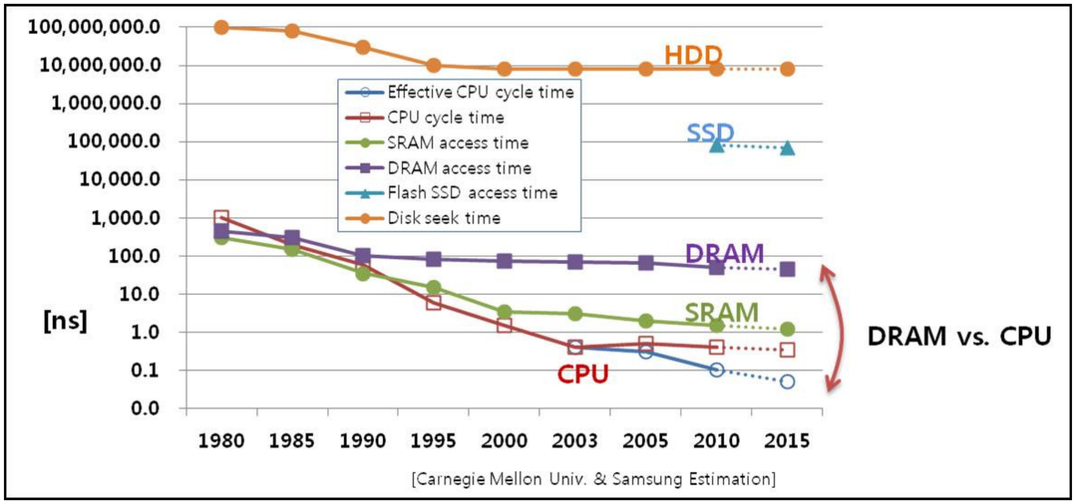
GPUs and CUDA 2
Memory and Warps

CS121 Parallel Computing

Fall 2021

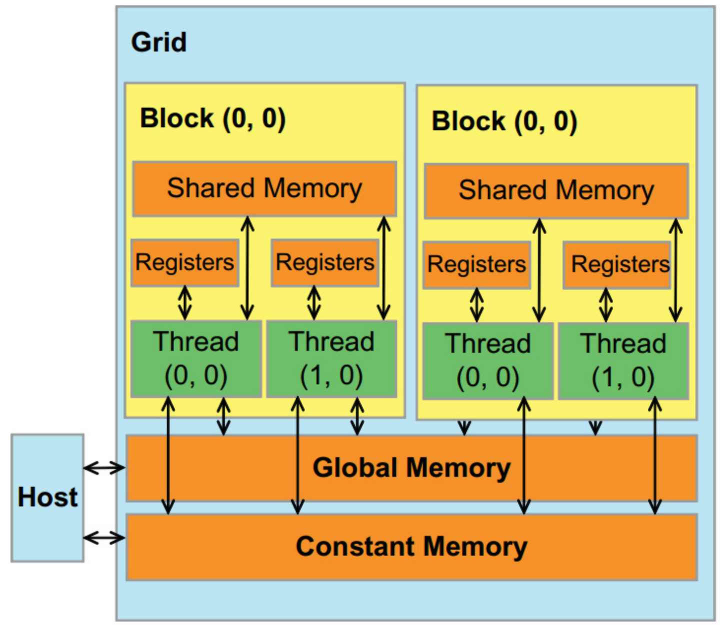
Need for speed



- Speed of code determined by amount of computation and memory accesses.
- Computation speed has been improving much faster than memory latency and bandwidth.
- Today, the main bottleneck is high memory latency and low memory bandwidth relative to CPU.
- But processors can access many different types of memory.
- Can write fast code if use right memory at right time.

GPU memory organization

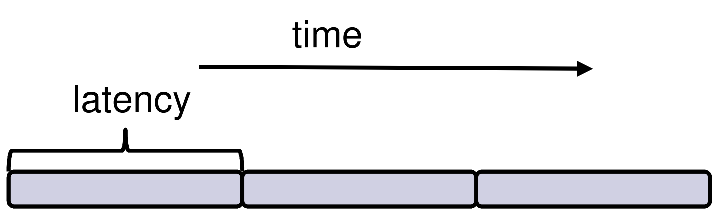
- GPU has several types of memory.
  - Different size, latency, bandwidth and scope.
  - Generally, the larger the size and scope, the slower and less bandwidth.
- Registers, shared memory, L1 cache are on-chip, much faster and higher bandwidth than global memory.
- L1 cache is controlled by hardware.
- In contrast, programmer controls what’s stored in shared memory.
- Shared memory size + L1 cache size = 64KB. User configurable.

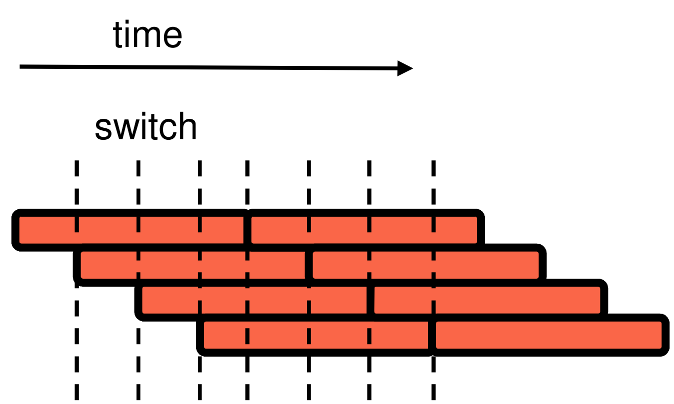


Type	Size	Latency (cycles)	Bandwidth	Visibility
Global	1-32 GB	400-800	900 GB/s	grid
Constant	64KB	cached		grid, read-only
Shared	48KB/16KB per SM	~20	12,000 GB/s	block
L1 cache	16KB/48KB per SM	~20	12,000 GB/s	block
Registers	64K per SM	~1		thread

Global memory latency

- Global memory has very high latency.
- If each thread waits (blocks) for a global memory operation to finish before doing the next operation, performance is very poor.
- Solution is to keep large pool of active threads.
- When one thread blocks doing a memory operation, switch to another thread.
  - "Massive multi-threading" (MMT).
- Total throughput high, even though each thread has high latency.





