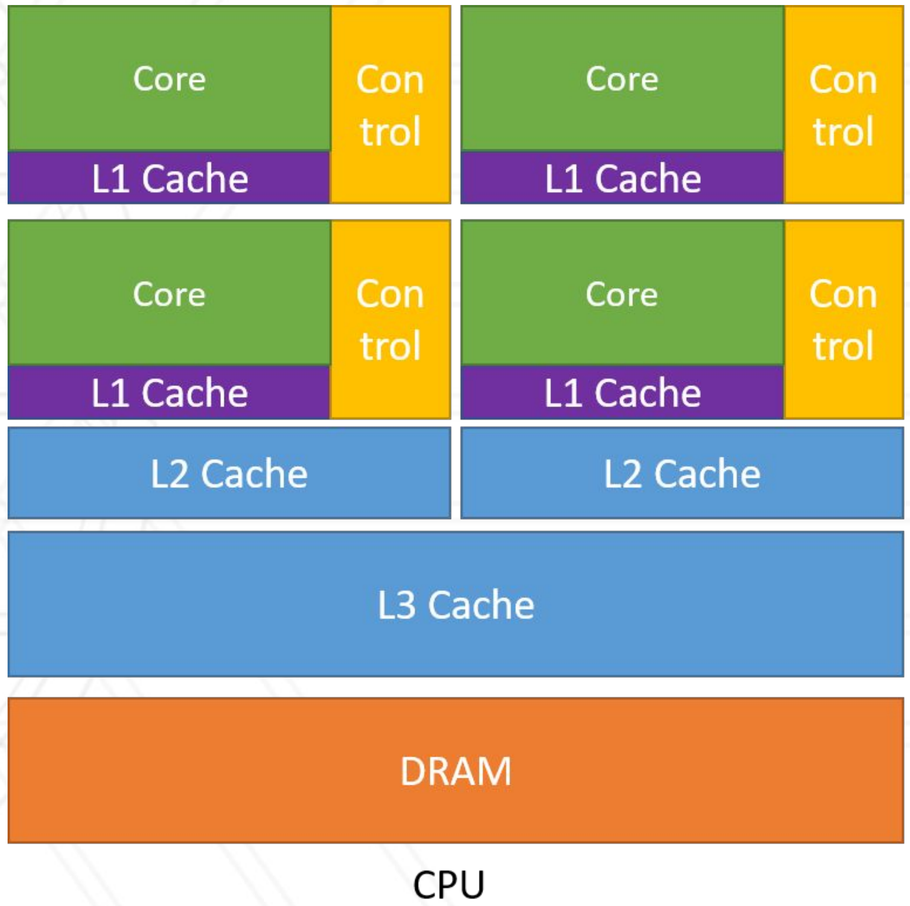
GPGPU Programming

Abhinav Bhatele, Daniel Nichols

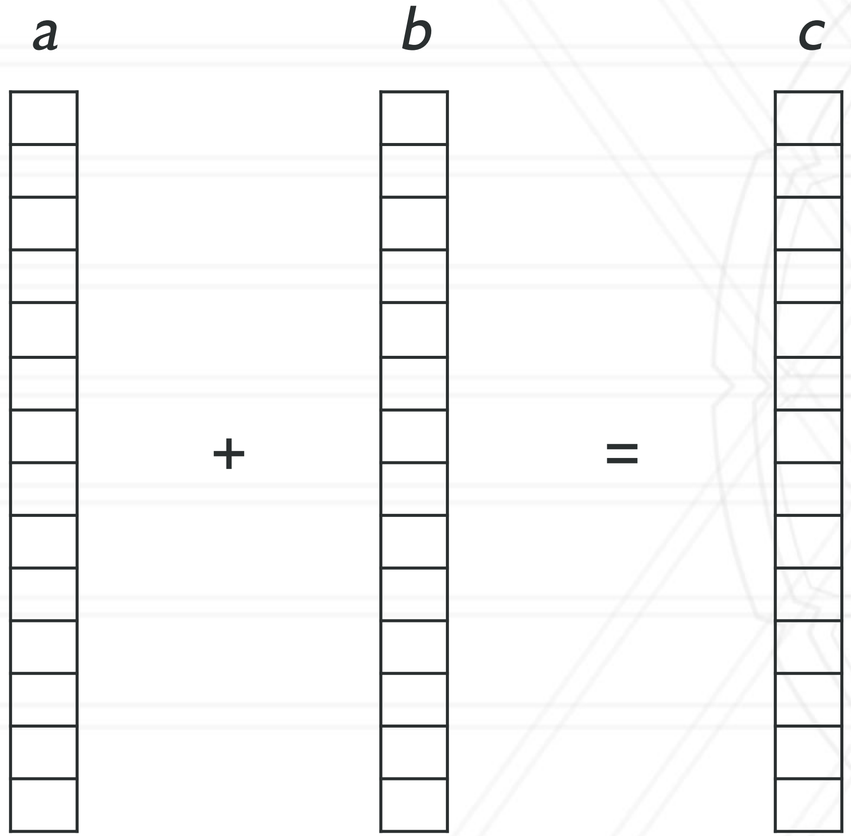
UNIVERSITY OF MARYLAND

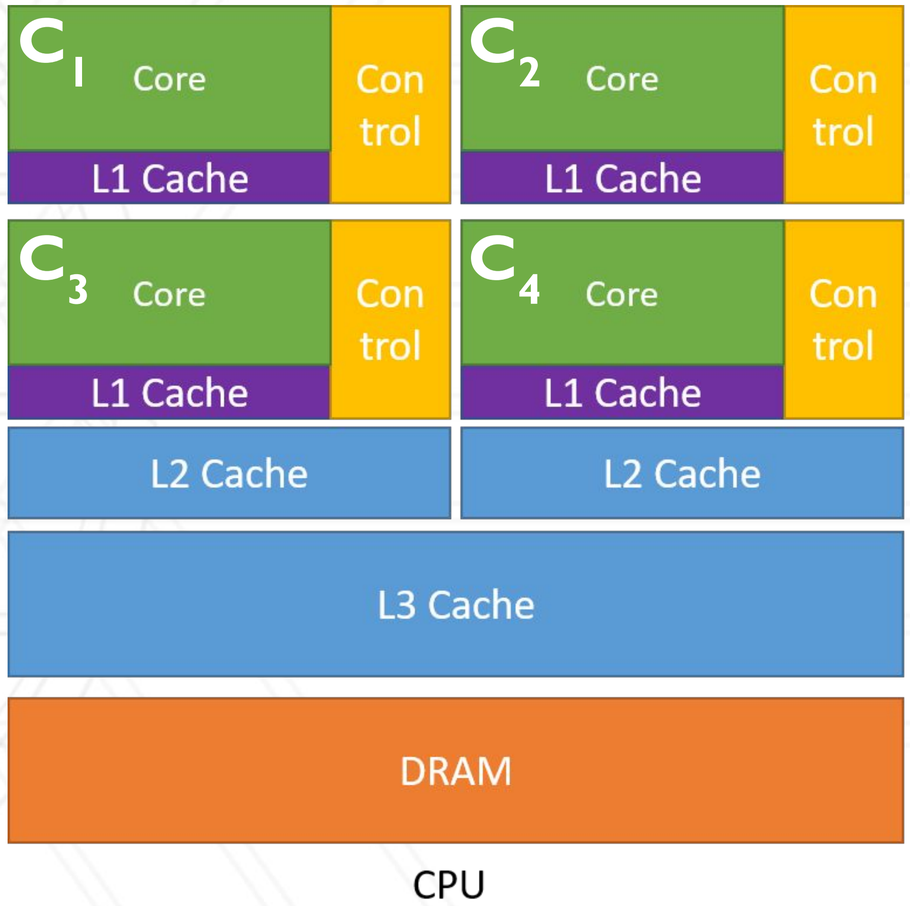
CPUs

- Modern CPUs are designed to reduce latency
  - High clock rate cores
  - Complex instruction sets
- Not great with throughput



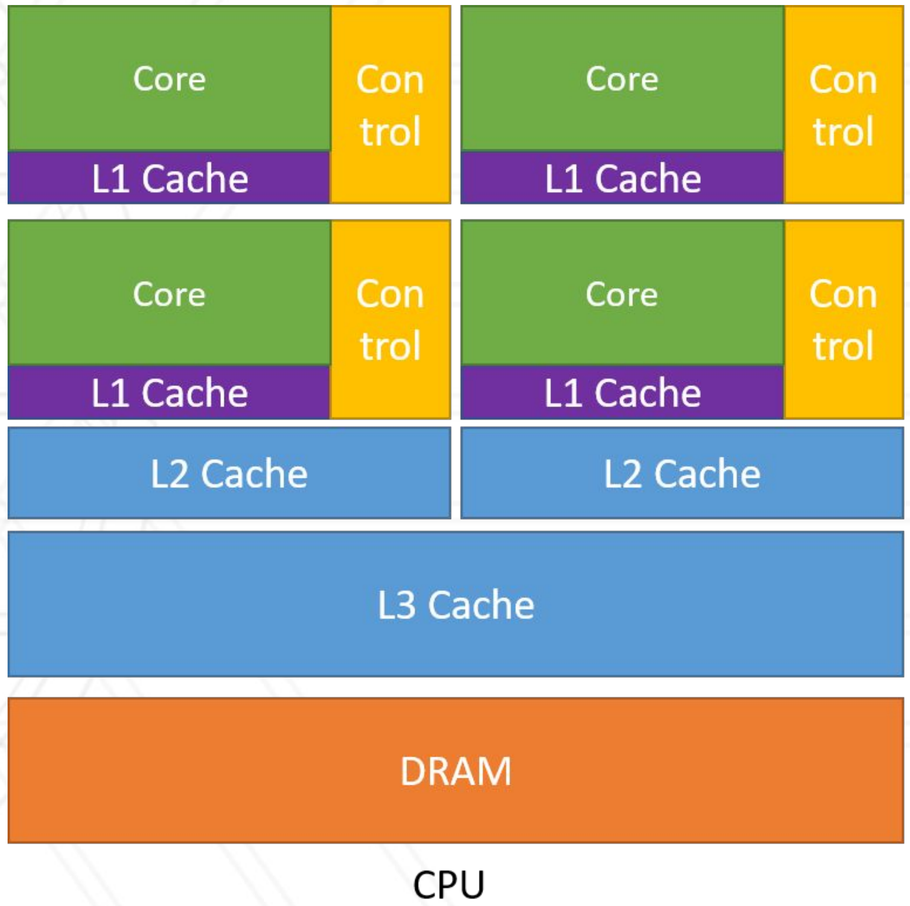
CPU Throughput Example: Vector Addition





CPU Throughput Example: Vector Addition

- Modern CPUs are designed to reduce latency
  - High clock rate cores
  - Complex instruction sets
- Not great with throughput
  - CPUs can process data in parallel by a factor of cores and maybe vector instruction width



GPUs

- CPU
  - few, fast cores
  - more hardware dedicated to control and caching
- GPU
  - many, "slow" cores
  - more hardware dedicated to compute

