SIMD and GPUs

CS/COE 1541 (Fall 2020)
Wonsun Ahn

SIMD Architectures

ISA not optimized for data parallel workloads

- This loop does multiply accumulate (MAC):

```
for (int i = 0; i < 64; i++) {
  y[i] = a * x[i] + y[i]
}
```

  - A common operation in digital signal processing

- Note how we apply the **same** MAC **operation on each data item**
  - This is how many data parallel workloads look like

- A conventional ISA (likes MIPS) is not optimal for encoding this
  - Results in wasted work and suboptimal performance
  - Let's look at the actual MIPS translation

MIPS code for $y(i) = a * x(i) + y(i)$

```
       l.d    $f0,0($sp)       ;$f0 = a
       addi   $s2,$s0,512      ;64 elements (64*8=512 bytes)
loop:  l.d    $f2,0($s0)       ;$f2 = x(i)
       mul.d  $f2,$f2,$f0      ;$f2 = a * x(i)
       l.d    $f4,0($s1)       ;$f4 = y(i)
       add.d  $f4,$f4,$f2      ;$f4 = a * x(i) + y(i)
       s.d    $f4,0($s1)       ;y(i) = $f4
       addi   $s0,$s0,8        ;increment index to x
       addi   $s1,$s1,8        ;increment index to y
       subu   $t0,$s2,$s0      ;evaluate i < 64 loop condition
       bne    $t0,$zero,loop   ;loop if not done
```

- Blue instructions don’t do actual computation. There for indexing and loop control.
  - Is there a way to avoid? Loop unrolling yes. But that causes code bloat!
- Red instructions do computation. But why decode them over and over again?
  - Is there a way to fetch and decode once and apply to all data items?

SIMD (Single Instruction Multiple Data)

- **SIMD (Single Instruction Multiple Data)**
  - An architecture for applying one instruction on multiple data items
  - ISA includes **vector instructions** for doing just that
    - Along with **vector registers** to hold multiple data items

- Using MIPS vector instruction extensions:

```
l.d      $f0,0($sp)    ;$f0 = scalar a
lv       $v1,0($s0)    ;$v1 = vector x (64 values)
mulvs.d  $v2,$v1,$f0   ;$v2 = a * vector x
lv       $v3,0($s1)    ;$v3 = vector y (64 values)
addv.d   $v4,$v2,$v3   ;$v4 = a * vector x + vector y
sv       $v4,0($s1)    ;vector y = $v4
```

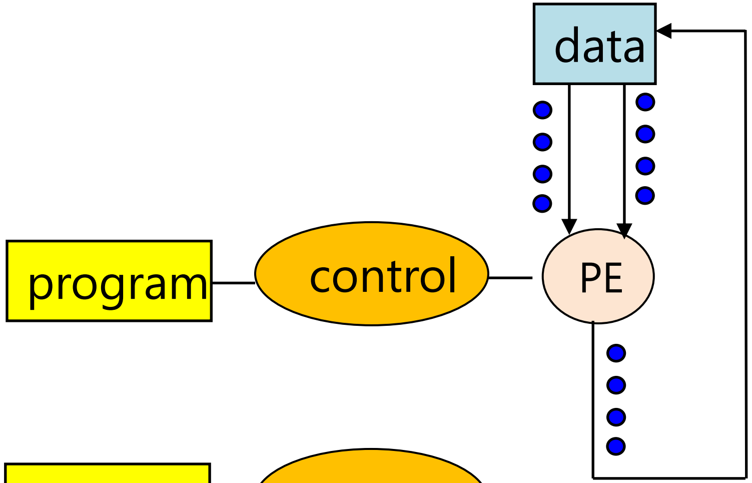
  - Note: no indexing and loop control overhead
  - Note: each instruction is fetched and decoded only once

University of Pittsburgh

SIMD Processor Design

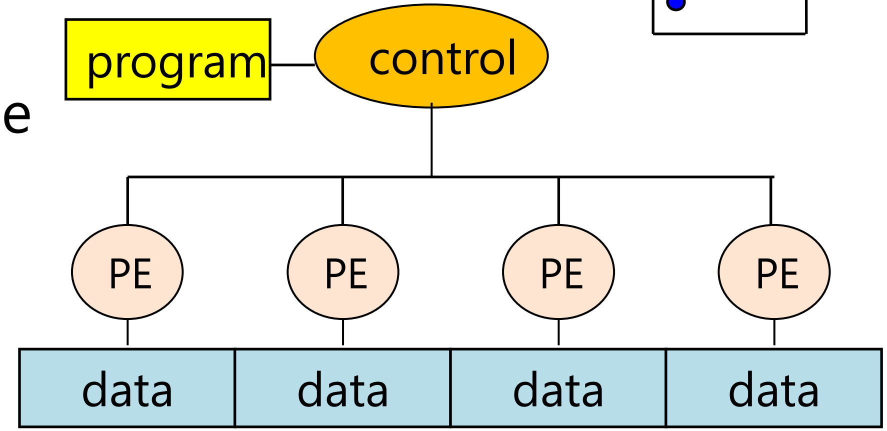
- How would you design a processor for the vector instructions?

1. One processing element (PE)
   - ○ Fetch and decode instruction once
   - ○ PE applies op on each data item
     - ▪ Item may be in vector register
     - ▪ Item may be in data memory



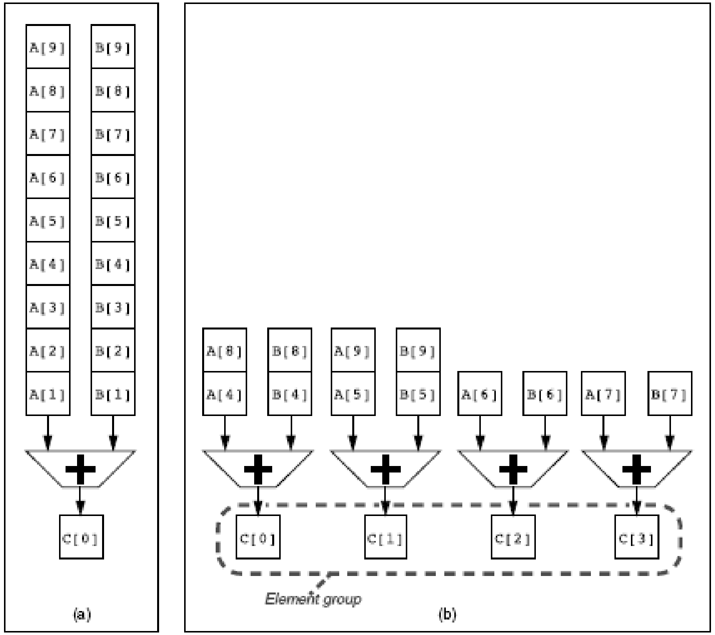
2. Multiple PEs in parallel
   - ○ Fetch and decode instruction once
   - ○ PEs apply op in parallel
     - ▪ In synchronous lockstep

