GPU Computing and CUDA Programming

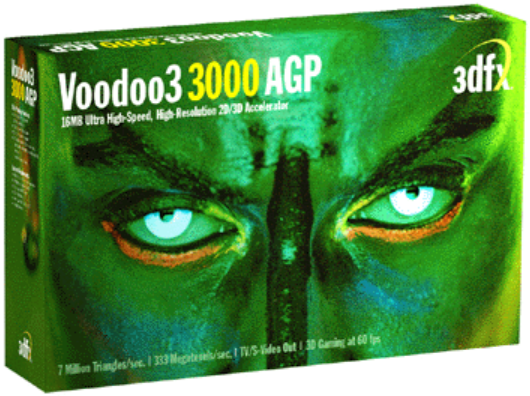
Computer Architecture 2

Fall 2020

Rui Fan

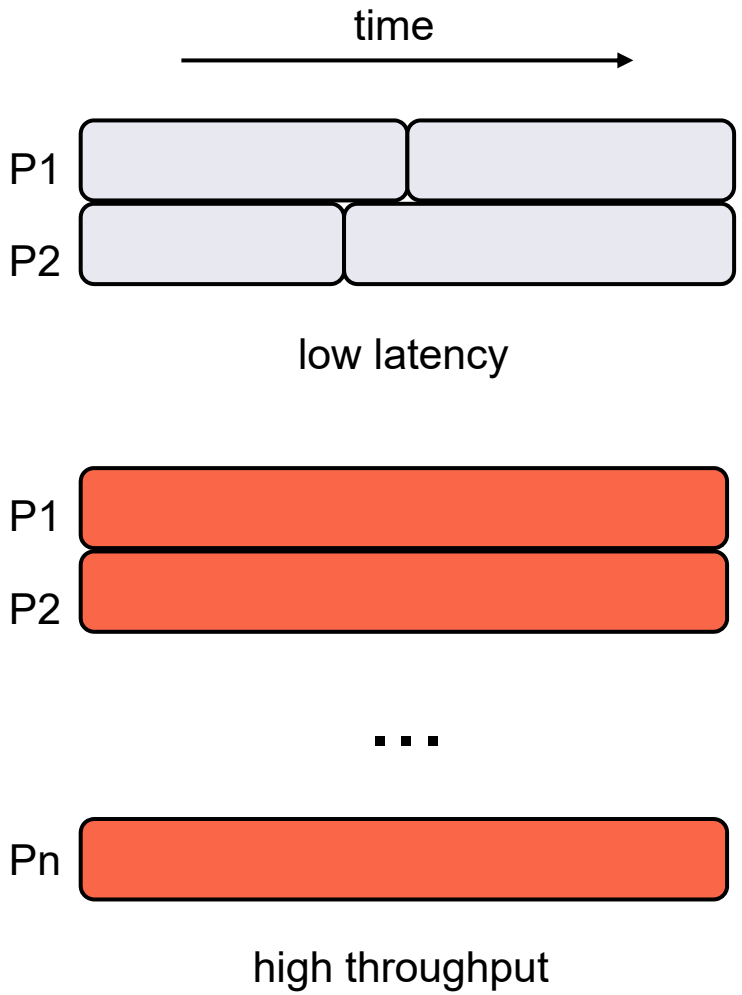
A brief history

- Graphics processing units (GPU) originally used to speed up 3D games.
- Need high throughput (lots of pixels), but parallelism abundant (compute pixels independently).
- Fancier games required programmable “pixel shaders”.
- Around 2006, Nvidia introduced Tesla, a programmable, general purpose GPU (GPGPU).
- GPUs now essential in machine learning, big data and HPC. Large amounts of research.
- GPUs have TFLOPS of performance, “supercomputer on a chip”.
- Also more energy efficient than CPUs, which is increasingly important.



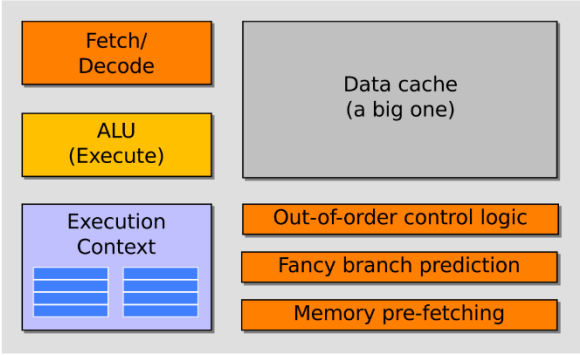
Latency vs throughput

- Up to now we looked at message passing and shared memory parallel computing using standard multicore processors.
- Multicore processors have a few cores, and try to minimize latency on each core.
- Throughput oriented parallel processors do each task slower, but have many cores, and so can do many tasks in parallel.
- Throughput processors can do more work per unit time.



