

SIMD and GPUs

CS 1541
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 (DAXPY)
- Note how we apply the **same** MAC **operation on each data item**
 - This is how many data parallel workloads look like
- A conventional ISA (like 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 $y(i) = a * x(i) + y(i)$

```
1.d    $f0,0($sp)      ;$f0 = a
      addi   $s2,$s0,512 ;64 elements (64*8=512 bytes)
loop: 1.d    $f2,0($s0)      ;$f2 = x(i)
      mul.d  $f2,$f2,$f0 ;$f2 = a * x(i)
      1.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:

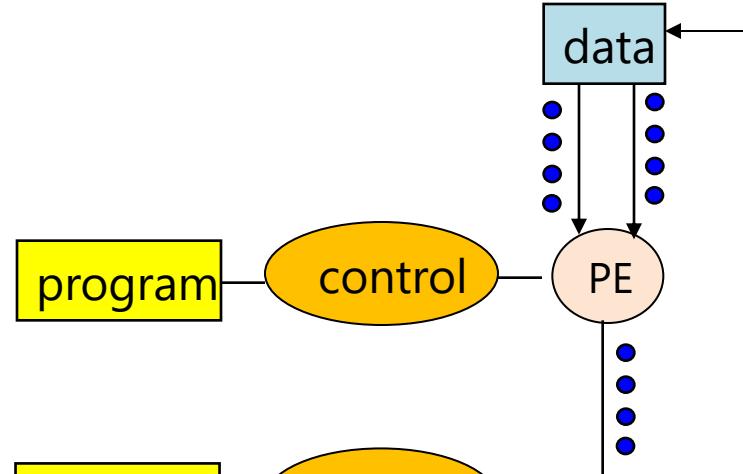
```
l.d      $f0,0($sp)    ;$f0 = scalar a  
l.v      $v1,0($s0)    ;$v1 = vector x (64 values)  
mulvs.d $v2,$v1,$f0   ;$v2 = a * vector x  
l.v      $v3,0($s1)    ;$v3 = vector y (64 values)  
addv.d  $v4,$v2,$v3   ;$v4 = a * vector x + vector y  
sv      $v4,0($s1)    ;vector y = $v4
```

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

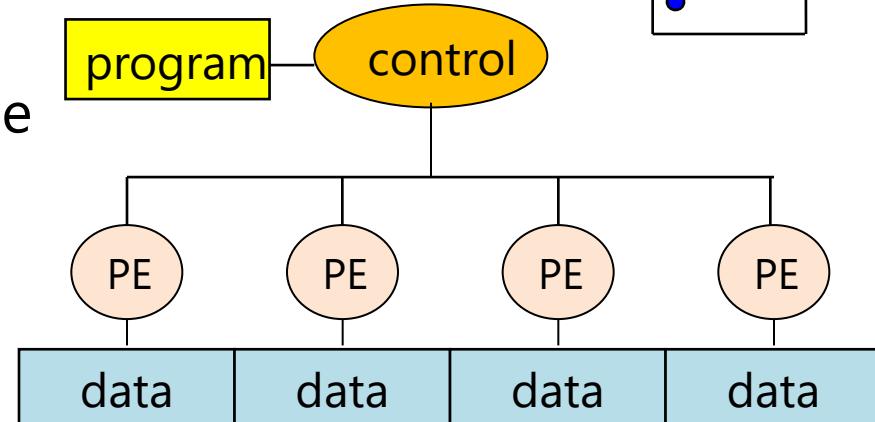
SIMD Processor Design

- 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

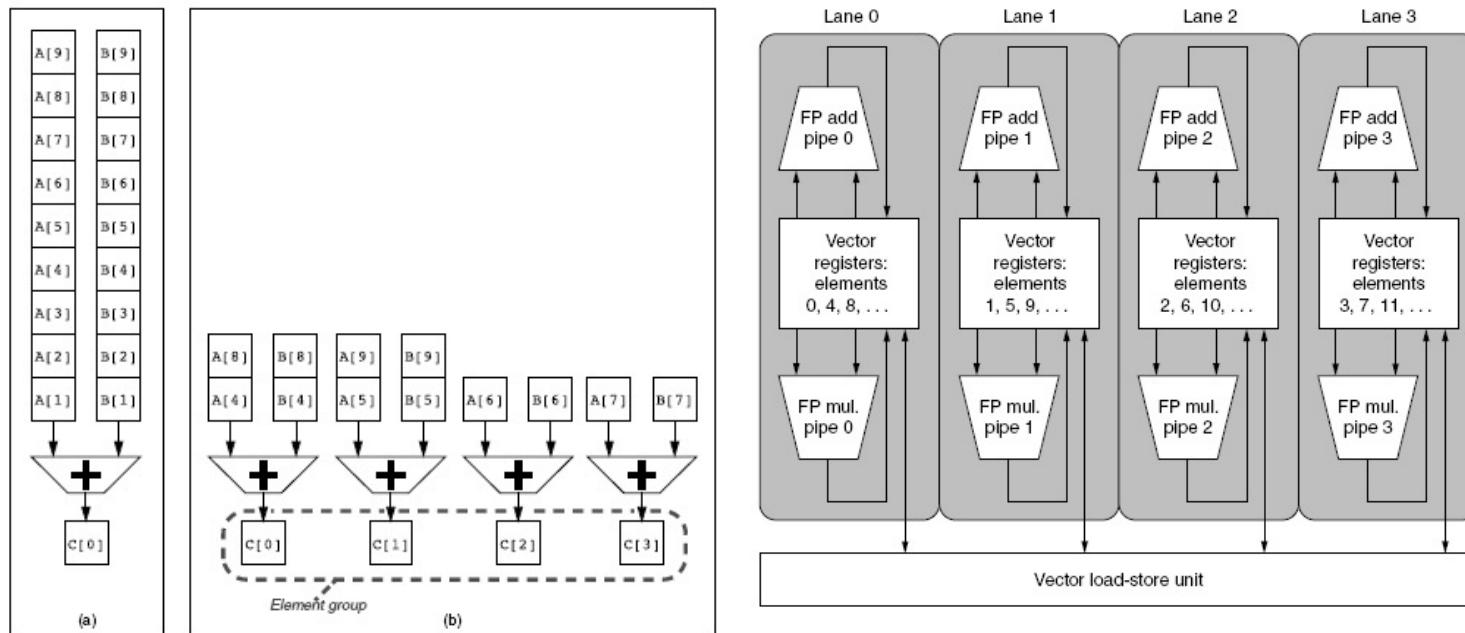


2. Multiple PEs in parallel
 - Fetch and decode instruction once
 - PEs apply op in parallel
 - In synchronous lockstep



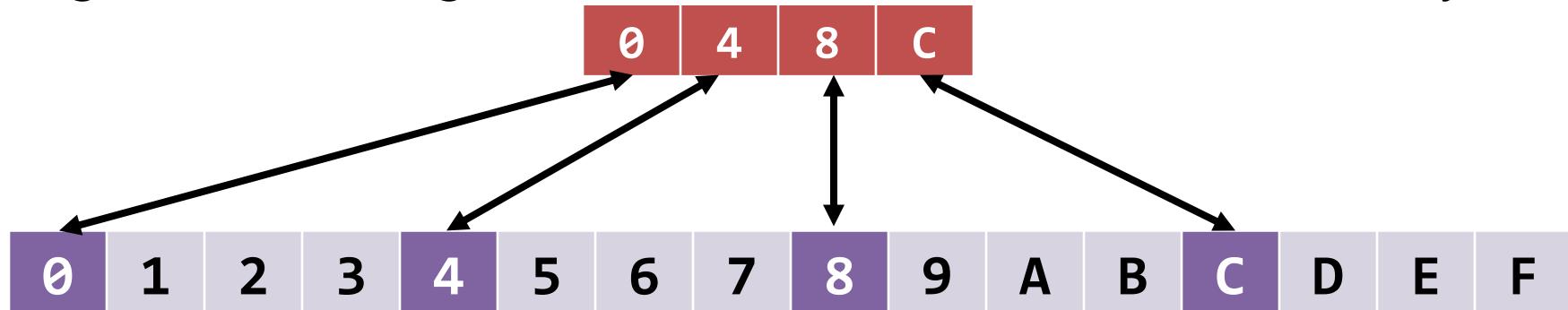
Example: Adding Two Vectors

- 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

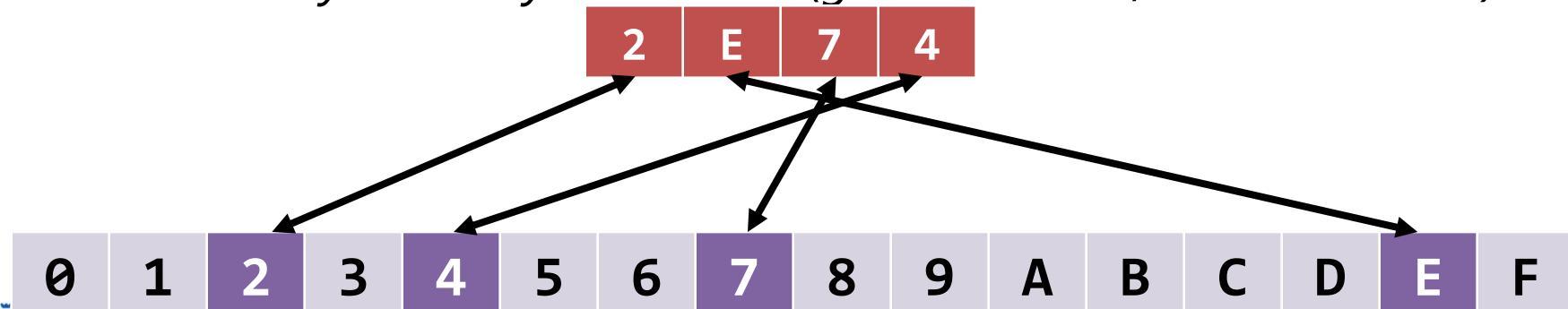


Vector Load-Store Unit

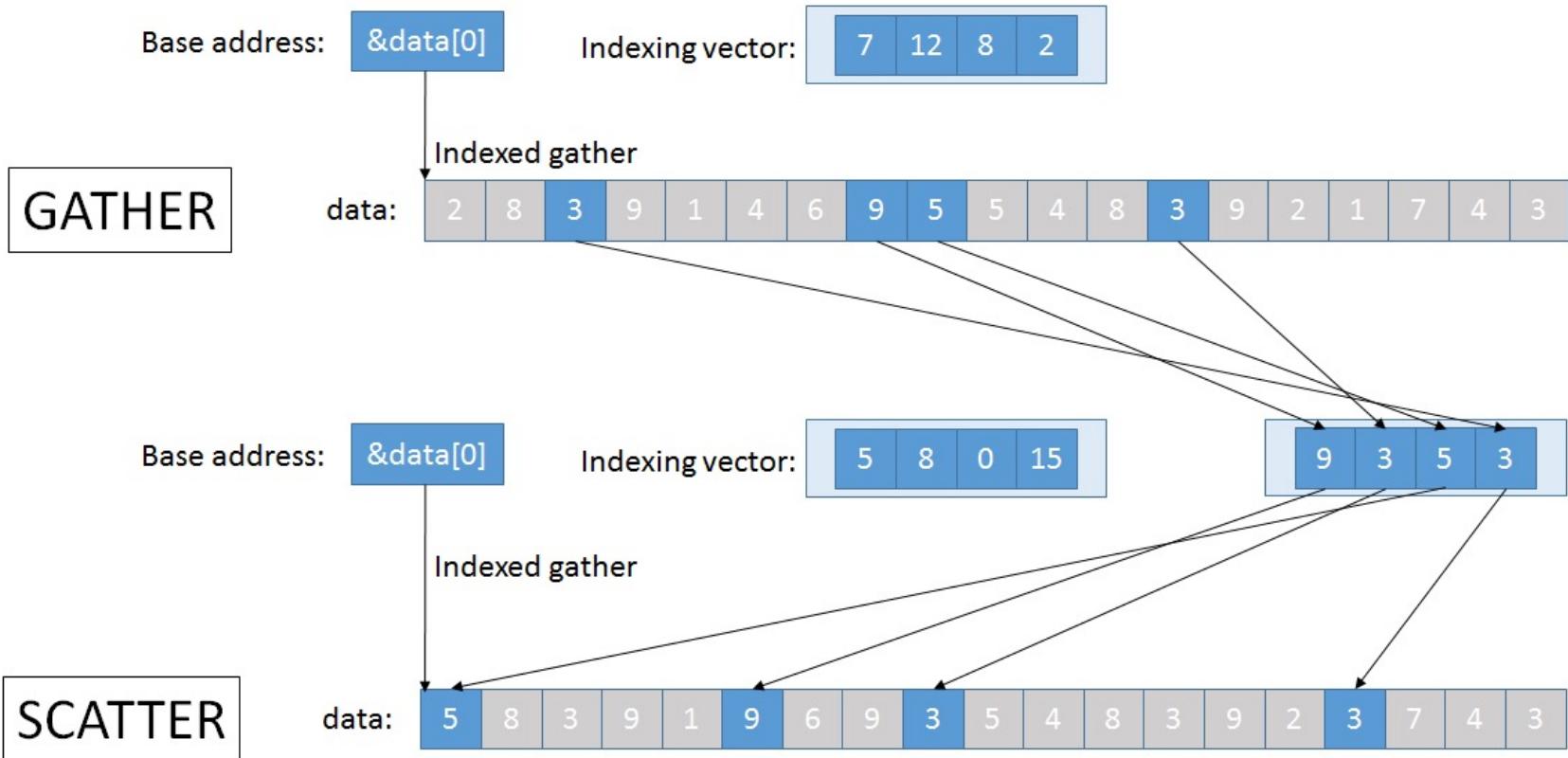
- *Striding* lets you load/store *non-contiguous* data from memory at regular offsets. (e.g. the first member of each struct in an array)



