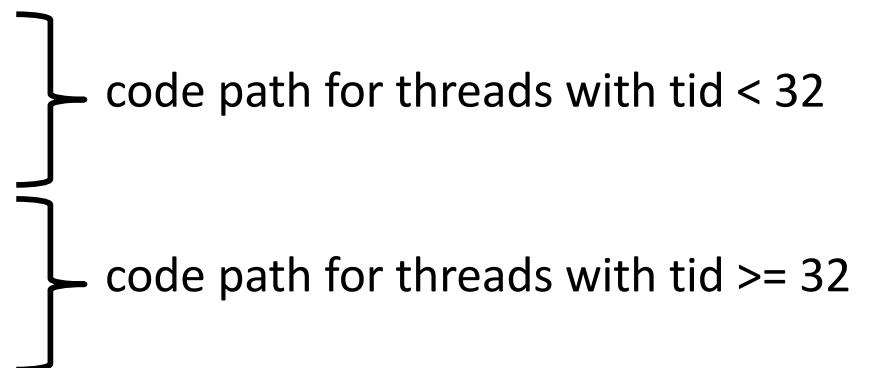
```
kernel<<<4, 8>>>(...);  
  
__global__ void kernel(...){  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
```



# How are CUDA threads organized?

- Thread coordinates offer a way to differentiate threads and identify thread-specific input data or code paths.
  - Co-relate data and computation, a mapping

```
__global__ void kernel(int *arr) {  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
    if (tid < 32) {  
        arr[tid] = f(arr[tid]);  
    } else {  
        arr[tid] = g(arr[tid]);  
    }  
}
```



Thread Divergence: useless code path is executed, but then disabled in SIMD execution model (EXE-commit, more later)

# How is GPU memory managed?

---

- CUDA Memory Management API
  - Allocation of GPU memory
  - Transfer of data from the host to GPU memory
  - Free-ing GPU memory
  - `Foo(int A[][]N) { }`

Host Function	CUDA Analogue
<code>malloc</code>	<code>cudaMalloc</code>
<code>memcpy</code>	<code>cudaMemcpy</code>
<code>free</code>	<code>cudaFree</code>

# How is GPU memory managed?

---

```
cudaError_t cudaMalloc(void **devPtr,  
                      size_t size);
```

- Allocate **size** bytes of GPU memory and store their address at **\*devPtr**

```
cudaError_t cudaFree(void *devPtr);
```

- Release the device memory allocation stored at **devPtr**
- Must be an allocation that was created using **cudaMalloc**

# How is GPU memory managed?

---

```
cudaError_t cudaMemcpy (   
    void *dst, const void *src, size_t count,   
    enum cudaMemcpyKind kind) ;
```

- Transfers count bytes from the memory pointed to by src to dst
- kind can be:
  - cudaMemcpyHostToHost,
  - cudaMemcpyHostToDevice,
  - cudaMemcpyDeviceToHost,
  - cudaMemcpyDeviceToDevice
- The locations of dst and src must match kind, e.g. if kind is cudaMemcpyHostToDevice then src must be a host array and dst must be a device array

# How is GPU memory managed?

---

```
void *d_arr, *h_arr;  
h_addr = ... ; /* init host memory and data */  
// Allocate memory on GPU and its address is in d_arr  
cudaMalloc((void **) &d_arr, nbytes);  
  
// Transfer data from host to device  
cudaMemcpy(d_arr, h_arr, nbytes,  
           cudaMemcpyHostToDevice);  
  
// Transfer data from a device to a host  
cudaMemcpy(h_arr, d_arr, nbytes,  
           cudaMemcpyDeviceToHost);  
  
// Free the allocated memory  
cudaFree(d_arr);
```

# CUDA Program Flow

---

- At its most basic, the flow of a CUDA program is as follows:
  1. Allocate GPU memory
  2. Populate GPU memory with inputs from the host
  3. Execute a GPU kernel on those inputs
  4. Transfer outputs from the GPU back to the host
  5. Free GPU memory

# Offloading Computation

```
// DAXPY in CUDA
__global__
void daxpy(int n, double a, double *x, double *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void) {
    int n = 1024;
    double a;
    double *x, *y; /* host copy of x and y */
    double *x_d, *y_d; /* device copy of x and y */
    int size = n * sizeof(double)
    // Alloc space for host copies and setup values
    x = (double *)malloc(size); fill_doubles(x, n);
    y = (double *)malloc(size); fill_doubles(y, n);

    // Alloc space for device copies
    cudaMalloc((void **)&d_x, size);
    cudaMalloc((void **)&d_y, size);

    // Copy to device
    cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);

    // Invoke DAXPY with 256 threads per Block
    int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x_d, y_d);

    // Copy result back to host
    cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(x); free(y);
    cudaFree(d_x); cudaFree(d_y);
    return 0;
}
```

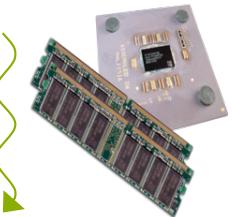
Memory allocation and data copy-in  
Offloading computation  
Data copy-out and deallocation