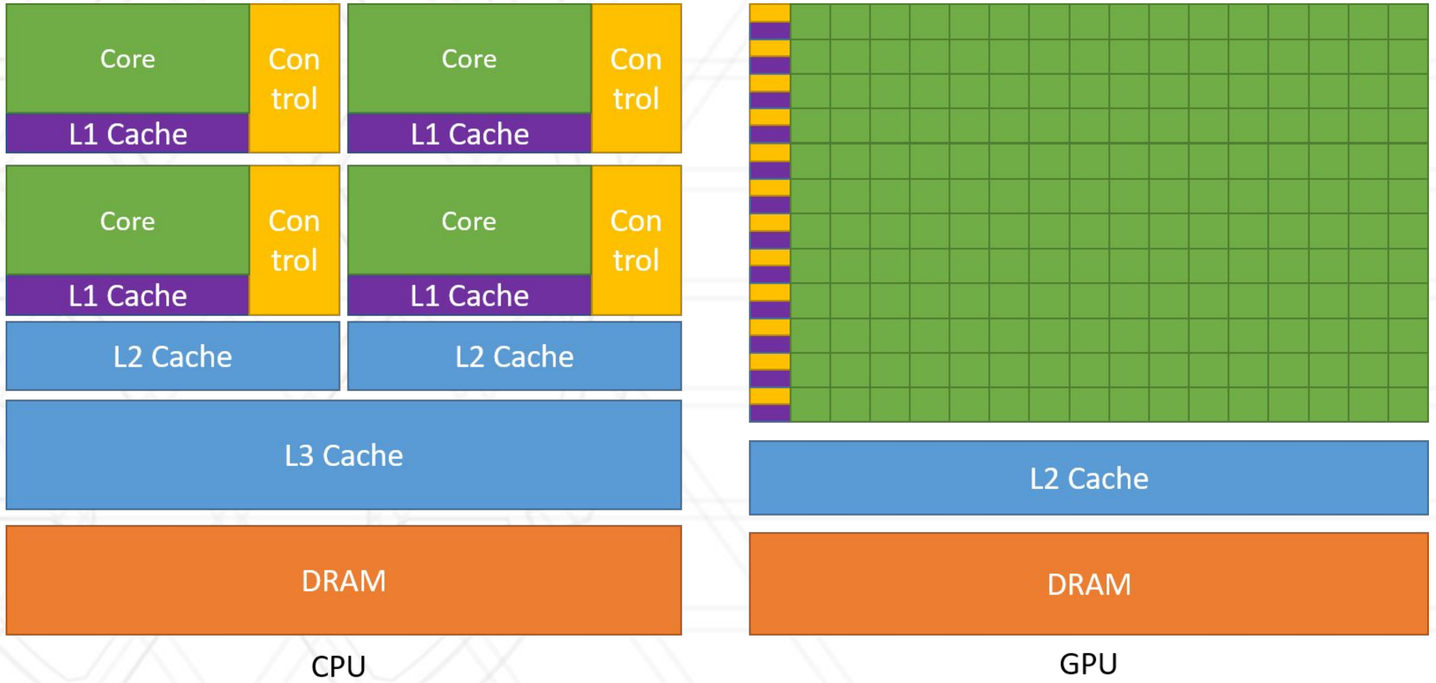
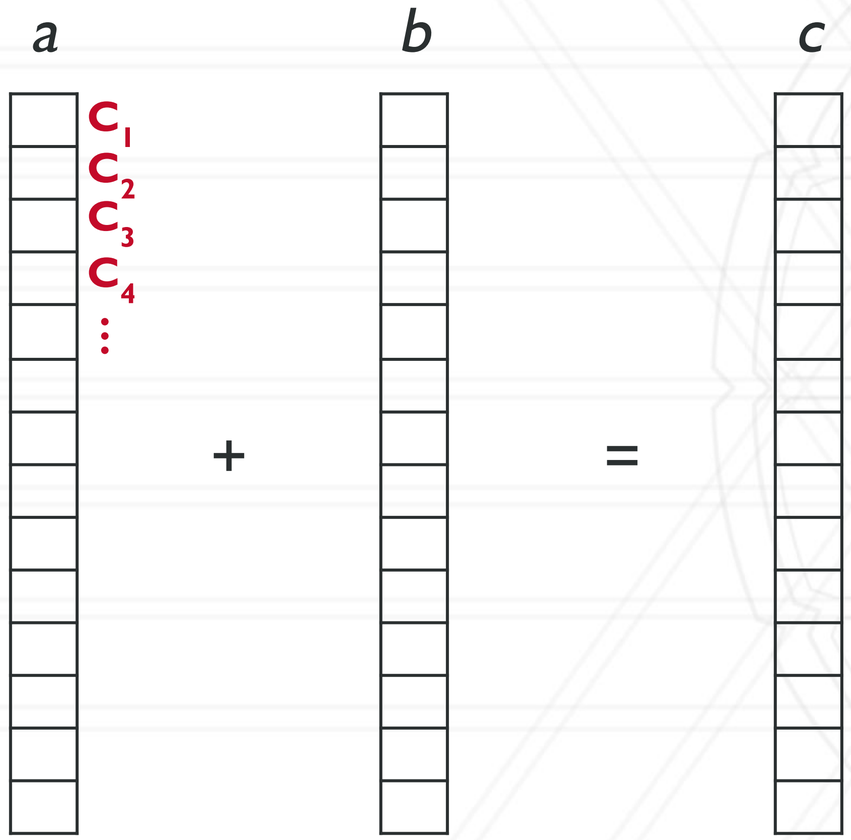
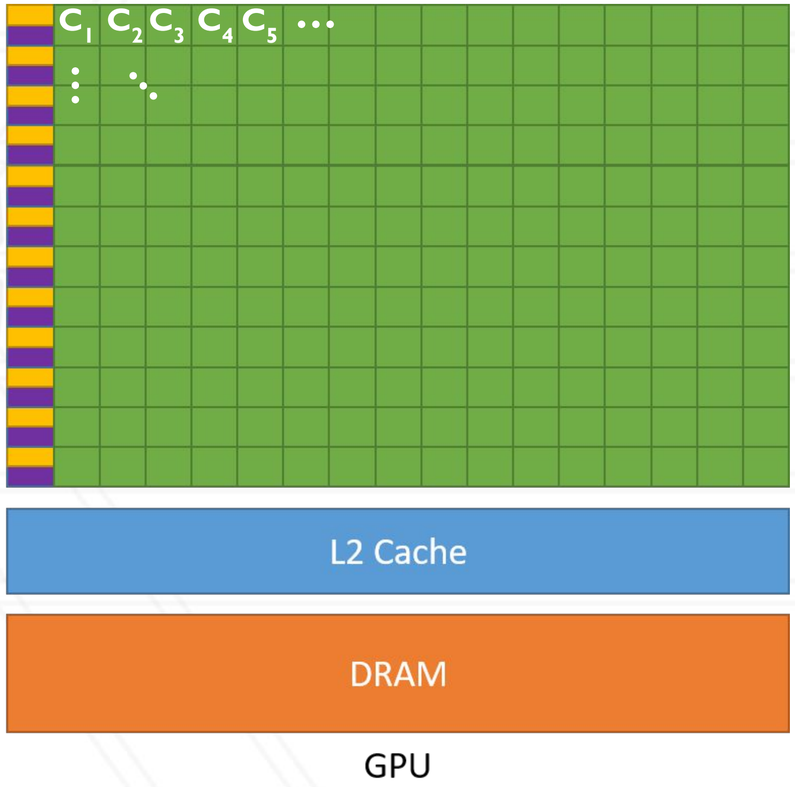


Image: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

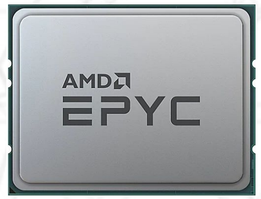
GPUs





CPUs vs GPUs: Some Numbers

- AMD Epyc 7742
  - 64 cores / 128 threads
  - 3.4 GHz boosted



- NVIDIA A100
  - 6912 FP32 cores
  - 3456 FP64 cores
  - 6912 INT32/FP32 cores
  - 1.4 GHz boosted

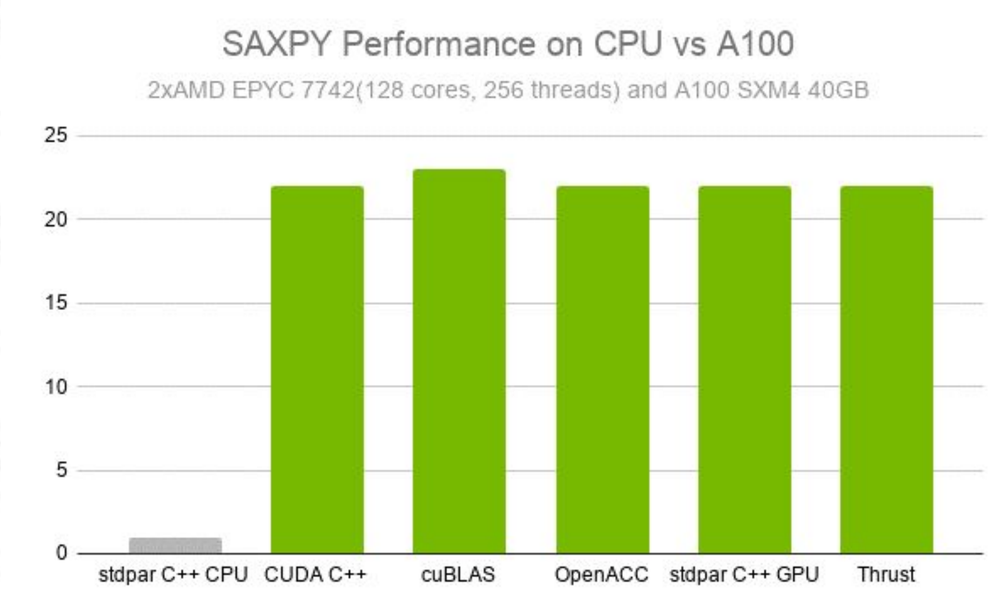
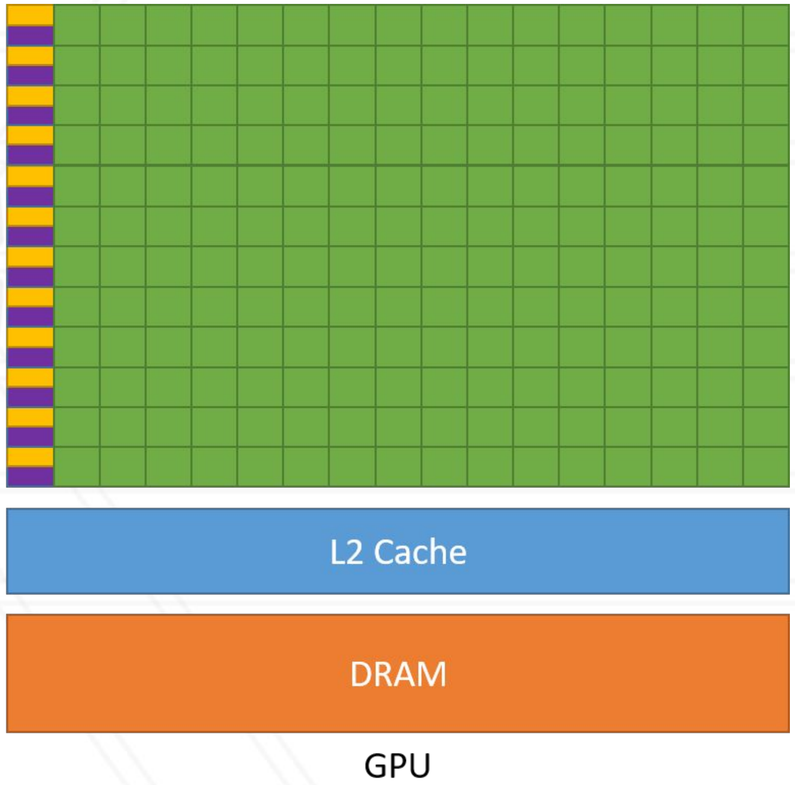


Image from https://developer.nvidia.com/blog/n-ways-to-saxpy-demonstrating-the-breadth-of-gpu-programming-options/