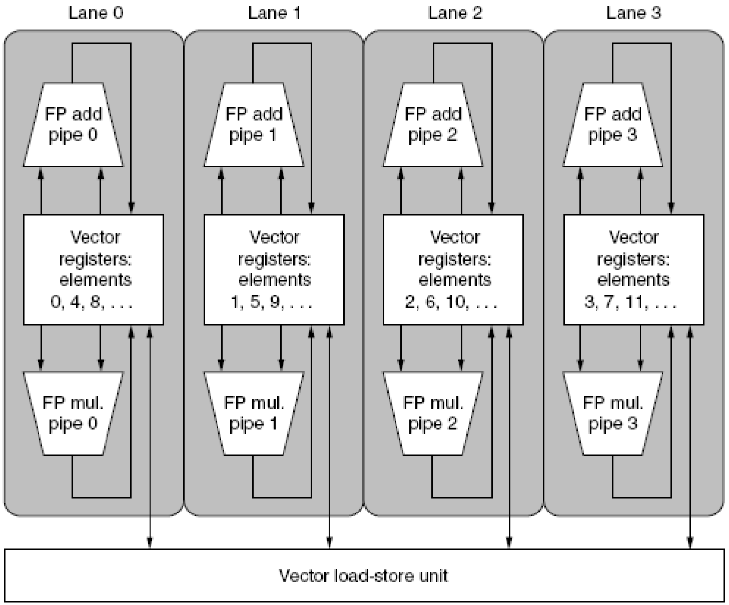
   → The more PEs, the faster!



Example: Adding Two Vectors

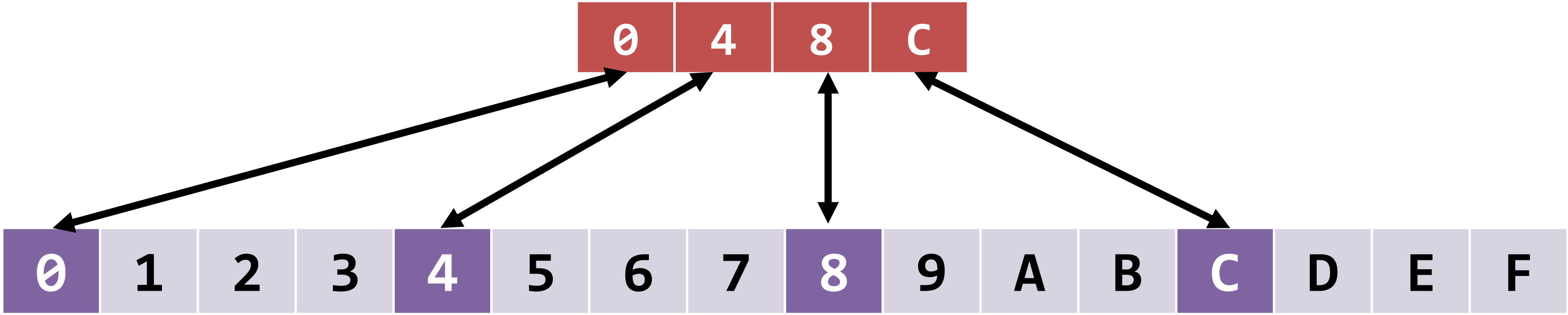
- Instead of having a single FP adder work on each item (a)
- Have four FP adders work on items in parallel (b)
- Each pipelined FP unit is in charge of pre-designated items in vector
  - For full parallelization, put as many FP units as there are items



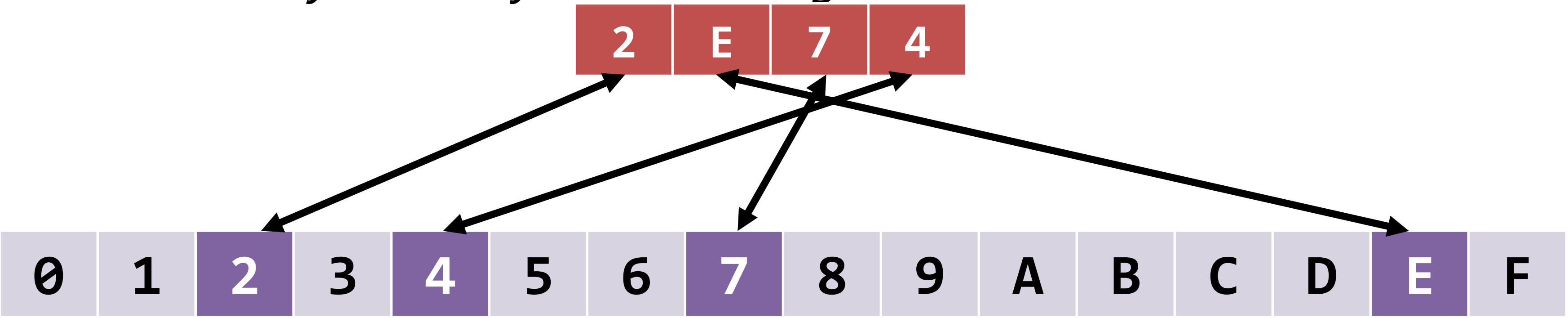

Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, . . .
Vector registers: elements 1, 5, 9, . . .
Vector registers: elements 2, 6, 10, . . .
Vector registers: elements 3, 7, 11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

Vector Load-Store Unit

- *Striding* lets you load/store *non-contiguous* data from memory at regular offsets. (e.g. the first member of each struct in an array)



- *Gather-scatter* lets you put pointers in a vector, then load/store from *arbitrary memory addresses. (gather = load, scatter = store)*



Vector Load-Store Unit

- Contiguous data items is still the best for performance
  - Means processor needs to access only one or a few cache blocks

- Strided or scattered accesses are possible but bad for performance
  - If any of the multiple cache blocks accessed miss, long latency
  - Accessing multiple blocks also consumes a lot of bandwidth

- If your vector program is slow, 99% it is because of memory latency

SIMD instructions in real processors

- x86 vector extensions
  - MMX, SSE, AVX, AVX-2
  - Current: AVX-512 (512-bit vector instructions)

- ARM vector extensions
  - VFP (Vector Floating Point)
  - Current: Neon (128-bit vector instructions)

