Today's Agenda: Cuda Programming Model and GPUs

- Recap of last time: thread hierarchy
- SIMD execution & how branching logic is executed
- Cuda memory model
- How Cuda threads are run & GPU scheduling
- Intro to project and some of the kernels you will be implementing:
  - Matrix Multiply
  - 1-d convolution

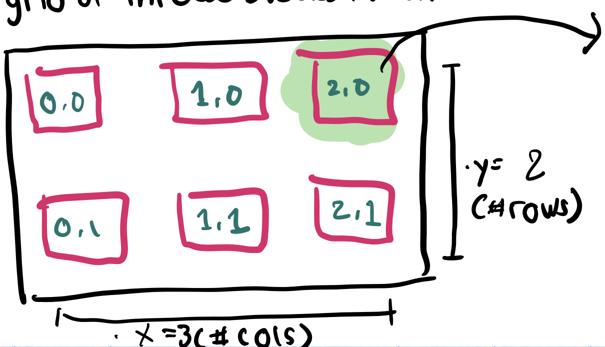
x

Thread Hierarchy Recap

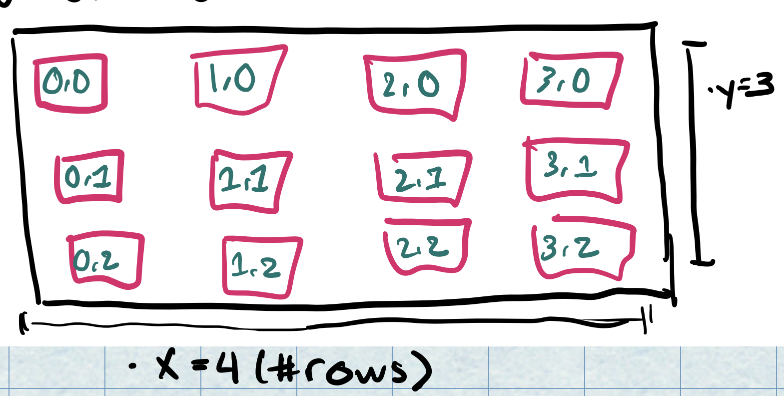
- When you launch a cuda kernel, you specify two template parameters:
  - **numBlocks** - the number of total thread blocks to launch
  - **threadsPerBlock** — within a block, the number of threads
- Each of these are up to 3d (code snippet shows 2d example)

```
const int Nx = 12;
const int Ny = 6;

// dim variable can be up to 3d, has .x, .y, and .z attributes

// shape inside of a thread block
dim3 threadsPerBlock(4, 3);
// shape of entire grid (here -- 3 by 2)
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y);

// A,B,C are allocated Nx x Ny float arrays

// this call launches 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A,B,C);

```

grid of threadblocks in ex:



grid of threads inside threadblock

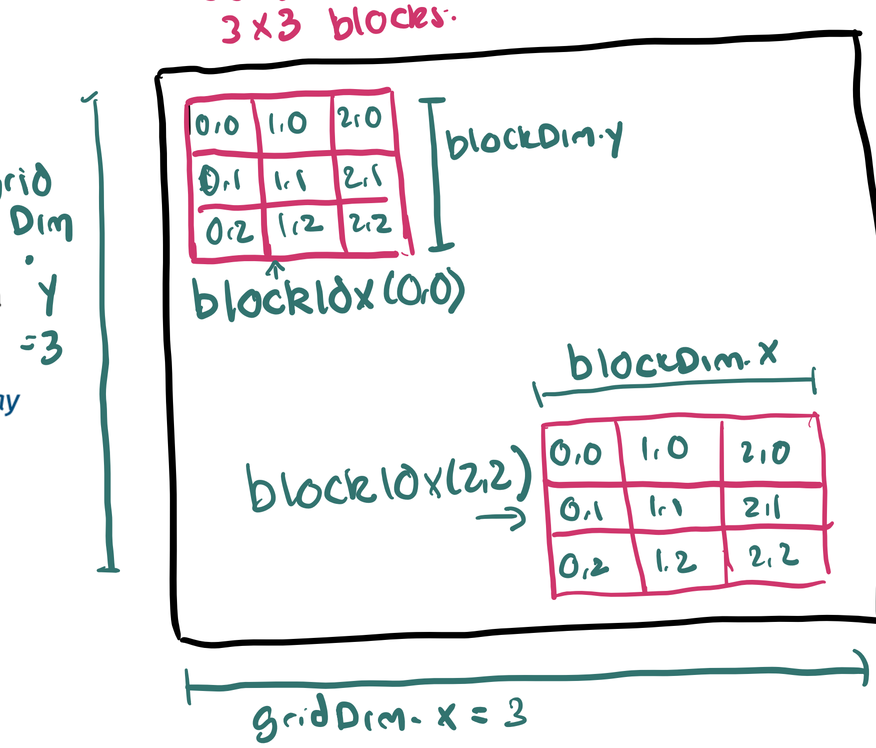


Grid, Block, Thread

Consider 3x3 cuda grid w/ 3x3 blocks.

- **`gridDim`** - dimensions of the grid
- **`blockIdx`** — block index within the grid
  ↳ 2-d vec
- **`blockDim`** — dimensions of a block
- **`threadIdx`** — thread index within a block
- Note that visually, row dimension *may either be x or y* and correspondingly col dimension *may either be y or x*

– this choice has perf implications (see HW)



Launching Cuda Threads & Clear Separation of Host and Device Code

Serial execution: host code runs as a part of a normal C/C++ application on a CPU

```
const int Nx = 12;
const int Ny = 6;

// shape inside thread block
dim3 threadsPerBlock(4, 3);
// shape of entire grid -- here 3x2
dim3 numBlocksPerThread(
        Nx/threadsPerBlock.x,
        Ny/threadsPerBlock.y
);

// assume A,B,C are allocated Nx x Ny float arrays
// we will go over allocation later

// this call launches 72 cuda threads
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks,ThreadsPerBlock(A,B,C)
```

host (serial execution)

Bulk launch of many CUDA threads: "launch a grid of CUDA thread blocks"; call returns when all threads are terminated

```
// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
    // not showing code
    int i = /*TODO*/;
    int j = /*TODO*/;

    // kernel here performs addition on one output cell of matrix
    C[j][i] = /*TODO*/;
}
```

device! (SIMD execution)

"global" denotes a CUDA kernel on GPU

*each thread can compute its overall grid threadId from:
threadIdx
blockIdx

Kernel executed in parallel on multiple CUDA cores

# of kernel invocations is NOT determined by the size of data, but explicitly declared by programmer

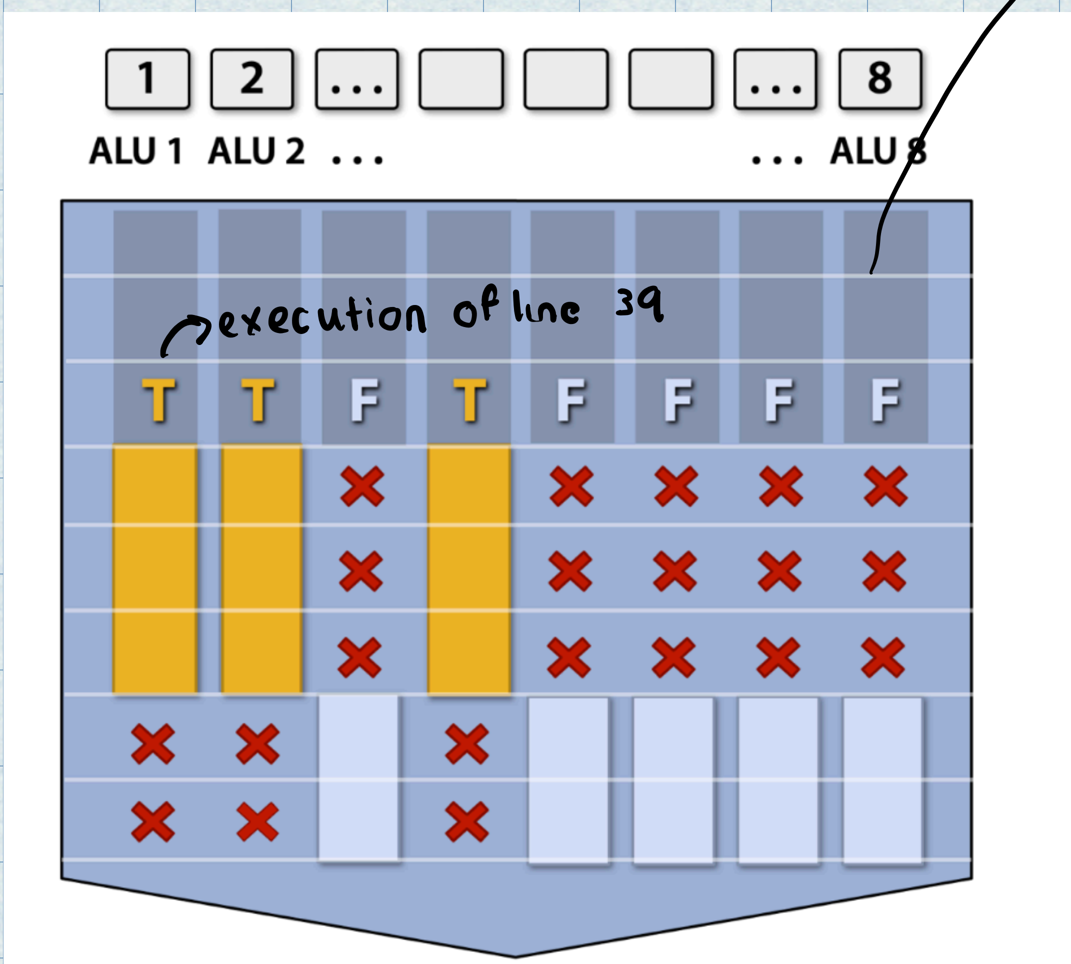
How does conditional execution work?

```
34 // conditional kernel definition
35 __global__ void f(float A[N])
36 {
37     int i = /*TODO*/;
38     float x = A[i];
39     if (x > 0) {
40         x = 2.0f * x;
41     } else {
42         x = exp(x, 5.0f);
43     }
44
45     A[i] = x;
46 }
```