Global memory latency

- Each SM has own scheduler to do thread switching.
  - Different from device Gigathread scheduler, which allocates thread blocks to SMs.
- Each thread's context (program counter, registers) always maintained in the SM.
  - SM has ~64K registers to allocate to ~1000 threads in a thread block.
  - Very fast, "zero overhead" thread switching.
- SM scheduler has "scoreboard" to keep track of which threads assigned to the SM are blocked / unblocked.
  - Keeps picking unblocked threads to run.
- Only effective if SM has many threads, so that there always exists some unblocked threads.
  - This is why SM can run ~1000 threads, though it only has ~30 cores.
- For high performance need many threads per SM.
  - High "occupancy".

Global memory bandwidth

- Massive multithreading not enough for performance.
  - Only addresses latency.
  - But doesn't help with other bottleneck, bandwidth.
- GPU's computing power is much higher than its global memory bandwidth.
  - Ex Compute:1.5 TFLOPS. Bandwidth: 200 GB/s.
- Recall matrix multiplication

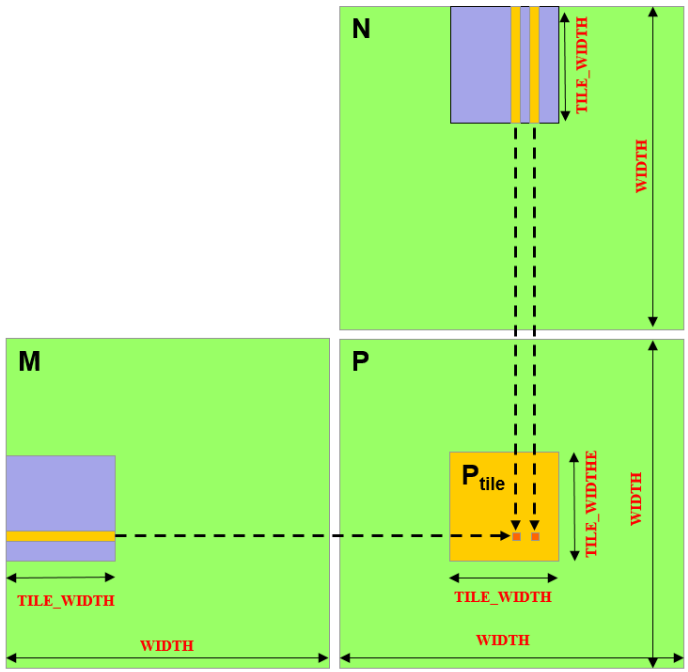
```
Pvalue += M[Row*Width+k] * N[k*Width+Col]
```

- 6 floating point ops (+, *) for 2 memory ops (read M and N).
  - Compute to global memory access (CGMA) ratio 3:1.
- 200 GB/s = 50G floating point vals / sec $\Rightarrow$ 150 GFLOPS.
  - 1/10 of theoretical peak!