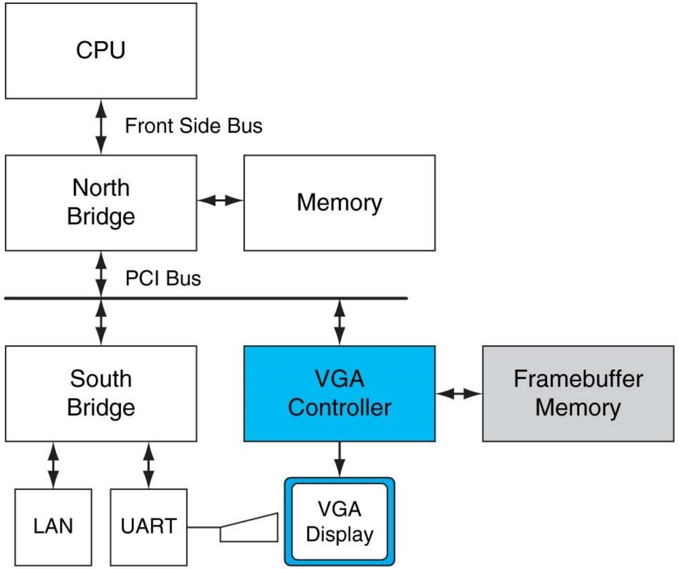
- Vector instructions have progressively become wider historically
  - Due to increase of data parallel applications

- Enter GPUs for general computing (circa 2001)