CUDA kernel

serial code

parallel exe on GPU

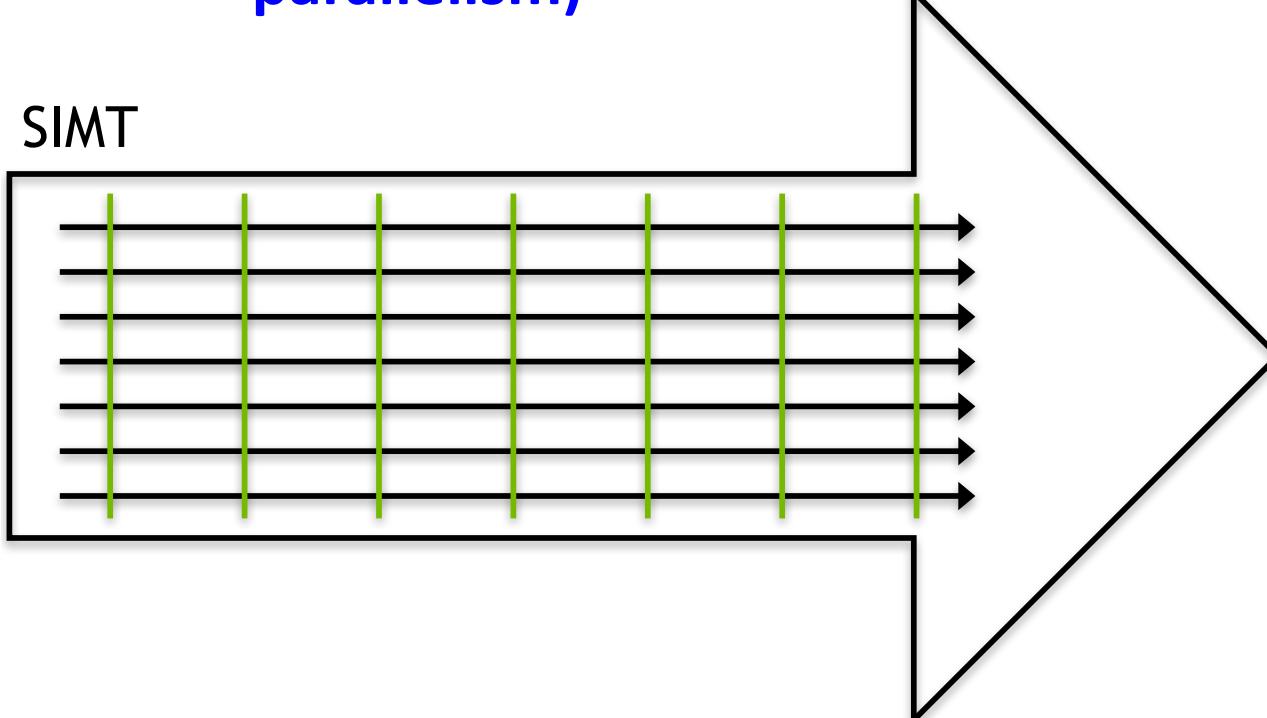
serial code



# GPU Multi-Threading (SIMD)

- NVIDIA calls it Single-Instruction, Multiple-Thread (SIMT)
  - Many threads execute the same instructions in lock-step
    - A warp (32 threads)
    - Each thread  $\approx$  vector lane; 32 lanes lock step
  - Implicit synchronization after every instruction (think vector parallelism)