remember: SIMD = single instruction multiple data → inside a warp, GPU can use multiple ALUs in parallel to execute SAME instruction. (example below has 8 ALUs)

Credits: https://mlsyscourse.org/slides/06-CUDA-programming.pdf

IDEA: STILL execute SIMD, but mask out /discard output of some ALUs)



after and b4, can continue at full perf.

*at each time unit, not all ALUs are doing useful work

- worst case on this ex: 1/8 peak performance

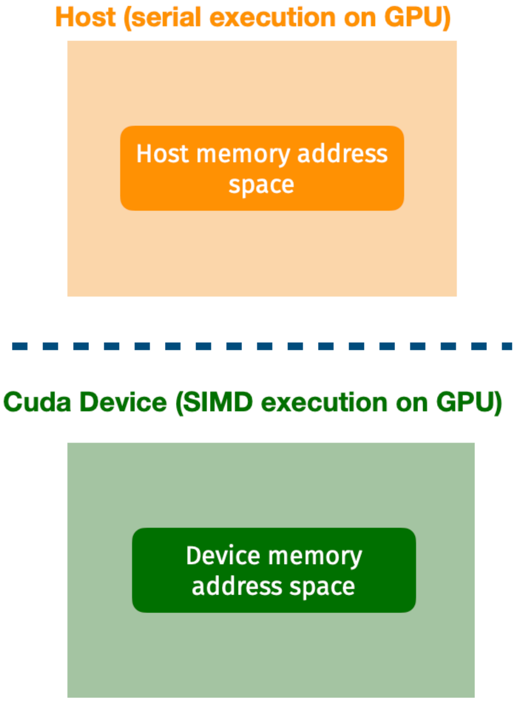
- this is called "coherence execution" vs. "divergent execution"

↓

same ins. sequence on all data elements (MOST efficient use of GPUs)

↓

- lack of conerence execution
- losing out on some of peak perf; should be minimized



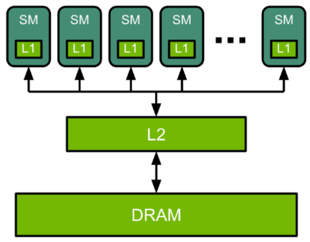
Cuda Memory Model

you can now allocate and copy data directly to device!