GPUs:
Graphical Processing Units

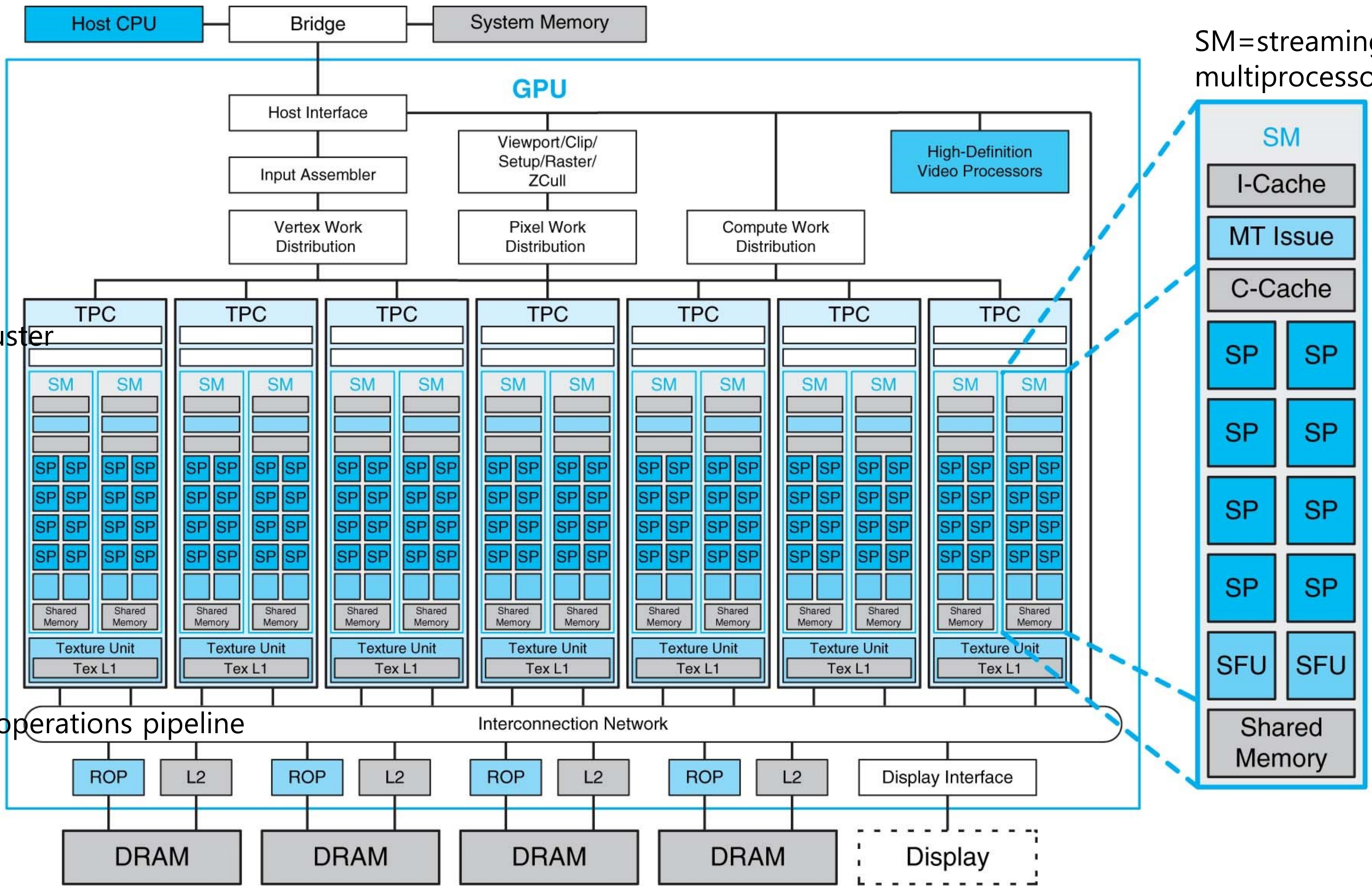
History of GPUs

- 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



University of Pittsburgh

Modern GPU architecture

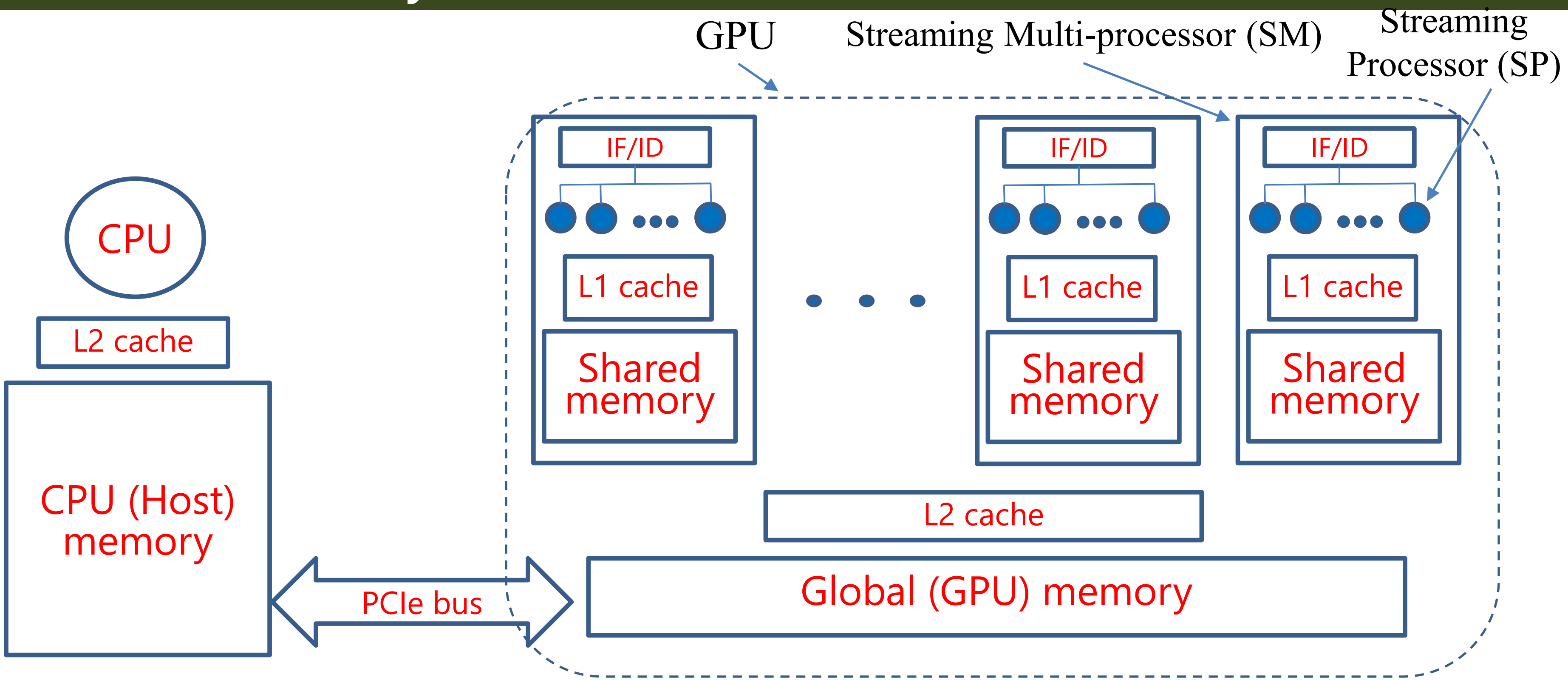


SM=streaming multiprocessor

TPC = texture processing cluster

ROP = raster operations pipeline

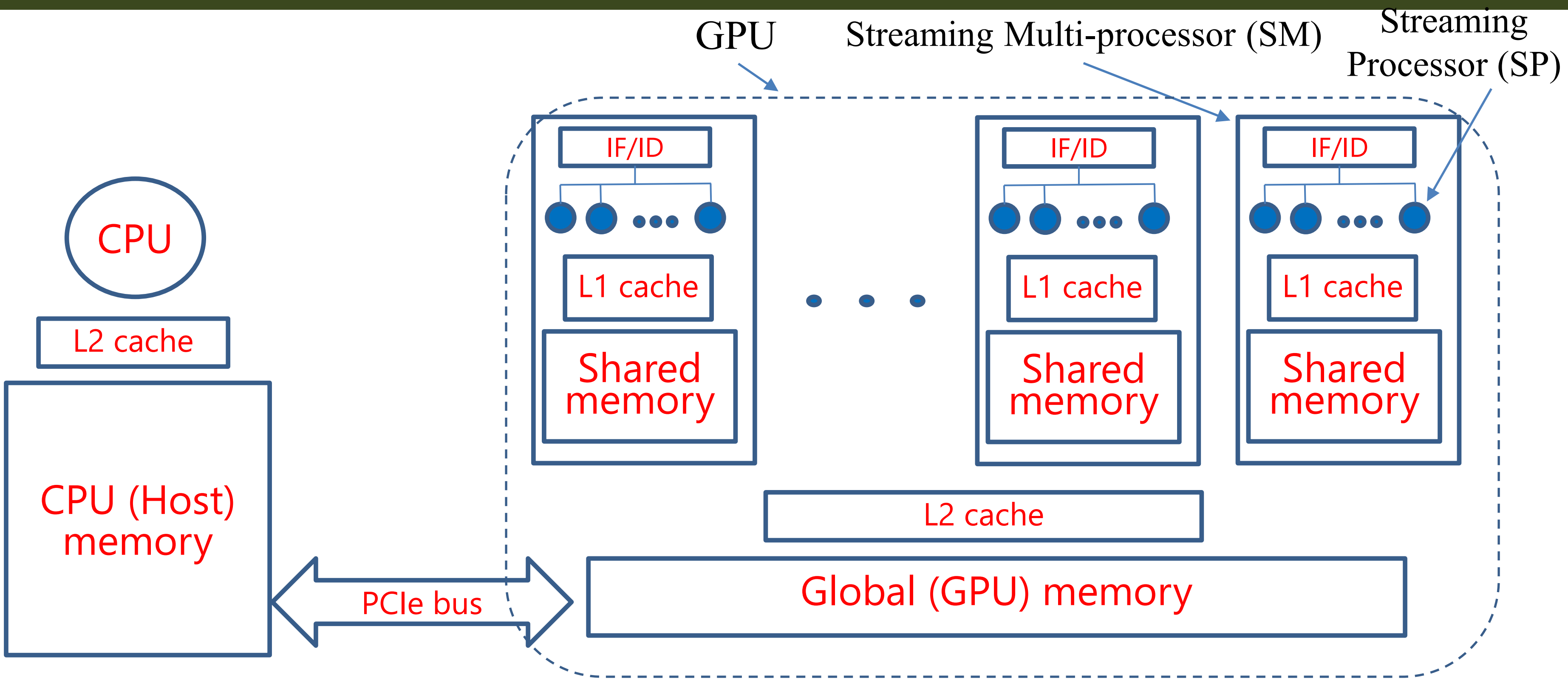
GPU is Really a SIMD Processor



- 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

University of Pittsburgh

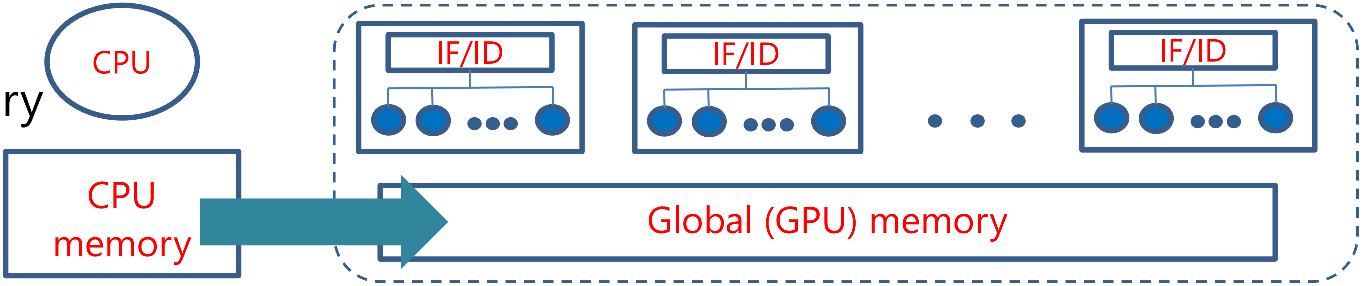
CPU-GPU architecture



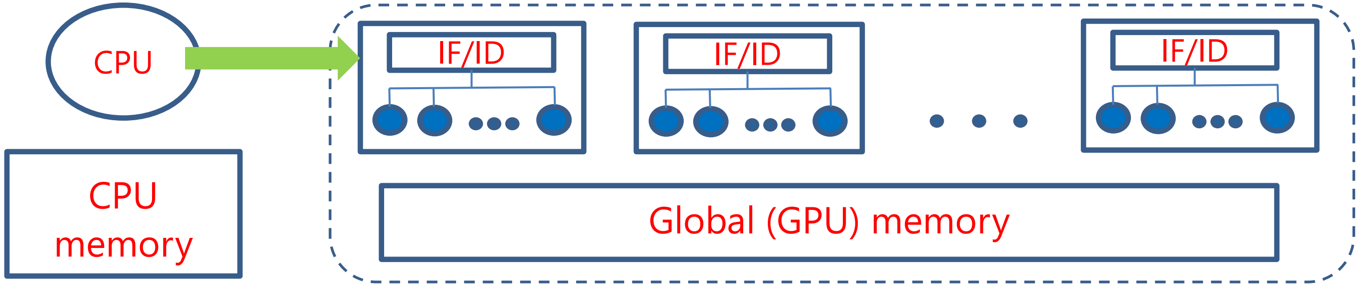
- Dedicated GPU memory separate from system memory