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



How does Gather-Scatter work?



When does Gather-Scatter make sense?

- Often data for scientific or AI applications are **sparse**.
 - Time-sampled points where a sensor measurement changes
 - In social networking, connections in a $N \times N$ friendship matrix
 - In neural networks, weights in a filter layer that are non-zero
- To save memory, sparse data is stored in sparse format:

A sparse matrix format				B corresponding full matrix							
	row	column	weight	0	1	2	3	4	5	6	7
0	1	2	1.5	0	0	0	0	0	0	0	0
1	1	6	4	-1	0	0	1.5	0	0	0	-4
2	3	4	0.3	1	0	0	0	0	0	0	0
3	4	6	1	-1	0	0	0	0	0	0	0.3
4	6	2	2	1	0	0	0	0	0	0	-1



How a sparse filter layer is stored in a Convolutional Neural Network (CNN)



When does Gather-Scatter make sense?

- Convolution works by applying filter like a shifting window:

1 <small>x1</small>	1 <small>x0</small>	1 <small>x1</small>	0	0
0 <small>x0</small>	1 <small>x1</small>	1 <small>x0</small>	1	0
0 <small>x1</small>	0 <small>x0</small>	1 <small>x1</small>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

row	column	weight
0	0	1
0	2	1
1	1	1
2	0	1
2	2	1

0	1	2	3
1	1	0	1
2	0	1	0
3	1	0	1

Filter (sparse matrix format)

- When applying filter on image, a gather needs to take place
 - Gather values in image in corresponding rows and columns
 - Create an image vector out of those gathered values
 - Do a dot (\bullet) product between image vector and filter vector

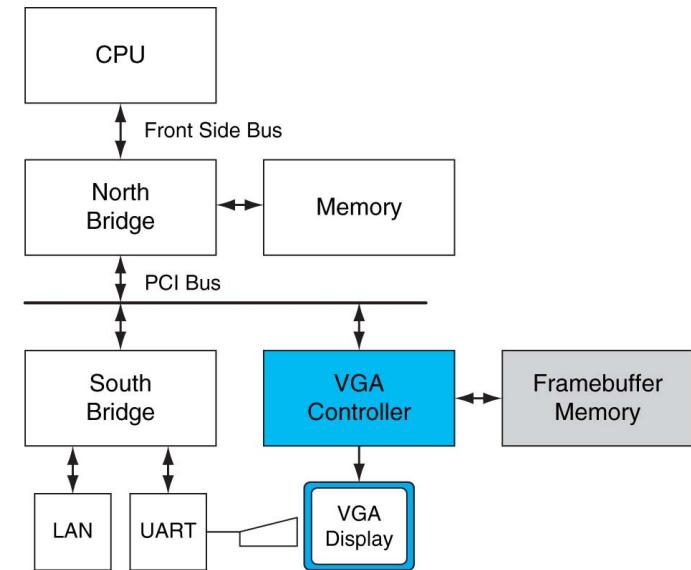
SIMD instructions in real processors

- x86 vector extensions
 - Historically: MMX, SSE, AVX, AVX-2
 - Current: AVX-512 (512-bit vector instructions)