NVIDIA Hardware Terminology

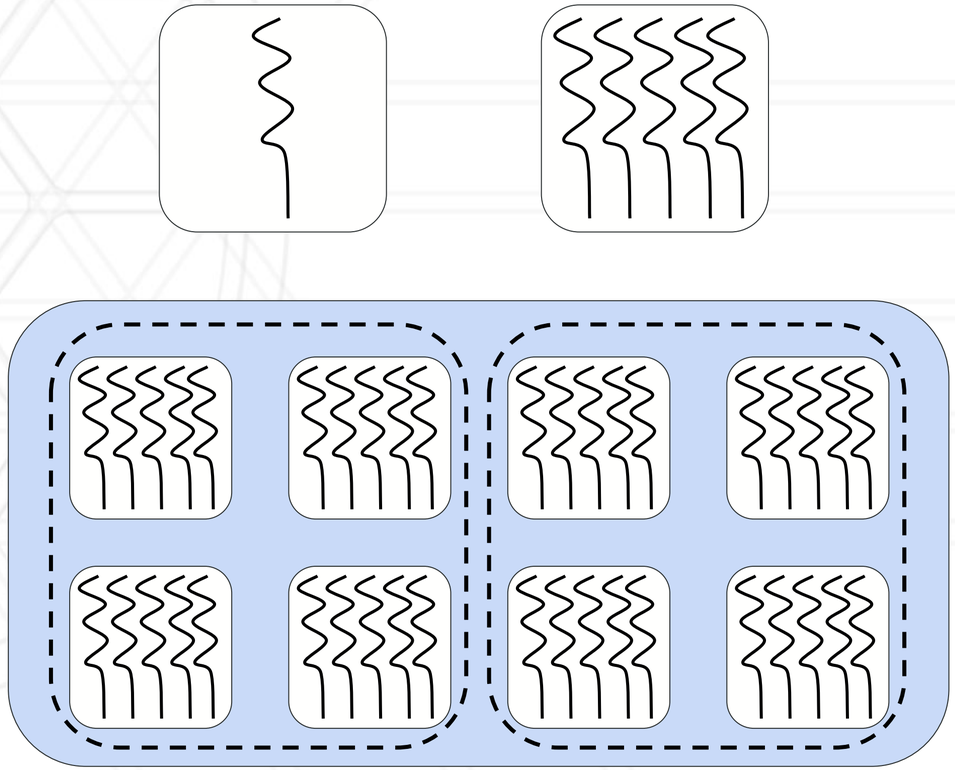
- CUDA Core
  - Single sequential execution unit
- Streaming Multiprocessor (SM)
  - Collection of CUDA cores
  - Shared L1 cache
  - Multiple “warp” schedulers per SM
- CUDA Capable Device / GPU
  - A collection of SMs + an L2 cache + DRAM



Example A100 SM

CUDA Software Abstractions

- CUDA
  - Language used to program NVIDIA GPUs
  - Software ecosystem of libraries, runtimes, compilers, drivers
- Thread
  - Sequential execution unit
- Block
  - A collection of concurrent threads
  - <= 1024 threads
- Block Cluster
  - H100 and later only
  - Groups of blocks within the grid
- Grid
  - A collection of blocks



Software to Hardware Mapping

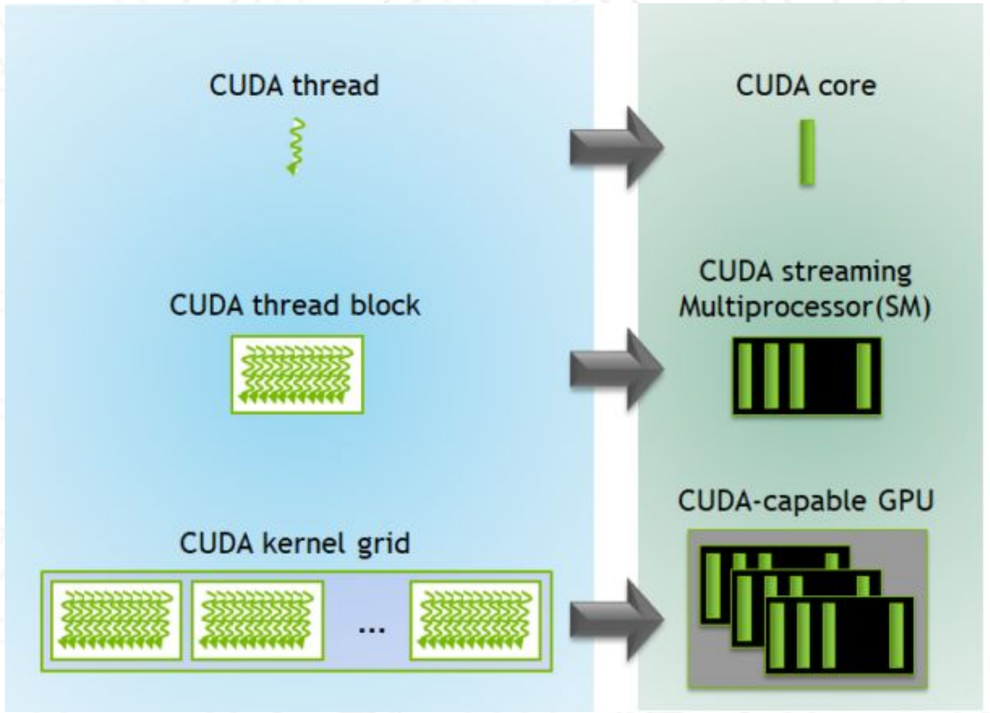


Image: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Anatomy of a CUDA Kernel

```
__global__ void saxpy(float *x, float *y, float alpha) {
    int i = threadIdx.x;
    y[i] = alpha*x[i] + y[i];
}

int main() {
    ...
    saxpy<<<1, N>>>(x, y, alpha);
    ...
}
```

__global__ denotes a *kernel*.
Called from CPU and run on GPU.

Anatomy of a CUDA Kernel

```
__global__ void saxpy(float *x, float *y, float alpha) {
    int i = threadIdx.x;
    y[i] = alpha*x[i] + y[i];
}

int main() {
    ...
    saxpy<<<1, N>>>(x, y, alpha);
    ...
}
```

Execution Configuration Syntax:
<<< # of blocks, threads per block >>>

threadIdx is the thread index 0...N

Kernels Running on the Device Example

Compute saxpy with N = 4

```
saxpy<<<1, 4>>>(x, y, alpha);
```

Call the kernel with 1 block and 4 threads per block.

Block 0

Thread 0

```
int i = threadIdx.x;
y[i] = alpha*x[i] + y[i];
```

Thread 1

```
int i = threadIdx.x;
y[i] = alpha*x[i] + y[i];
```

Thread 2

```
int i = threadIdx.x;
y[i] = alpha*x[i] + y[i];
```

Thread 3

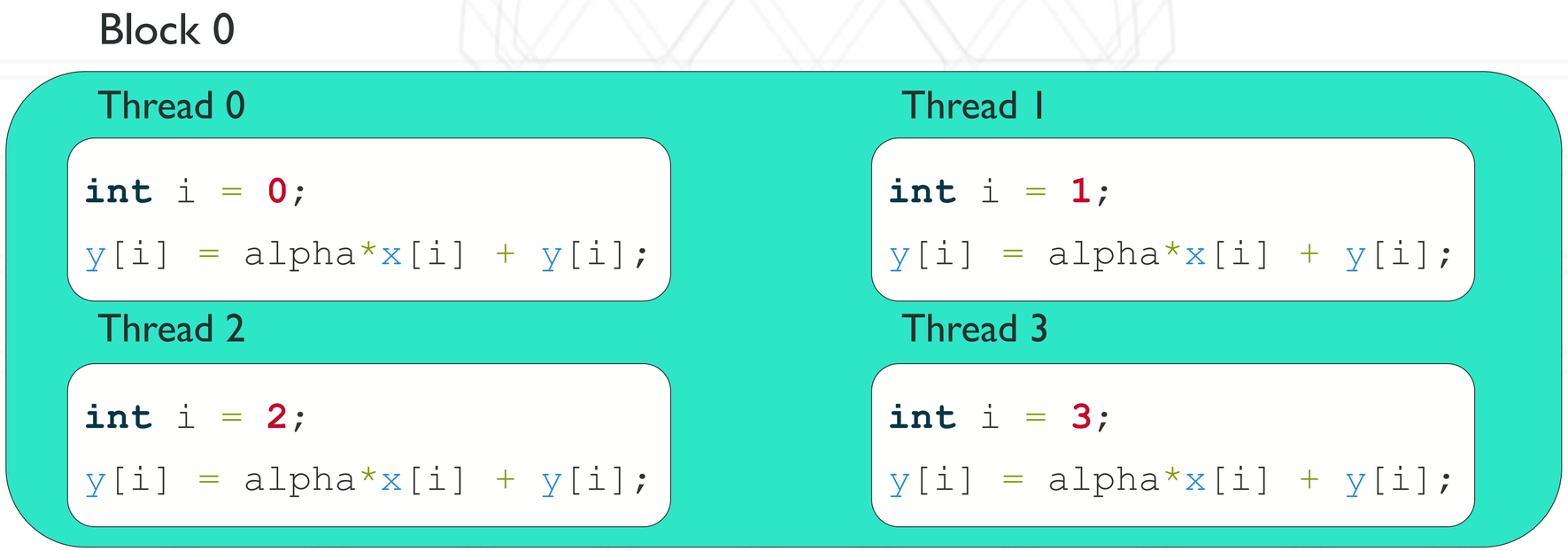
```
int i = threadIdx.x;
y[i] = alpha*x[i] + y[i];
```

Kernels Running on the Device Example

Compute saxpy with N = 4

```
saxpy<<<1, 4>>>(x, y, alpha);
```

Call the kernel with 1 block and 4 threads per block.



Possible Issues?

```
__global__ void saxpy(float *x, float *y, float alpha) {
    int i = threadIdx.x;
    y[i] = alpha*x[i] + y[i];
}

int main() {
    ...
    saxpy<<<1, N>>>(x, y, alpha);
    ...
}
```

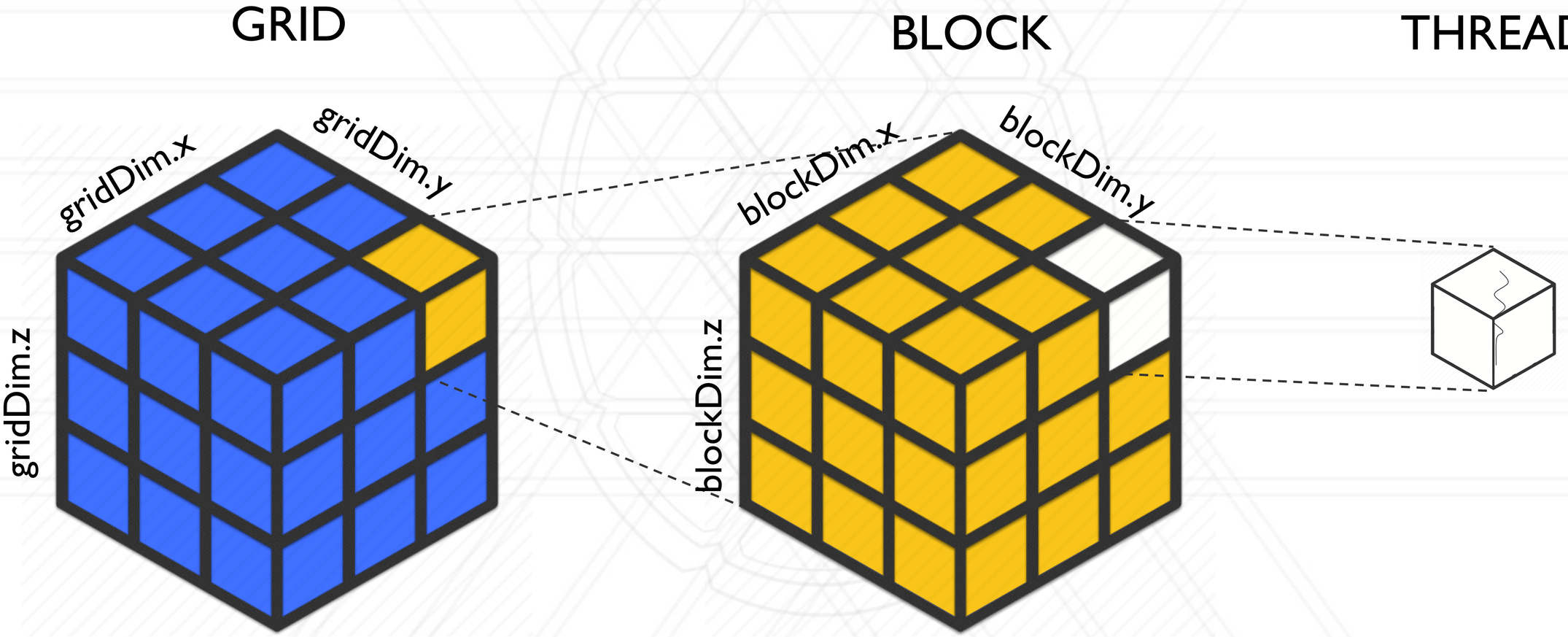
Multiple Blocks

```
__global__ void saxpy(float *x, float *y, float alpha, size_t N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
      y[i] = alpha*x[i] + y[i];
}

int main() {
    ...
    saxpy<<<⌈N/block_size⌉, block_size>>>(x, y, alpha);
    ...
}
```