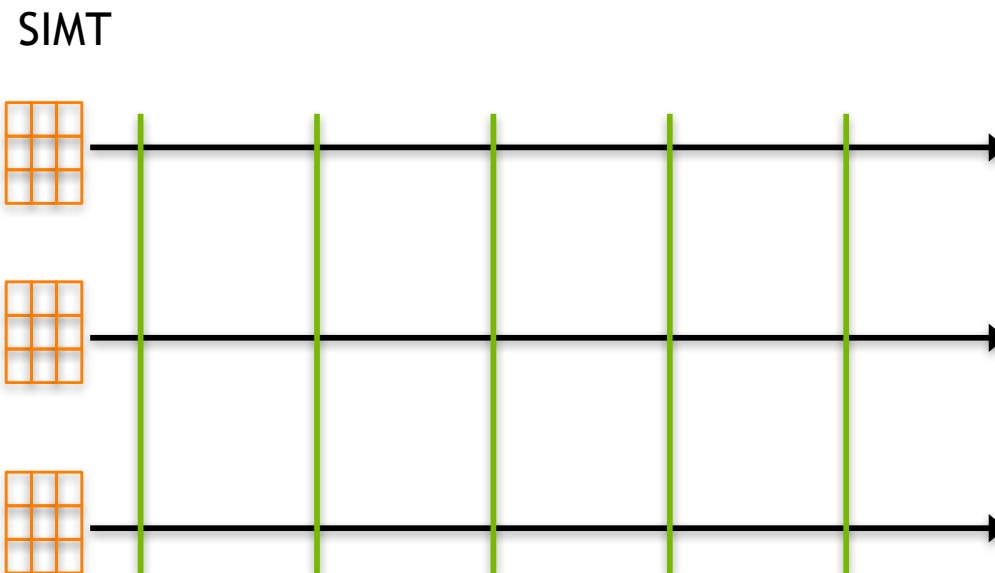
SIMT



# GPU Multi-Threading

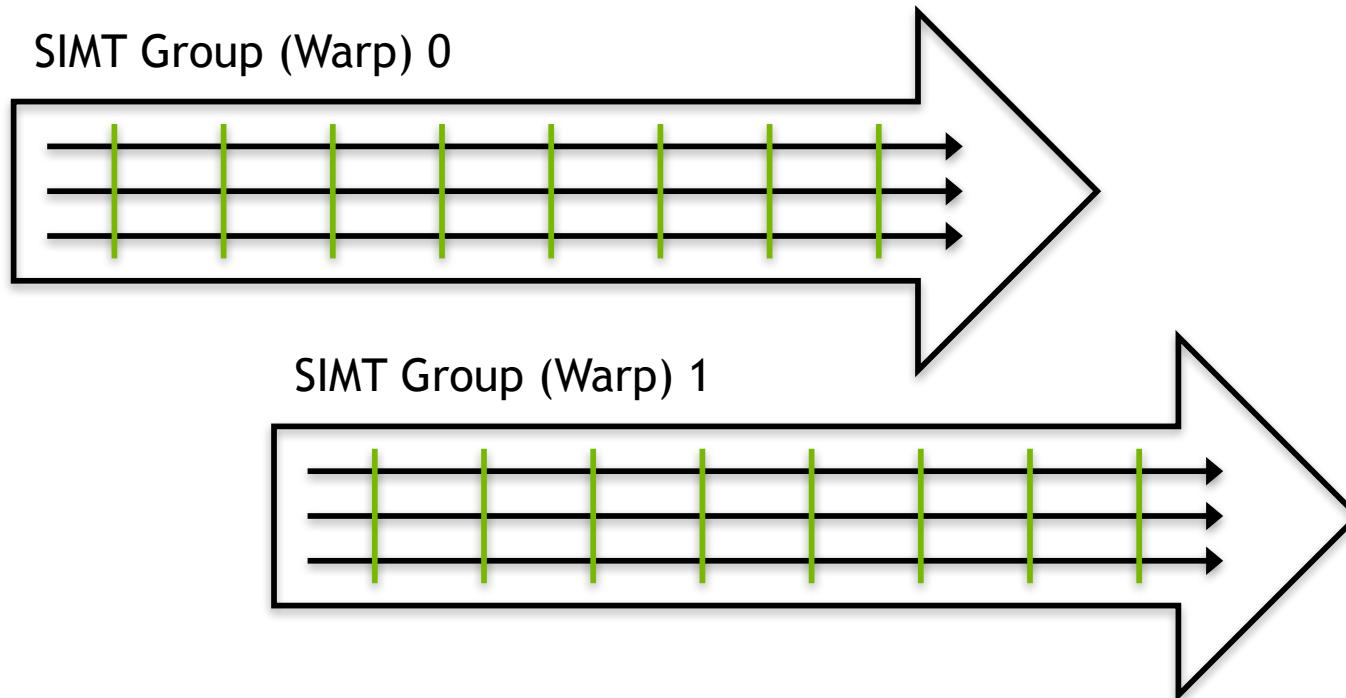
---

- In SIMT, all threads share instructions but operate on their own private registers, allowing threads to store thread-local state



# GPU Multi-Threading

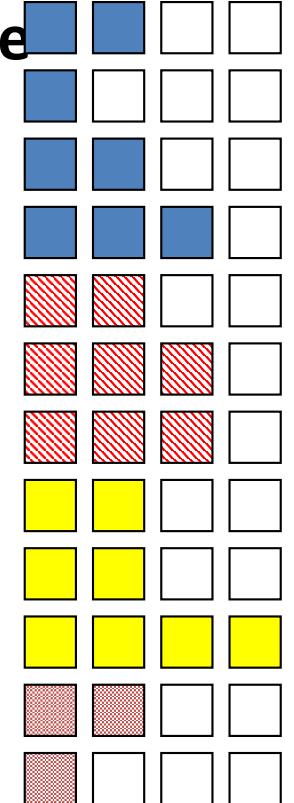
- GPUs execute many groups of SIMT threads in parallel
  - Each executes instructions independent of the others



