University of Pittsburgh
School of Computing and Information

Graphics Processing Units (GPUs)

Xulong Tang

Slides adapted from Sharan Chetlur (NVIDIA), David Patterson, John L. Hennessy, Wonsun Ahn

1

Flynn's Taxonomy

SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	(MIMD) Multiple instruction stream Multiple data stream

classic von Neumann

Does it make sense?
Yes, systolic array.

2

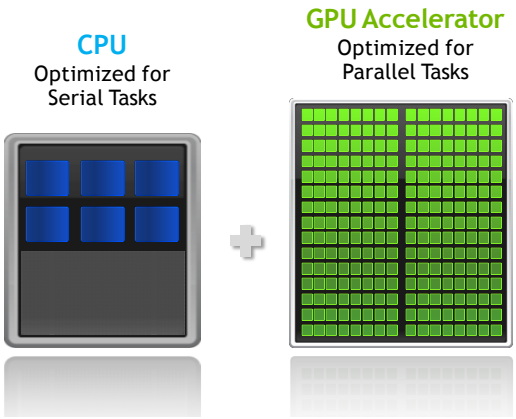
Graphics Processing Units:

SIMD + Multi-threading (SIMT)

3

GPU ACCELERATED COMPUTING

10X PERFORMANCE 5X ENERGY EFFICIENCY



4

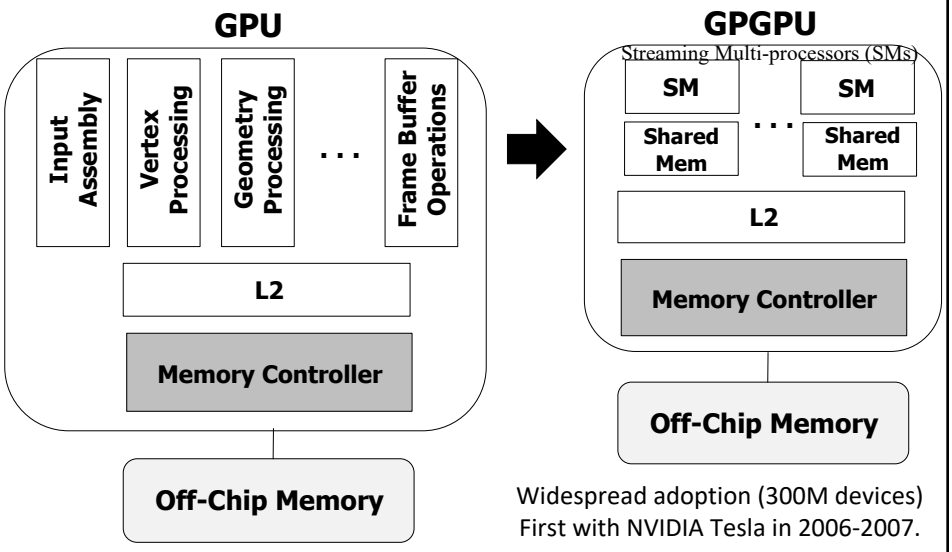
Graphical Processing Units

- Given the hardware invested to do graphics well, how can we supplement it to improve performance of a wider range of applications?

- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is "Single Instruction Multiple Thread"
  - SIMD is not exposed to programmers.

5

From GPU to GPGPU



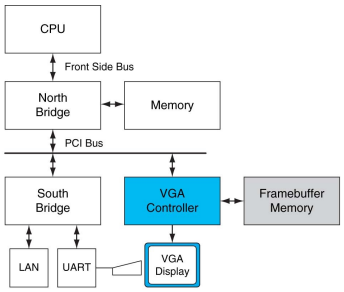
Widespread adoption (300M devices)
First with NVIDIA Tesla in 2006-2007.