Make sure we have enough threads for each element

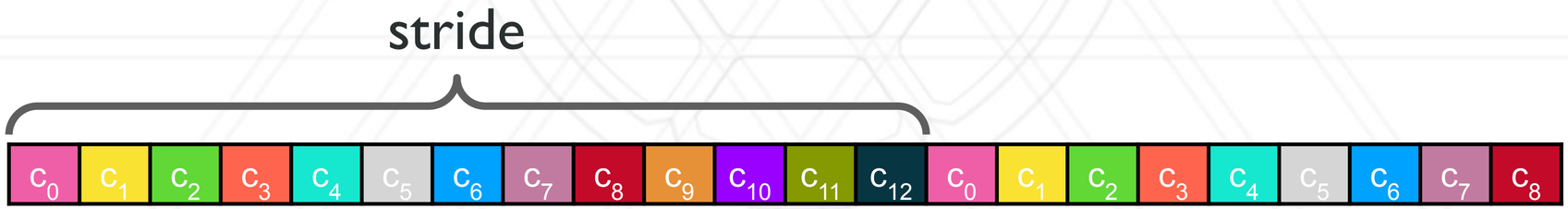
Grid and Block Dimensions



Striding

```
__global__ void saxpy(float *x, float *y, float alpha, int N) {
    int i0 = blockDim.x * blockIdx.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = i0; i < N; i += stride)
        y[i] = alpha*x[i] + y[i];
}
```



Questions?

Matrix Multiply

- Standard matrix multiply
- How can we parallelize on a GPU?