# Warp Switching

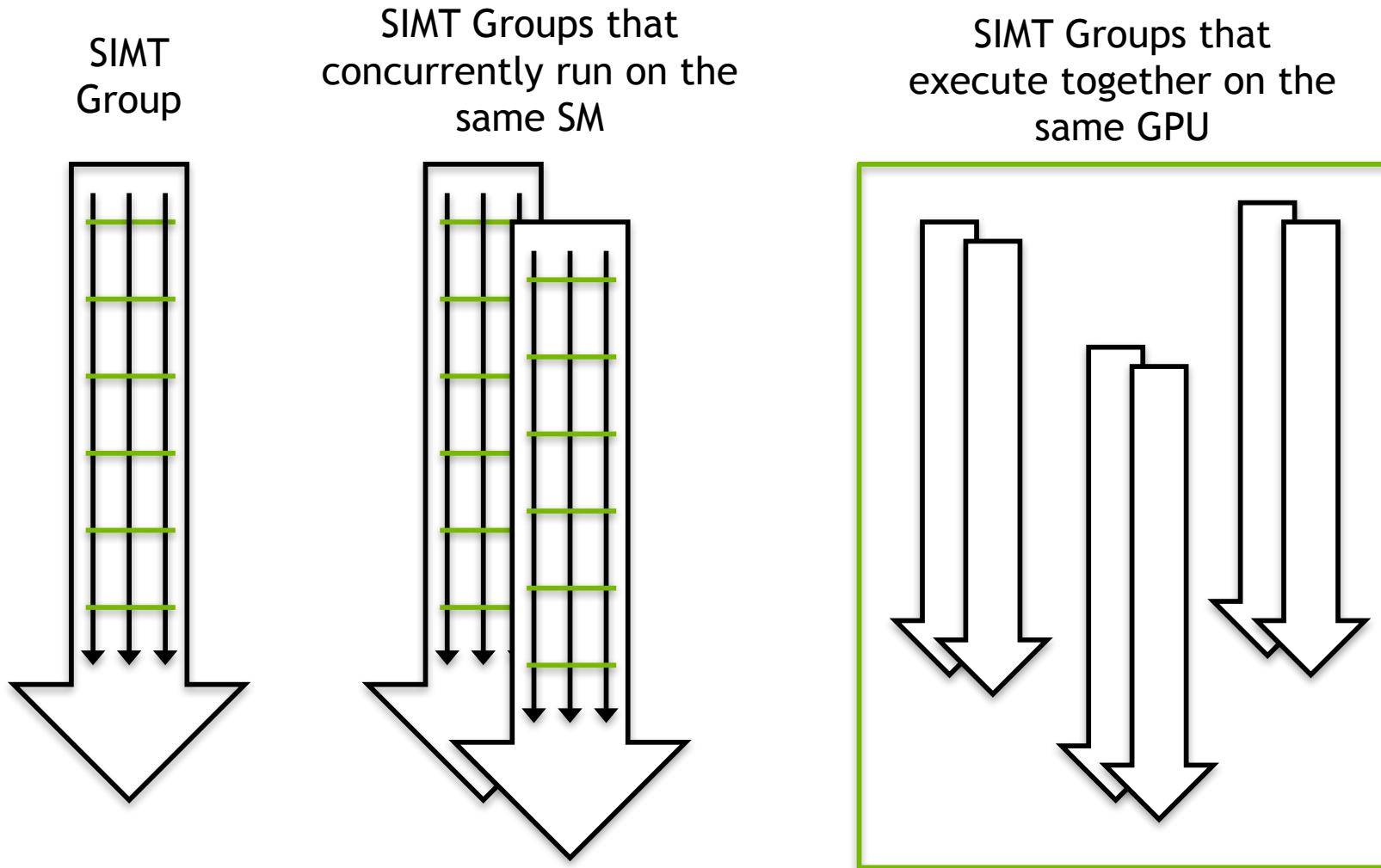
SMs can support more concurrent SIMT groups than core count would suggest → **Coarse grained multiwarpping (the term I coined)**

- Similar to coarse-grained multi-threading
- Each thread persistently stores its own state in a private register set
  - Enable very efficient context switching between warps
- SIMT warps block if not actively computing
  - Swapped out for other, no worrying about losing state
    - Keeping blocked SIMT groups scheduled on an SM would waste cores



# Execution Model to Hardware

- This leads to a nested thread hierarchy on GPUs



# NVIDIA PTX (Parallel Thread Execution) ISA

---

- **Compiler target (Not hardware ISA)**
  - **Similar to X86 ISA, and use virtual register**
  - **Both translate to internal form (micro-ops in x86)**
    - **X86's translation happens in hardware at runtime**
    - **NVIDIA GPU PTX is translated by software at load time**
- **Basic format (d is destination, a, b and c are operands)**  
opcode.type d, a, b, c;

---

Type	.type Specifier
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating Point 16, 32, and 64 bits	.f16, .f32, .f64