- ARM vector extensions
 - Historically: VFP (Vector Floating Point)
 - Current: Neon (128-bit vector instructions)
- Vector instructions have progressively become wider historically
 - Due to increase of data parallel applications
 - Due to their power efficiency
 - Compared to fetching, decoding, scheduling multiple instructions
 - Good way to increase FLOPS while staying within TDP limit
- Enter GPUs for general computing (circa 2001)

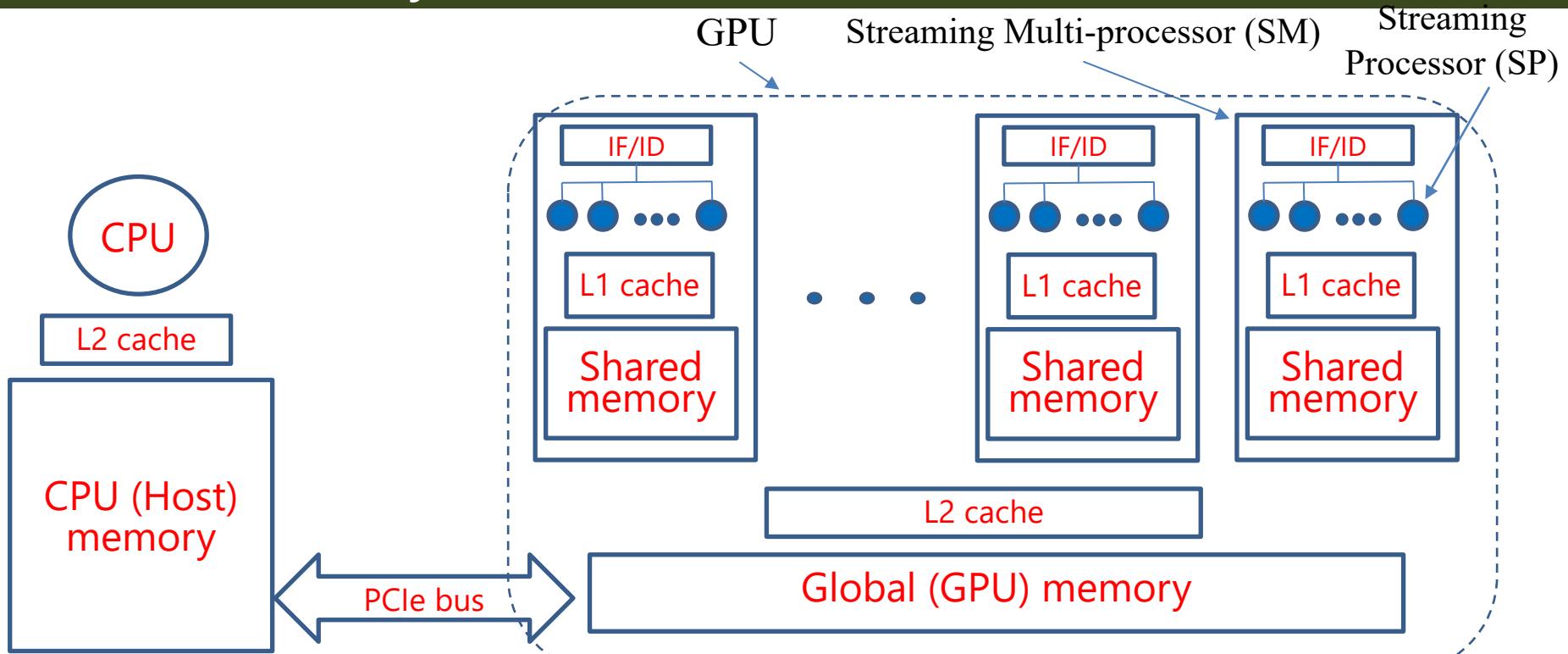
GPUs: Graphical Processing Units

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

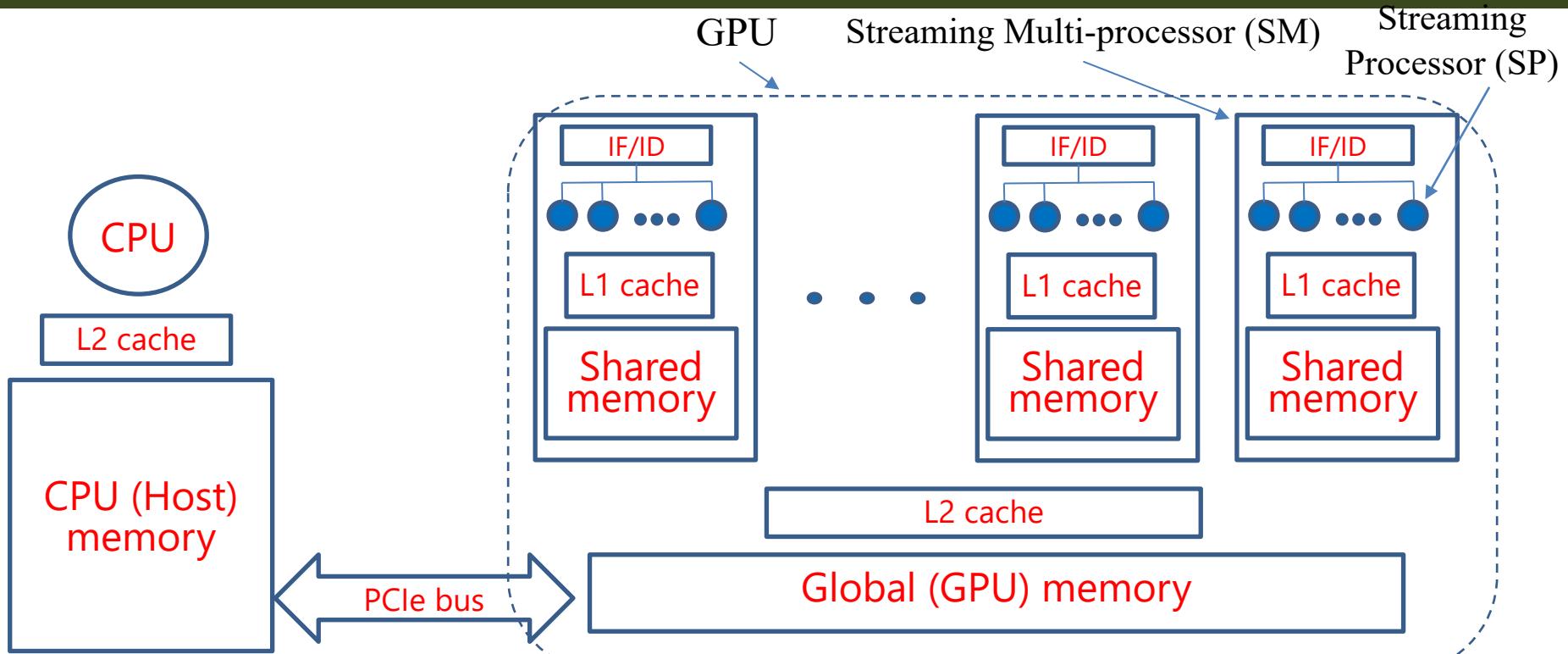


GPU is Really a SIMD Processor



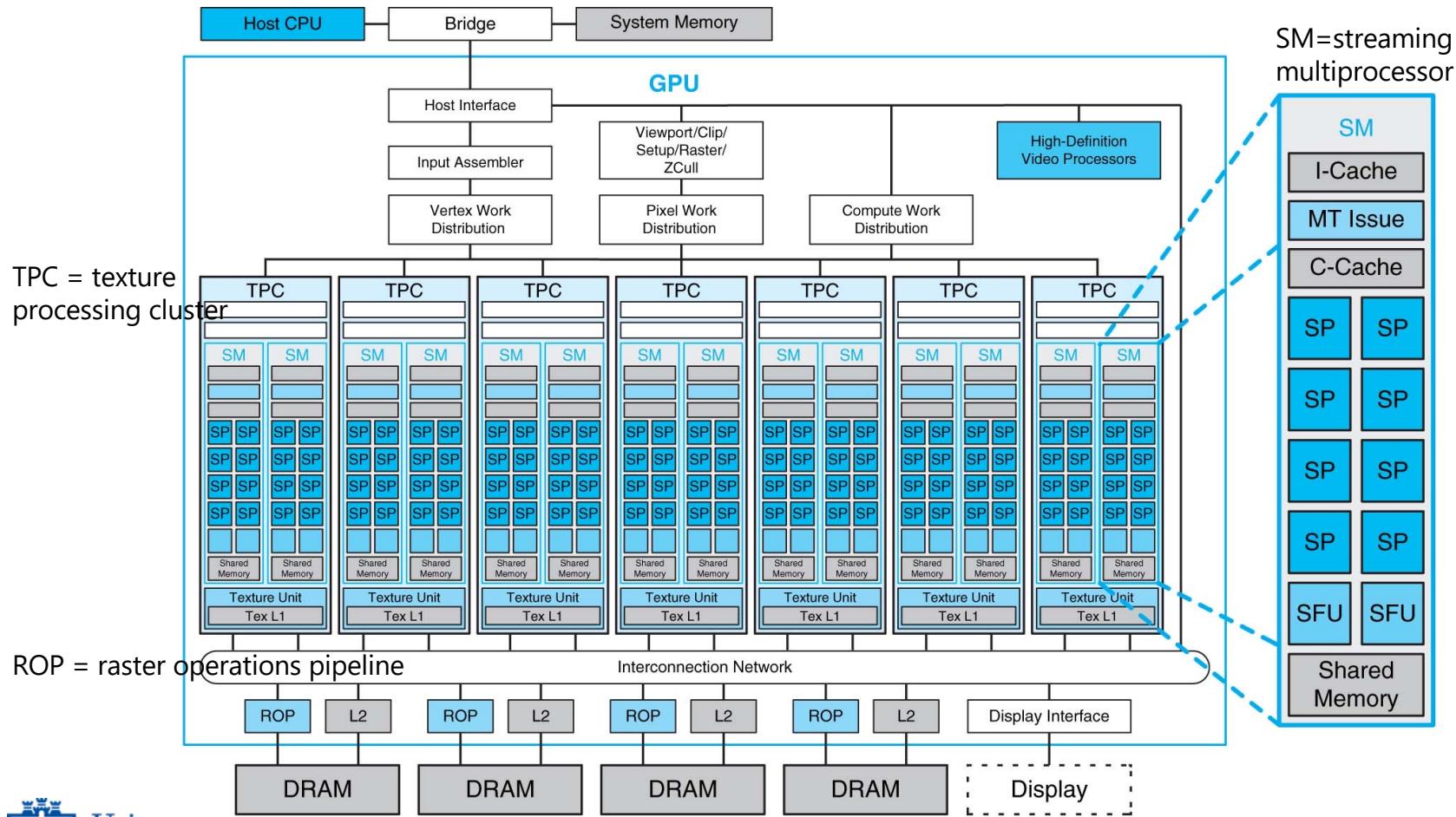
- Logically, a GPU is composed of **SMs** (Streaming Multi-processors)
 - An SM is a **vector** unit that can process multiple pixels (or data items)
- Each SM is composed of **SPs** which work on each pixel or data item

CPU-GPU architecture



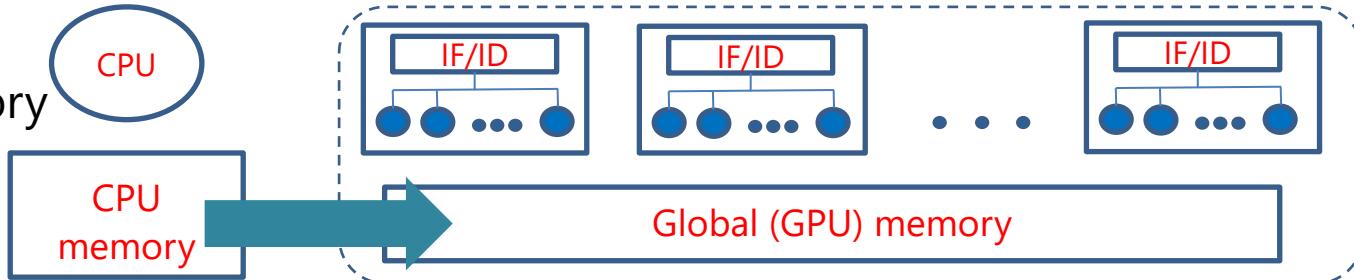
- Dedicated GPU memory separate from system memory
- Code and data must be transferred to GPU memory for it to work on it
 - Through PCI-Express bus connecting GPU to CPU

Modern GPU architecture

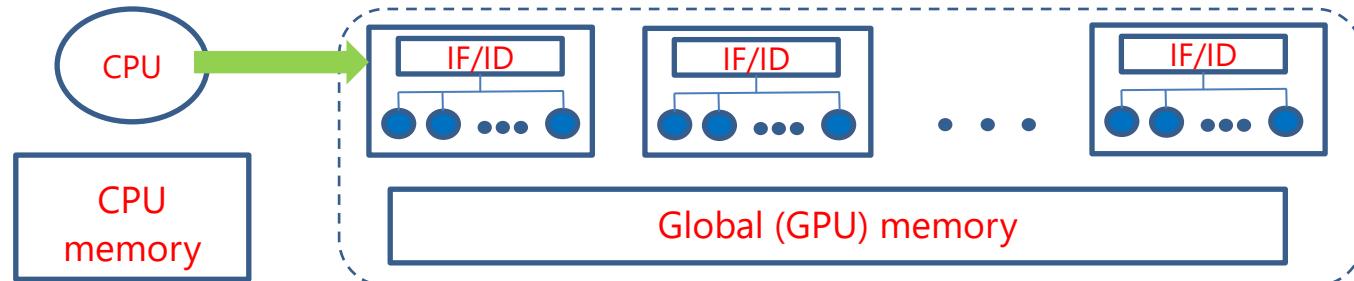


GPU Programming Model

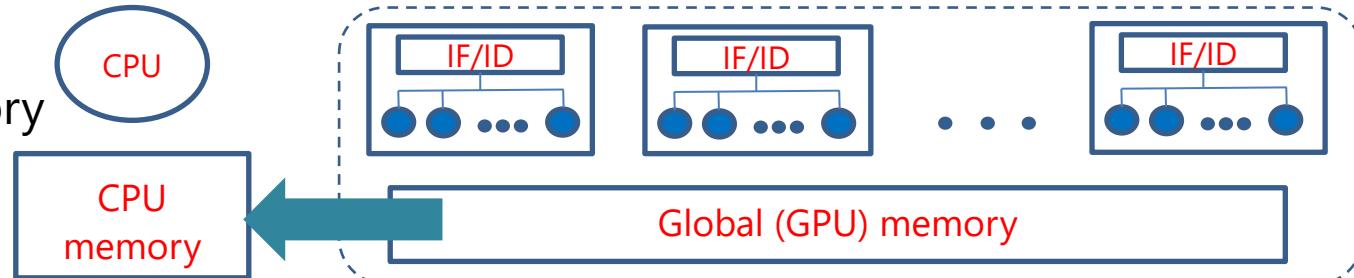
Copy data from CPU memory to GPU memory



Launch the **kernel**



Copy data from GPU memory to CPU memory



GPU Programming Model

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 (...)

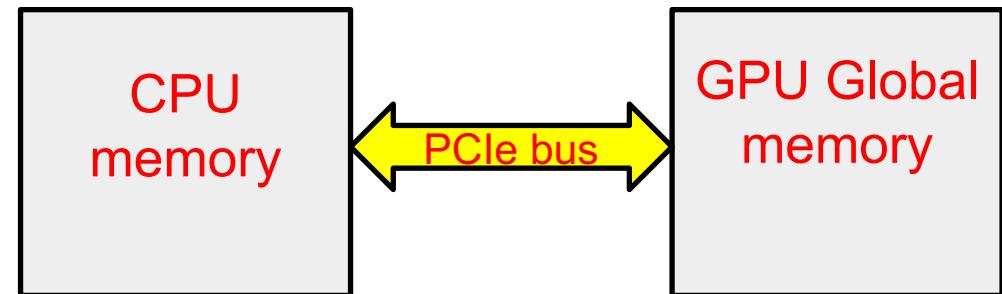
Implementation of GPU kernel
kernel: Function executed on the GPU

GPU Programming Model: Copying Data