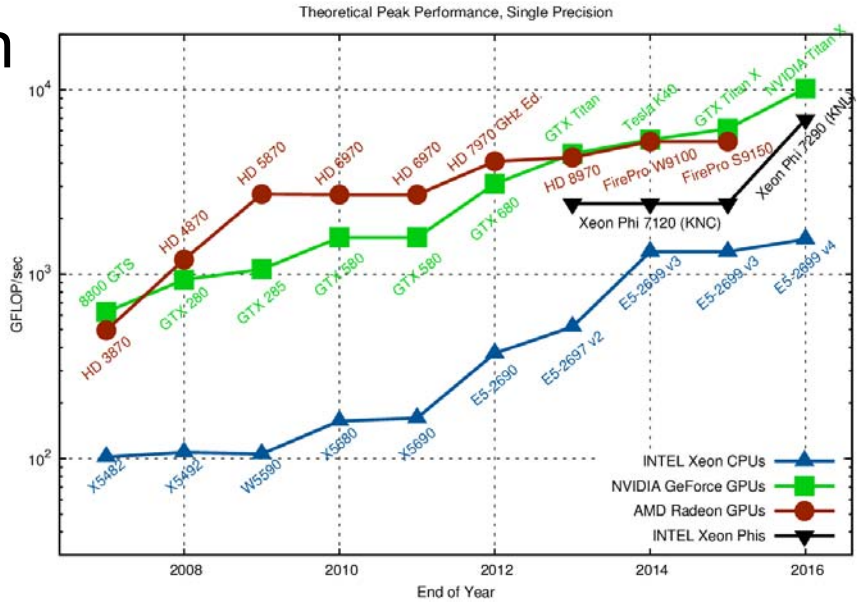
GPU vs CPU architecture

- CPU has many complex features to lower latency.
  - Consumes lots of die space.
  - Less space for compute units.
- GPU only has basic processor units, and shares them among the cores.
  - Each core slower.
  - But lots of them.
- Nvidia Tesla P100 has 56 SMs and 64 cores per SM.
  - Runs 3584 threads simultaneously, 11 TFLOPS of performance.
  - 16 GB of memory, 720 GB/s of bandwidth.
- Intel Xeon E7-8890 v4 runs 48 threads simultaneously (using hyperthreading), about 3 TFLOPS.



The right choice(?)

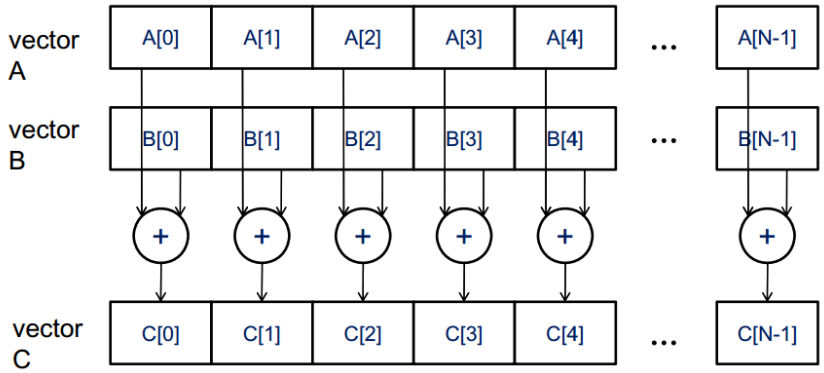
- GPUs >10 times faster than CPUs for many problems.
  - Even more speedup for specialized applications.
- GPUs also much more energy efficient.
- Titan (20 petaflops) uses 18,688 Nvidia Tesla K20X GPUs.
- Best for data parallel tasks.
- GPU is based on SIMD architecture.
- Less effective for irregular computations (branching, synchronization).



Source: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Data parallelism

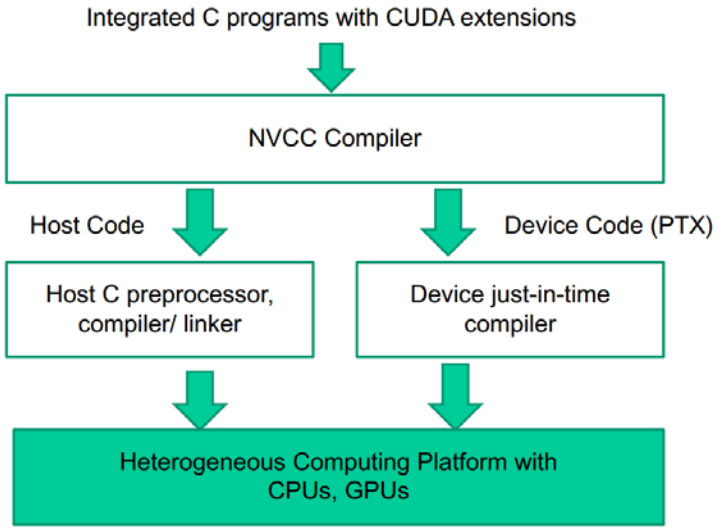
- Apply same operation to multiple data items.
- Vector addition.



- Linear algebra (matrix-vector, matrix-matrix multiplication).
- Computer graphics.
- Data analysis (convolutions, FFT).
- Finite elements.
- Simulations.
- “Big data”, data mining and machine learning.

CUDA

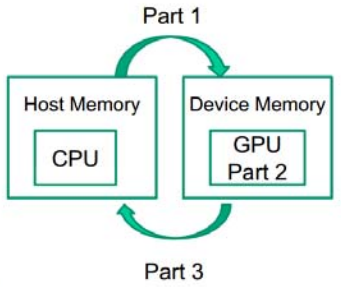
- Compute Unified Device Architecture.
- Easily use GPU as coprocessor for CPU.
- Popular Nvidia platform for programming GPUs.
  - An extension of C language.
  - Compiler, debugger, profilers provided.
- Other platforms include OpenCL and OpenACC.
  - OpenCL is similar CUDA, but more portable.
    - Same source code can be compiled for GPUs, CPUs, FPGAs, etc.
    - Somewhat lower performance than CUDA.
  - OpenACC similar to OpenMP, i.e. write GPU code using simple directives.
    - Compiler takes care of parallelization.
    - Significantly lower performance than CUDA.