# Basic PTX Operations (ALU, MEM, and Control)

Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	$d = a + b;$	
	sub.type	sub.f32 d, a, b	$d = a - b;$	
	mul.type	mul.f32 d, a, b	$d = a * b;$	
	mad.type	mad.f32 d, a, b, c	$d = a * b + c;$	multiply-add
	div.type	div.f32 d, a, b	$d = a / b;$	multiple microinstructions
	rem.type	rem.u32 d, a, b	$d = a \% b;$	integer remainder
	abs.type	abs.f32 d, a		memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64
	neg.type	neg.f32 d, a		ld.space.type ld.global.b32 d, [a+off] $d = *(a+off);$ load from memory space
	min.type	min.f32 d, a, b		st.space.type st.shared.b32 [d+off], a $*(d+off) = a;$ store to memory space
	max.type	max.f32 d, a, b		tex.nd.dtype.btype tex.2d.v4.f32.f32 d, a, b $d = tex2d(a, b);$ texture lookup
Special Function	setp.cmp.type	setp.lt.f32 p, a, b		atom.global.add.u32 d,[a], b atomic { $d = *a;$ $*a =$ } atomic read-modify-write
	numeric .cmp = eq, ne, lt, le, gt, ge; unor			atom.global.cas.b32 d,[a], b, cop(*a, b); } operation
	mov.type	mov.b32 d, a		atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32
	selp.type	selp.f32 d, a, b, p		branch @p bra target if (p) goto target; conditional branch
	cvt.dtype.atype	cvt.f32.s32 d, a		call call (ret), func, (params) ret = func(params); call function
	special .type = .f32 (some .f64)			ret ret return; return from function call
	rcp.type	rcp.f32 d, a		bar.sync bar.sync d wait for threads barrier synchronization
	sqrt.type	sqrt.f32 d, a		exit exit exit; terminate thread execution
	rsqrt.type	rsqrt.f32 d, a	$d = 1/sqrt(a);$	reciprocal square root
	sin.type	sin.f32 d, a	$d = sin(a);$	sine
Logical	cos.type	cos.f32 d, a	$d = cos(a);$	cosine
	lg2.type	lg2.f32 d, a	$d = log(a)/log(2)$	binary logarithm
	ex2.type	ex2.f32 d, a	$d = 2 ^ a;$	binary exponential
	logic.type = .pred,.b32, .b64			
	and.type	and.b32 d, a, b	$d = a \& b;$	
	or.type	or.b32 d, a, b	$d = a   b;$	
	xor.type	xor.b32 d, a, b	$d = a ^ b;$	
	not.type	not.b32 d, a, b	$d = \sim a;$	one's complement
	cnot.type	cnot.b32 d, a, b	$d = (a==0)? 1:0;$	C logical not
	shl.type	shl.b32 d, a, b	$d = a << b;$	shift left
	shr.type	shr.s32 d, a, b	$d = a >> b;$	shift right