Exploiting data reuse

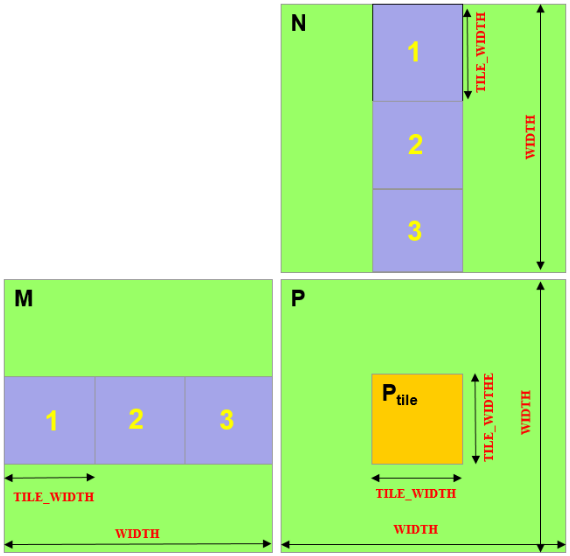
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
         float Pvalue = 0;
         for (int k = 0; k < Width; ++k)
         Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
         d_P[Row*Width+Col] = Pvalue; } }
```

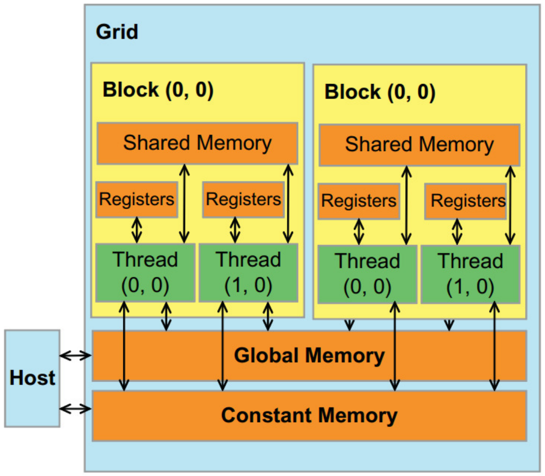
- P_{tile} contains a block of TILE_WIDTH^2 threads.
- Every thread loads all data it needs by itself.
  - TILE_WIDTH^2 threads in P_{tile} each loads 2*TILE_WIDTH data $\Rightarrow$ 2*TILE_WIDTH^3 global memory reads.
- But notice all threads in P_{tile} need data from purple tiles.
  - Purple data can be reused!
- Threads in P_{tile} cooperate to load purple tiles, eliminating redundant global memory reads.
  - Only 2*TILE_WIDTH^2 global memory reads in total. A factor of TILE_WIDTH less!



Shared memory and tiled MM

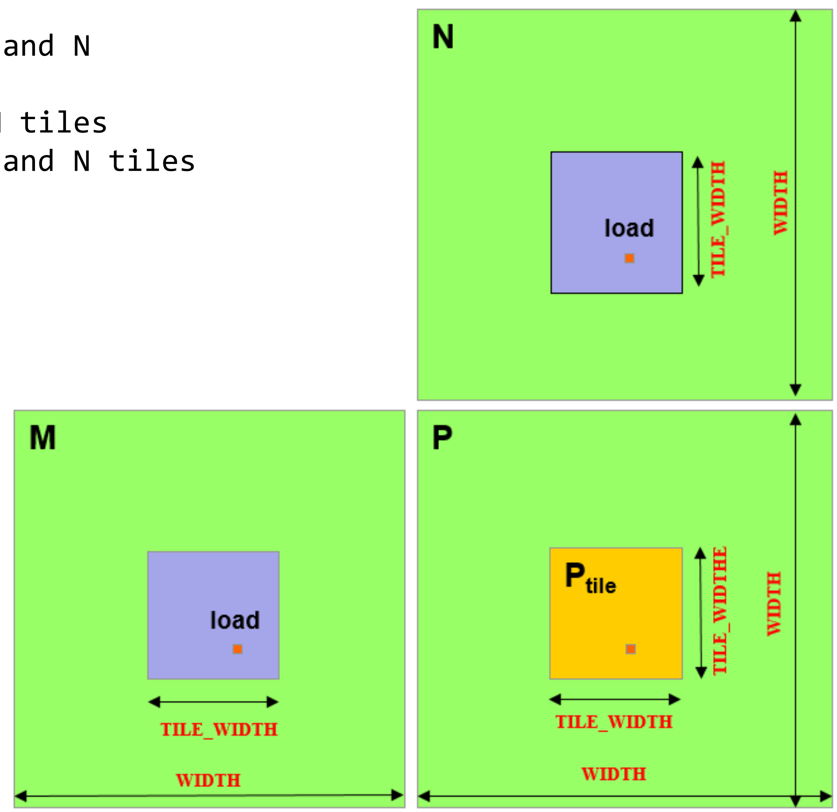
- Tiled MM is a memory efficient method of performing matrix multiplication using shared memory.
- Break M, N into tiles and multiply tile by tile.
  - Work in phases.
  - Exploit data reuse in each phase.
- # phases = WIDTH / TILE_WIDTH
- In phase i, threads in P_{tile} cooperatively load i^{th} tile from M, N in global memory into shared memory.
- Then each thread reads a row and column of data from shared memory.
- After all threads finished with the tiles, next two tiles loaded, overwriting current ones.





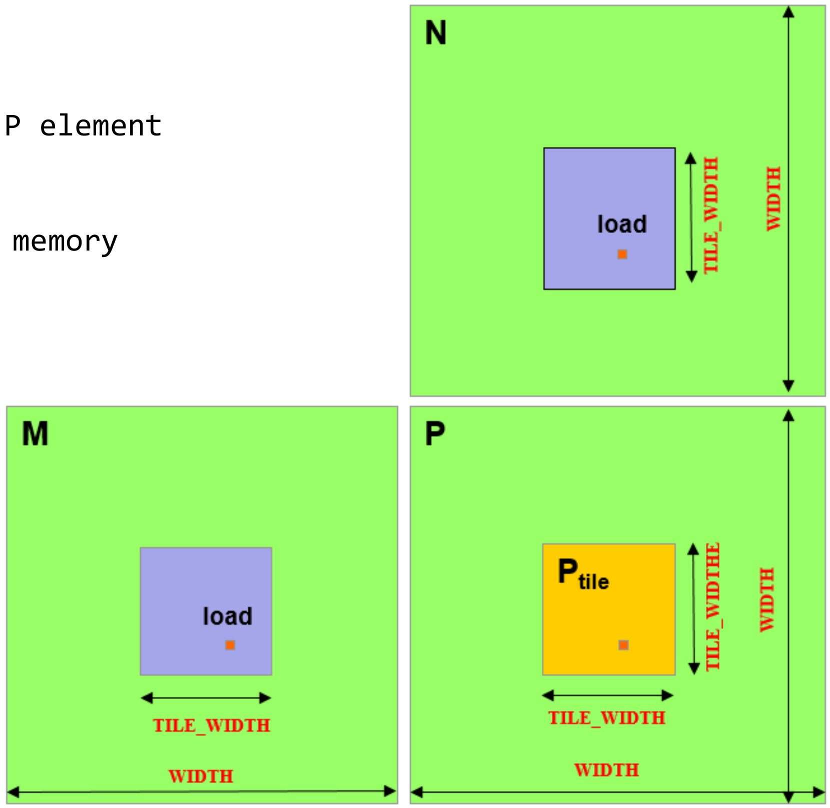
Tiled matrix multiplication

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
// Allocate space for M and N block in shared memory

// Thread in block (bx, by) works on (bx, by)'th P_tile
// Thread with ID (tx, ty) works on element (tx, ty) in its P_tile

// Loop over the M and N tiles required to compute the P element
for (int m = 0; m < WIDTH/TILE_WIDTH; m++) {
     // Load one element from M and N into shared memory
     // How do we calculate which elements to load?
     // Wait till all threads finished loading from M and N

     // Compute dot product from shared memory M and N tiles
     // Wait till all threads in block finish using M and N tiles
     }
// Write value of P element back to global memory
}
```



Tiled matrix multiplication

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
__shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
__shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

int bx = blockIdx.x;  int by = blockIdx.y;
int tx = threadIdx.x; int ty = threadIdx.y;

// Thread identifies row and column of P element to work on
int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;

float Pvalue = 0;
// Loop over the M and N tiles required to compute the P element
for (int m = 0; m < WIDTH/TILE_WIDTH; m++) {

     // Collaboratively load M and N tiles into shared memory
     ds_M[ty][tx] = d_M[Row*WIDTH + m*TILE_WIDTH+tx];
     ds_N[ty][tx] = d_N[Col+(m*TILE_WIDTH+ty)*WIDTH];

     // Wait till all threads finished loading tiles
     __syncthreads();

     // Compute dot product from tiles
     for (int k = 0; k < TILE_WIDTH; ++k)
          Pvalue += ds_M[ty][k] * ds_N[k][tx];