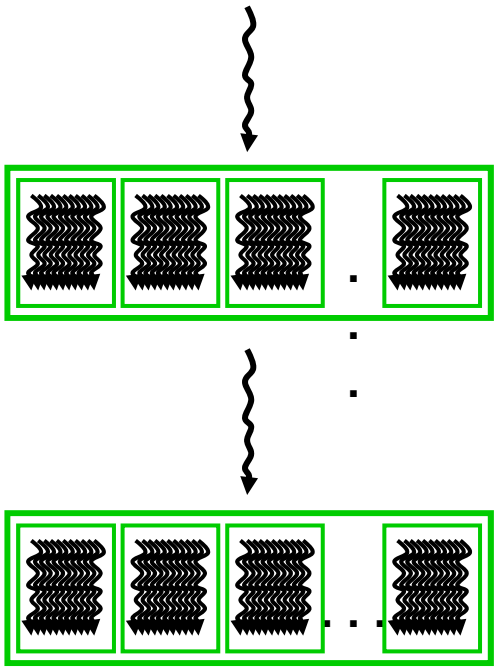


CUDA steps

- Write C program with CUDA annotations and compile.
- Start CUDA program on host (CPU).
- Run mostly serial parts on host.
- For parallel part
  - Allocate memory on device (GPU).
  - Transfer data to device.
  - Specify number of device threads.
  - Invoke device kernel.
- Can pass control back to CPU and repeat.

```
#include <cuda.h>
…
void vecAdd(float* A, float*B, float* C, int n)
{
  int size = n* sizeof(float);
  float *A_d, *B_d, *C_d;
  …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```





CUDA functions

- Use labels to declare host and device functions.

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- Allocate memory on device.

```
cudaMalloc((void **) &x, size)
```

- Transfer memory.
  - Let x be some host data and d_x be a pointer to device memory.
  - From host to device (send input).

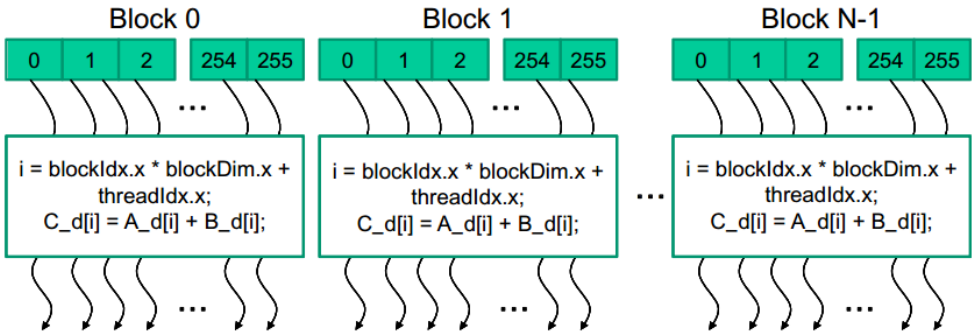
```
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice)
```

  - From device to host (receive output).

```
cudaMemcpy(x, d_x, size, cudaMemcpyDeviceToHost)
```

CUDA functions

- When calling kernel, must specify number of threads.
  - Threads grouped into blocks.
  - Specify number of blocks, and number of threads per block.



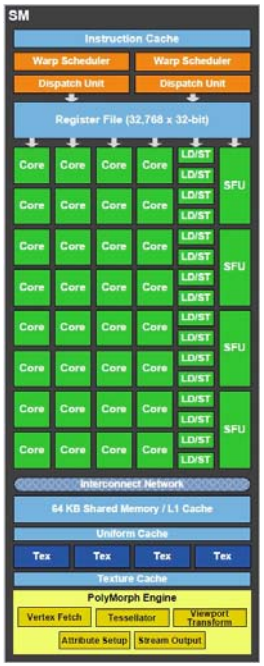
- Invoke kernel.
  - Let `n` be total # threads, `t` be # threads per block.
    - Start `ceil(n/t)`thread blocks with **`t`** threads each.
  - `KernelFunction<<<ceil(n/t), t>>>(args)`
  - `ceil` ensures we have at least n threads.

Vector addition code

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n) {
    int i = threadId.x + blockDim.x * blockId.x;
    if (i<n) C[i] = A[i] + B[i];
}

void vecAdd(float* A, float* B, float* C, int n) {
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    vecAddKernel<<<ceil(n/256), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}

int main() {
    vecAdd(A_h, B_h, C_h, N);
}
```

Why two levels of threads?

- A grid of thread blocks is easier to manage than one big block of threads.
- GPU has 1000's of cores, grouped into 10's of streaming multiprocessors (SMs).
  - Each SM has its own memory, scheduling.
  - Each SM has e.g. 64 cores (P100 architecture).
- GPU can start millions of threads, but they don't all run simultaneously.
- Scheduler (Gigathread Engine) packs up to ~1000 threads into one block and assigns the block to an SM.
  - The threads have consecutive IDs.
  - Several thread blocks can be assigned to an SM at same time.
  - Threads in a block don't execute simultaneously either.
    - They run in warps of 32 threads; more later.