Slide credit: Manish Arora

6

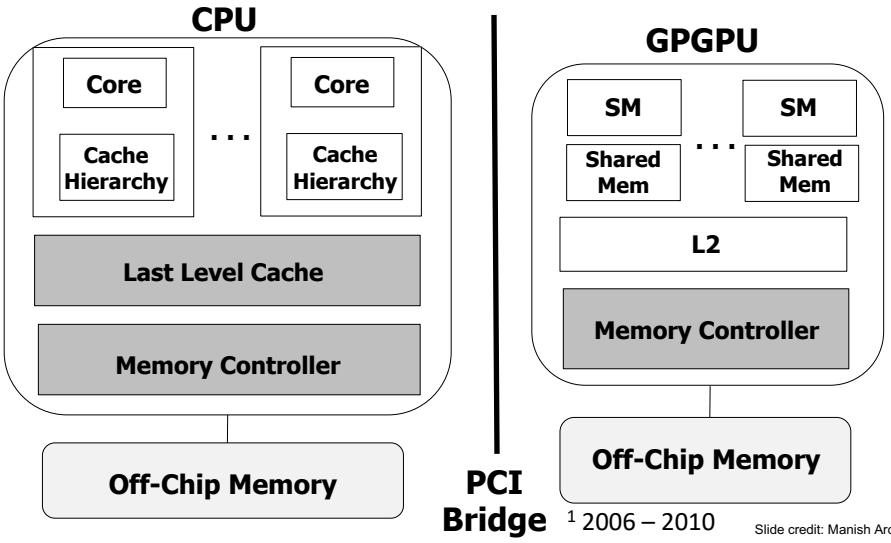
History of GPUs

- VGA (Video graphic array) has been around since the early 90's
  - A display generator connected to some (video) RAM
- By 2000, VGA controllers were handling almost all graphics computation
  - Programmable through OpenGL, Direct 3D API
  - APIs allowed accelerated vertex/pixel processing:
    - Shading
    - Texture mapping
    - Rasterization
  - Gained moniker Graphical Processing Unit
- 2007: First general purpose use of GPUs
  - 2007: Release of CUDA language
  - 2011: Release of OpenCL language



7

Consumer Hardware[1]



[1] 2006 – 2010

Slide credit: Manish Arora

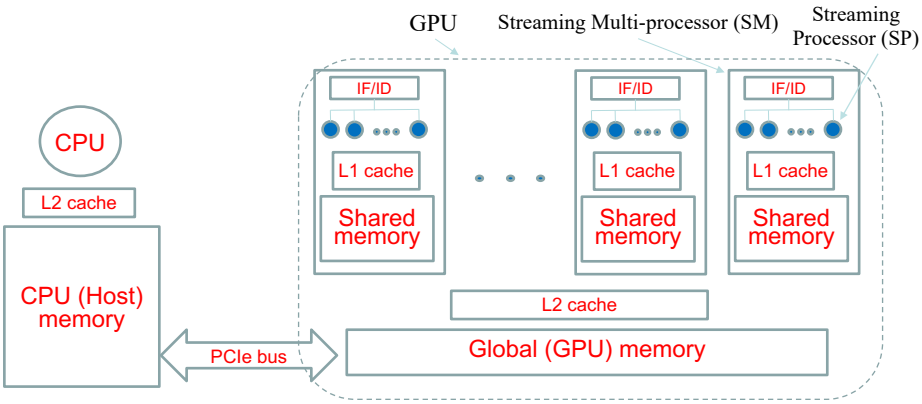
8

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

9

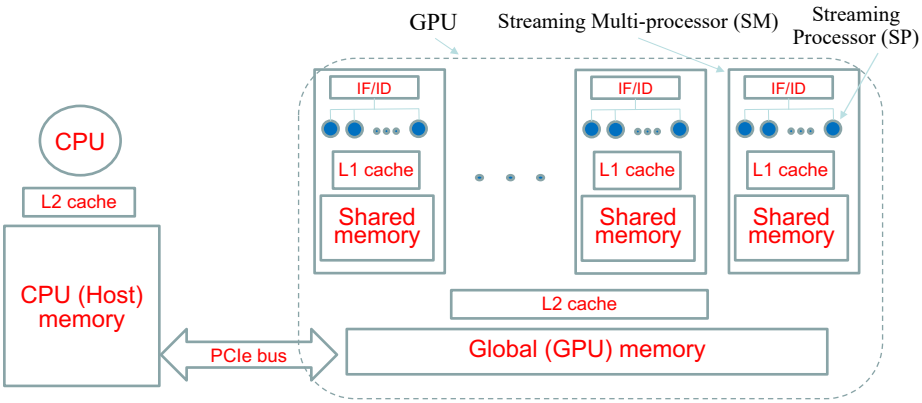
GPU is Really a SIMD Processor



- Logically, a GPU is composed of **SM**s (Streaming Multi-processors)
  - An SM is a **vector** unit that can process multiple pixels (or data items)
- Each SM is composed of **SPs** which work on each pixel or data item