     __synchthreads();
     }
d_P[Row*Width+Col] = Pvalue;
}
```



Shared memory and tiled MM

- Decreases total number of reads from global memory compared to naive algorithm.
- Consider an nxn matrix with mxm blocks.
- Naive algorithm
  - n^2 elements each require 2n global memory reads. $2n^3$ total.
- Shared memory algorithm
  - $(n/m)^2$ blocks in total.
  - Each block does n/m phases.
  - In each phase, block does $2m^2$ reads from global memory.
  - Total $(n/m)^2*(n/m)*2m^2 = 2n^3/m$ accesses.
  - A factor of m less!
- Doesn't decrease overall number of memory accesses.
  - Threads do $2n^3$ reads to shared memory instead of global memory.
- But get much better performance, because shared memory bandwidth is 10X global memory bandwidth.

Tiled matrix multiply performance

- With 16x16 tiles, decrease global memory usage by factor of 16.
- Can get (200GB/4B)*(3 FLOP / 1B)*16 = 2400 GFLOPS, compared to 150 GFLOPS from before!
- Each thread block uses 16 x 16 x 4B x (2 matrices) = 2KB of shared memory.
- Even if only 16KB shared memory, can still run 8 blocks per SM, which is enough to achieve full occupancy.
  - Each block has 256 threads, so 6 blocks enough to saturate SM with 1536 thread capacity.
- If use 32x32 tiles, then 1024 threads per tile / thread block, so only one thread block per SM.
  - Only 2/3 occupancy if SM can run 1536 threads.
  - Note the tradeoff between improving bandwidth and occupancy.

Effective use of shared memory

- General strategy is to find reused data and load it into fast, high bandwidth memory.
  - Very effective in practice.
  - Same principle as caches, except that programmer controls what's in shared memory.
  - Data reuse is also called temporal locality.
- Using shared memory is sometimes hard, since it's so small.
  - 48KB shared mem / SM, vs. 12 GB global mem.
  - On a CPU, 32KB L1 cache vs 32 GB main mem.
- Designing algorithms with high temporal locality is one of main techniques for getting fast code.

CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

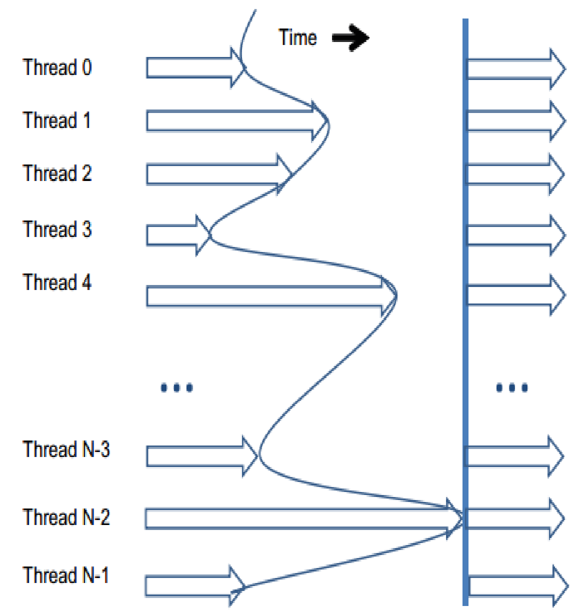
- `__device__` is optional when used with `__shared__`, or `__constant__`.
- Automatic variables without any qualifier reside in a register.
  - Except per-thread arrays, which reside in global memory.