```
/* malloc in GPU global memory */
cudaMalloc (void **pointer, size_t nbytes);
/* free mallocoed 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 cudaMemcpyKind dir);
```

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice



Example: Copying array a to array b using the GPU

Data Movement Example



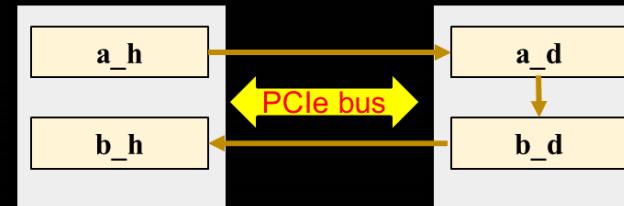
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    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) ;
}
```

GPU Programming Model: Launching the Kernel

CPU program
(serial code)



cudaMemcpy (...)



Function <<<nb,nt >>>



cudaMemcpy (...)



global_ Function (...)

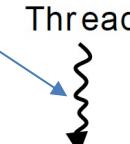
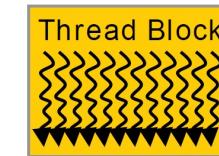
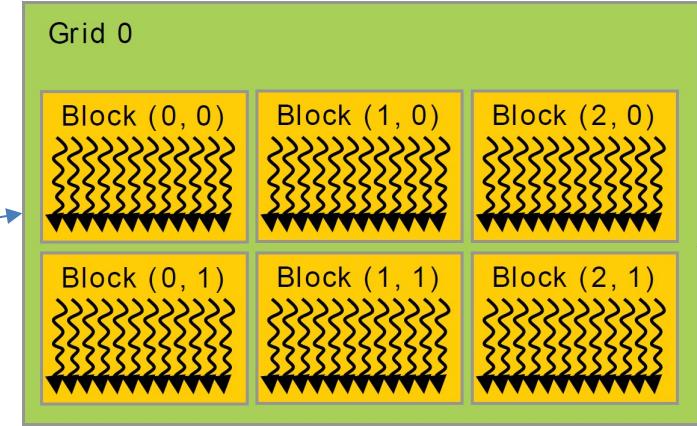


Copy data from CPU
memory to GPU memory

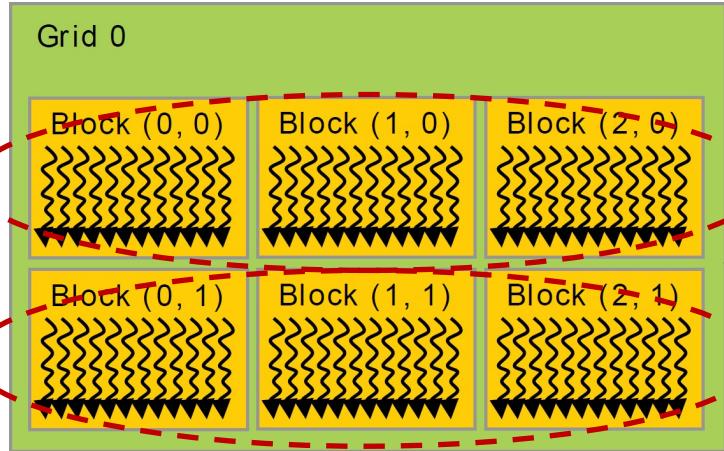
Launch a **kernel** with nb
blocks, each with nt threads

Copy results from GPU
memory to CPU memory

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

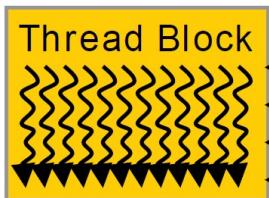


The Execution Model

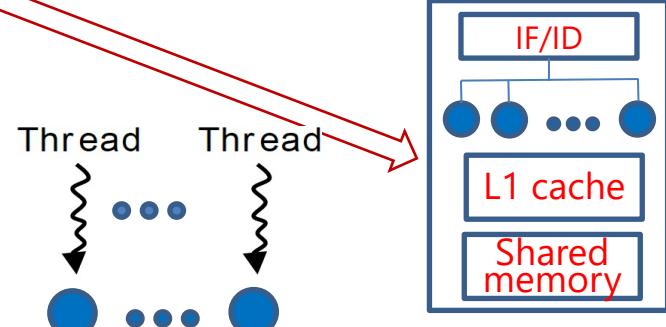
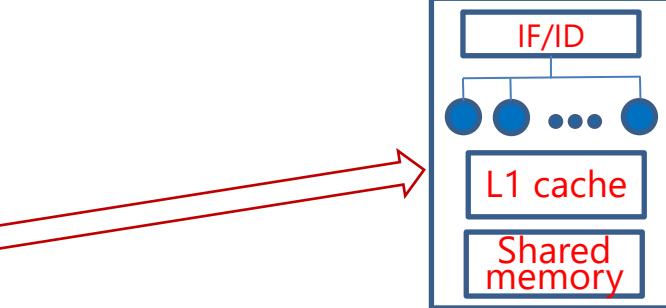


- The **thread blocks** are dispatched to **SMs**
- The number of blocks dispatched to an SM depends on the SM's resources (registers, shared memory, ...).

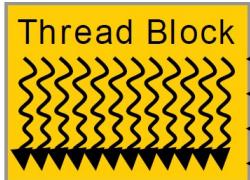
Blocks not dispatched initially are dispatched when an SM frees up after finishing a block



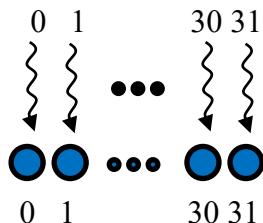
- When a block is dispatched to an SM, each of its threads executes on an SP in the SM.



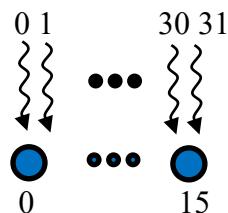
A thread block is executed in warps



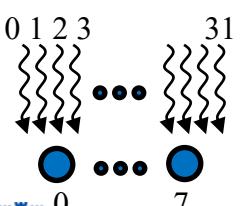
- Each block (up to 1K threads) is divided into groups of 32 threads (called **warps**) – empty threads are used as fillers.
- A warp executes as a SIMD **vector instruction** on the SM.
- Depending on the number of SPs per SM:



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



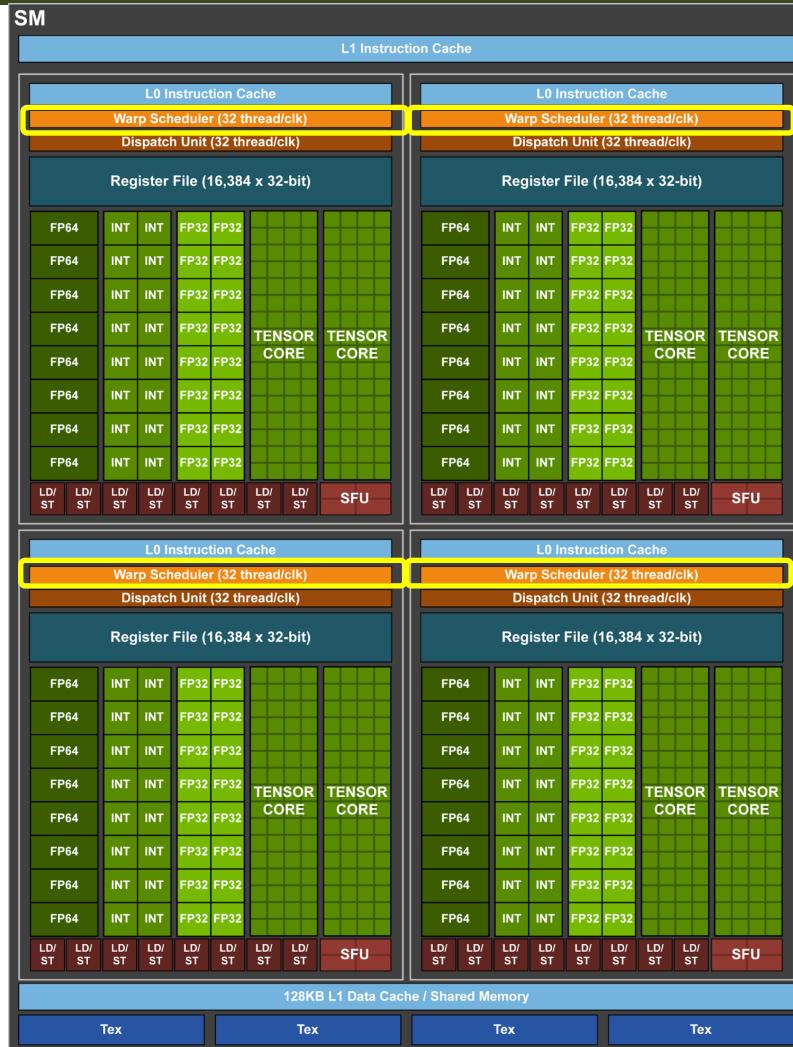
- If 16 SP per SM → 2 threads are time multiplexed on 1 SP (16 lanes of execution, 2 threads per lane)



- If 8 SP per SM → 4 threads are time multiplexed on 1 SP (8 lanes of execution, 4 threads per lane)

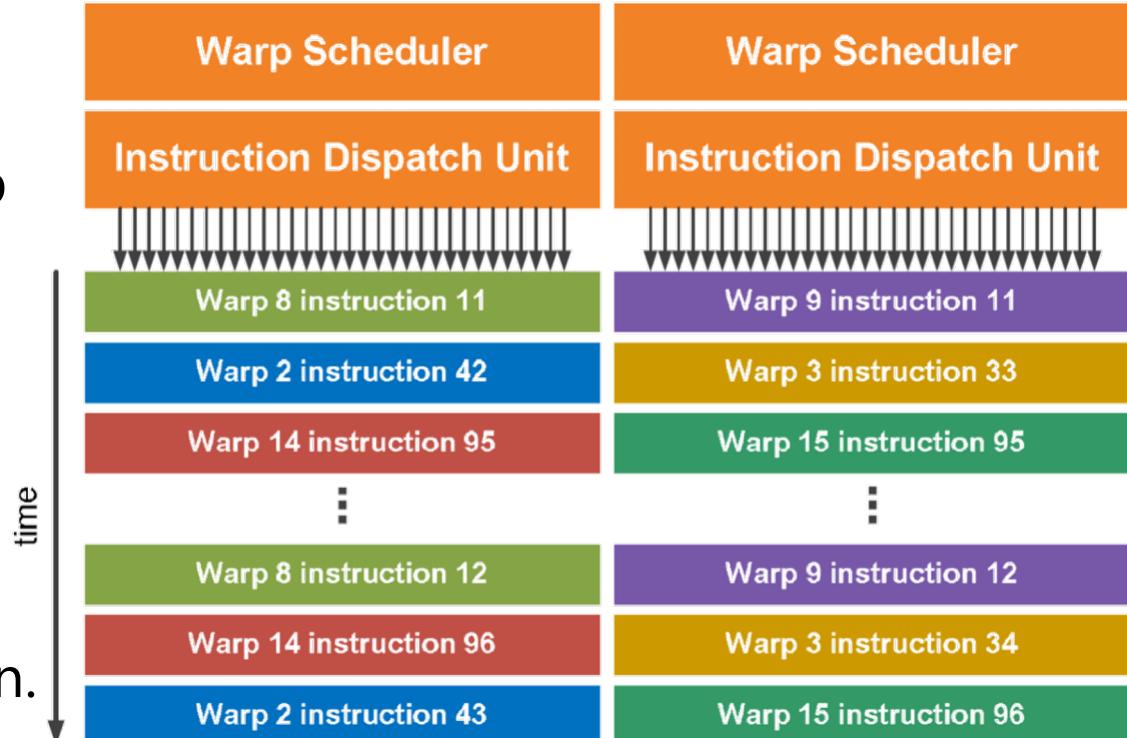
A SM is composed of one or more warp schedulers

- In this SM, there are 4 warp schedulers.
- Warps in thread block are divided statically among warp schedulers.
- E.g. for 4 schedulers:
 - Scheduler 1: Warp 0, Warp 4, ...
 - Scheduler 2: Warp 1, Warp 5, ...
 - Scheduler 3: Warp 2, Warp 6, ...
 - Scheduler 4: Warp 3, Warp 7, ...
- Round robin assignment to ensure equal distribution of warps