- Code and data must be transferred to GPU memory for it to work on it
  - Through PCI-Express bus connecting GPU to CPU

GPU Programming Model

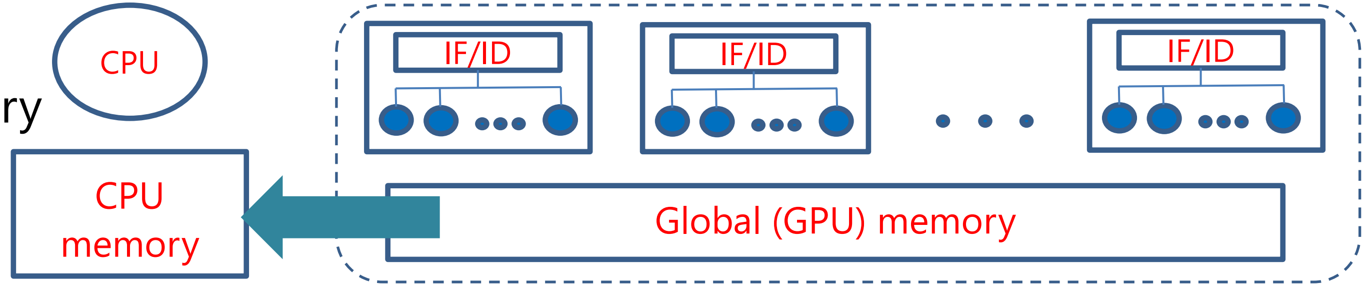
Copy data from CPU memory to GPU memory



Launch the **kernel**



Copy data from GPU memory to CPU memory

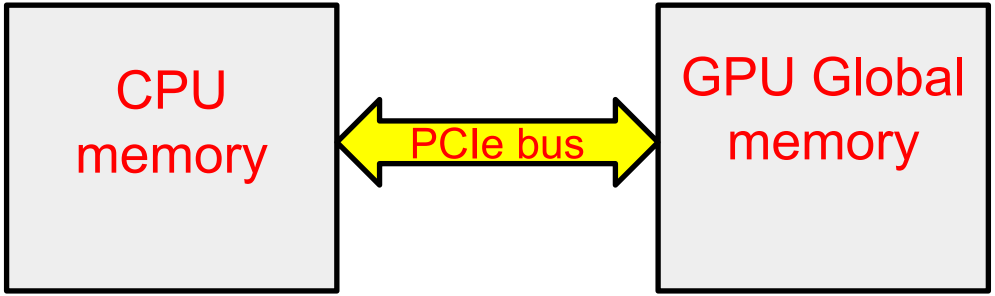


GPU Programming Model: Exchanging Data

```
cudaMalloc (void **pointer, size_t nbytes); /* malloc in GPU global memory */
cudaMemset (void **pointer, int value, size_t count);
cudaMemcpy(void *dest, void *src, size_t nbytes, enum cudaMemcpyKind dir)
cudaFree(void **pointer) ;
```