Synchronization

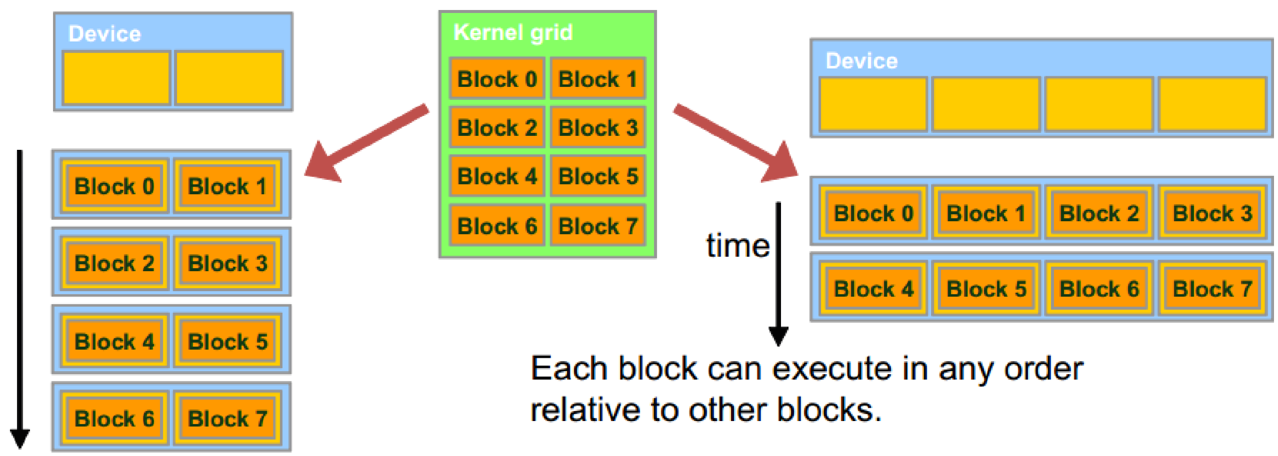
- In algorithms up to now, threads mostly worked independently.
  - E.g. it's ok if say thread 1 took 3 steps but thread 2 took 7.
- Sometimes threads need to coordinate. I.e. they must all finish some step before any thread can go on to next step.
  - Ex In MM, threads must finish loading M, N tiles into shared mem before any thread can start its dot product.
- Threads within a block can synchronize using `__syncthreads()`.
  - Scheduler ensures a thread reaching `__syncthreads()` statement blocks until all threads in block also reach that `__syncthreads()`.
  - This is a barrier synchronization.

Synchronization

- Must be careful with synchronization.
- Ex If code has `__syncthreads()` but some thread in block goes into infinite loop, block will never finish!
- Ex If `__syncthreads()` occurs in `if` block, then all threads must go through the `if`, or none do. Otherwise block never finishes.
- Ex If `__syncthreads()` occurs in both branches of `if-then-else` code block, all threads must go through same branch, or block never finishes.
- `__syncthreads()` can also cause wasted work. Threads arriving earlier at the barrier wait for later threads.
  - Decreases number of schedulable threads. May hurt latency hiding.
- Avoid synchronization if possible.



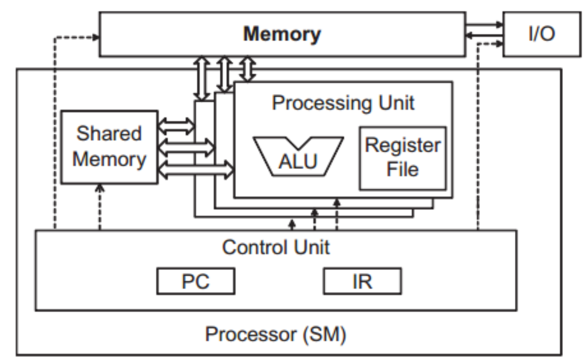
Synchronization



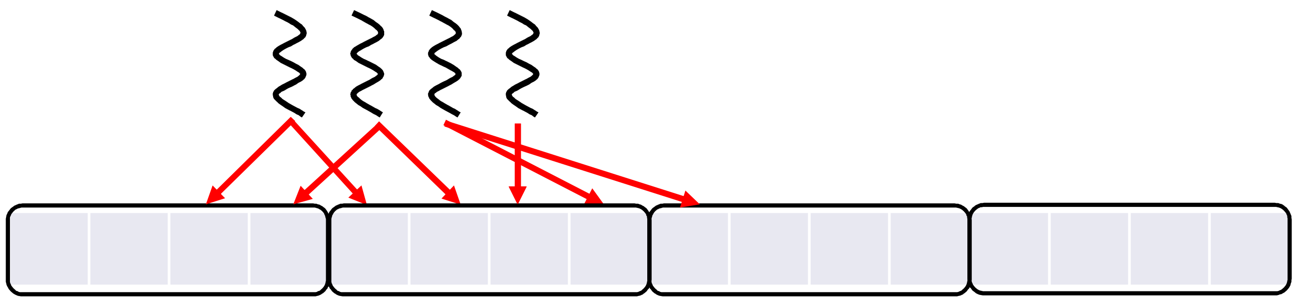
- Threads in different blocks can’t synchronize.
- This allows blocks to execute in any order, and scale transparently to GPUs with more SMs.
- Downside is if need inter-block synchronization, must wait till all blocks in kernel finish, then start a new kernel.

Thread warps

- An SM contains one or more SIMD (single instruction multiple data) processors.
  - Each SIMD processor contains multiple cores that run the same command on different data.
- The unit of “SIMDness” is a warp of 32 threads.
  - An entire warp of threads runs at a time.
  - A thread block is divided into warps with consecutive threadIdx.x values.
- Execution is fast when entire warp “does the same thing”.
  - Different warps can do different things without performance loss.
- It’s much slower when there’s non-coalesced memory accesses, control flow divergence or bank conflicts.