# NVIDIA PTX GPU ISA Example

## DAXPY

```
__global__
void daxpy(int n, double a, double *x, double *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

shl.s32 R8, blockIdx, 9 ; Thread Block ID \* Block size  
(512 or 29)

add.s32 R8, R8, threadIdx ; R8 = i = my CUDA thread ID

ld.global.f64 RD0, [X+R8] ; RD0 = X[i]

ld.global.f64 RD2, [Y+R8] ; RD2 = Y[i]

mul.f64 R0D, RD0, RD4 ; Product in RD0 = RD0 \* RD4  
(scalar a)

add.f64 R0D, RD0, RD2 ; Sum in RD0 = RD0 + RD2 (Y[i])

st.global.f64 [Y+R8], RD0 ; Y[i] = sum (X[i]\*a + Y[i])

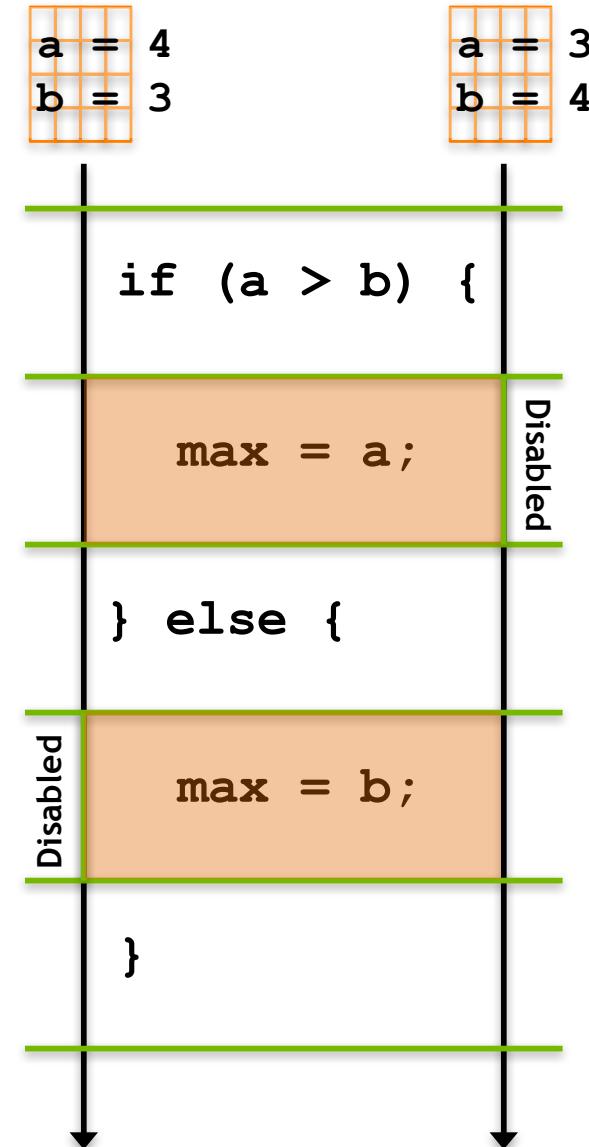
# Conditional Branching in GPU

---

- Like vector, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each core
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - ...and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

# Conditional Branching in GPU

- Instruction lock-step execution by multi-threads
- SIMT threads can be “disabled” when they need to execute instructions different from others in their group
  - Mask and commit
- Branch divergence
  - Hurt performance and efficiency



# PTX Example

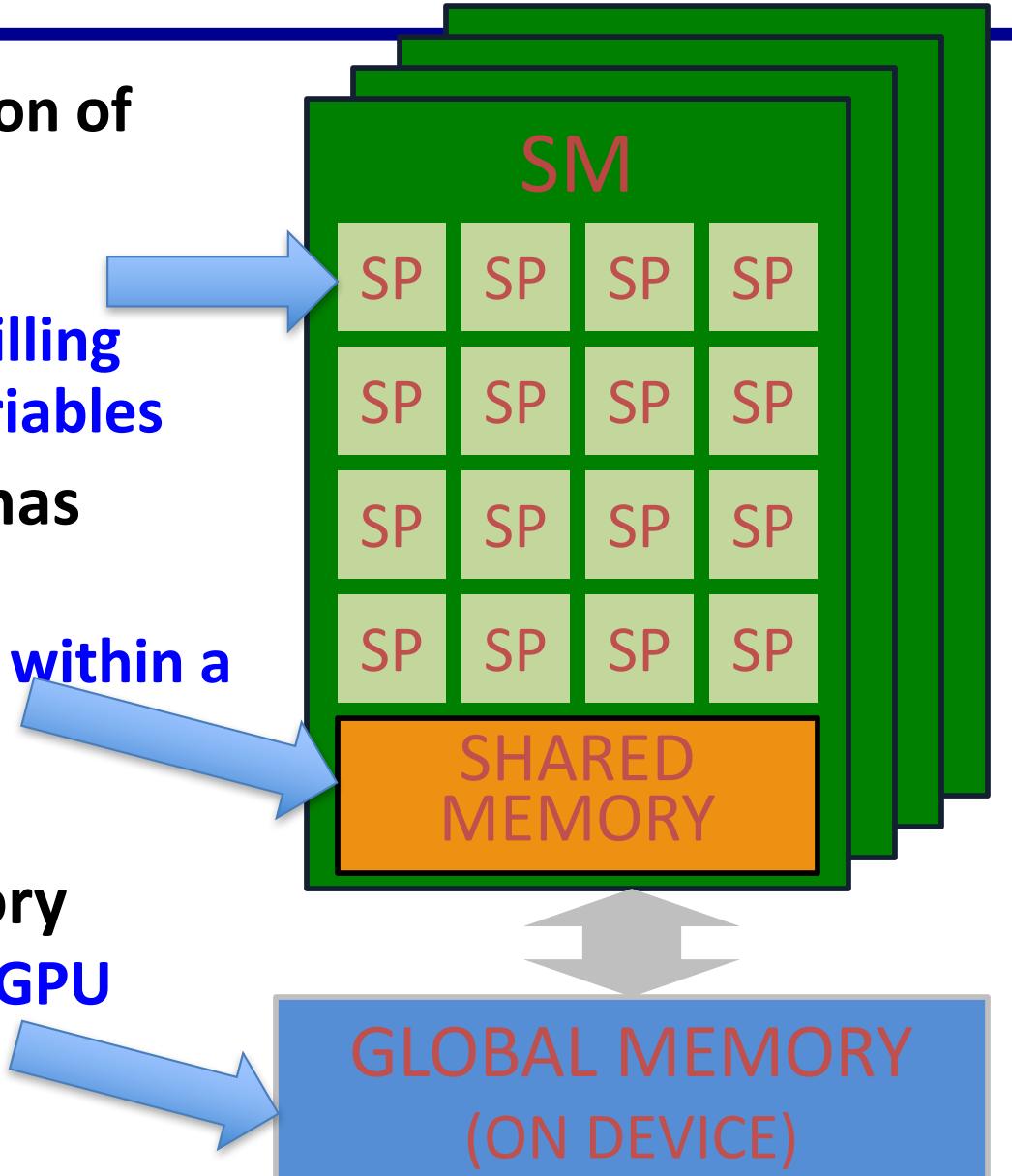
---