enum cudaMemcpyKind

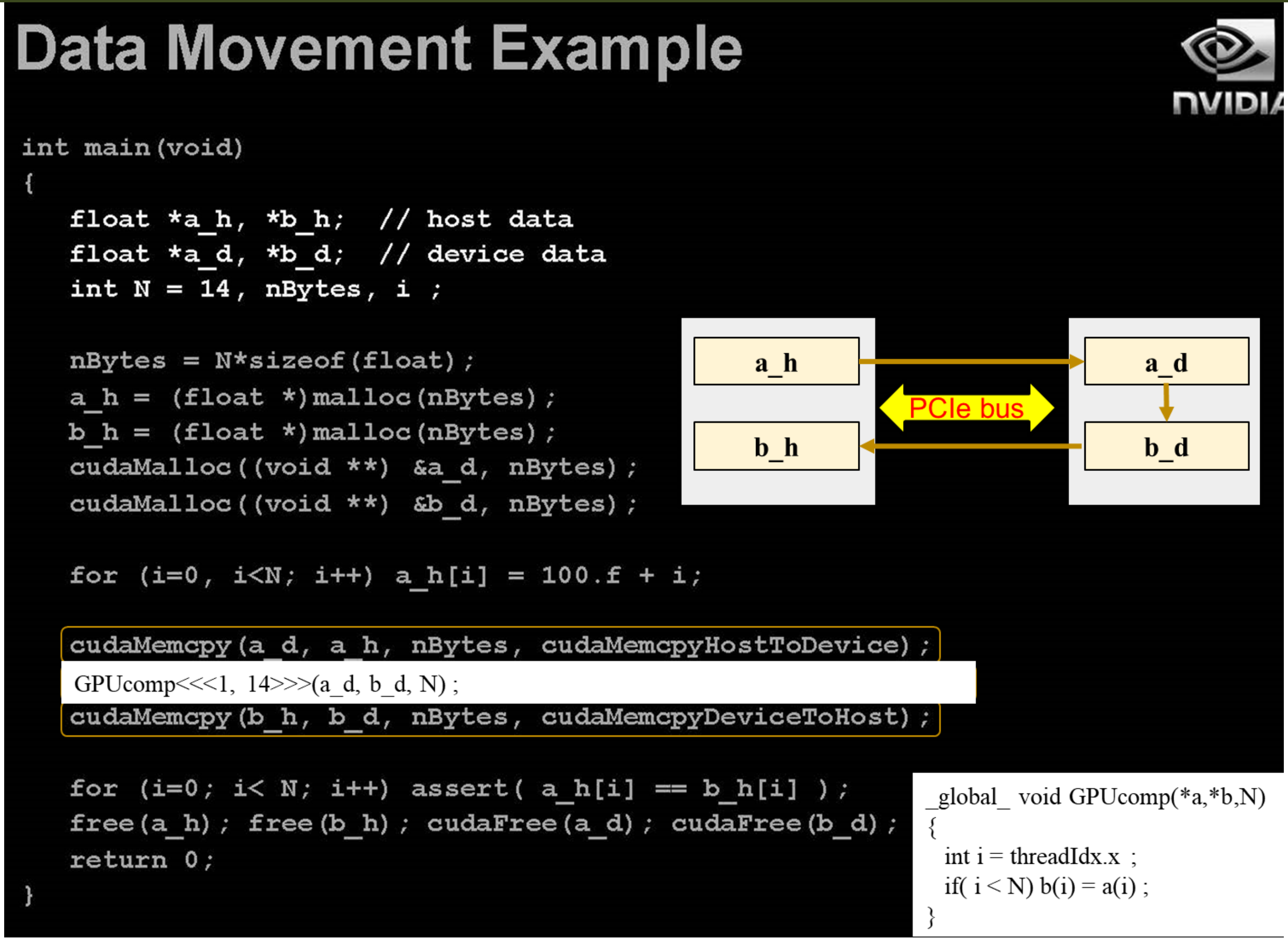
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice



Notes:

- cudaMemcpy() blocks CPU thread until copy is complete
- cudaMemcpy() does not start copying until previous CUDA calls complete