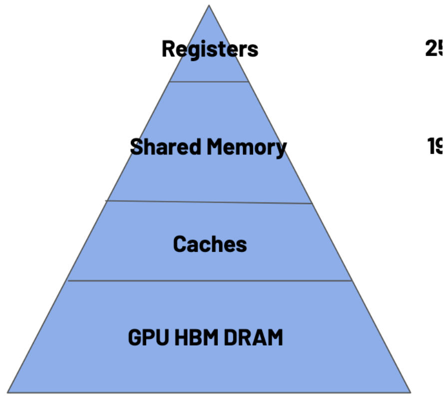
```
float *A = new float[N];

// populate host address space pointer A
for (int i = 0; i < N; i++) {
    A[i] = (float)i;
}

int bytes = sizeof(float) * N;
float* deviceA;
cudaMalloc(&deviceA, bytes);
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);

// note: 'deviceA[i]' is a now an invalid operation in host code!
// only from device code can you manipulate deviceA directly
```

How can kernels take advantage of shared memory?

- Recall three different types of memory available to kernels:
  - **Per-thread private memory** (readable and writable by only this thread)
  - **Per-block shared memory** (readable and writeable by all threads in a block)
  - **Device global memory** (readable/ writable by all threads)

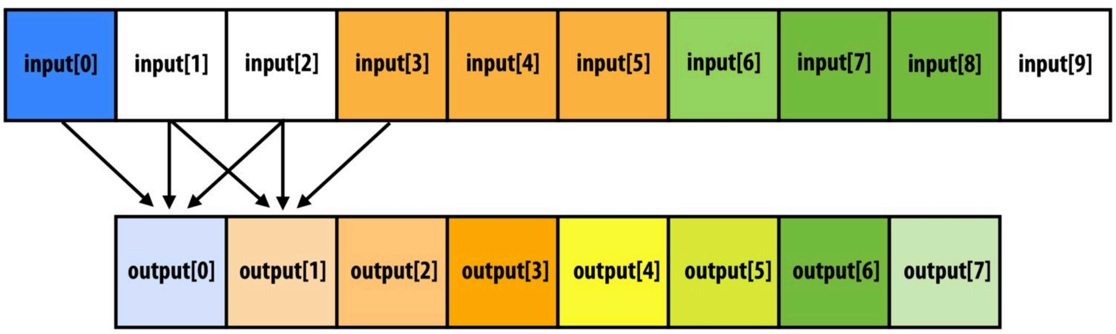


https://docs.nvidia.com/deeplearning/



*like in CPUs, programmers can't directly load/store into caches

Consider 1D Convolution



$$output[i] = (inp[i] + inp[i+1] + inp[i+2]) / 3.F$$

Credits: https://gfxcourses.stanford.edu/cs149/fall23/lecture/gpucuda/slide_35

Consider 1D Convolution

```
#define THREADS_PER_BLOCK 128

__global__ void convolve(int N, float* inp, float* out)
{
    float result = 0.0f;
    // not shown: manipulate result
    output[index] = /*TODO*/
}
```

- result is thread-local
- output (and input, not shown) are from global HBM

Credits: https://gfxcourses.stanford.edu/cs149/fall23/lecture/gpucuda/slide_35

idea:
- declare shared block mem
- fill in, synchronize
- each thread does local work referencing shared mem

Consider 1D Convolution

```
#define THREADS_PER_BLOCK 128

__global__ void convolve(int N, float* inp, float* out)
{
    int index = /*TODO*/;

    __shared__ float support[THREADS_PER_BLOCK + 2]; // per-block
    // not shown: fill in support!

    __syncthreads();

    // now compute output based on data in support
}
```

→ shared per-block memory

↳ threads must work together to fill support!

> barrier for all threads in a block!

Credits: https://gfxcourses.stanford.edu/cs149/fall23/lecture/gpucuda/slide_35

*think: what would "naive" convolve have in terms of # of memory accesses? what about reuse shared memory operation?

Cuda Synchronization Primitives

- `__syncthreads()` - wait for all *threads in a block* to arrive at some point → used to wait until shared mem. is filled in convolve
- Atomic operations
  - `float atomicAdd(float *addr, float amt);`
  - Can happen on *both global and shared memory*
- Host/device synchronization
  - Implicit barrier across all threads at return of kernel
    ↳ to coordinate scheduling of dependent kernels from host

What happens when a Cuda kernel is compiled?