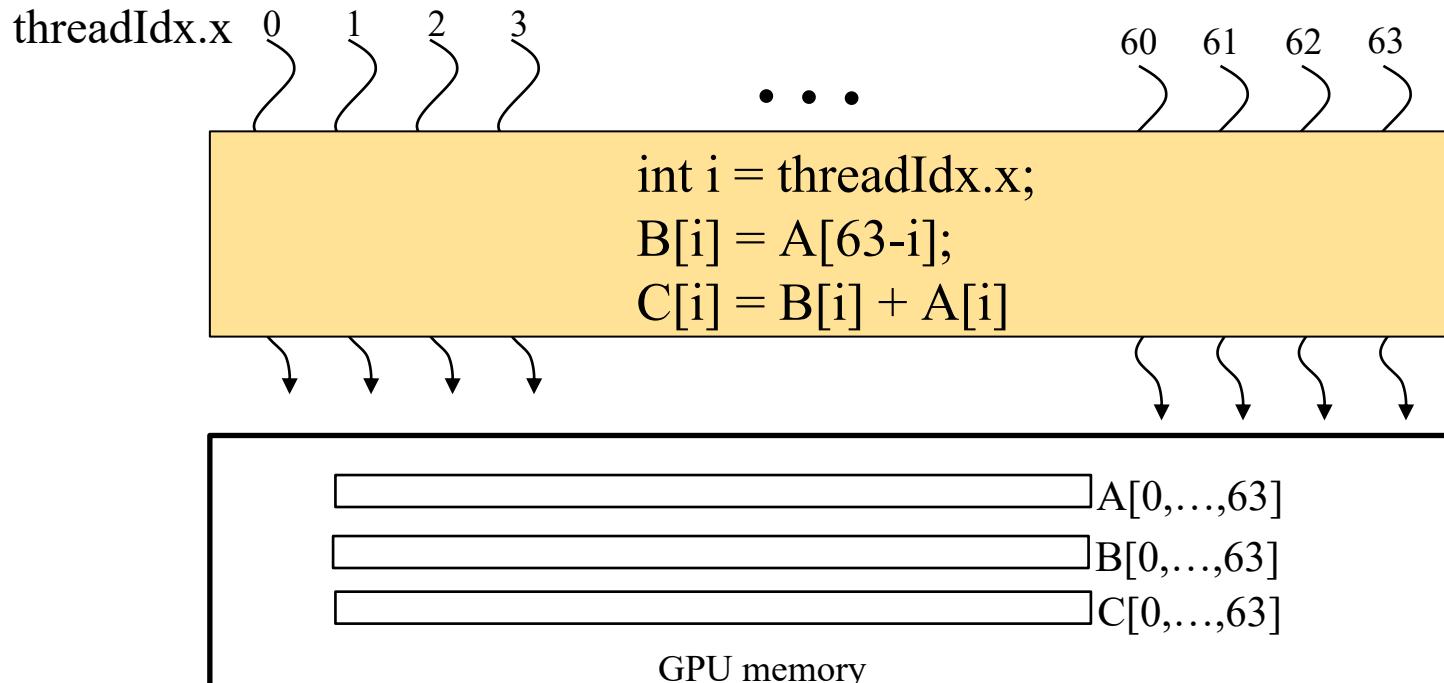
Warp scheduling enables latency hiding

- Warp scheduler can hide bubbles (just like a superscalar scheduler)
 - But without an instruction queue and out-of-order execution
- How?
 - In-order execution.
 - Switch to different warp when a bubble is about to form.
- Warp can come from any thread block in SM
 - More thread blocks can lead to higher utilization.



All threads execute the same code

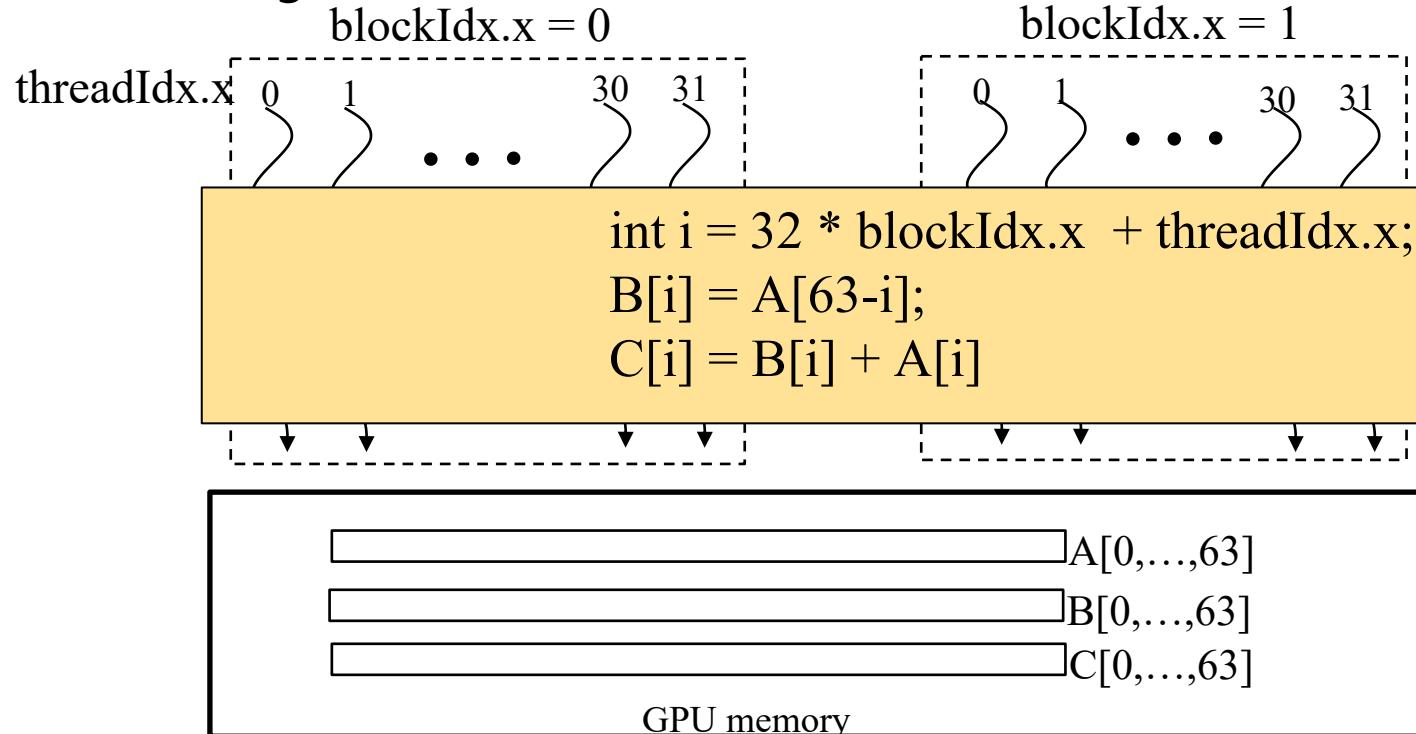
- Launched using **Kernel <<<1, 64>>>** : 1 block with 64 threads



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

Blocks of Threads

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

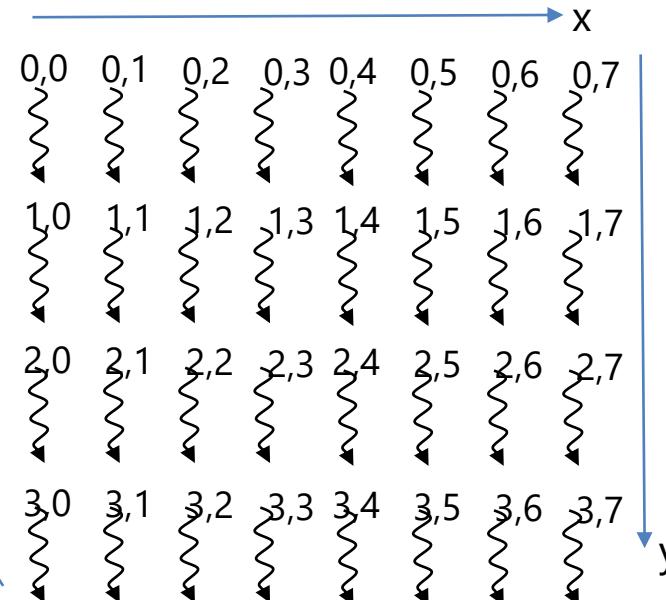
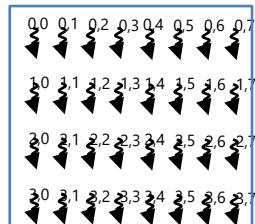
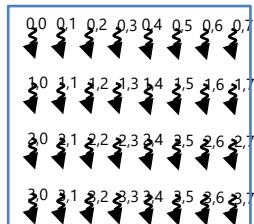
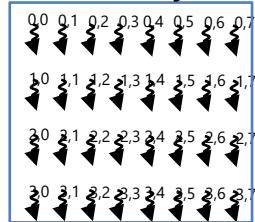
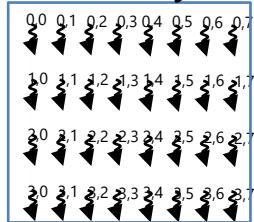


- Each thread block has a unique “block index” → **blockIdx.x**
- Each thread has a unique **threadIdx.x** within its own block
- Can compute a global index from the `blockIdx.x` and `threadIdx.x`

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: Computing the global index

```
void main ()
```

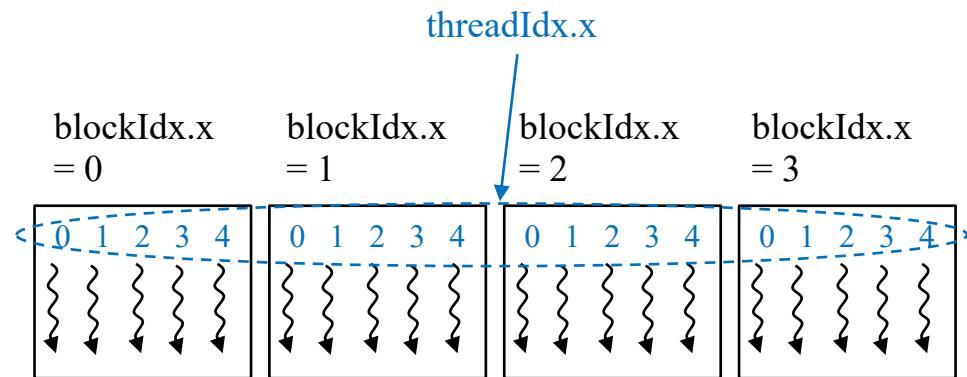
```
{  cudaMalloc (int* &a, 20*sizeof(int));  
  cudaMalloc (int* &b, 20*sizeof(int));  
  cudaMalloc (int* &c, 20*sizeof(int));  
  ...  
  kernel<<<4,5>>>(a, b, c) ;  
  ...  
}
```

```
_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;  
}
```

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 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



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



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];
}
```

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 ()
{
    ...
    saxpy_serial(n, 2.0, x, y);
    ...
}
```

```
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 $y(i) = a * x(i) + y(i)$

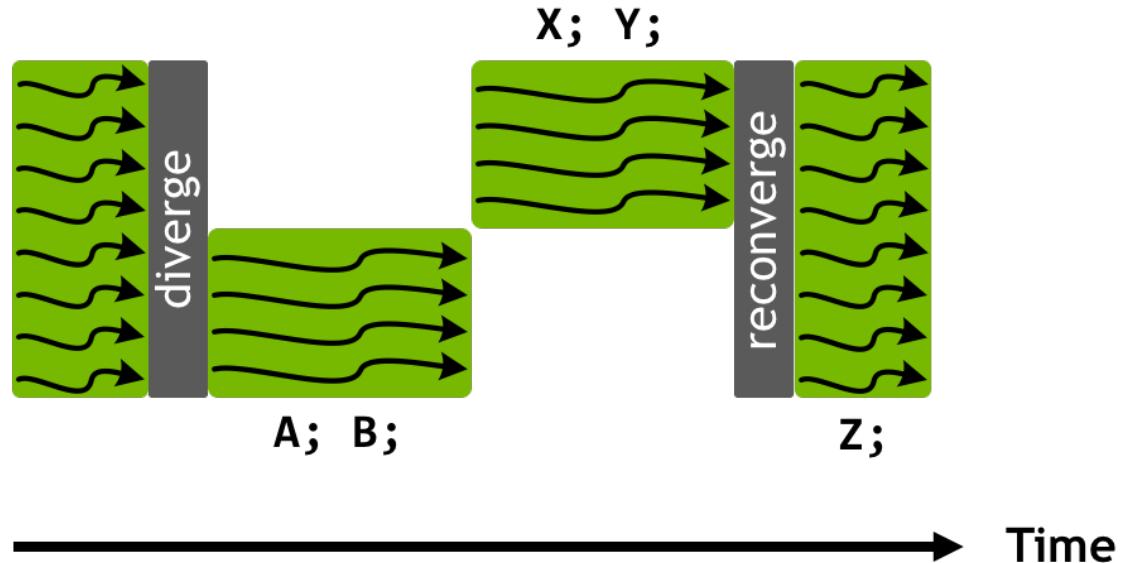
- What happens when $n = 1$?