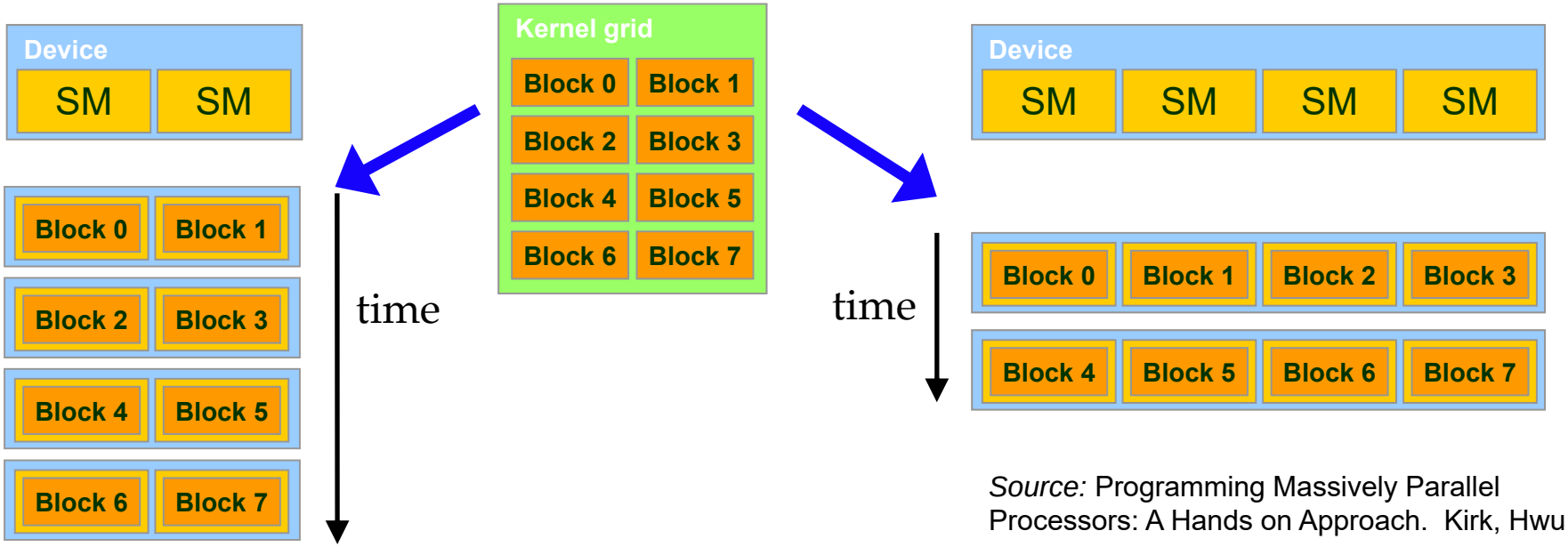


Why two levels of threads?

- A thread block assigned to an SM uses resources (registers, shared memory) on the SM.
  - All assigned threads are pre-allocated resources.
    - Since we know the block size when we invoke the kernel, the SM knows how much resources to assign.
  - This makes switching between threads very fast.
    - No dynamic resource allocation.
    - SM has huge number (e.g. 64K) of registers, so no register flush when switching threads.
- Each SM has its own (warp) scheduler to manage threads assigned to it.
- When all threads in a block finishes, the resources are freed.
- Then Gigathread Engine schedules a new block to the SM, using the freed resources.
- At any time, SM only needs to manage a block of a few thousand threads, instead of entire grid of millions of threads.

Synchronization

- Different blocks can execute in any order.
  - Allows CUDA to easily scale to more SMs on higher end GPUs.
  - Ex For 2 SM GPU, can assign blocks 0,1,2,3,4,5... For 4 SM GPU, assign 0,1,2,3,4,5,6,7...
- Drawback is different blocks can't synchronize, e.g. can't force block 2 to run after block 1 finishes.
  - Your code must not depend on a particular block ordering.

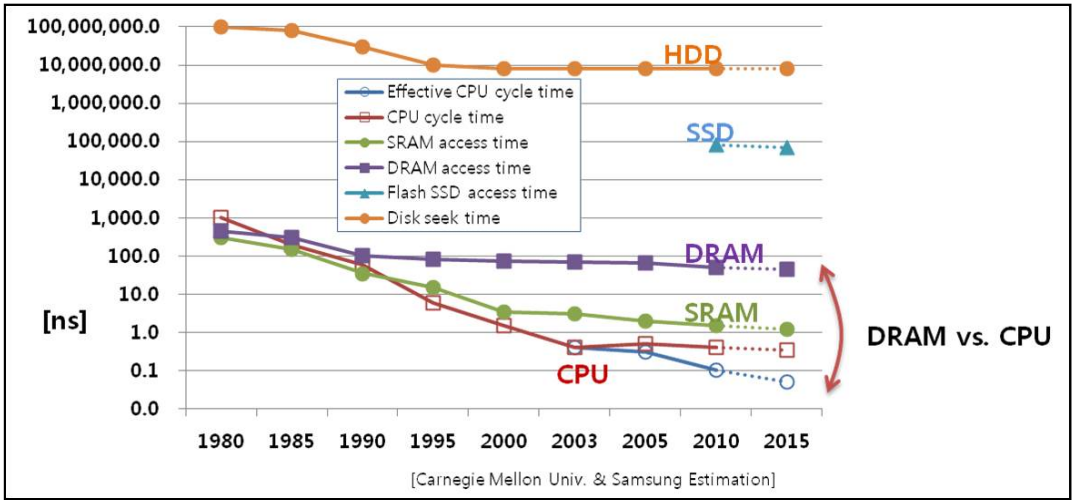


Source: Programming Massively Parallel Processors: A Hands on Approach. Kirk, Hwu

Synchronization

- Suppose you want to synchronize blocks, e.g. make sure some blocks do statement 1 before other blocks do statement 2.
- Can only do this by putting 2 statements in different kernels.
  - Launch first kernel with all blocks doing statement 1.
  - Then launch second kernel with all blocks doing statement 2.
  - Kernel launches relatively expensive, so this is an expensive form of synchronization.
- Threads within a block can do barrier synchronization using `__syncthreads()`.
  - More on this in later lecture.