- A compiled CUDA device binary will contain:
  - Program text (actual instructions)
  - Info about required resources:
    - # of threads per block
    - Amount of local data per thread needed to be allocated per thread
    - Amount of shared data needed to be allocated per thread block
  - Important: kernel can be run on *any GPU,* no matter how many S.M.s are there in that version → one GPU might have 64 S.M.s, another might have 128

How does thread-block assignment work?

(8000 thread-blocks)

↳ suppose we have a kernel where each thread block:

→ 128 threads per block

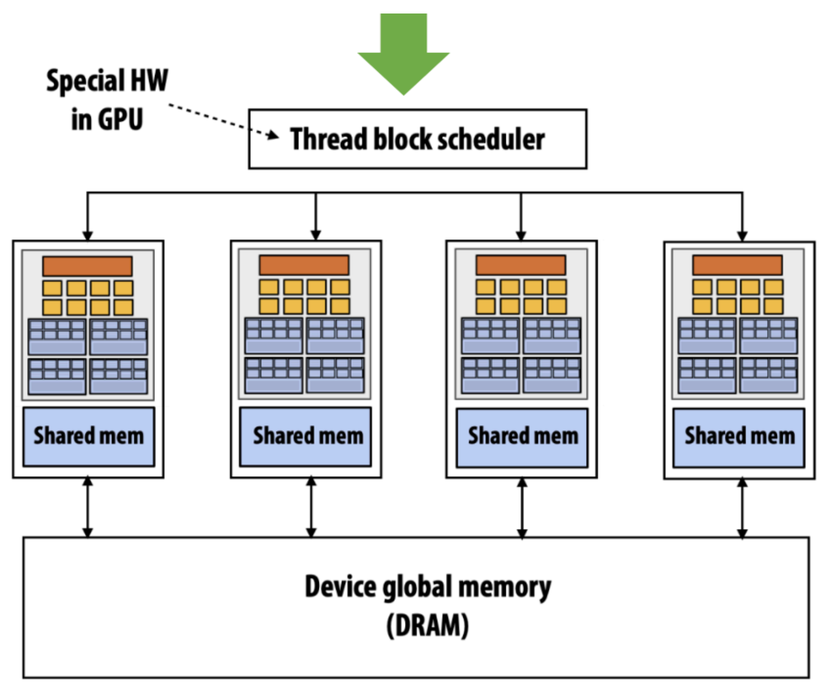
→ 520 bytes shared mem per block

→ (128 · 8) bytes of private thread mem.

↳ Host launches kernel from host:

Cuda thread-block assignment

- ***MAJOR ASSUMPTION***: thread blocks can be launched in any order! (So thread blocks must not have dependencies)
- GPU maps thread blocks to cores using a dynamic scheduling policy respecting each thread block's resource requirements (details not important)

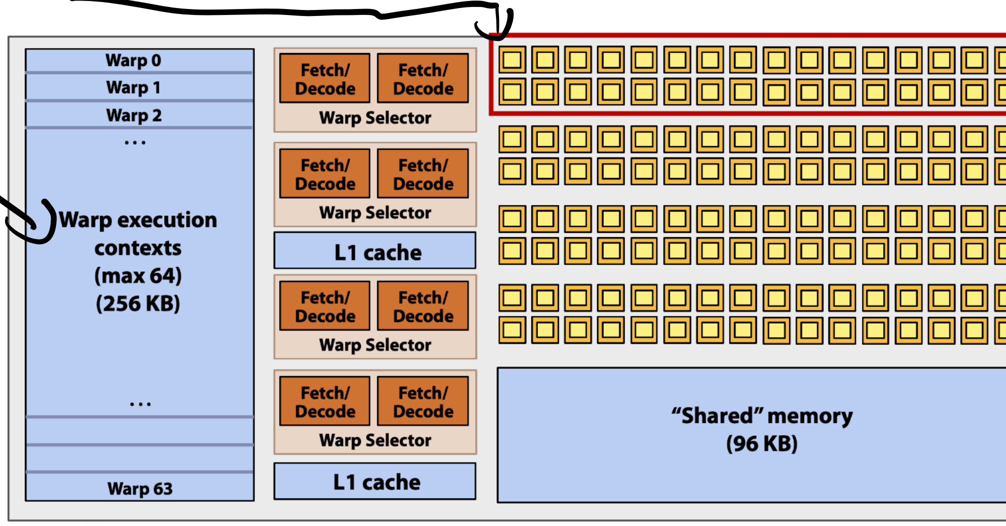


ttps://gfxcourses.stanford.edu/cs149/fall23/lecture/gpucuda/slide_43

Internal Look at Streaming MultiProcessor

- Recall, a streaming multiprocessor must contain *execution contexts* for each thread it is executing
  - Typically, S.M. will have max # concurrent warps
  - For 64 concurrent warps, GPU can maintain 64 * 32 = ~2K total cuda threads
  - Suppose our S.M. has 96 KB of shared memory