10

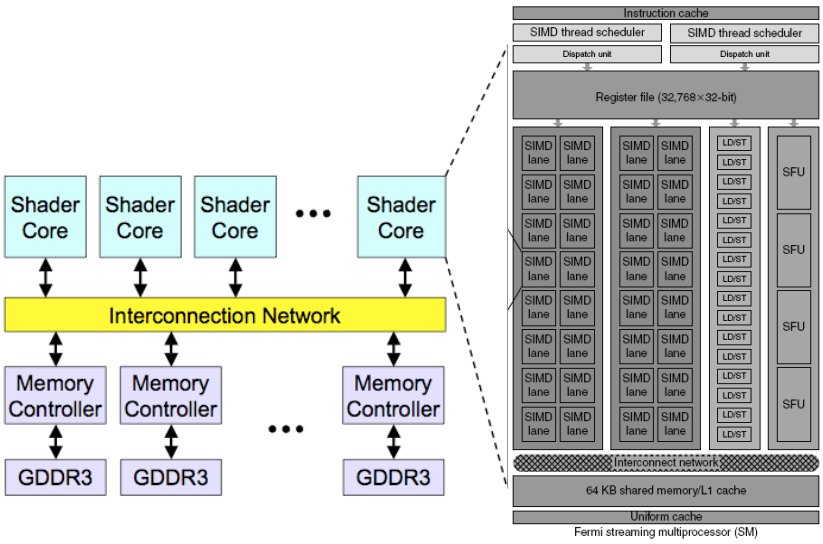
CPU-GPU architecture



- Dedicated GPU memory separate from system memory
- Code and data must be transferred to GPU memory for it to work on it
  - Through PCI-Express bus connecting GPU to CPU