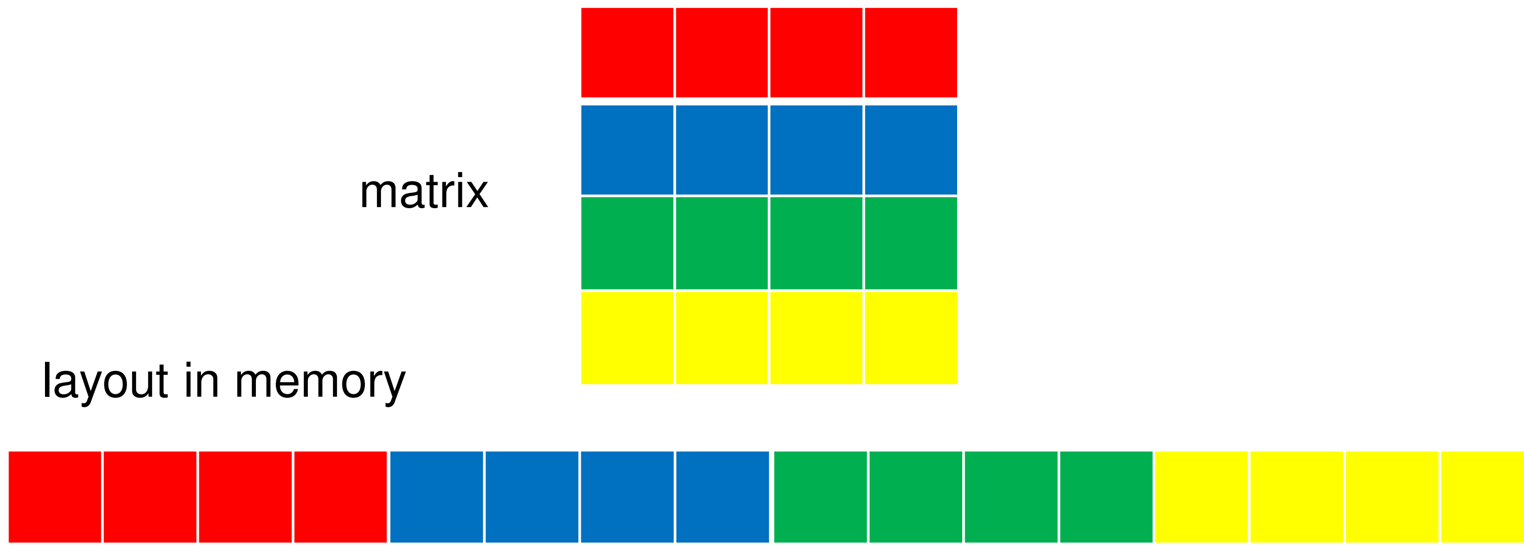


Memory coalescing



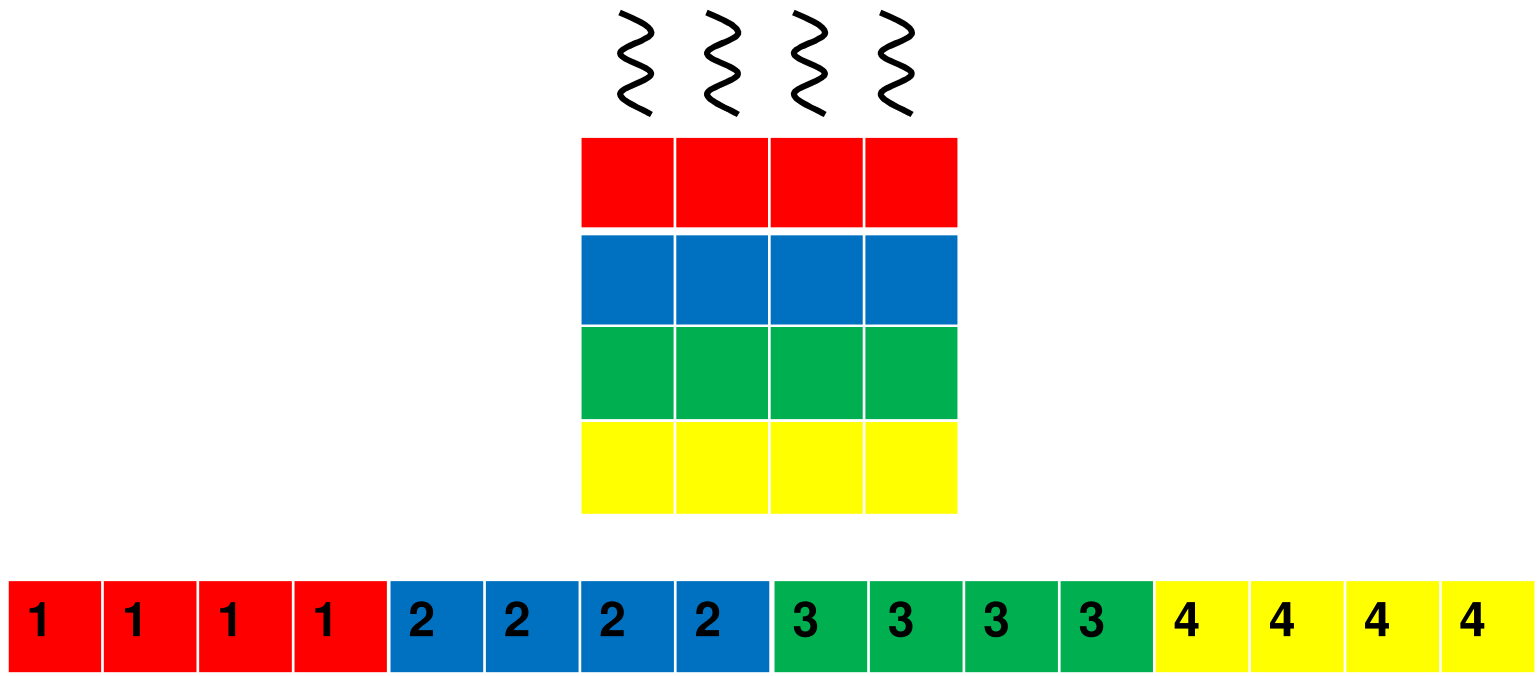
- Global memory divided into segments of 128 B (= 32 floats).
- Suppose SM executes a warp of 32 threads all executing a SIMD instruction reading from global memory.
  - If all 32 locations being read lie in one segment, hardware detects this and only transfers one segment (128 B) from global memory to SM.
    - Access is coalesced.
  - If locations lie in k different segments, k*128 B are transferred.
    - Access is uncoalesced.
  - In worst case, transfer 32*128 B = 4KB to read 32 floats!
    - Huge waste of limited global memory bandwidth.
- For good performance, make global memory accesses as coalesced as possible.

Coalescing example



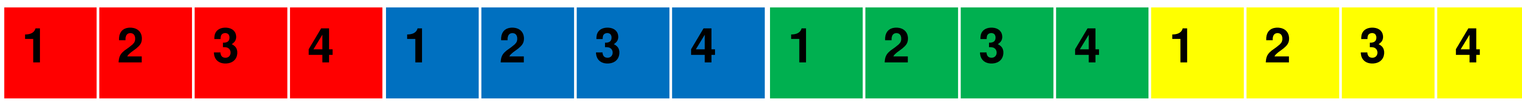
- Say we have 4x4 matrix, stored in row major format.
- Suppose segments are 4 elements wide.
  - I.e. can transfer 4 consecutive elements in one step.
- We have warp of 4 threads, and want to iterate through matrix either row by row, or column by column.

Coalescing example



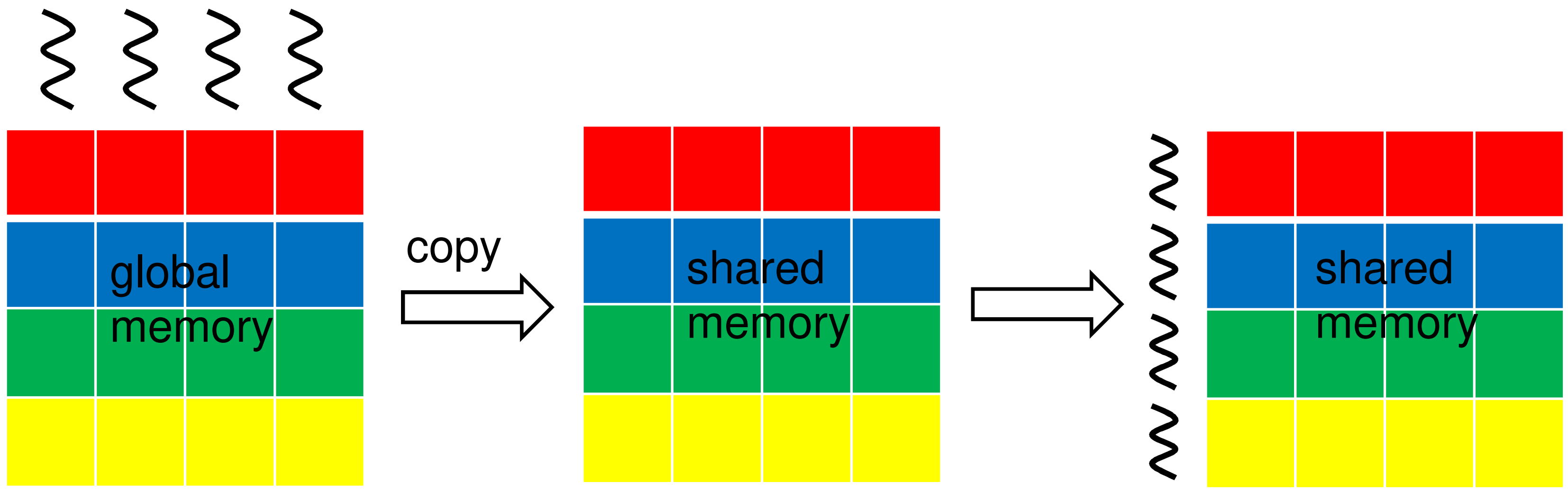
- When iterating by row, we naturally map one thread to each column.
  - Need 4 iterations in total.
- Numbers show locations accessed each iteration.
  - Locations all consecutive. All iterations coalesced.

Coalescing example



- When iterating by column, map one thread per row.
  - In iteration 1, access locations 0,4,8,12.
  - In iteration 2, access locations 1,5,9,13. Etc.
  - Each iteration accesses nonconsecutive locations.
    - All accesses noncoalesced.

Improving coalescing



- Only global memory has bandwidth penalty for noncoalesced accesses.
- Shared memory has much smaller penalty for scattered accesses.
- To improve coalescing, first do coalesced read from global to shared memory. Then make scattered accesses to shared memory.
- Ex To read matrix by column, first read it by row and copy to matrix in shared memory. Then read shared memory matrix by column.
- Once again shows flexibility of shared memory vs global memory.

Warp divergence

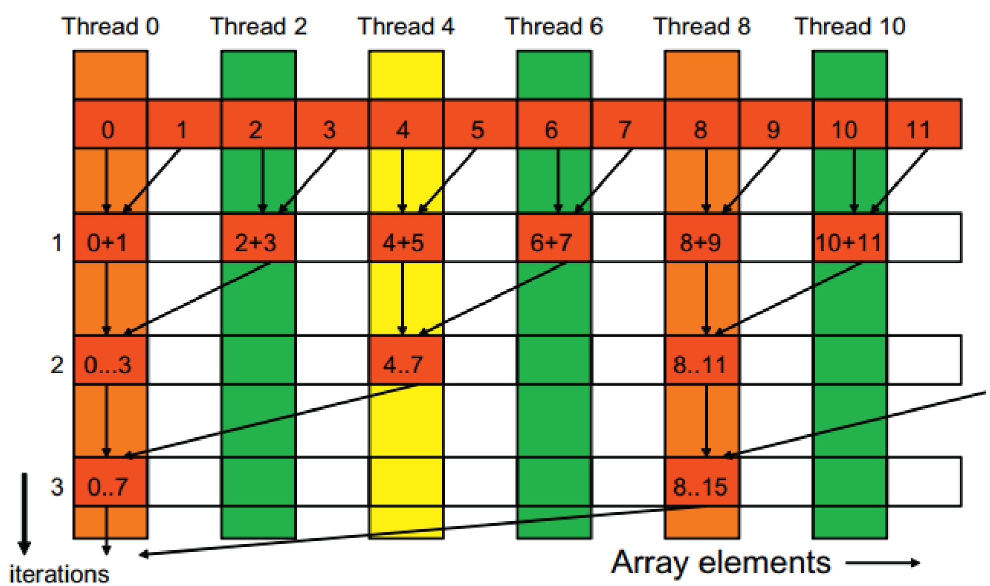
- Since SM is SIMD, efficient when all threads run same instruction.
  - SM finishes a warp in one pass.
- But if code has branches, threads can run different instructions.