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        for (k=0; k<P; k++)
            C[i][j] += A[i][k]*B[k][j];
```

Matrix Multiply

- All C_{ij} can be computed independently
- 2-D thread decomposition
- Thread (*i*, *j*) can compute C_{ij}
  - Dot product of A row i and B column j

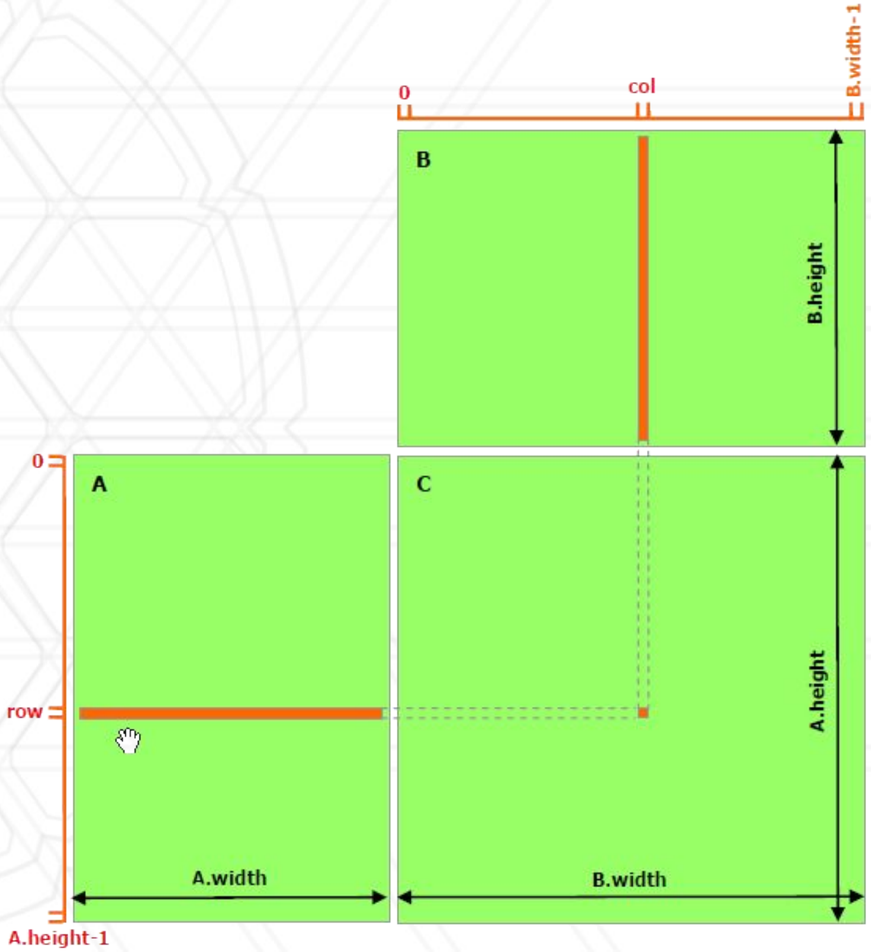


Image: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Issues?

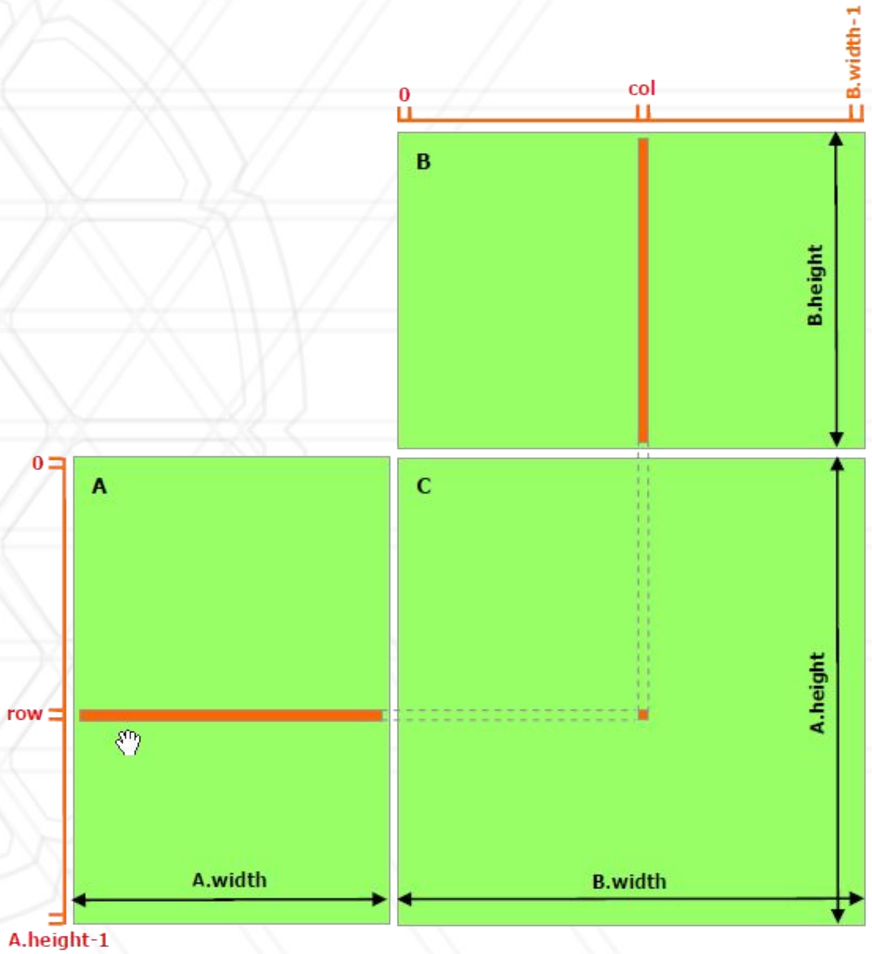
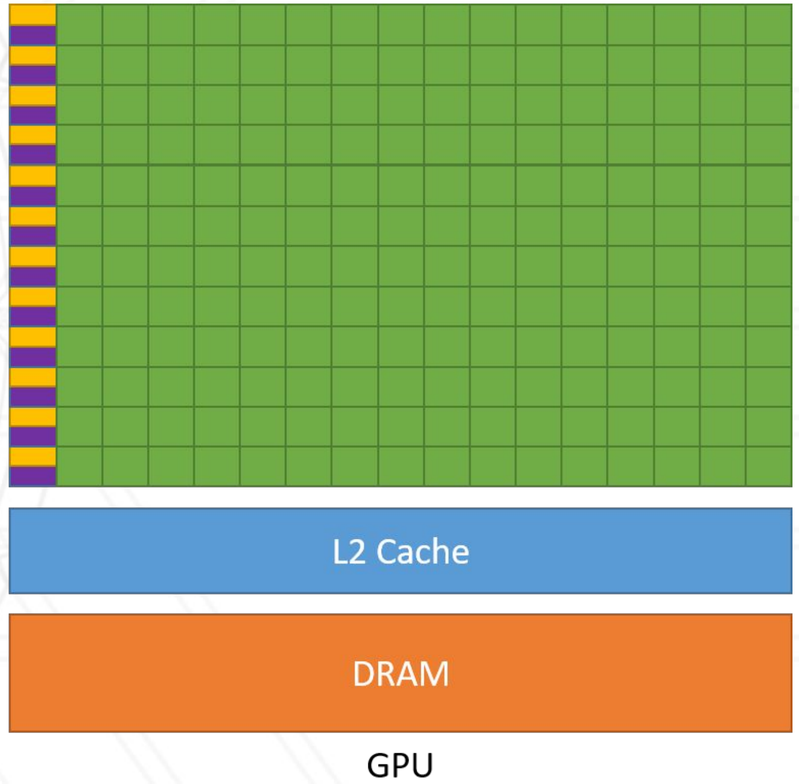


Image: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

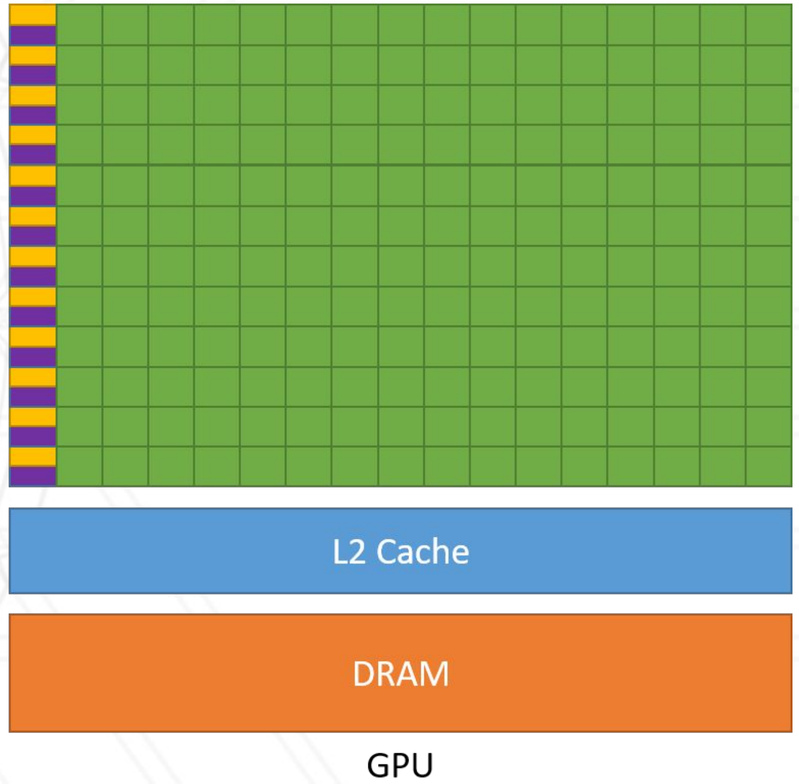
Issues?

- Poor data re-use
  - Every value of A and B is loaded from global memory
  - A is read N times
  - B is read M times
- How can we improve data re-use?



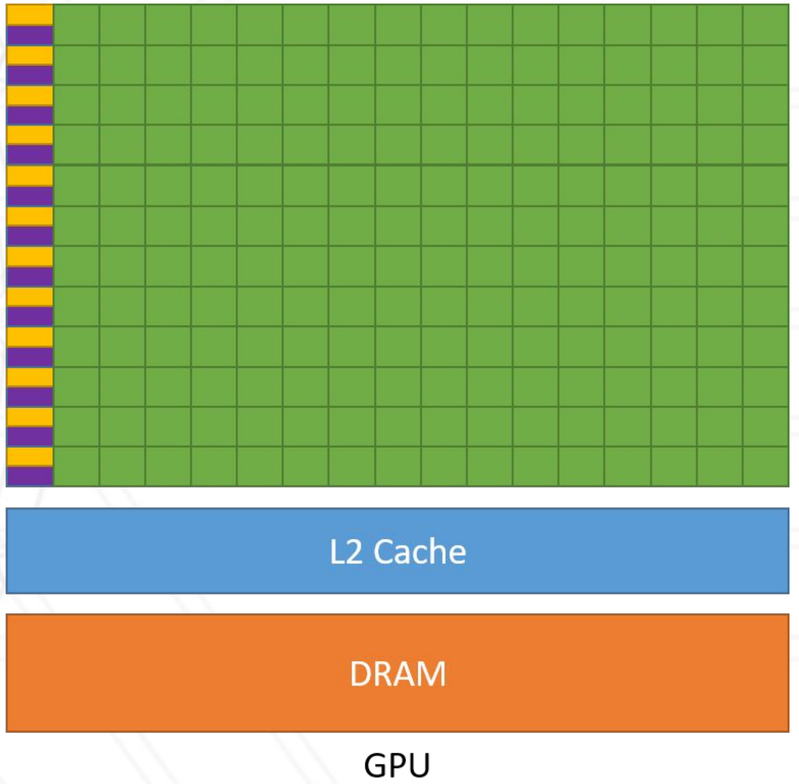
Shared Memory

- Local
  - thread only
- Shared
  - threads in a block
- Distributed Shared
  - blocks in a cluster
  - H100 and later
- Global
  - all threads

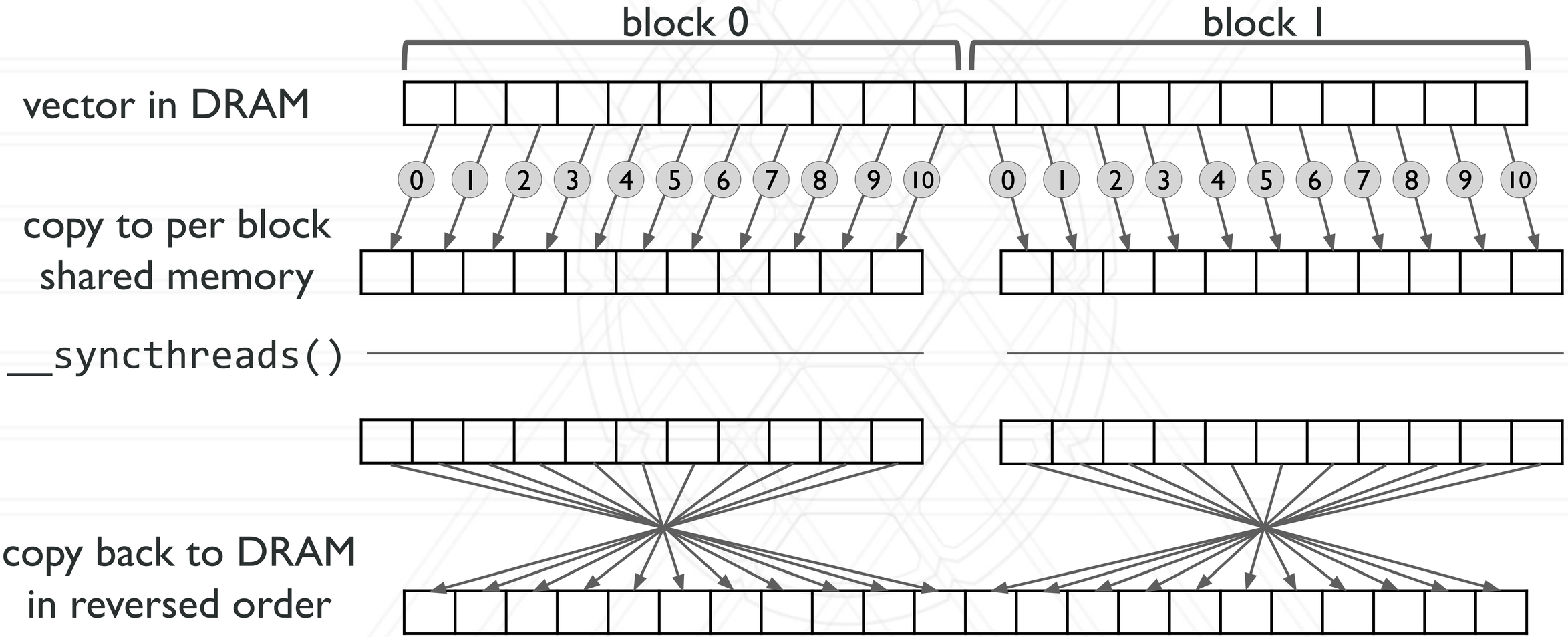


Shared Memory

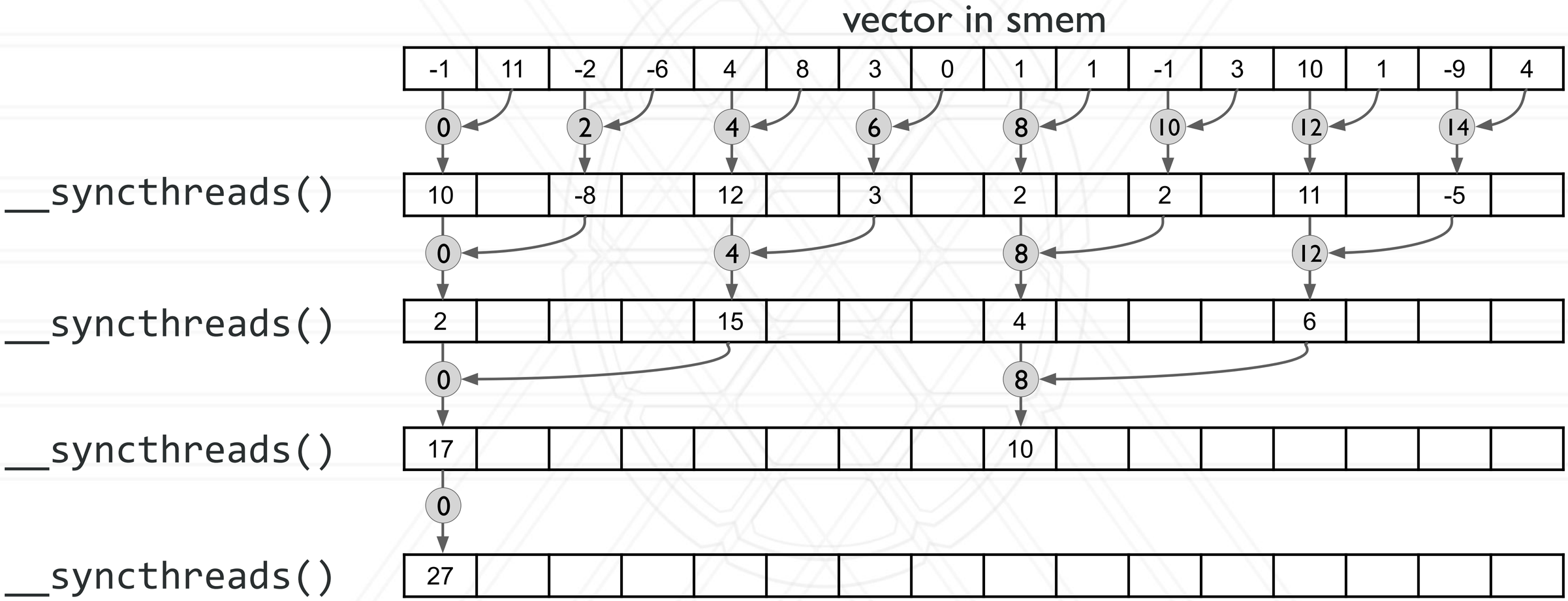
- __shared__
  - denotes static memory shared between threads in a block
- __syncthreads()
  - synchronizes threads in a block



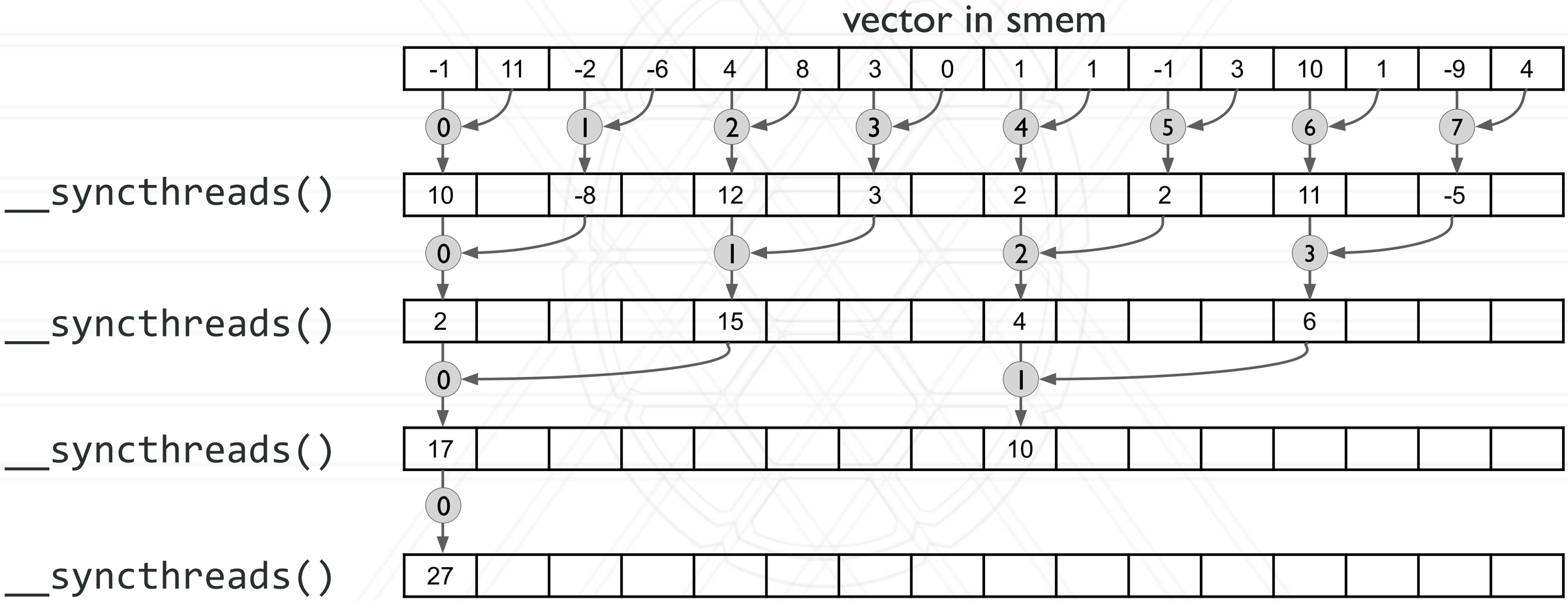
Example: Reversing in a Block



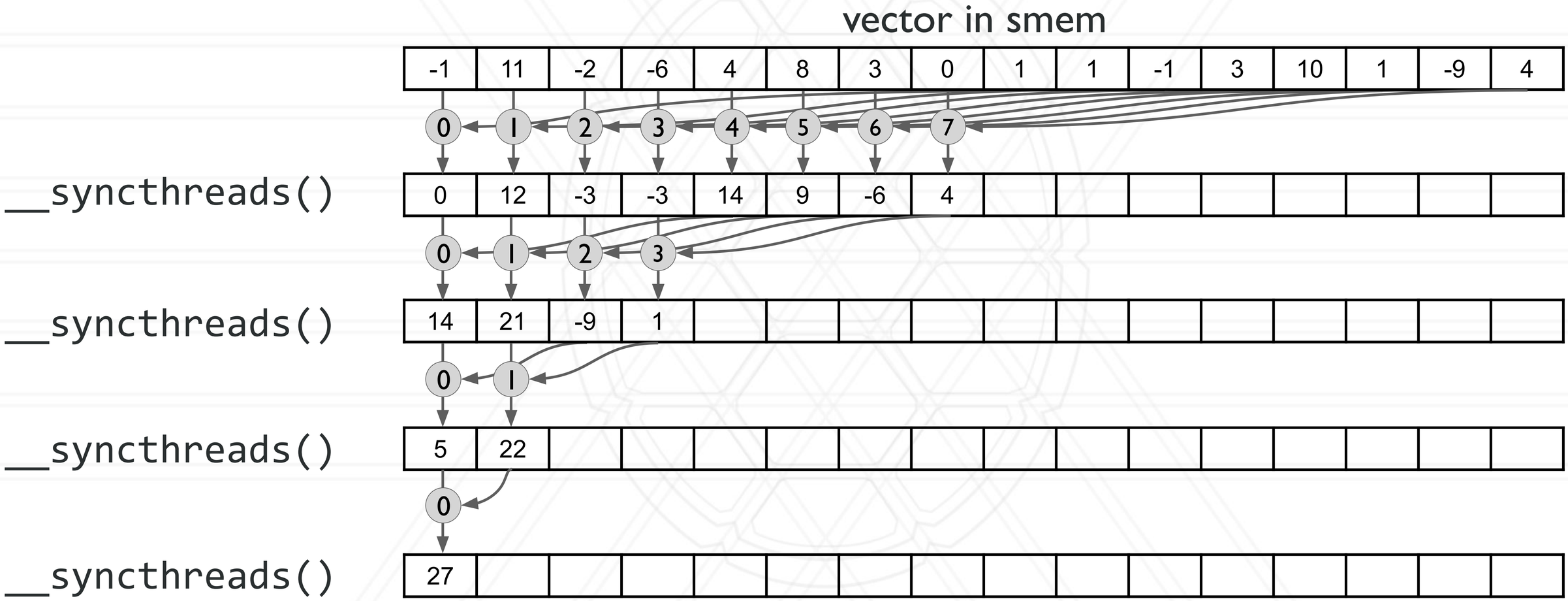