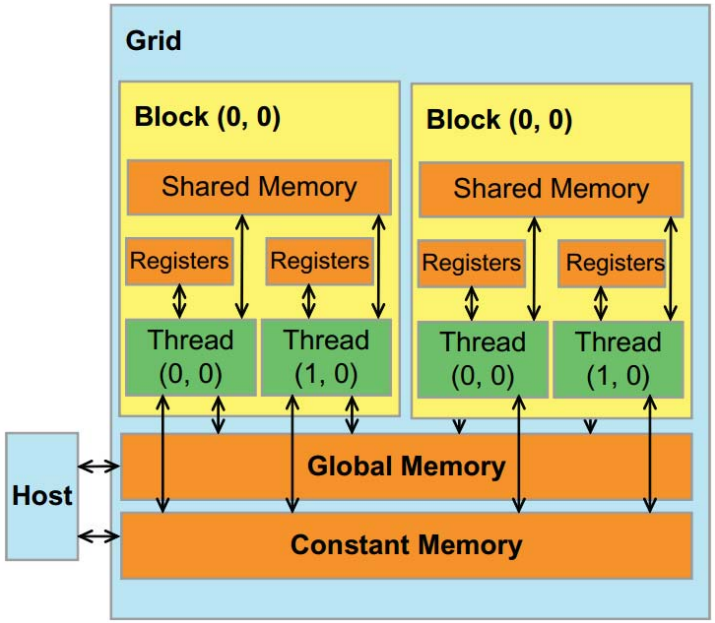
Need for speed



- 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

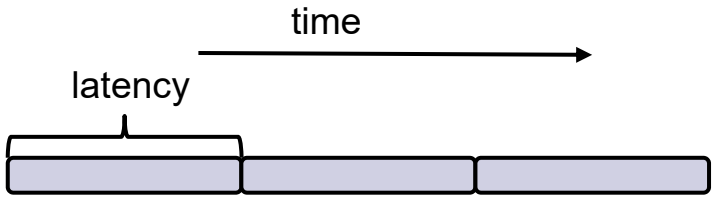
- GPU has several types of memory.
  - Different size, latency, bandwidth and scope.
  - Generally, the larger the size and scope, the slower and less bandwidth.
- Registers, shared memory, L1 cache are on-chip, much faster and higher bandwidth than global memory.
- L1 cache is controlled by hardware.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

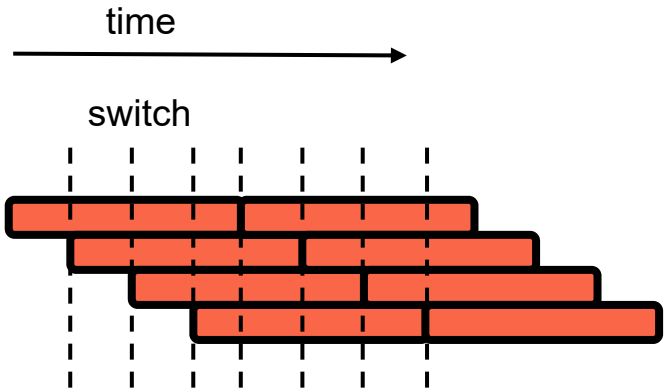


Type	Size	Latency (cycles)	Bandwidth	Visibility
Global	1-12 GB	400-800	150 GB/s	grid
Constant	64KB	cached		grid, read-only
Shared	48KB/16KB per SM	~20	1,500 GB/s	block
L1 cache	16KB/48KB per SM	~20	1,500 GB/s	block
Registers	64K per SM	~1	8,000 GB/s	thread

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

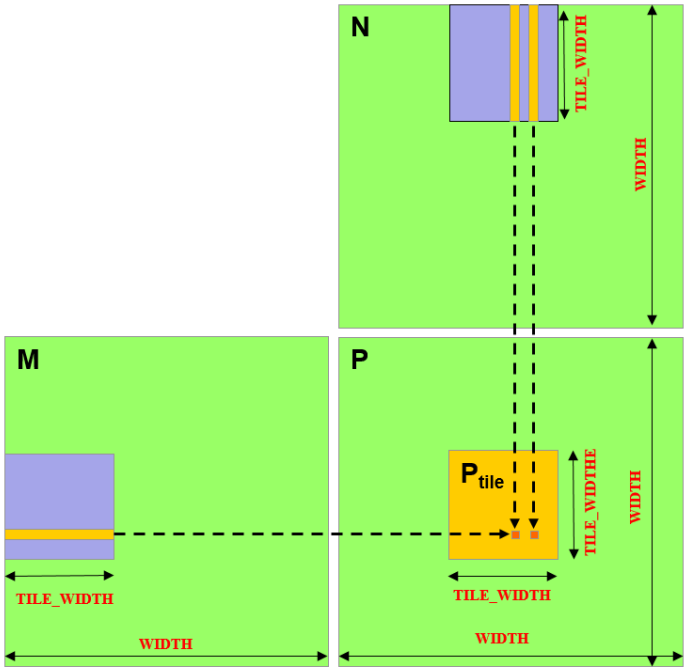
```
Pvalue += M[Row*Width+k] * N[k*Width+Col]
```