```
_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

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**.

GPU Performance

Lesson 1: Parallelism is Important

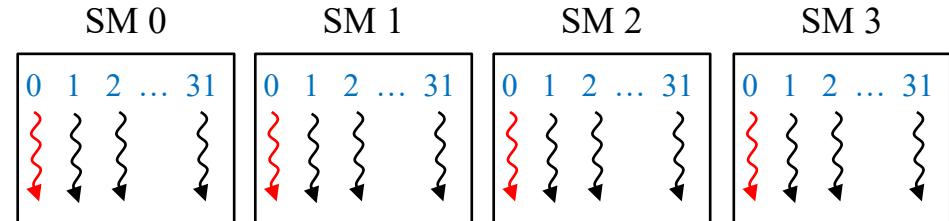
Thread Level Parallelism

- Superscalars and VLIWs are useful only if...
 - Program exhibits ILP (Instruction Level Parallelism) in the code
- GPUs are useful only if...
 - Program has **TLP (Thread Level Parallelism)** in the code
 - TLP can be expressed as the number of threads in the code
- How that TLP is laid out in the kernel is also important
 - How many **threads** are in a thread block
 - If less than threads in warp, some **SPs** may get unused
 - How many **thread blocks** are in the grid
 - If less than number of SMs, some **SMs** may get unused
 - If not careful, your GPU may get **underutilized**

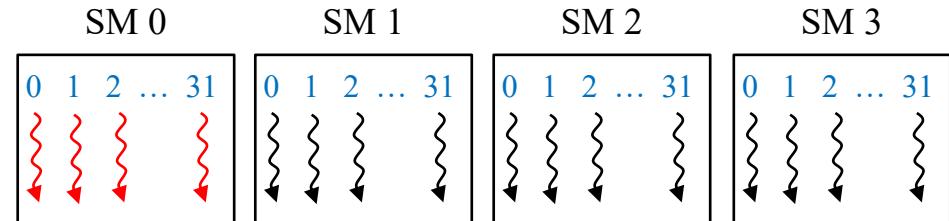
Example: Kernels with Bad Layout

- Suppose there are 4 SMs in GPU with 32 SPs in each SM.
 - Case 1, 2 below have enough TLP (1024 threads) but bad layout.
 - Utilized threads are marked in red. Rest are unused.

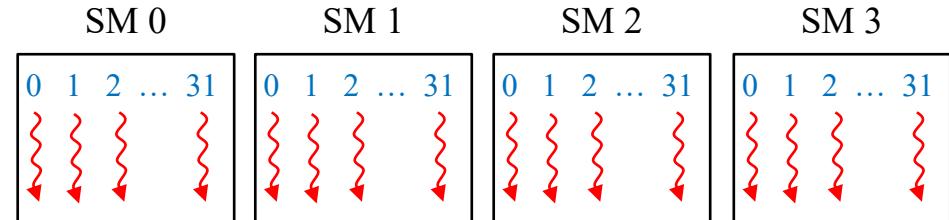
- Case 1: Not enough threads
`kernel<<<1024, 1>>(...);`



- Case 2: Not enough blocks
`kernel<<<1, 1024>>(...);`



- Balanced threads and blocks
`kernel<<<32, 32>>(...);`
`kernel<<<16, 64>>(...);`
`kernel<<<4, 256>>(...);`



Lesson 2: Bandwidth is Important

Example: Computing $y(i) = A(i,j) * x(j)$

C program (on CPU)

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

CUDA program (on CPU+GPU)

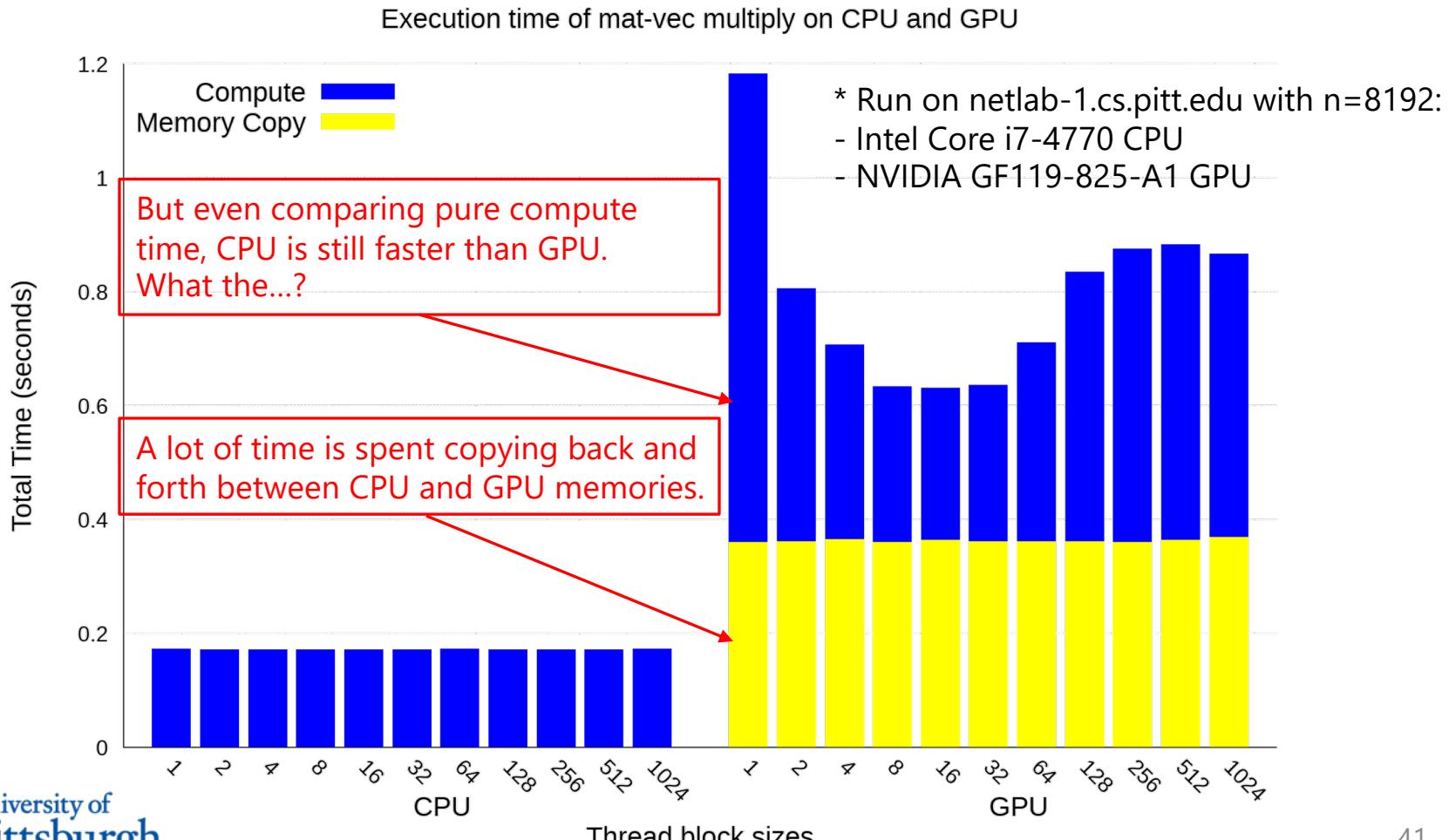
```
void mv_gpu(float* y, float* A, float* x, int n) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < n) {  
        for (int j = 0; j < n; j++)  
            y[i] += A[i*n + j] * x[j];  
    }  
}
```

```
void main ()  
{  
    ...  
    mv_cpu(y, A, x, n);  
    ...  
}
```

```
void main ()  
{  
    ...  
    int nblocks = (n + block_size - 1) / block_size;  
    mv_gpu <<<nblocks, block_size>>> (y, A, x, n);  
    ...  
}
```

Performance Results for $y(i) = A(i,j) * x(j)$

- Guess what? CPU is faster than GPU!



Performance Results for $y(i) = A(i,j) * x(j)$

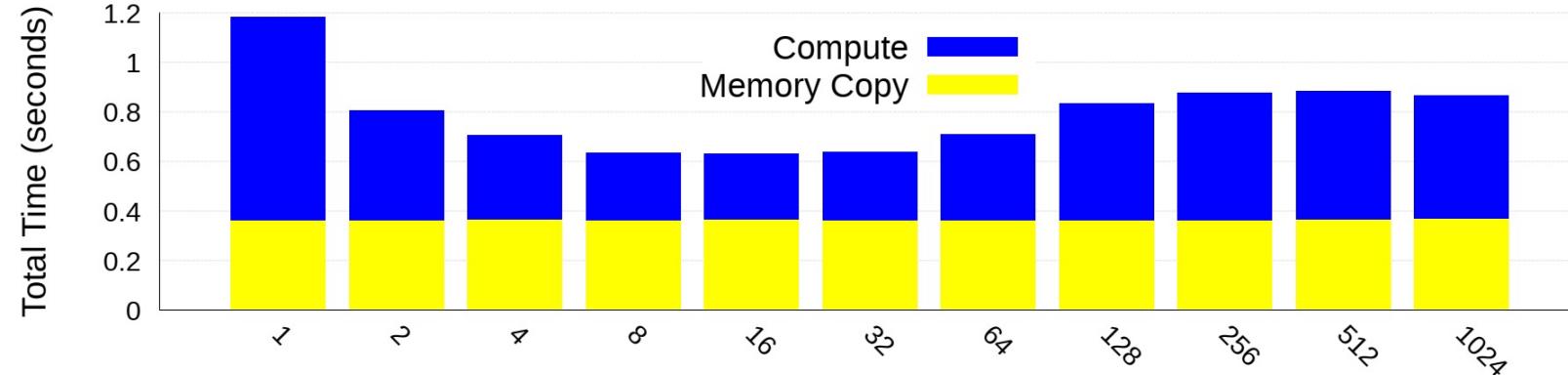
- Was it because the GPU was wimpy and can't do enough **FLOPS**?
- NVIDIA GF119-825-A1 is a Fermi GPU Capability 2.1
 - Clock rate: 1046 MHz (X 2 for warp execution)
 - Number of SMs: 1
 - Number of SPs per SM: 48
 - Max FLOPS = $1046 \text{ MHz} * 2 * 1 * 48 = \mathbf{100.4 \text{ GFLOPS}}$
- What was the FLOPS achieved?
 - $y[i] += A[i*n + j] * x[j] = 2 \text{ FP ops each iteration for } n * n \text{ iterations}$
 - $n = 8192$, so FP ops = $8192 * 8192 * 2 = 134 \text{ M}$
 - Time = 0.27 seconds (shortest at 32 thread block size)
 - FLOPS = $134 \text{ M} / 0.27 = \mathbf{496 \text{ MFLOPS}}$
 - **Not even close to the limit!**