Example: Sum


vector in smem
__syncthreads()
__syncthreads()
__syncthreads()
__syncthreads()
-1 11 -2 -6 4 8 3 0 1 1 -1 3 10 1 -9 4
0 2 4 6 8 10 12 14
10 -8 12 3 2 2 11 -5
0 4 8 12
2 15 4 6
0 8
17 10
0
27

Example: Sum

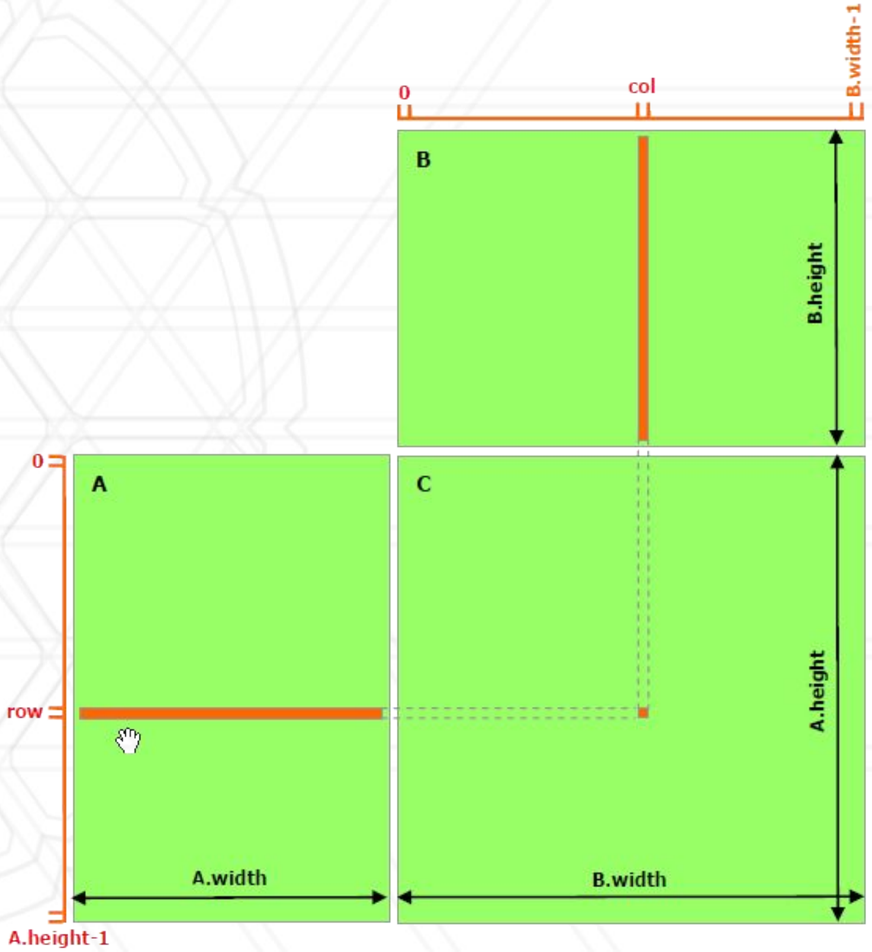

vector in smem
__syncthreads()
__syncthreads()
__syncthreads()
__syncthreads()
-1 11 -2 -6 4 8 3 0 1 1 -1 3 10 1 -9 4
0 1 2 3 4 5 6 7
10 -8 12 3 2 2 11 -5
0 1 2 3
2 15 4 6
0 1
17 10
0
27

Example: Sum


vector in smem
-1 11 -2 -6 4 8 3 0 1 1 -1 3 10 1 -9 4
0 1 2 3 4 5 6 7
__syncthreads()
0 12 -3 -3 14 9 -6 4
0 1 2 3
__syncthreads()
14 21 -9 1
0 1
__syncthreads()
5 22
0
__syncthreads()
27

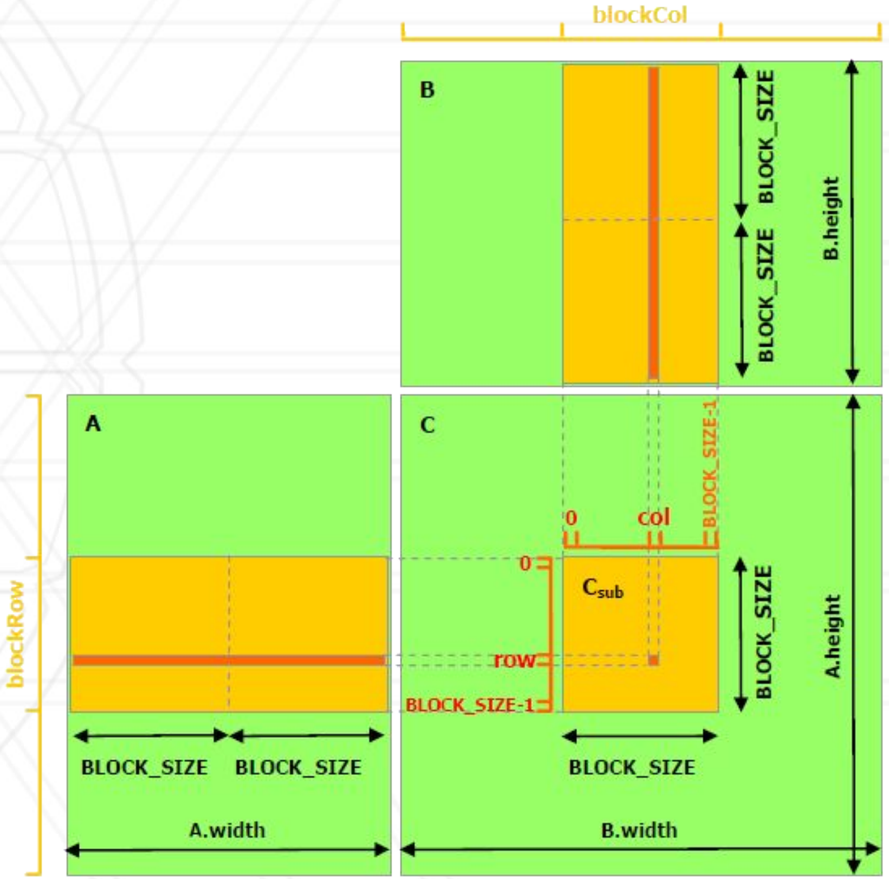
Matrix Multiply with Shared Memory

- How can we speed up matrix multiply with shared memory?



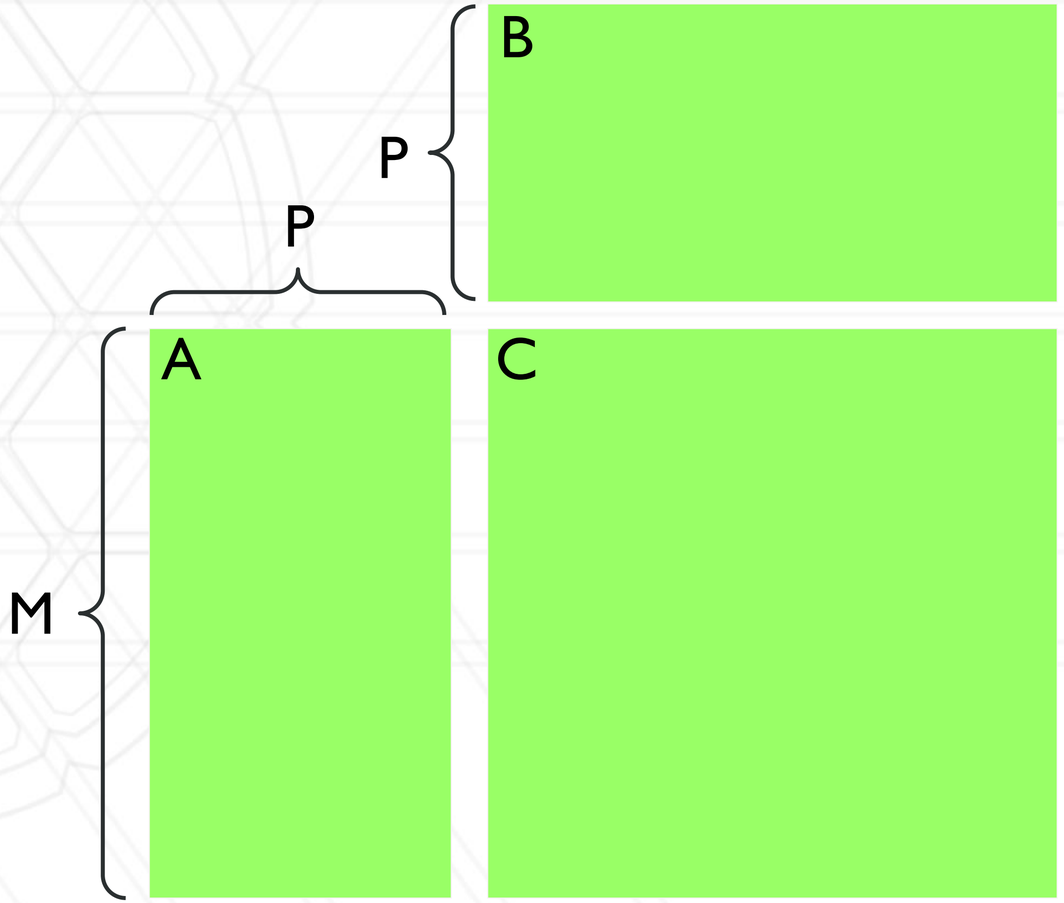
Matrix Multiply with Shared Memory

- Block computation
- Each block computes a submatrix of C
- Load re-used values of A and B into shared memory



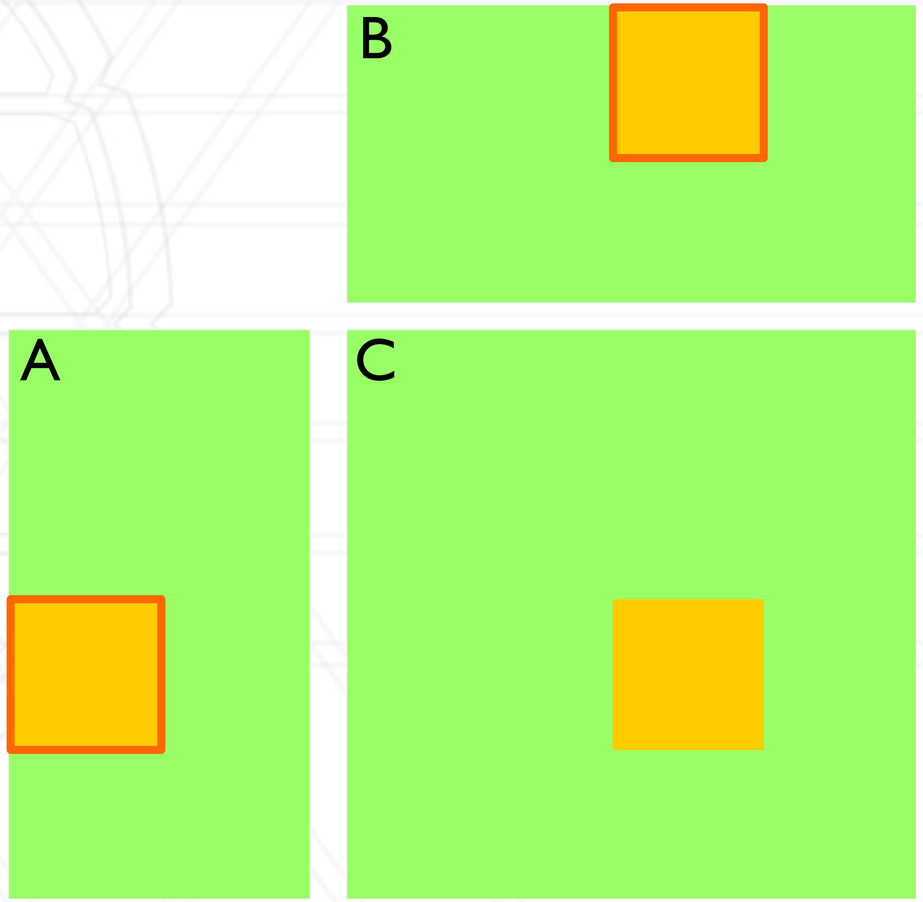
Matrix Multiply with Shared Memory

- Compute C=AB



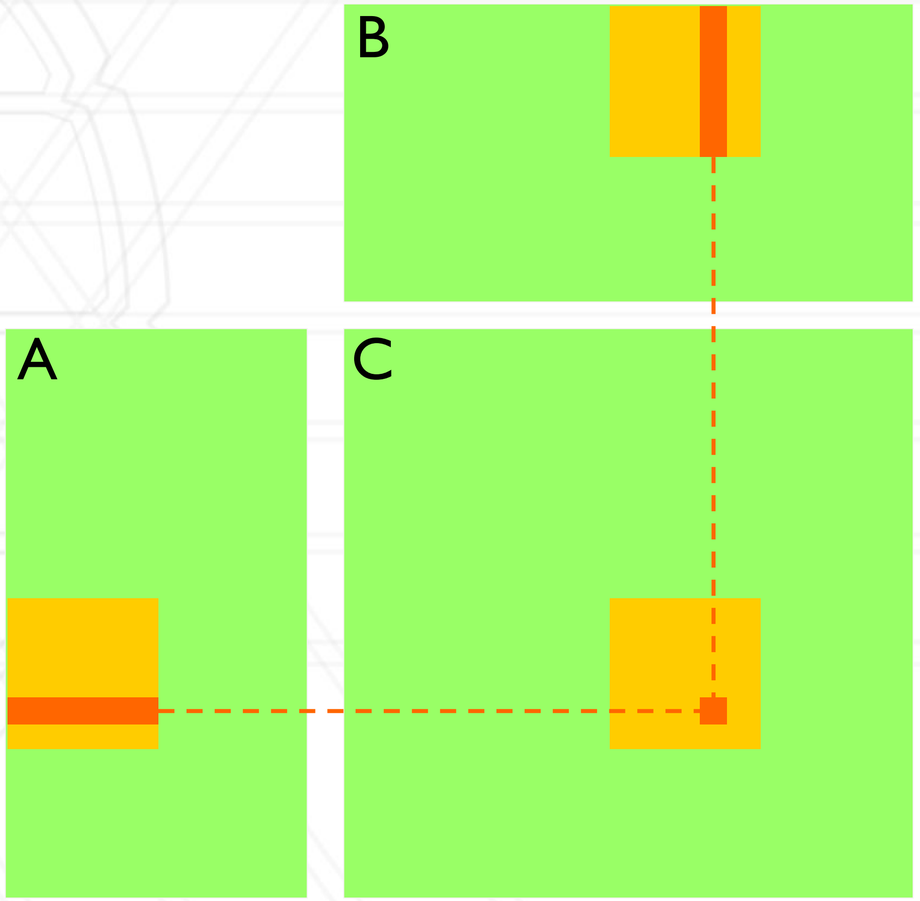
Matrix Multiply with Shared Memory

- Compute C=AB
- Block (i,j) compute submatrix C_{ij}
  - Save A & B submatrices into shared memory



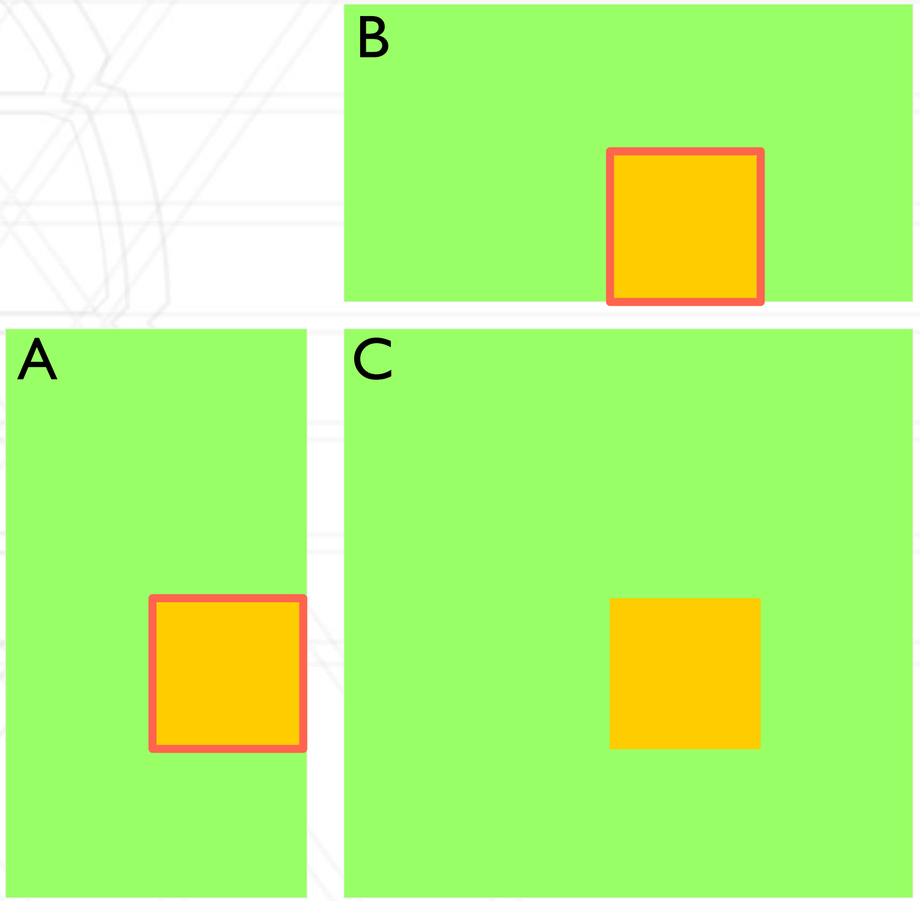