- 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

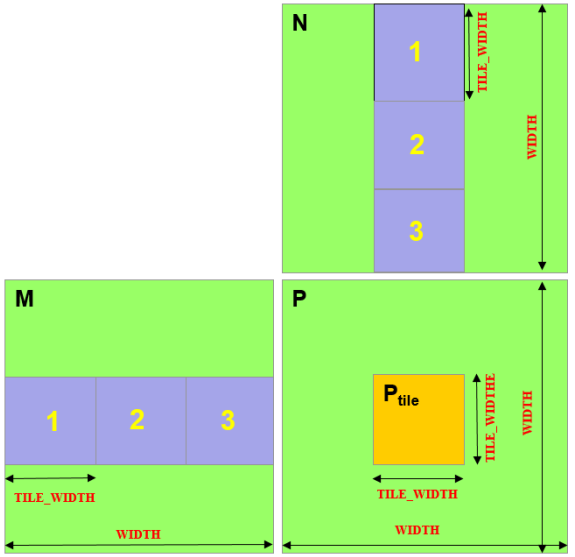
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
     int Row = blockIdx.y*blockDim.y+threadIdx.y;
     int Col = blockIdx.x*blockDim.x+threadIdx.x;

     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; } }
```

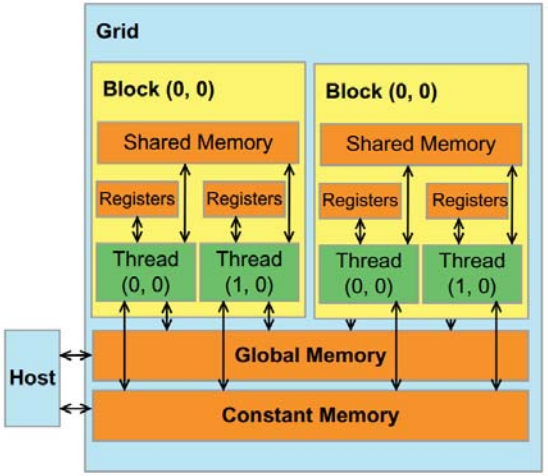
- P_{tile} contains a block of TILE_WIDTH^2 threads.
- Every thread loads all data it needs by itself.
  - TILE_WIDTH^2 threads in P_{tile} each loads 2*TILE_WIDTH data $\Rightarrow$ 2*TILE_WIDTH^3 global memory reads.
- But notice all threads in P_{tile} need data from purple tiles.
  - Purple data 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

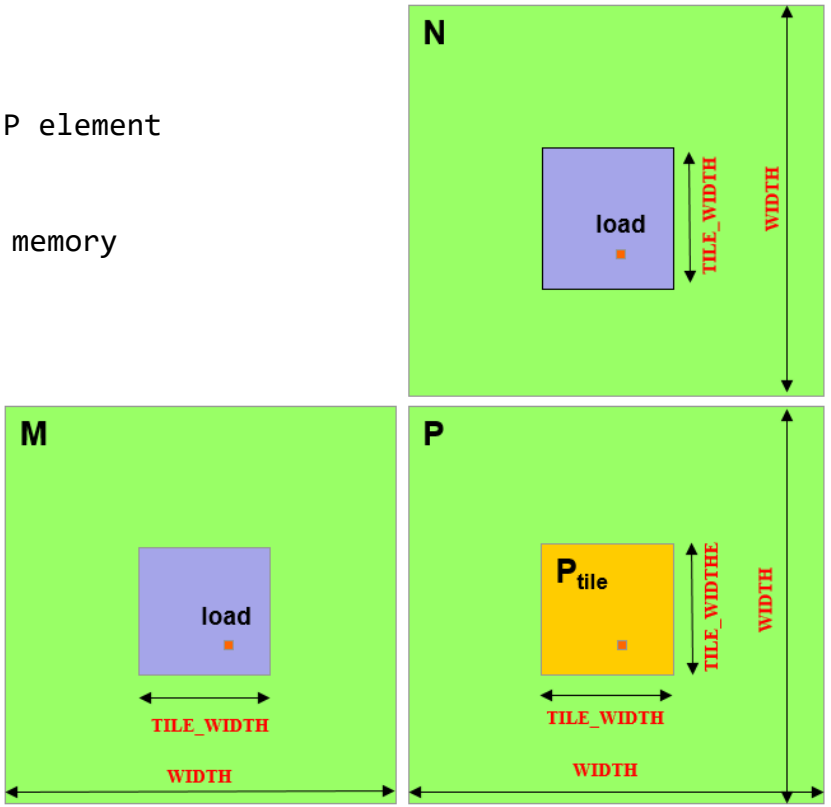
- Tiled MM is a memory efficient method of performing matrix multiplication using shared memory.
- Break M, N into tiles and multiply tile by tile.
  - Work in phases.
  - Exploit data reuse in each phase.
- # phases = WIDTH / TILE_WIDTH
- In phase i, threads in P_{tile} cooperatively load i^{th} tile 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) {
__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;
}
```