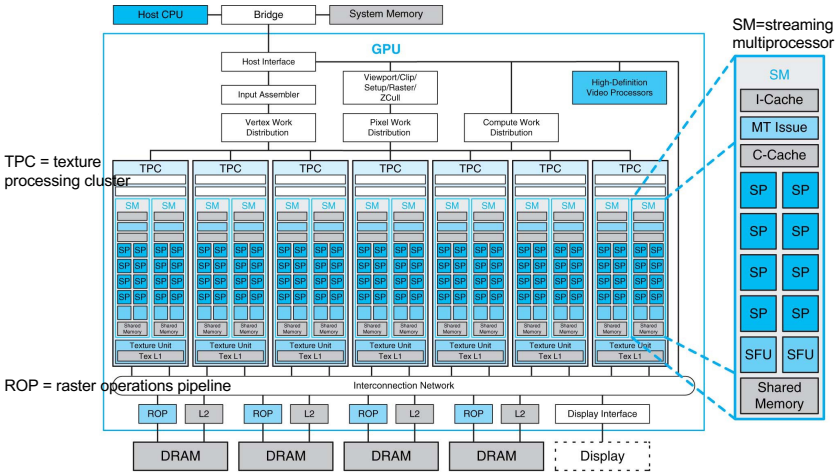
11

High-Level View of a GPU



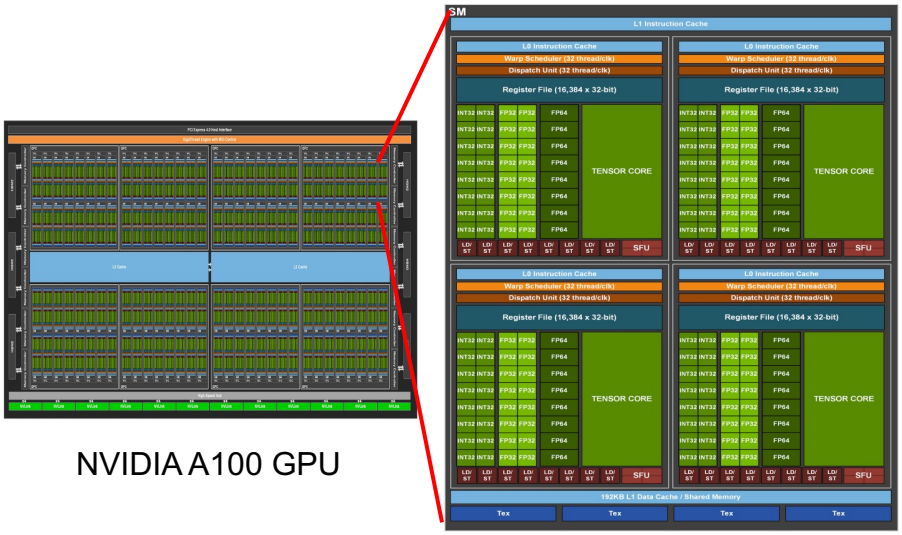
12

More recent GPU architecture



13

Modern GPU architecture

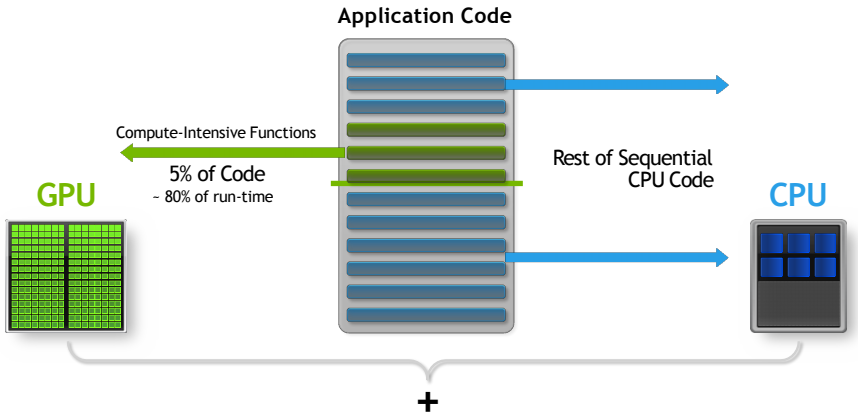


NVIDIA A100 GPU

14

Source: NVIDIA volta GPU whitepaper

HOW GPU ACCELERATION WORKS



15

Introduction to CUDA Programming

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

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

16

GPU Programming Model

CPU program
(serial code)

cudaMemcpy (…)

Copy data from CPU memory to GPU memory

Function <<<nb,nt >>>

Launch kernel on GPU

cudaMemcpy (…)

Copy results from GPU memory to CPU memory

global Function (…)

Implementation of GPU kernel
kernel: Function executed on the GPU

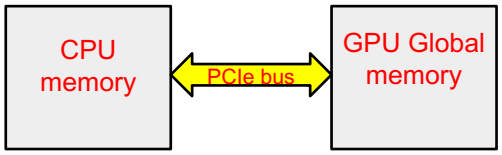
17

GPU Programming Model: Copying Data

```
/* malloc in GPU global memory */
cudaMalloc (void **pointer, size_t nbytes);
/* free malloced GPU global memory */
cudaFree(void **pointer) ;
/* initialize GPU global memory with value */
cudaMemset (void *pointer, int value, size_t count);
/* copy to and from between CPU and GPU memory */
cudaMemcpy(void *dest, void *src, size_t nbytes, enum cudaMemcopyKind dir);
```

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice



18

Example: Copying array a to array b using the GPU

Data Movement Example

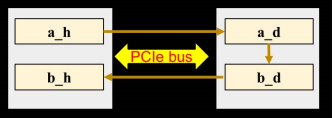
```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   GPUcomp<<<1, 14>>>(a_d, b_d, N);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```