Matrix Multiply with Shared Memory

- Compute C=AB
- Block (i,j) compute submatrix C_{ij}
  - Save A & B submatrices into shared memory
  - Accumulate partial dot product into C



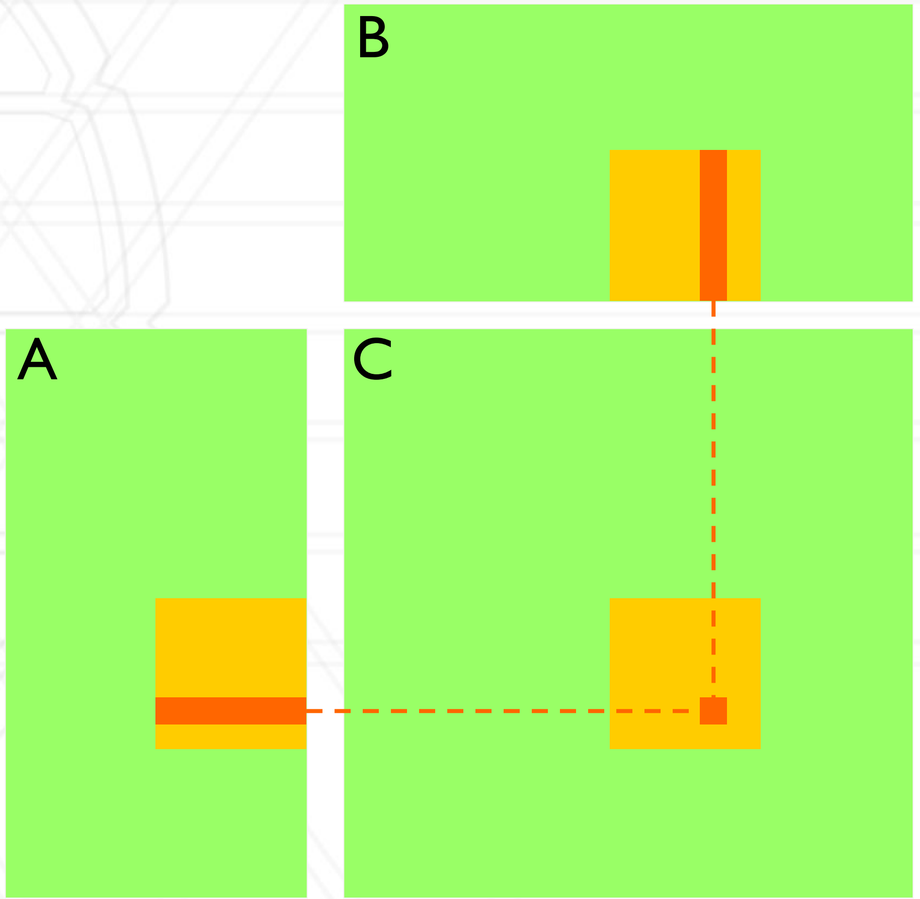
Matrix Multiply with Shared Memory

- Compute C=AB
- Block (i,j) compute submatrix C_{ij}
  - Save A & B submatrices into shared memory
  - Accumulate partial dot product into C



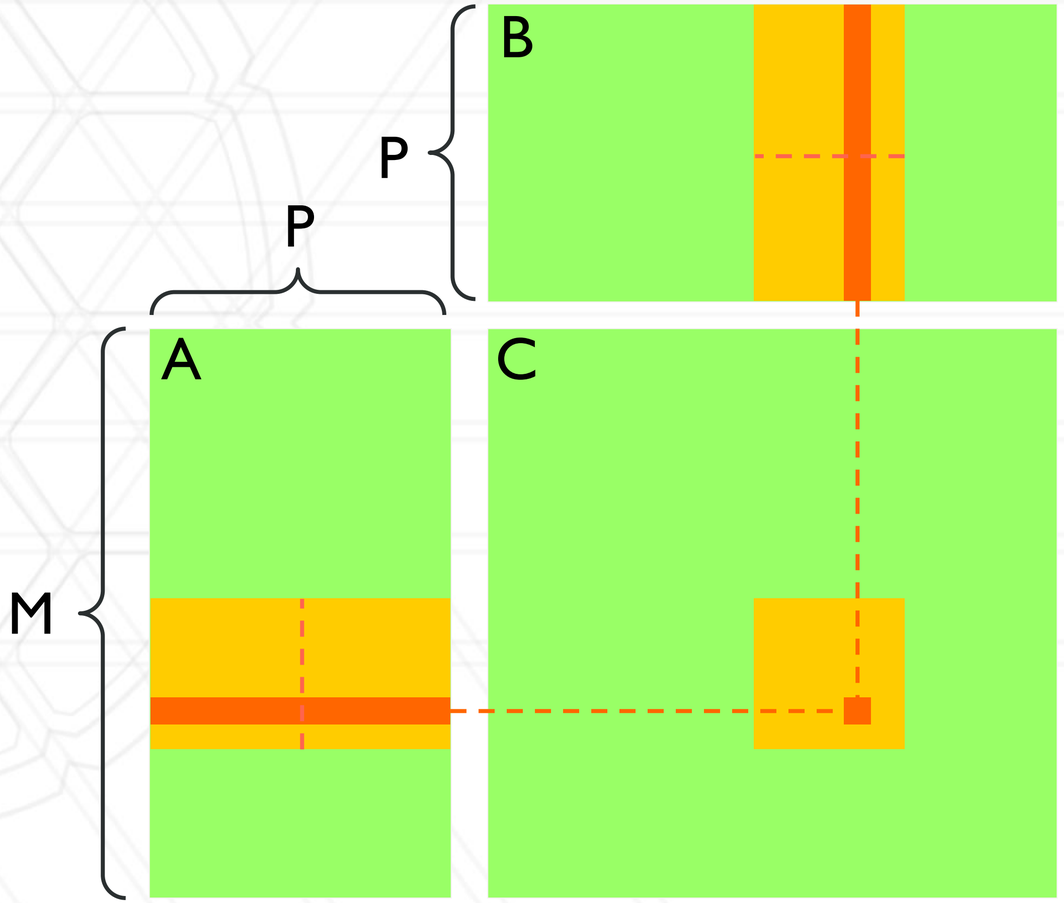
Matrix Multiply with Shared Memory

- Compute C=AB
- Block (i,j) compute submatrix C_{ij}
  - Save A & B submatrices into shared memory
  - Accumulate partial dot product into C



Matrix Multiply with Shared Memory

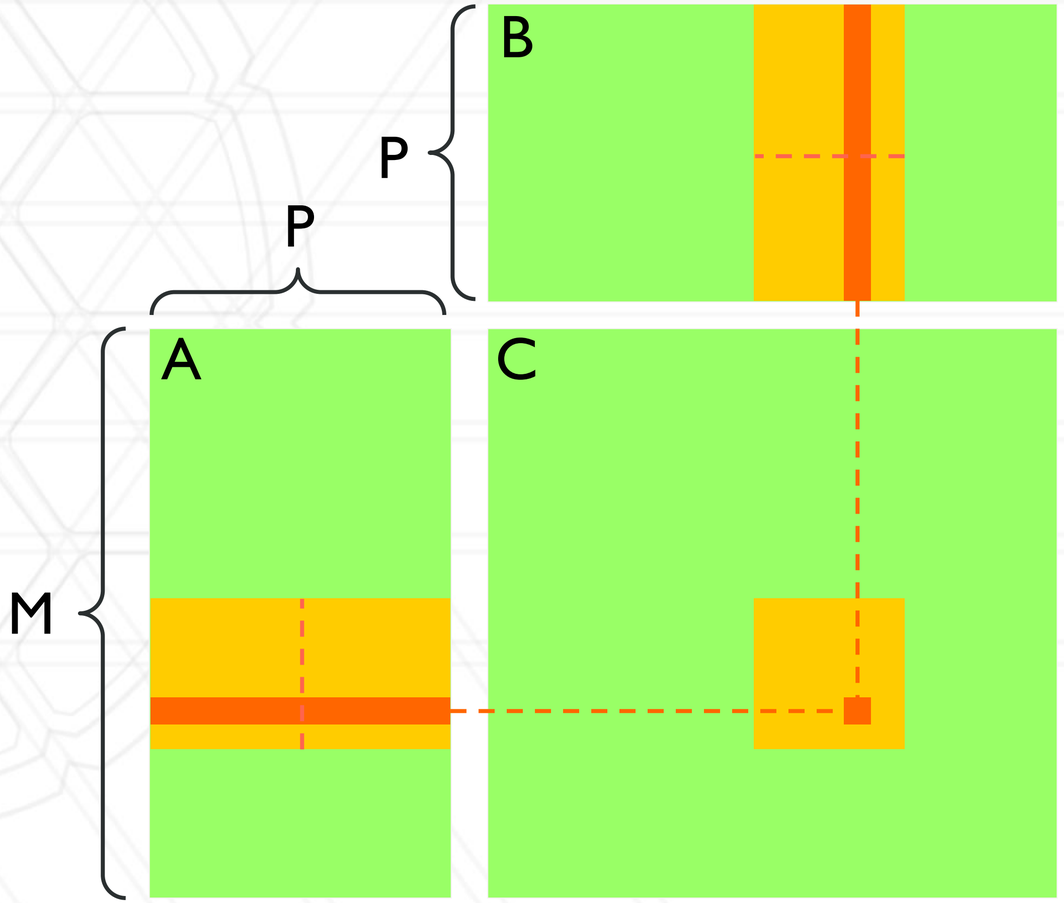
- Compute C=AB
- Block (i,j) compute submatrix C_{ij}
  - Save A & B submatrices into shared memory
  - Accumulate partial dot product into C
- A is read N / block_size times
- B is read M / block_size times
- Data reads from global memory are reduced by an order of the block size



Matrix Multiply with Shared Memory

Algorithm	Time (s)
Simple CPU	170.898
Simple GPU	1.997
Shared Memory	0.091
CuBLAS	0.017

A, B are 2048x2048



Questions?

Streams

- CUDA kernels execute in streams
- Kernels in the same stream execute sequentially
- Kernels in separate streams can execute concurrently