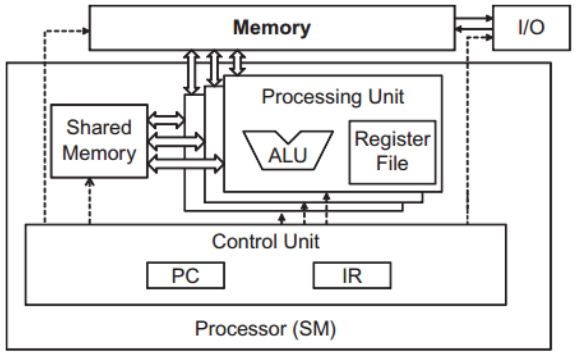


Tiled matrix multiply performance

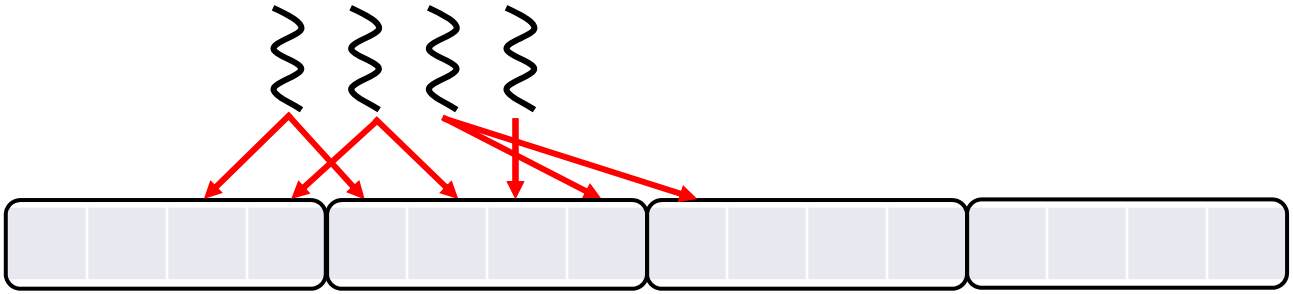
- With 16x16 tiles, decrease global memory usage by factor of 16.
- Can get (200GB/4B)*(3 FLOP / 1B)*16 = 2400 GFLOPS, compared to 150 GFLOPS from before!
- Each thread block uses 16 x 16 x 4B x (2 matrices) = 2KB of shared memory.
- Even if only 16KB shared memory, can still run 8 blocks per SM, which 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.

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.

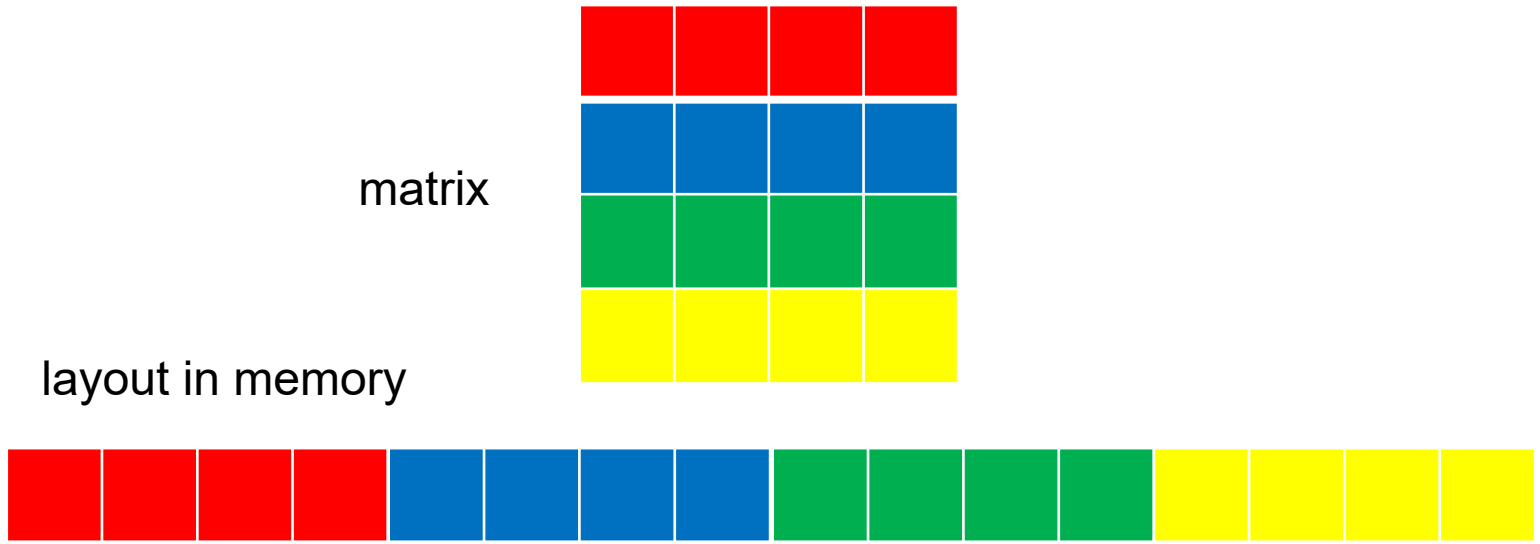


Memory coalescing



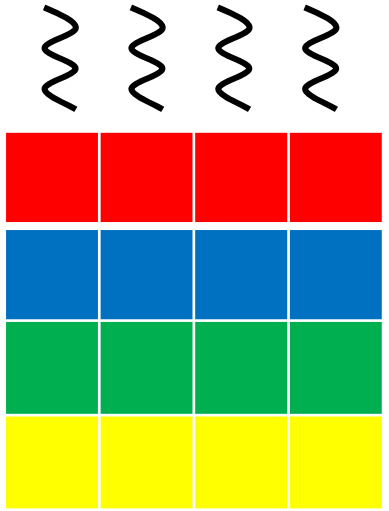
- Global memory divided into segments of 128 B (= 32 floats).
- Suppose SM executes a warp of 32 threads all executing a SIMD instruction reading from global memory.
  - If all 32 locations being read lie in one segment, hardware detects this and only transfers one segment (128 B) from global memory 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