Performance Results for $y(i) = A(i,j) * x(j)$

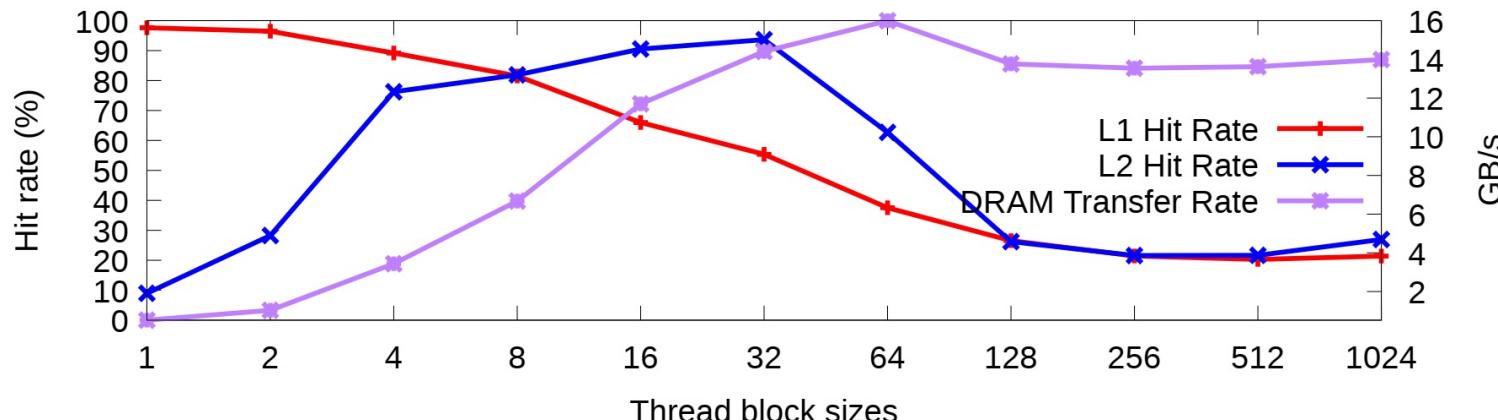
- Could it be that the GPU didn't have enough **memory bandwidth**?
- NVIDIA GF119-825-A1 is a Fermi GPU Capability 2.1
 - Memory Type: DDR3
 - Memory Bandwidth: **14.00 GB/s**
- GPUs also have Performance Monitoring Units (PMUs)
 - NVIDIA Profiler (nvprof) provides an easy way to read them:
<https://docs.nvidia.com/cuda/profiler-users-guide/index.html>
 - Let's use the PMU to profile the following:
 - DRAM Transfer Rate (GB/s)
 - L1 Hit Rate (%)
 - L2 Hit Rate (%)

Memory Wall Hits Again

Execution time of mat-vec multiply on GPU

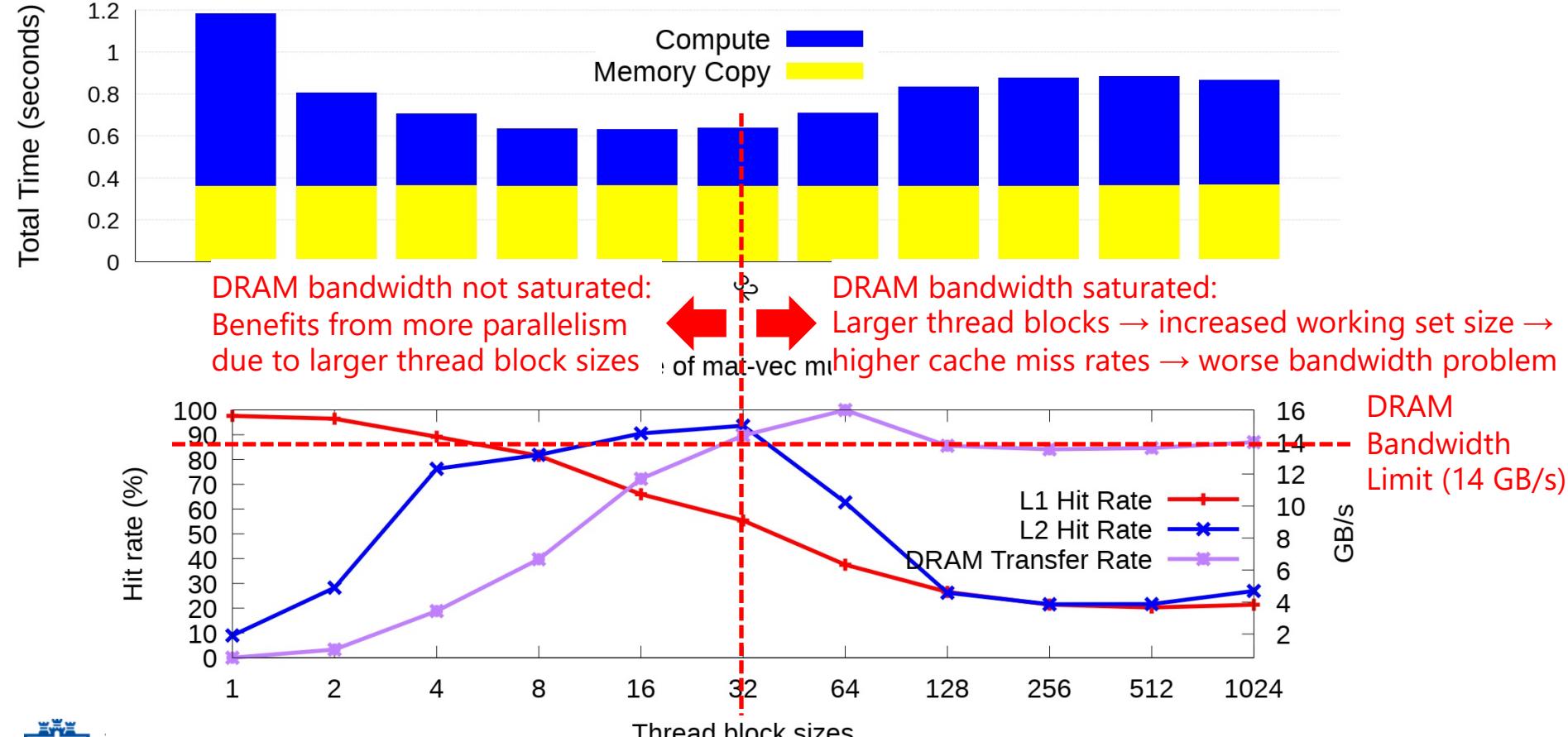


Memory profile of mat-vec multiply on GPU



Memory Wall Hits Again

Execution time of mat-vec multiply on GPU



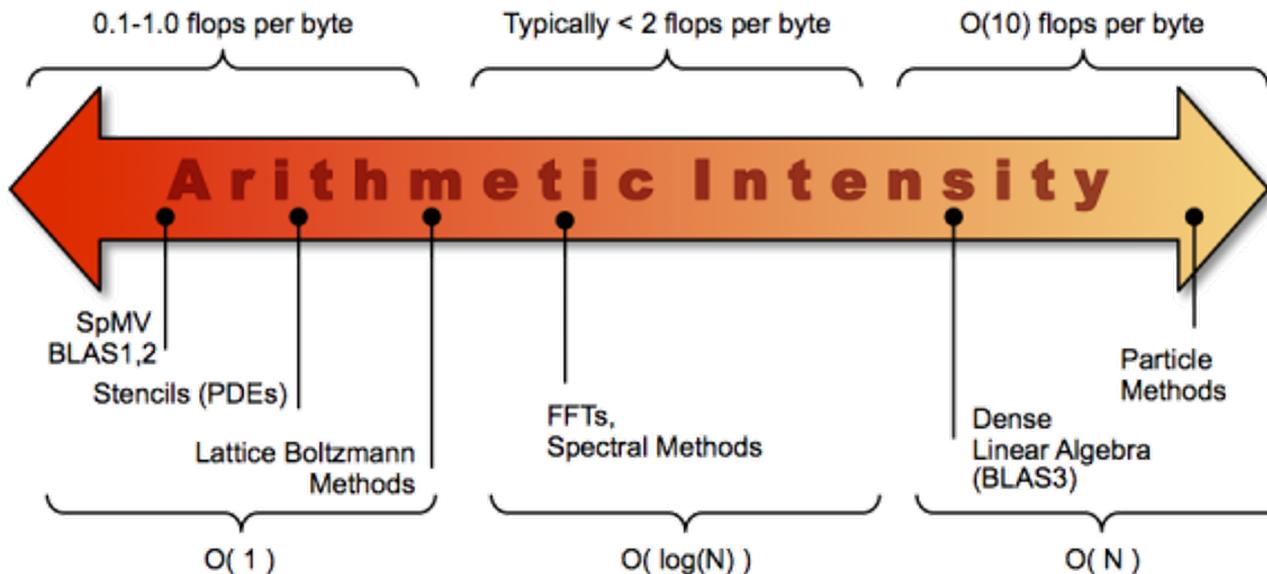
Is there a way we can reach max FLOPS?

- Let's take a look at the GPU design metrics again:
 - Max FLOPS = 100.4 GFLOPS
 - Memory Bandwidth: 14.00 GB/s
- To sustain max FLOPS, you need to do a lot of work per byte
 - $100.4 \text{ GFLOPS} / 14.00 \text{ GB/s} = 7.17 \text{ FP ops / byte}$
 - Or, about **28 FP ops / float** (4 bytes) fetched from memory
 - Otherwise, the memory bandwidth cannot sustain the FLOPS
- All GPUs have this problem with **memory bandwidth**:
 - It's easy to put in more SMs using transistors for Moore's Law
 - Your memory bandwidth is limited due to your DDR interface

Arithmetic Intensity: A property of the program

- How many **FP ops / float** for our mat-vec multiplication?
 - $y[i] += A[i*n + j] * x[j]$ each iteration with $n * n$ iterations
 - FP ops = **$2 * n * n$** (one multiply and one add)
 - Float accesses = **$n * n + 2n$** (1 matrix and 2 vector accesses)
 - That's counting only cold misses but could be even more
 - So approx. **2 FP ops / float** (a far cry from 28 FP ops / float)
 - This metric is called **arithmetic intensity**
- Arithmetic intensity is a **property of the program** needed by GPUs
 - Just like TLP (thread-level-parallelism) is needed by GPUs
 - Matrix-vector multiplication has low intensity
 - Fundamentally not suited for fast GPU computation

Arithmetic Intensity: A property of the program



* Courtesy of Lawrence Berkeley National Laboratory:

<https://crd.lbl.gov/departments/computer-science/par/research/roofline/introduction/>

- **BLAS**: Basic Linear Algebra Subprograms
 - BLAS 1: Vector operations only (e.g. saxpy) → Bad intensity
 - BLAS 2: **General Matrix-Vector Multiplication (GeMV)** → Bad intensity
 - BLAS 3: **General Matrix Multiplication (GeMM)** → Good intensity

Matrix-Matrix Multiply: Good Arithmetic Intensity

- Matrix-multiplication:

```
for (int i=0; i<n; i++)
```

```
    for (int j=0; j<n; j++)
```

```
        for (int k=0; k<n; k++)
```

```
            C[i*n + j] += A[i*n + k] * B[k*n + j];
```

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

- What's the arithmetic intensity for this program?