```
_global_ void GPUcomp(*a,*b,N)
{
  int i = threadIdx.x ;
  if( i < N) b(i) = a(i) ;
}
```

19

GPU Programming Model: Launching the Kernel

CPU program
(serial code)

cudaMemcpy (…)

Copy data from CPU memory to GPU memory

Function <<<nb,nt >>>

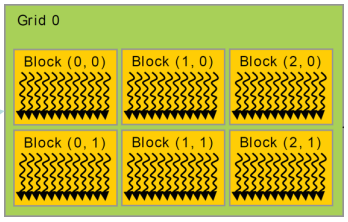
Launch a kernel with *nb* blocks, each with *nt* threads

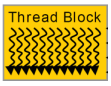
cudaMemcpy (…)

Copy results from GPU memory to CPU memory

global Function (…)

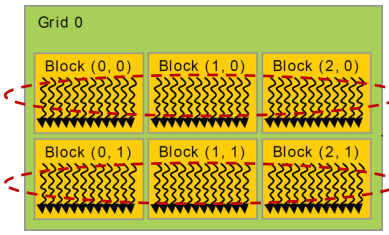
Implementation of kernel (the function run by each GPU thread)

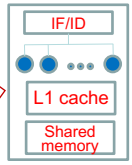




20

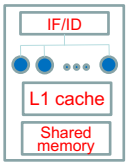
The Execution Model

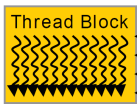




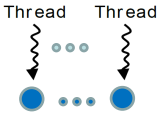
- The **thread blocks** are dispatched to **SM**s
- The number of blocks dispatched to an SM depends on the SM's resources (registers, shared memory, …).

Blocks not dispatched initially are dispatched when an SM frees up after finishing a block



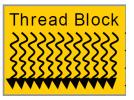


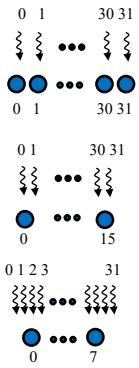
- When a block is dispatched to an SM, each of its threads executes on an SP in the SM.



21

The Execution Model



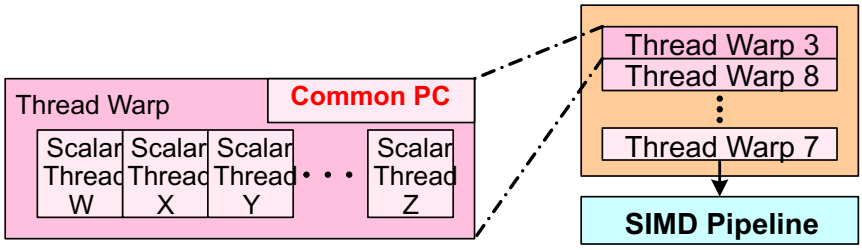


- Each block (up to 1K threads) is divided into groups of 32 threads (called **warps**) – empty threads are used as fillers.
- A warp executes as a SIMD **vector instruction** on the SM.
- Depending on the number of SPs per SM:
  - If 32 SP per SM → 1 thread of a warp executes on 1 SP (32 lanes of execution, one thread per lane)
  - If 16 SP per SM → 2 threads are time multiplexed on 1 SP (16 lanes of execution, 2 threads per lane)
  - If 8 SP per SM → 4 threads are time multiplexed on 1 SP (8 lanes of execution, 4 threads per lane)

22

Concept of "Thread Warps" and SIMT

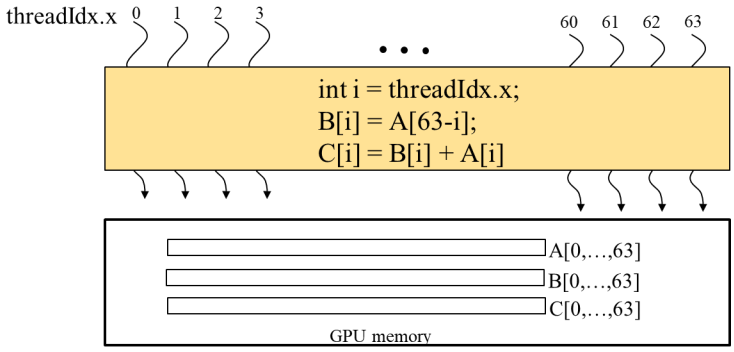
- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same kernel
- Warp: The threads that run lengthwise in a woven fabric …



23

All threads execute the same code

- Launched using **Kernel <<<1, 64>>>** : 1 block with 64 threads

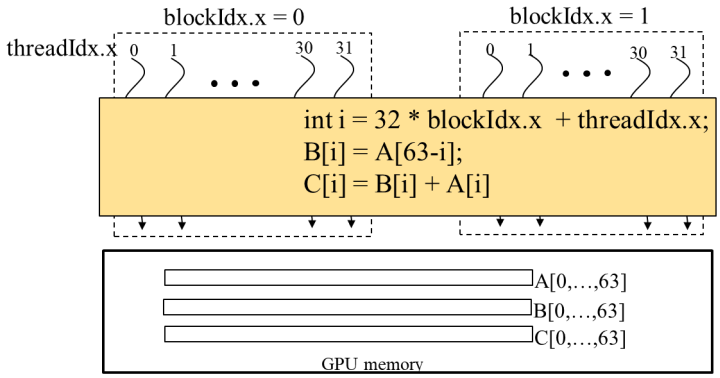


- Each thread in a thread block has a unique "thread index" → **threadIdx.x**
- The same sequence of instructions can apply to different data items.

24

Blocks of Threads

- Launched using **Kernel <<<2, 32>>>** : 2 blocks of 32 threads

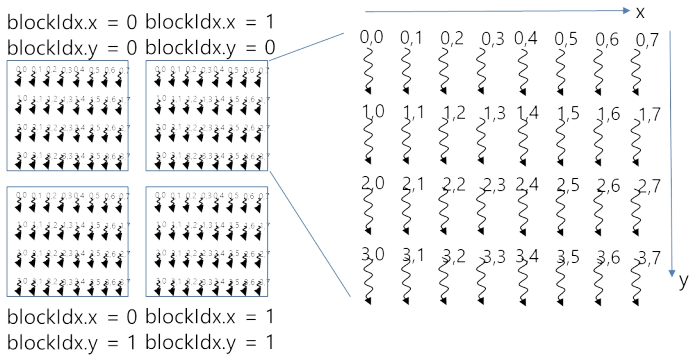


- Each thread block has a unique "block index" → **blockIdx.x**
- Each thread has a unique **threadIdx.x** within its own block
- Can compute a global index from the blockIdx.x and threadIdx.x

25

Two-dimensions grids and blocks

- Launched using **Kernel <<<(2, 2), (4, 8)>>>** : 2X2 blocks of 4X8 threads

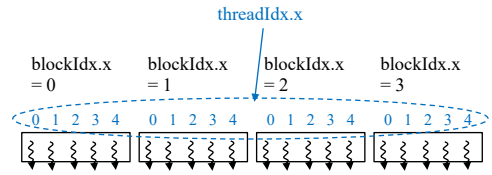


- Each block has two indices **(blockIdx.x, blockIdx.y)**
- Each thread in a thread block has two indices **(threadIdx.x, threadIdx.y)**

26

Example: Computing the global index