Example: Copying array a to array b



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) ;
}
```

GPU Programming Model: Launching the Kernel

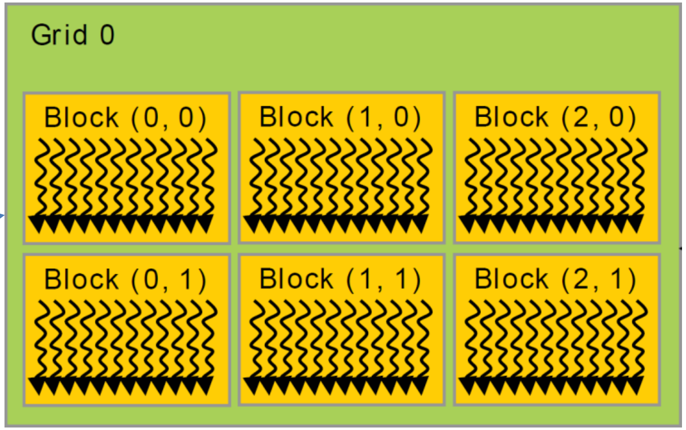
CPU program
(serial code)

cudaMemcpy (…)

Copy data from CPU memory to GPU memory

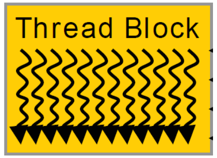
Function <<<nb,nt >>>

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



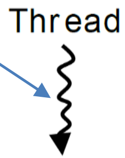
cudaMemcpy (…)

Copy results from GPU memory to CPU memory

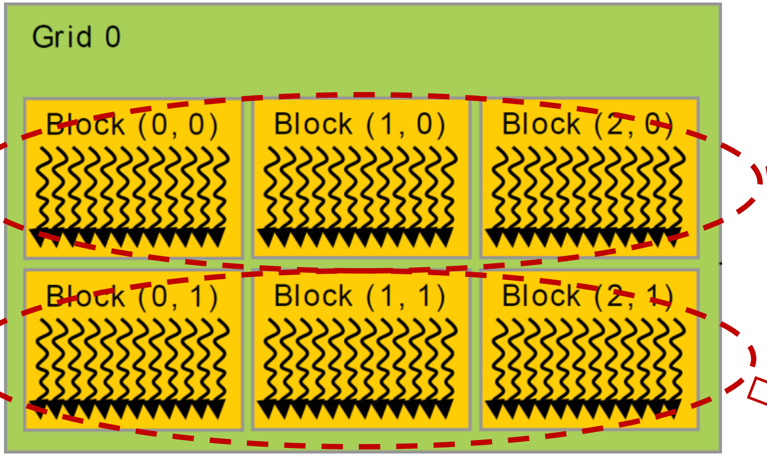


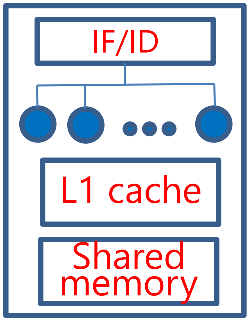
global Function (…)

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



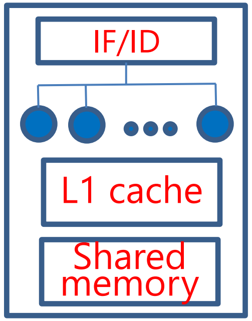
The Execution Model

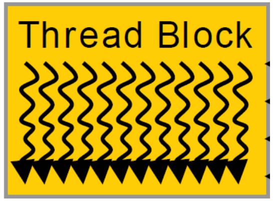




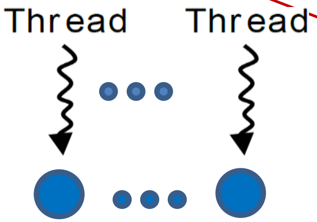
- The **thread blocks** are dispatched to **SM**s
- The number of blocks dispatched to an SM depends on the SM’s resources (registers, shared memory, …).

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

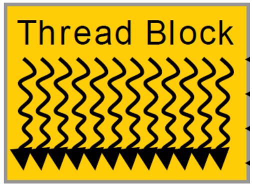




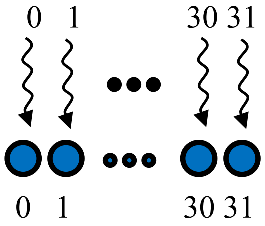
- When a block is dispatched to an SM, each of its threads executes on an SP in the SM.



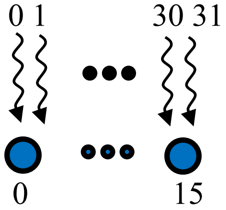
The Execution Model



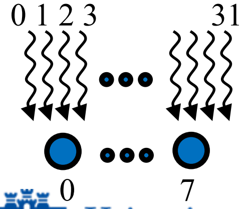
- Each block (up to 1K threads) is divided into groups of 32 threads (called **warps**) – empty threads are used as fillers.
- A warp executes as a SIMD **vector instruction** on the SM.
- Depending on the number of SPs per SM:



  - ○ If 32 SP per SM → 1 thread of a warp executes on 1 SP (32 lanes of execution, one thread per lane)



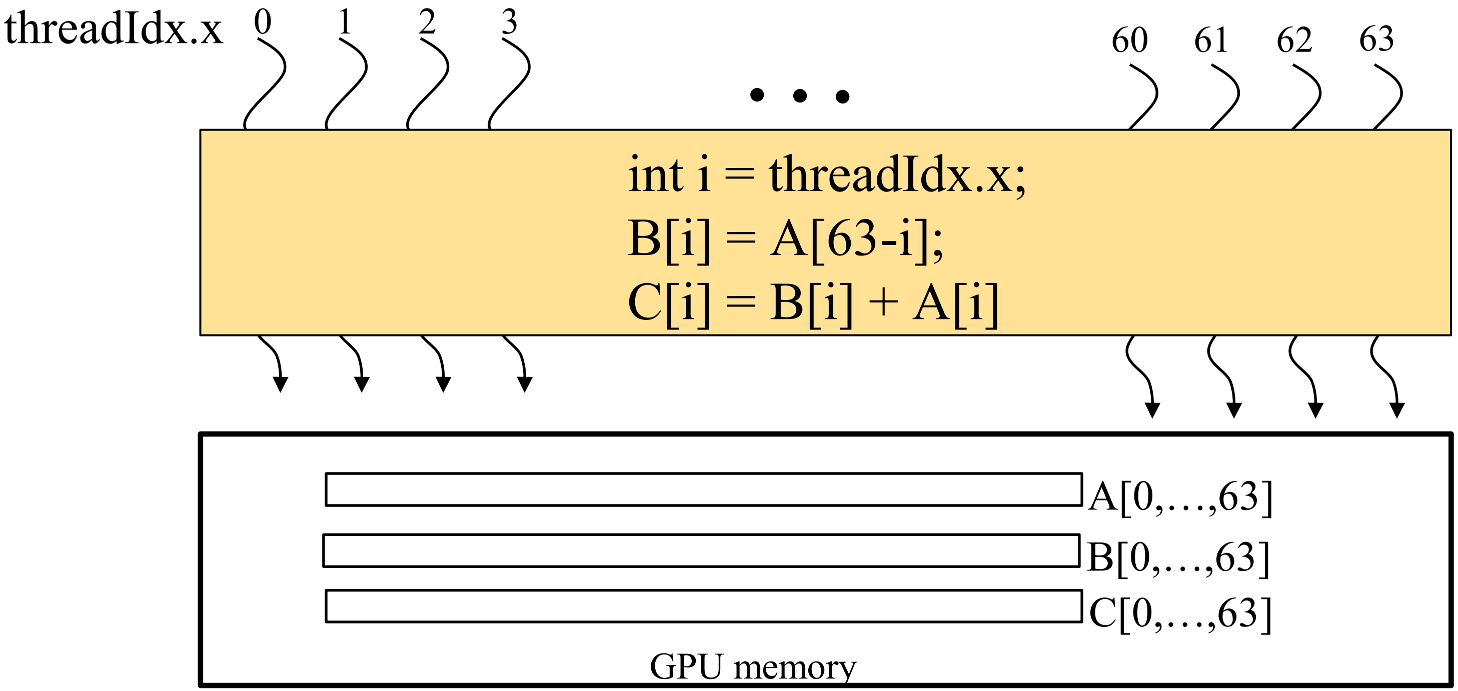
  - ○ If 16 SP per SM → 2 threads are time multiplexed on 1 SP (16 lanes of execution, 2 threads per lane)



  - ○ If 8 SP per SM → 4 threads are time multiplexed on 1 SP (8 lanes of execution, 4 threads per lane)
  - ○ Some SMs have more than 32 SPs and run 2 warps in parallel!

All threads execute the same code

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

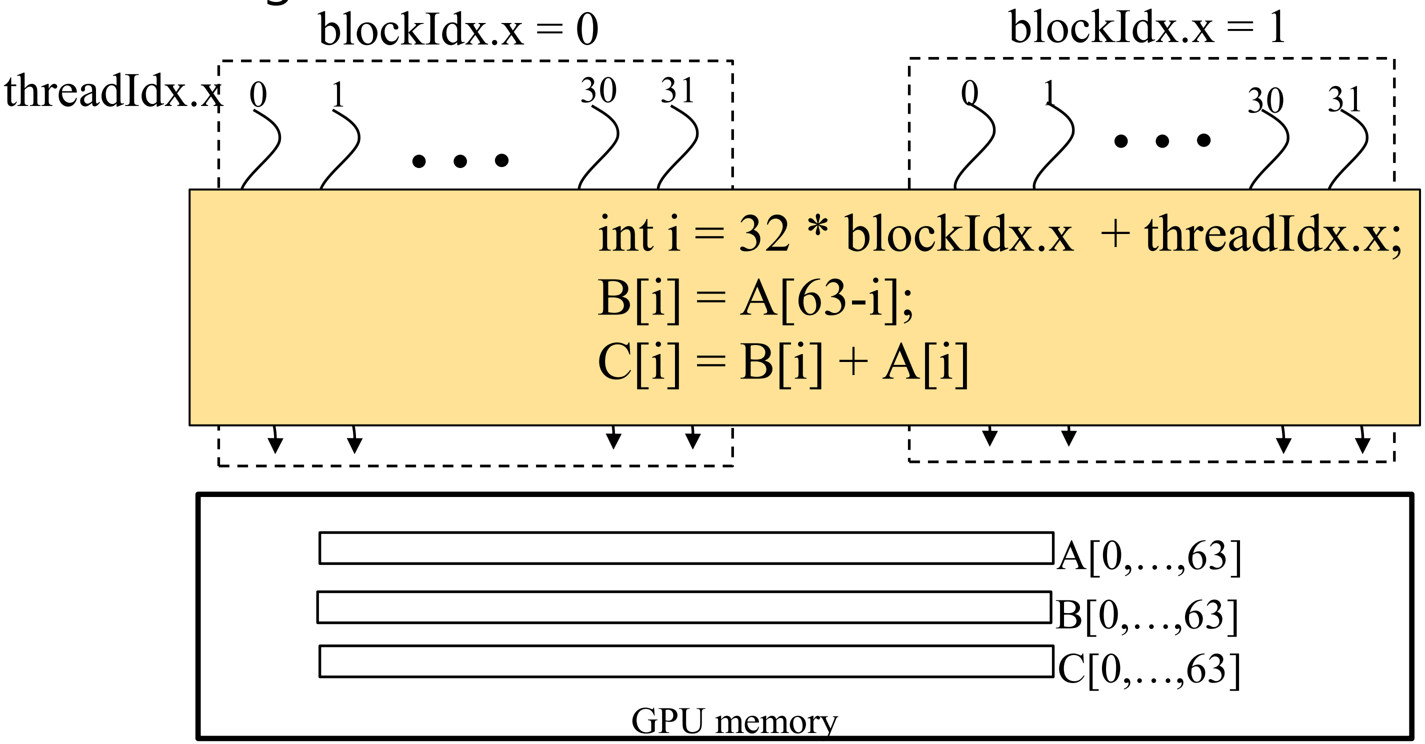