```
if (threadIdx.x % 3 == 0)  i += 1;
else if (threadIdx.x % 3 == 1)  i -= 1;
else i *= 2;
```

  - Called warp divergence.
- If threads have k branches, SM takes k passes to run warp.
  - In each pass, runs all threads of one branch, which all run same instruction.
- In worst case, SM takes 32 passes to run one warp!

Divergence example

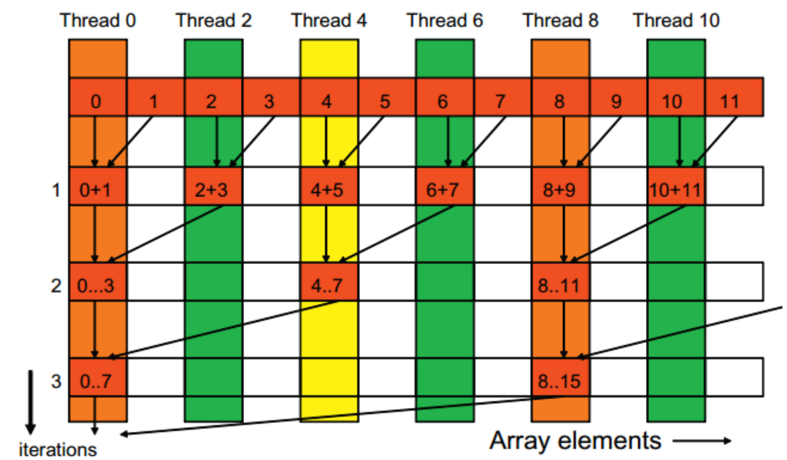
- Reduction produces single number from array of numbers.
  - Ex Returns sum, max, or min of array.
  - Very commonly used operation.
- Can be computed in parallel using reduction tree.
  - With n values and n threads, do O(n) additions, take O(log n) iterations.



Divergence example

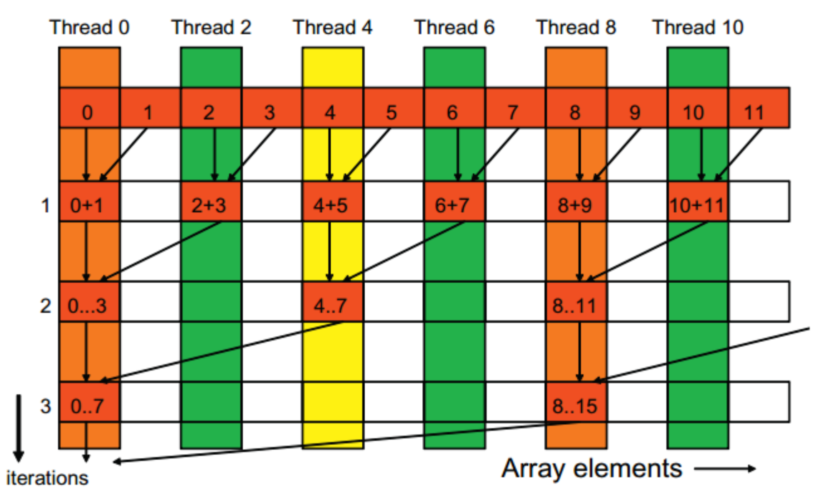
```
int t = threadId.x;
for (int stride = 1; stride < blockDim.x; stride *= 2) {
     __syncthreads();
     if (t % (2*stride) == 0)
          partialSum[t] += partialSum[t+stride];
}
```

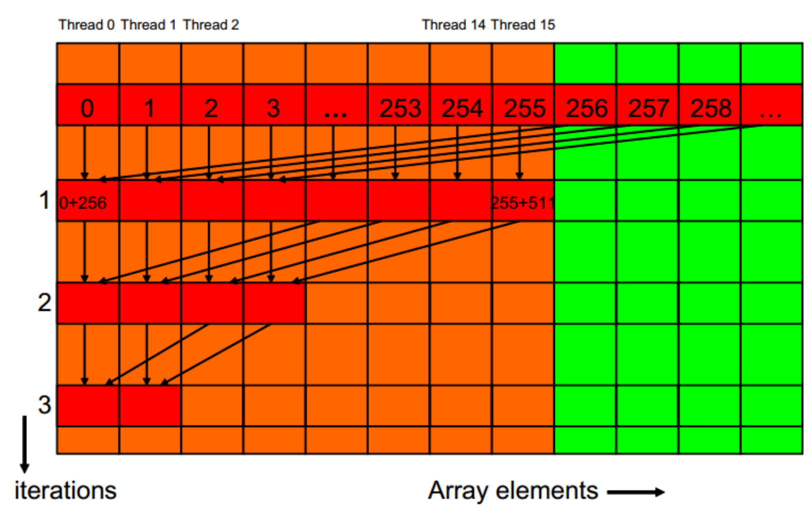
- Number of threads working halves every iteration.
- Each working thread adds value from stride away into its location in output array.
  - stride=1 in iteration 1, then 2, 4, 8, etc.
- `__syncthreads()` ensures all values from one iteration complete before computing next iteration.



Reducing divergence

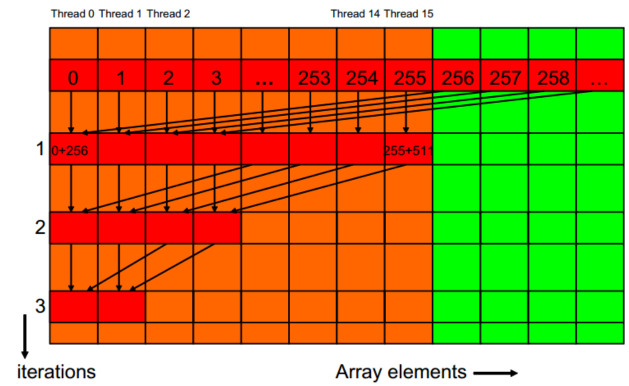
- Each iteration has 2 branches, the threads with (t % (2*stride) == 0), and the other threads.
  - Each iteration takes 2 passes to complete.
- Can modify algorithm to decrease amount of divergence.
  - Still tree based, but first add values from far away.
    - Iter 1: first 256 threads add value from 256 away.
    - Iter 2: first 128 threads add value from 128 away.
    - Iter 3: first 64 threads add value from 64 away. Etc.
  - Work, time complexity same as before.





Reducing divergence