SIMD functional units
- (32 ☐s) or 32 "units" share control
- 1 mul / add per clock



https://mlsyscourse.org/slides/06-CUDA-programming.pdf

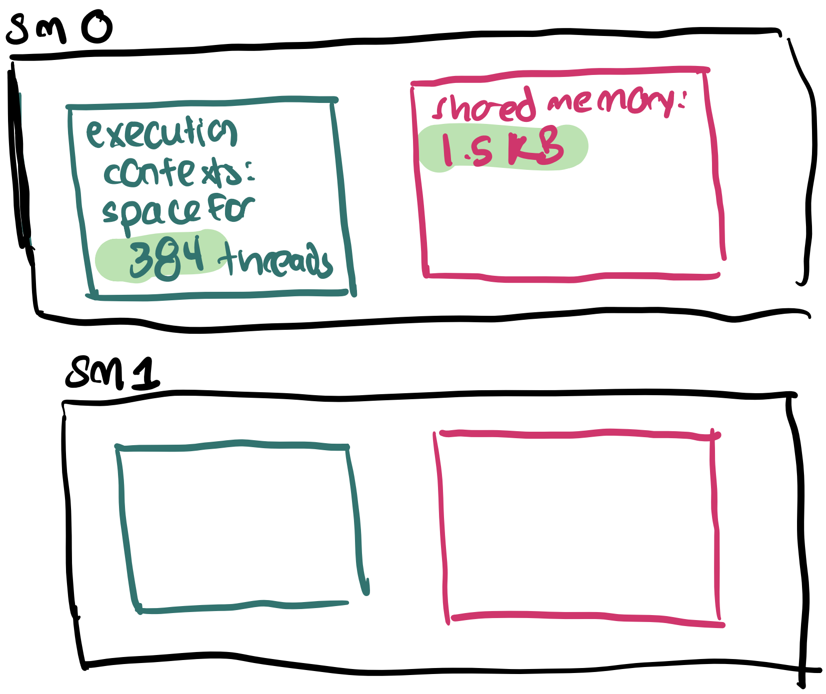
Running a thread block:

→ per clock:
- select up to 4 runnable warps from up to 64 resident warps ("thread-level" parallelism)
- select up to 2 runnable instructions per warp (instruction-level parallelism)
  - ↳ if 2 instructions DO <u>NOT</u> depend on each-other at all

Sequence of steps to run a cuda kernel (e.g., convolve)

- ***Example convolve kernel's requirement***:
  - Each thread block has 128 CUDA threads
  - Each thread block has 512 bytes of shared memory
- Suppose we have 2 streaming multiprocessors available
- How would they execute 1000 thread-blocks?

• max # threads: 32 max warps (32 · 32 max # threads)



→ 1st threadblock (0)

→ gets scheduled on SM 1, execution context threads 0-127, uses 520 bytes of shared memory at 0x0

add padding

→ 2nd threadblock (1)

→ SM 1, execution context threads 128-255, uses 520 bytes of shared memory at 0x520

→ 3rd threadblock: (2)

- put on SM 2!
- would run out of memory on SM1 (3 x 520 > 1.5 KB)

→ ... and so on and so forth

INTRO TO PROJECT 3

- warmup: implement saxpy ($Y = AX + B$)
  - ↳ but also learn how to measure memory BW and compute BW
- PART 1: Implementing and optimizing SGEMM
  0) on CPU
  1) Naive sgemm
  2) global memory coalescing
     - ↳ intuition: when threads WITHIN a WARP make accesses to CONSECUTIVE MEMORY - the processor executing these warps can make a single memory access
  3) shared memory caching
  4) 1D Thread Tiling
  5) 2D Thread Tiling
  6) Generalize to non-divisible matrix sizes
- Part 2: 1D - convolution