- FP ops = **2 * n * n * n** (one multiply and one add)

- Float accesses = **3 * n * n** (3 matrix accesses)

- If we only have cold misses and no capacity misses

- Arithmetic intensity = $2 * n / 3 = \mathbf{0.66 * n} = \mathbf{O(n)}$

- Implication: The larger the matrix size, the better suited for GPUs!
 - Important result for deep learning and other apps

Example: Computing $C(i, j) = A(i, k) * B(k, j)$

C program (on CPU)

```
void mm_cpu(float* C, float* A, float* B,  
int n) {  
    for (int i=0; i<n; i++)  
        for (int j=0; j<n; j++)  
            for (int k=0; k<n; k++)  
                C[i*n + j] += A[i*n + k] * B[k*n + j];  
}
```

CUDA program (on CPU+GPU)

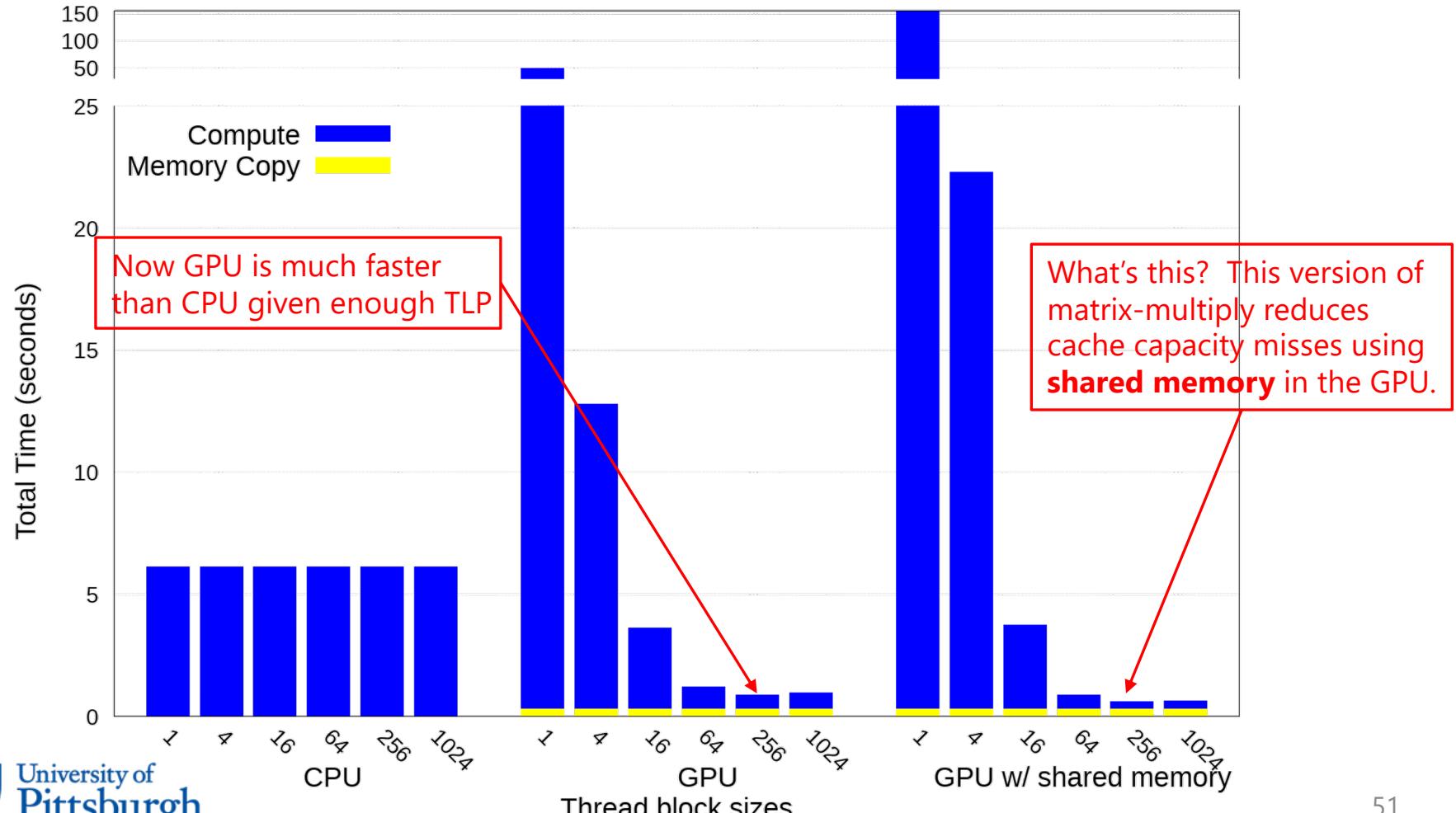
```
void mm_gpu(float* C, float* A, float* B, int n) {  
    float Cvalue = 0;  
    int i = blockIdx.y * blockDim.y + threadIdx.y;  
    int j = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int k = 0; k < n; ++k)  
        Cvalue += A[i * n + k] * B[k * n + j];  
    C[i * n + j] = Cvalue;  
}
```

```
void main ()  
{  
    mm_cpu(C, A, B, n);  
}
```

```
void main ()  
{  
    dim3 dimBlock(block_size, block_size);  
    dim3 dimGrid(n / dimBlock.x, n / dimBlock.y);  
    mm_gpu <<<dimGrid, dimBlock>>> (C, A, B, n);  
}
```

Performance Results for $C(i,j) = A(i,k) * B(k,j)$

Execution time of mat-mat multiply on CPU and GPU

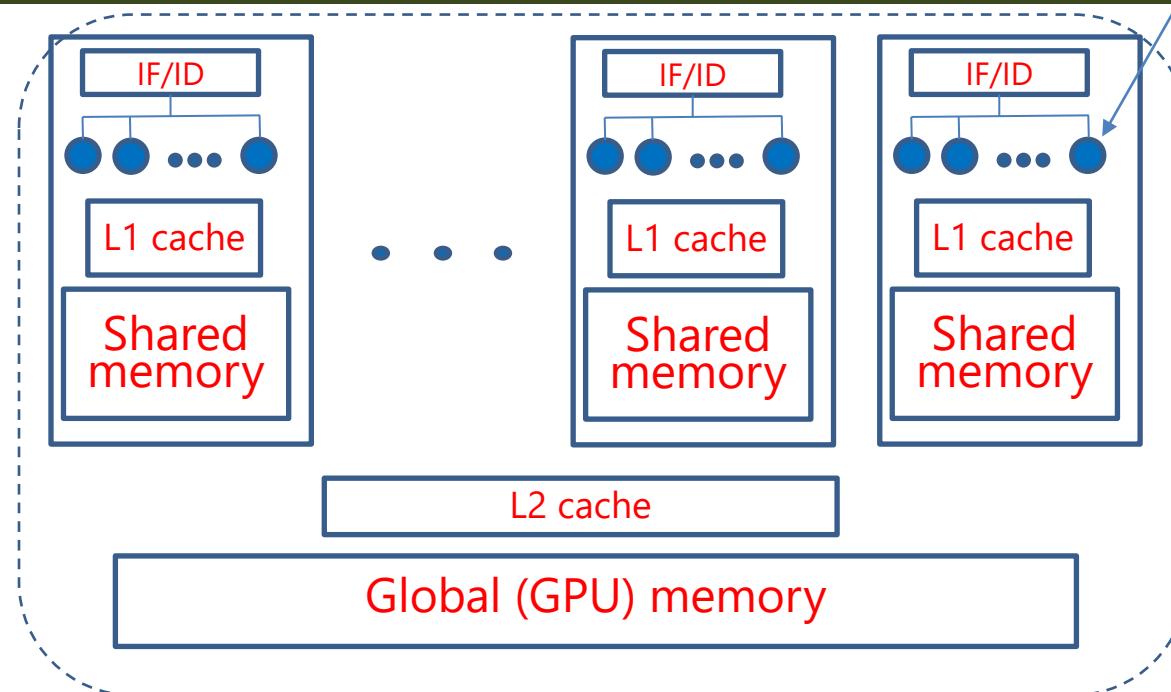


Capacity Misses can Reduce Arithmetic Intensity

```
for (int i=0; i<n; i++)  
    for (int j=0; j<n; j++)  
        for (int k=0; k<n; k++)  
            C[i*n + j] += A[i*n + k] * B[k*n + j];
```

- The ideal arithmetic intensity for this program was:
 - FP ops = **$2 * n * n * n$** (one multiply and one add)
 - Float accesses = **$3 * n * n$** (3 matrix accesses)
 - Arithmetic intensity = $2 * n / 3 = 0.66 * n = \mathbf{O(n)}$
- Only if we have **no capacity misses**. What if we did have them?
 - For $B[k*n + j]$, reuse distance is the entire matrix of B
 - If $B[k*n + j]$ always misses, memory accesses is in the order of **n^3** !

So what is Shared Memory?



- **Shared Memory:** memory shared among threads in a thread block
 - Variables declared with `_shared_` modifier live in shared memory
 - Is same as L1 cache in terms of latency and bandwidth!
 - Storing frequently used data in shared memory can save on bandwidth

Loop Tiling with Shared Memory

- Store a “tile” within matrix in shared memory while operating on it
 - Can reduce accesses to DRAM memory

- Code in: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory-in-matrix-multiplication-c-ab>

```
shared __ float aTile[TILE_DIM][TILE_DIM],  
        bTile[TILE_DIM][TILE_DIM];
```

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
```

```
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

```
float sum = 0.0f;
```

```
aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
```

```
bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
```

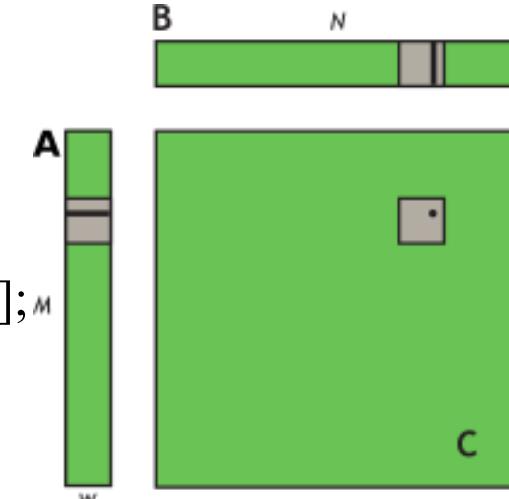
```
__syncthreads();
```

```
for (int i = 0; i < TILE_DIM; i++)
```

```
    sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
```

```
c[row*N+col] = sum;
```

- Assumption: $TILE_DIM = w$. What if $w > TILE_DIM$?



Loop Tiling with Shared Memory

- TILE_DIM is limited by amount of shared memory. What if $w > \text{TILE_DIM}$?
 - Now must load A and B tiles $w / \text{TILE_DIM}$ times per thread block

- Now code will look like:

```
__shared__ float aTile[TILE_DIM][TILE_DIM],  
            bTile[TILE_DIM][TILE_DIM];
```

```
float sum = 0.0f;
```

```
for (int t = 0; t < w / TILE_DIM; t++) {
```

```
    ...
```

```
    aTile[threadIdx.y][threadIdx.x] = ...;
```

```
    bTile[threadIdx.y][threadIdx.x] = ...;
```

```
    __syncthreads();
```

```
    for (int i = 0; i < TILE_DIM; i++)
```

```
        sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
```

```
    __syncthreads();
```

```
}
```

```
c[row*N+col] = sum;
```