```
cudaStream_t stream;
...
kernel<<<grid, block, 0, stream>>>(x, b);
```

More info https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

Streams

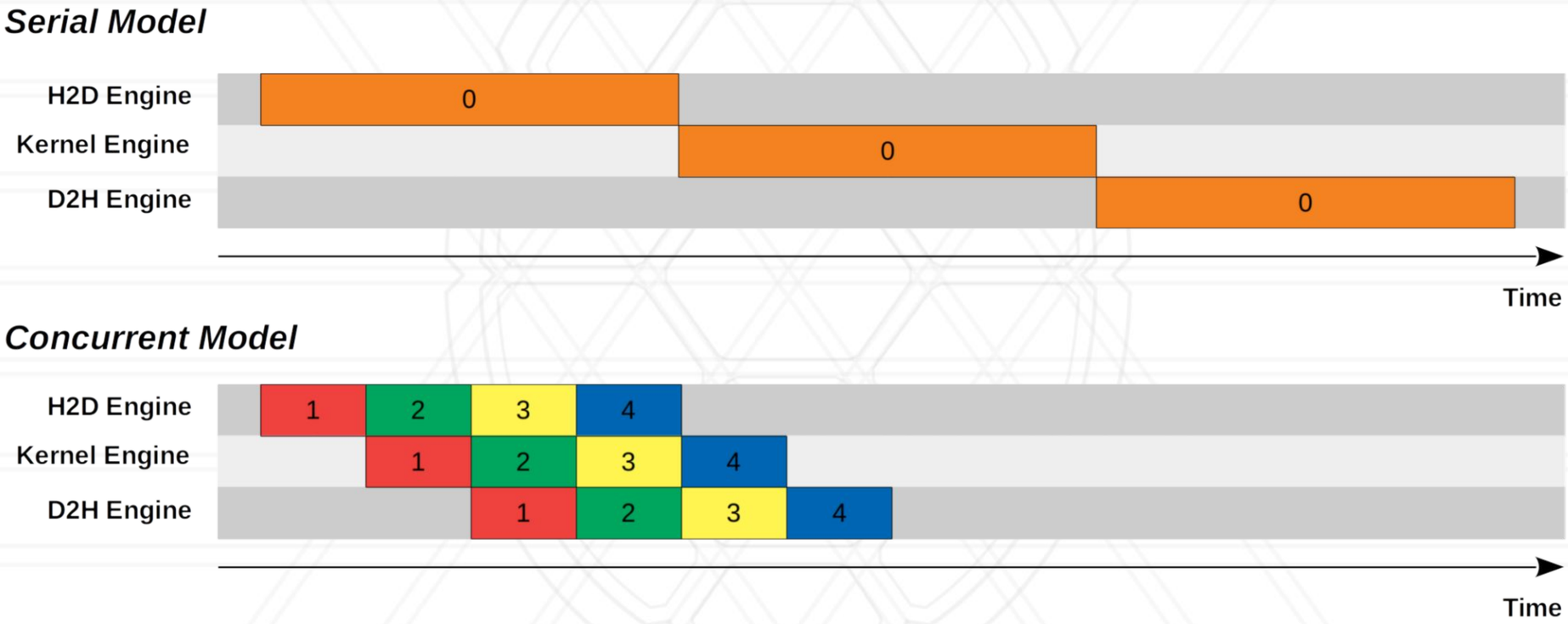


Image from https://leimao.github.io/blog/CUDA-Stream/

GPU Performance Optimization

- Profiling
  - Nsight Systems: https://developer.nvidia.com/nsight-systems
- Common performance issues
  - Host ↔ Device memory copying
  - Memory, memory, memory
  - Register pressure
  - Warp divergence
  - Occupancy

UNIVERSITY OF
MARYLAND