```
void main ()
{  cudaMalloc (int* &a, 20*sizeof(int));
   cudaMalloc (int* &b, 20*sizeof(int));
   cudaMalloc (int* &c, 20*sizeof(int));
   …
   kernel<<<4,5>>>(a, b, c) ;
   …
}
_global_ void kernel(int *a, *b, *c)
{ int i = blockIdx.x * blockDim.x + threadIdx.x ;
  a[i] = i ;
  b[i] = blockIdx.x;
  c[i] = threadIdx.x;
}
```



NOTE: Each block will consist of one warp – only 5 threads in warp will do useful work. (Other 27 threads will execute no-ops.)

Global Memory

a[]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b[]	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3
c[]	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

27

Example: Computing $y(i) = a * x(i) + y(i)$

C program (on CPU)

```
void saxpy_serial(int n, float a, float
 *x, float *y)
 {
   for(int i = 0; i<n; i++)
         y[i] = a * x[i] + y[i];
 }
```

```
void main ()
{
  …
  saxpy_serial(n, 2.0, x, y);
  …
}
```

CUDA program (on CPU+GPU)

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

```
void main ()
{ …
  // cudaMalloc arrays X and Y
  // cudaMemcpy data to X and Y
  int NB = (n + 255) / 256;
  saxpy_gpu<<<NB, 256>>>(n, 2.0, X, Y);
  // cudaMemcpy data from Y
}
```

28

Example: Computing $y(i) = a * x(i) + y(i)$

- What happens when n = 1?

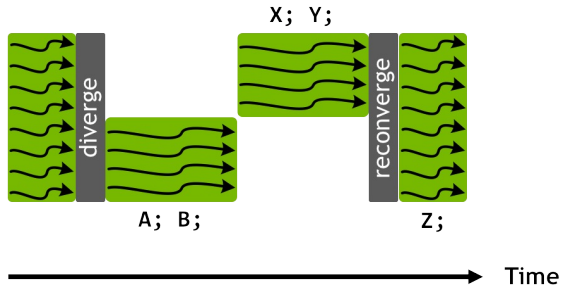
```
_global_void saxpy_gpu(int n, float a, float *X, float *Y)
{
  int i = blockIdx.x*blockDim.x +  threadIdx.x;
  if (i < n ) Y[i] = a * X[i] + Y[i];
}
…..
saxpy_gpu<<<1, 256>>>(1, 2.0, X, Y); /* X and Y are both sized 1! */
```

- "if (i < n)" condition prevents writing beyond bounds of array.
- But that requires some threads within a **warp** not performing the write.
  - But a warp is a single vector instruction. How can you branch?
  - "if (i < n)" creates a **predicate** "mask" vector to use for the write
  - Only thread 0 has predicate turned on, rest has predicate turned off

29

GPUs Use Predication for Branches

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



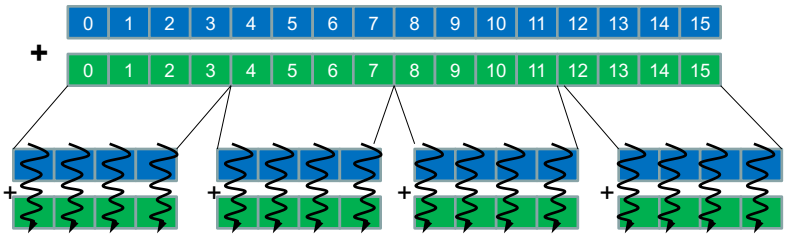
- Each thread computes own predicate for condition threadIdx.x < 4
- Taken together, 32 threads of a warp create a 32-bit predicate mask
- Mask is applied to warps for A, B, X, and Y.
- Just like for VLIW processors, this can lead to **low utilization**.

30

SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements

Let's assume N=16, blockDim=4 → 4 blocks

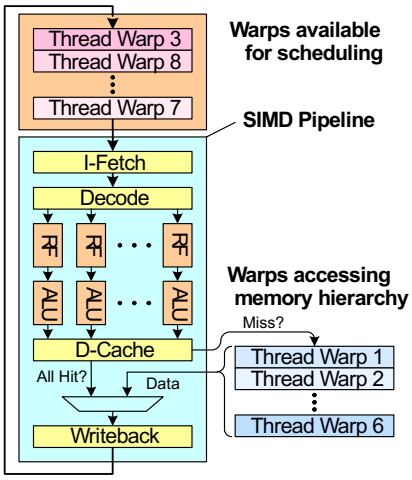


Slide credit: Hyesoon Kim

36

Latency Hiding with "Thread Warps"

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- No OS context switching
- Memory latency hiding
  - Graphics has millions of pixels



Slide credit: Tor Aamodt

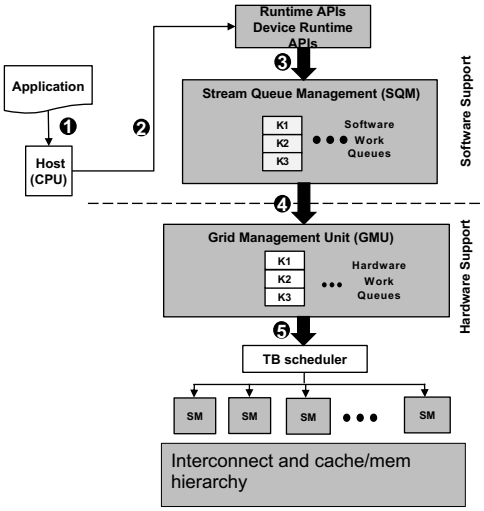
37

Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions
- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

38

CUDA Execution



39