```
int t = threadId.x;
for (int stride = blockDim.x; stride > 1; stride /= 2) {
     __syncthreads();
     if (t < stride)
          partialSum[t] += partialSum[t+stride];
}
```



- Why is divergence reduced?
  - In first iteration, first 256 threads = 8 warps all do if, last 8 warps all don't.
  - In second iteration, first 128 threads = 4 warps do if, last 12 don't.
  - No divergence till 5th iteration.
- From 5th iteration on, 16, 8, 4, 2 threads from first warp do if.
- 8 iterations, only 4 have divergence.
- Can you get rid of the 4 iterations with divergence?

Bank conflicts

- Shared memory is arranged in banks.
  - A bank stores a set of 4B data.
  - Allows parallel accesses. Threads can access different banks at same time.
- If n banks, then address x is stored in bank x % n.
  - Current GPUs have 32 banks.
- If threads in a warp access different banks, completes in one pass.
- If k > 1 threads access different addresses in same bank, get k-way bank conflict.
  - Accesses serialize, takes k passes to complete accesses.
  - Unless all threads access same value, which then gets broadcast in one pass.
- Different warps don’t have bank conflicts.

bank 0	**0**	**4**	**8**	**12**	**16**
bank 1	**1**	**5**	**9**	**13**	**17**
bank 2	**2**	**6**	**10**	**14**	**18**
bank 3	**3**	**7**	**11**	**15**	**19**

Bank conflict example

- Suppose 4 banks, warp size = 4 and block size = 4.
- Want each thread to load two values from global memory into shared memory.
- If thread loads consecutive locations, 2 way bank conflict.
- If thread loads locations block size apart, no bank conflicts!

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid + 1] = global[2*tid+1];
```

```
int tid = threadIdx.x;
shared[tid] = global[tid];
shared[tid + blockDim.x] =  global[tid +
     blockDim.x];
```