```
if (X[i] != 0)
    X[i] = X[i] - Y[i];
else X[i] = Z[i];
```

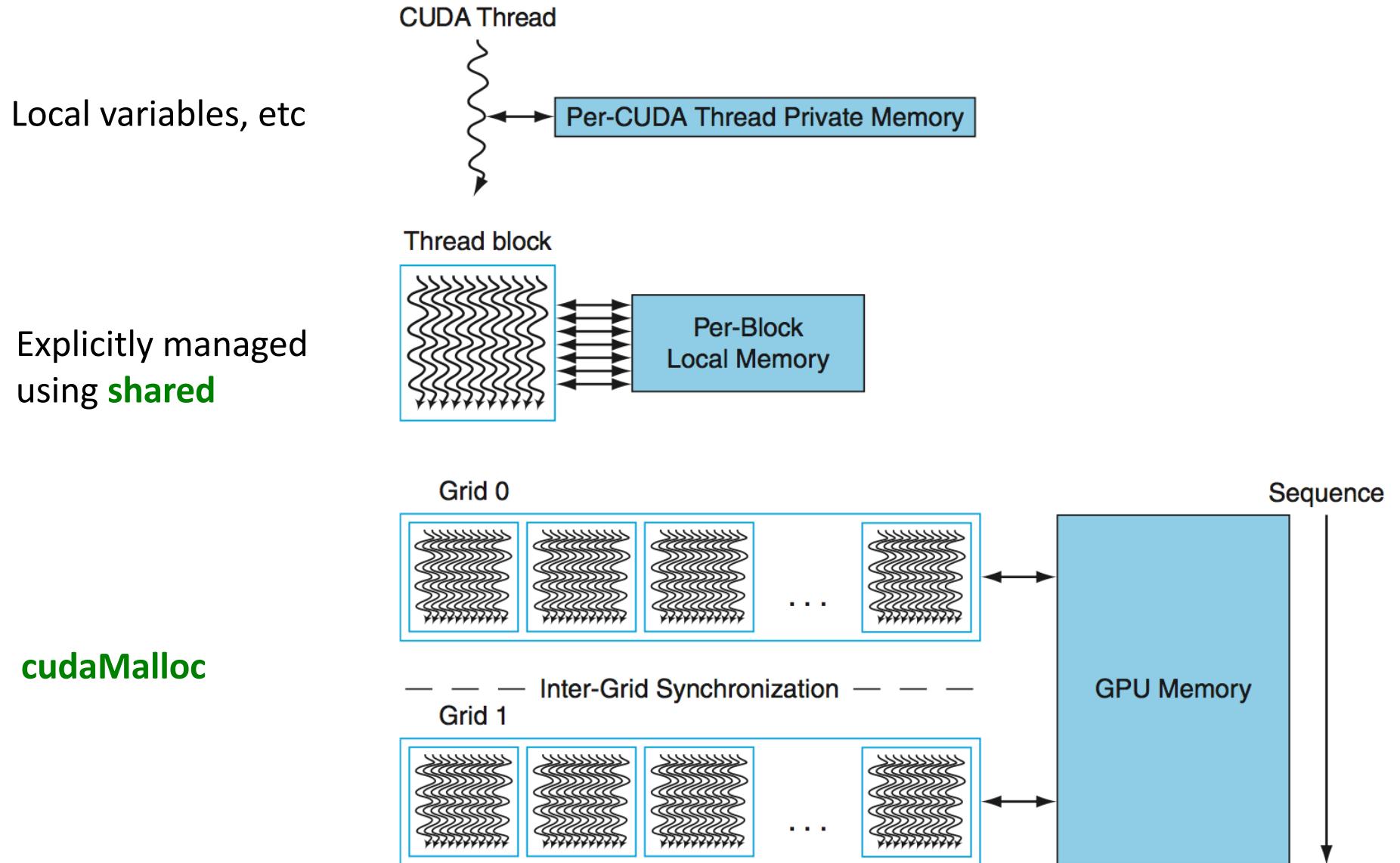
	ld.global.f64 RD0, [X+R8]	; RD0 = X[i]
	setp.neq.s32 P1, RD0, #0	; P1 is predicate register 1
	@!P1, bra ELSE1, *Push	; Push old mask, set new mask bits
		; if P1 false, go to ELSE1
	ld.global.f64 RD2, [Y+R8]	; RD2 = Y[i]
	sub.f64 RD0, RD0, RD2	; Difference in RD0
	st.global.f64 [X+R8], RD0	; X[i] = RD0
	@P1, bra ENDIF1, *Comp	; complement mask bits
		; if P1 true, go to ENDIF1
ELSE1:	ld.global.f64 RD0, [Z+R8]	; RD0 = Z[i]
	st.global.f64 [X+R8], RD0	; X[i] = RD0
ENDIF1:	<next instruction>, *Pop	; pop to restore old mask

# NVIDIA GPU Memory Structures

- Each core has private section of off-chip DRAM
  - “Private memory”
  - Contains stack frame, spilling registers, and private variables
- Each SM processor also has local memory
  - Shared by cores/threads within a SM/block
- Memory shared by SM processors is GPU Memory
  - Host can read and write GPU memory



# GPU Memory for CUDA Programming



# Shared Memory Allocation

---

- Shared memory can be allocated statically or dynamically
- Statically Allocated Shared Memory
  - Size is fixed at compile-time
  - Can declare many statically allocated shared memory variables
  - Can be declared globally or inside a device function
  - Can be multi-dimensional arrays

```
__shared__ int s_arr[256][256];
```

# Shared Memory Allocation

---

- **Dynamically Allocated Shared Memory**
  - Size in bytes is set at kernel launch with a third kernel launch configurable
  - Can only have one dynamically allocated shared memory array per kernel
  - Must be one-dimensional arrays

```
__global__ void kernel(...) {
    extern __shared__ int s_arr[];
    ...
}

kernel<<<nblocks, threads_per_block,
shared_memory_bytes>>>(...);
```

# GPU Memory

- **More complicated**
- **Different usage scope**
- **Different size, and performance**
  - **Latency and bandwidth**
  - **Read-only or R/W cache**

SIMT Thread Groups on a GPU

SIMT Thread Groups on an SM

SIMT Thread Group

Registers Local Memory

On-Chip Shared Memory

Global Memory

Constant Memory

Texture Memory

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

# GPU and Manycore Architecture

---

We only INTRODUCE the programming interface and architecture

For more info:

- <http://docs.nvidia.com/cuda/>
- **Professional CUDA C Programming, John Cheng Max Grossman Ty McKercher September 8, 2014, John Wiley & Sons**

Other Related info

- **AMD GPU and OpenCL**
- **Programming with Accelerator using pragma**
  - **OpenMP and OpenACC**

# Loop-Level Parallelism

---

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence
- Example 1:

```
for (i=999; i>=0; i=i-1)  
    x[i] = x[i] + s;
```

- No loop-carried dependence

# Loop-Level Parallelism

- Example 2:

```
for (i=0; i<100; i=i+1) {  
    S1: A[i+1] = A[i] + C[i];  
    S2: B[i+1] = B[i] + A[i+1];  
}
```

Circular dependency  
and not parallelizable  
(or vectorizable)

- S1 and S2 use values computed by S1 and S2 in previous iteration: **loop-carried dependency** → **serial execution**
  - $A[i] \rightarrow A[i+1]$ ,  $B[i] \rightarrow B[i+1]$
- S2 uses value computed by S1 in same iteration → not loop carried
  - $A[i+1] \rightarrow A[i+1]$

# Loop-Level Parallelism

---

- Example 3:

```
for (i=0; i<100; i=i+1) {  
    A[i] = A[i] + B[i];      /* S1 */  
    B[i+1] = C[i] + D[i];    /* S2 */  
}
```

S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel

- Transform to:

```
A[0] = A[0] + B[0];  
for (i=0; i<99; i=i+1) {  
    B[i+1] = C[i] + D[i];  
    A[i+1] = A[i+1] + B[i+1];  
}  
B[100] = C[99] + D[99];
```

# Loop-Level Parallelism

---

- Example 4:

```
for (i=0;i<100;i=i+1) {
```

```
    A[i] = B[i] + C[i];      /* S1 */
```

```
    D[i] = A[i] * E[i];      /* S2 */
```

```
}
```



No need to store  $A[i]$  in  $S1$  and then load  $A[i]$  in  $S2$

- Example 5:

```
for (i=1;i<100;i=i+1) {
```

```
    Y[i] = Y[i-1] + Y[i];
```

```
}
```



Recurrence: for exploring pipelining parallelism between iterations

# Finding dependencies

---

- Assume indices are affine:
  - $a \times i + b$  (i is loop index and a and b are constants)
- Assume:
  - Store to  $a \times i + b$ , then
  - Load from  $c \times i + d$
  - $i$  runs from  $m$  to  $n$
  - Dependence exists if:
    - Given  $j, k$  such that  $m \leq j \leq n, m \leq k \leq n$
    - Store to  $a \times j + b$ , load from  $a \times k + d$ , and  $a \times j + b = c \times k + d$

# Finding dependencies

---

- Generally cannot determine at compile time
- Test for absence of a dependence:
  - GCD test:
    - If a dependency exists,  $\text{GCD}(c,a)$  must evenly divide  $(d-b)$

- Example:

```
for (i=0; i<100; i=i+1) {  
    X[2*i+3] = X[2*i] * 5.0;  
}
```

$a=2$ ,  $b=3$ ,  $c=2$ , and  $d=0$ , then  $\text{GCD}(a,c)=2$ , and  $d-b=-3$ . Since 2 does not divide  $-3$ , no dependence is possible.

# Finding dependencies

---

- Example 2:

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}
```

```
for (i=0; i<100; i=i+1 {  
    T[i] = X[i] / c; /* Y renamed to T to remove  
    X1[i] = X[i] + c; /* X renamed to X1 to remove  
    Z[i] = T[i] + c; /* Y renamed to T to remove  
    Y[i] = c - T[i];  
}
```

- True dependencies:
  - S1 to S3 and S1 to S4 because of Y[i], not loop carried
- Antidependence:
  - S1 to S2 based on X[i] and S3 to S4 for Y[i]
- Output dependence:
  - S1 to S4 based on Y[i]

# Reductions

---

- **Reduction Operation:**

```
for (i=9999; i>=0; i=i-1)  
    sum = sum + x[i] * y[i];
```

- **Transform to...**

```
for (i=9999; i>=0; i=i-1)  
    sum [i] = x[i] * y[i];  
for (i=9999; i>=0; i=i-1)  
    finalsum = finalsum + sum[i];
```

- **Do on p processors:**

```
for (i=999; i>=0; i=i-1)  
    finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- **Note: assumes associativity!**

# Dependency Analysis

---

- Mostly done by compiler before vectorization
  - Can be conservative if compiler is not 100% sure
- For programmer:
  - Write code that can be easily analyzed by compiler for vectorization
  - Use explicit parallel model such as OpenMP or CUDA

```
15 #pragma omp parallel for \
16     shared(a,b,c,chunk) private(i) \
17     schedule(static,chunk)
18     for (i=0; i < n; i++)
19         c[i] = a[i] + b[i];
20 }
```

<https://computing.llnl.gov/tutorials/openMP/>

# Wrap-Ups (Vector, SIMD and GPU)

---

- Data-level parallelism