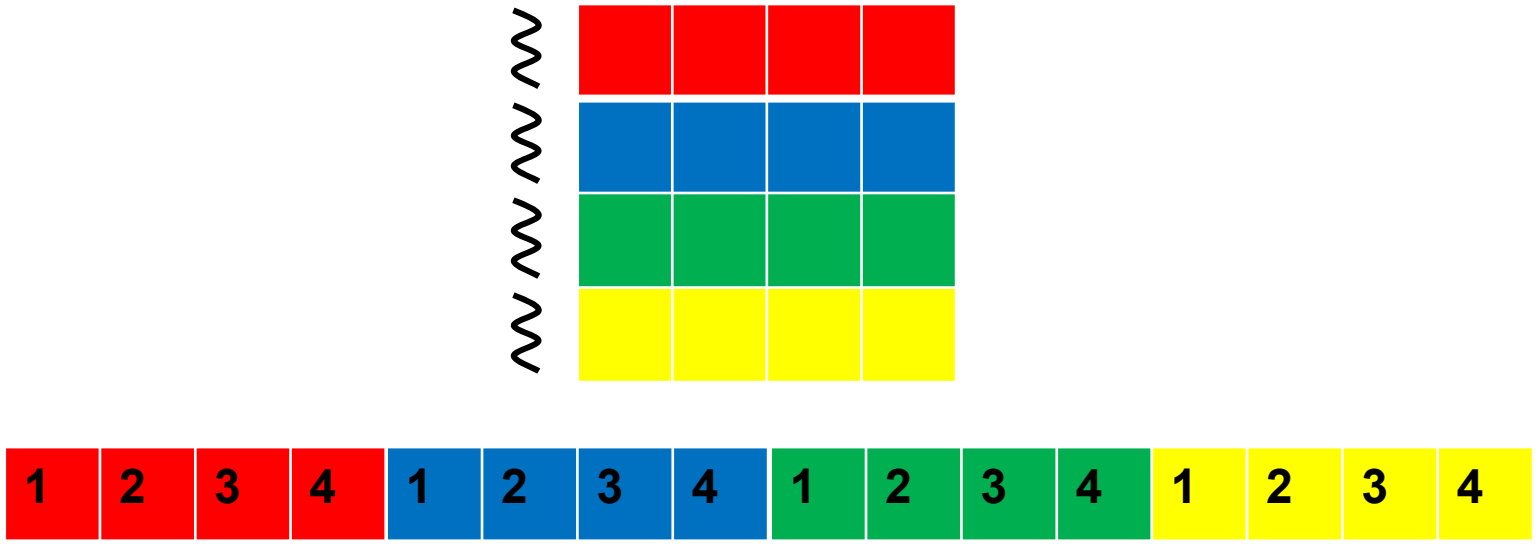
- Say we have 4x4 matrix, stored in row major format.
- Suppose segments are 4 elements wide.
  - I.e. can transfer 4 consecutive elements in one step.
- We have warp of 4 threads, and want to iterate through matrix either row by row, or column by column.

Coalescing example



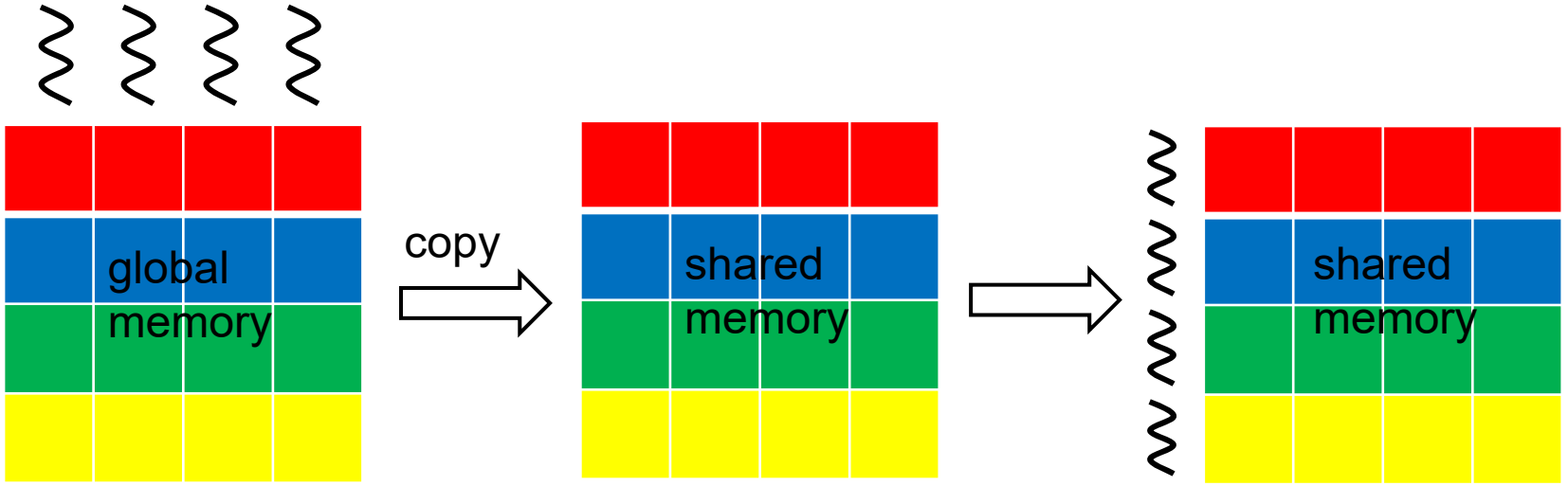
- When iterating by row, we naturally map one thread to each column.
  - Need 4 iterations in total.
- Numbers show locations accessed each iteration.
  - Locations all consecutive. All iterations coalesced.

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!

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.