```
int i = threadIdx.x;
B[i] = A[63-i];
C[i] = B[i] + A[i]
```

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

Blocks of Threads

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



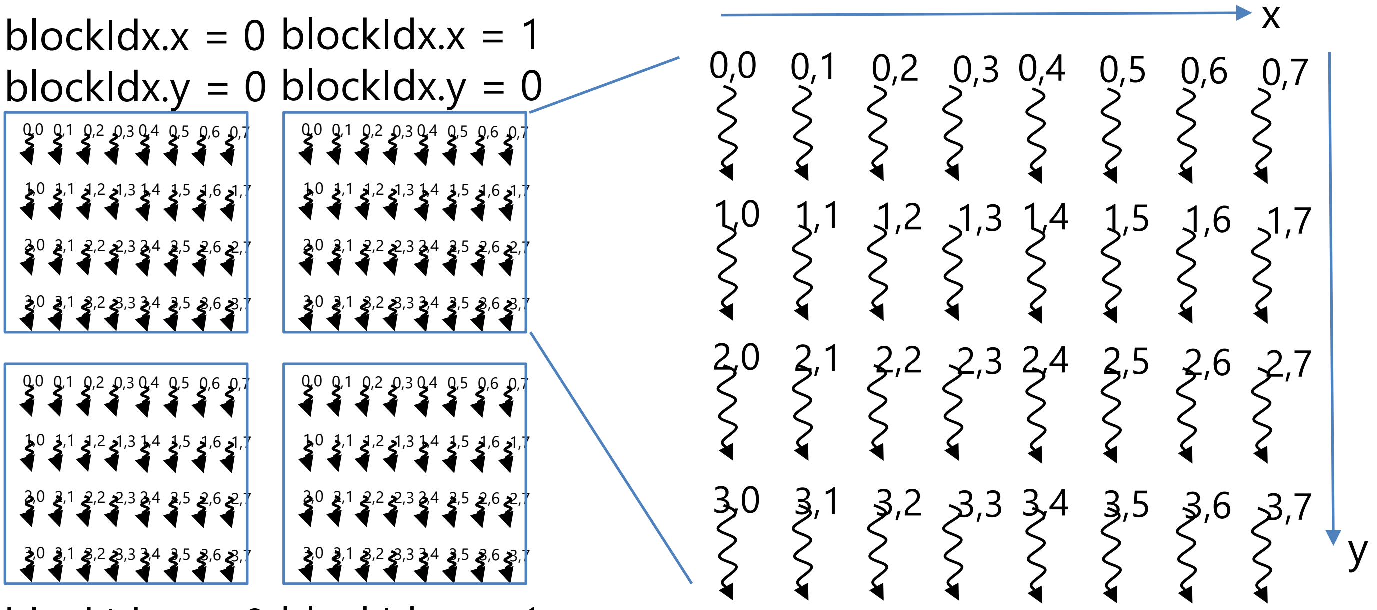
```
int i = 32 * blockIdx.x  + threadIdx.x;
B[i] = A[63-i];
C[i] = B[i] + A[i]
```

- 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

Two-dimensions grids and blocks

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

blockIdx.x = 0 blockIdx.x = 1
blockIdx.y = 0 blockIdx.y = 0

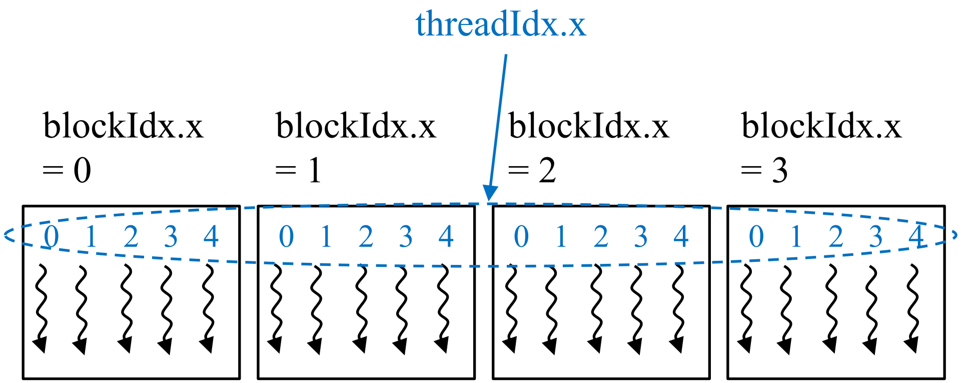


blockIdx.x = 0 blockIdx.x = 1
blockIdx.y = 1 blockIdx.y = 1

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

Example

```
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 the warp will do useful work and the 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

Example: Computing y = ax + y

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
}
```

Example: Computing y = ax + y

- 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** vector to use for the write
  - Only thread 0 has predicate turned on, rest has predicate turned off