Computing with GPU
Part 2

Topics: data sharing within a thread-block
expressing parallelism using 1d and 2d grids and thread-blocks

Hands on: dot product and matrix-vector multiplication

Reductions: result = dot(x,x)

Sequential

```
result = 0.0;
for (size_t  i = 0; i < N; ++i)
    result += x[i]*x[i];
```

Parallel: shared memory

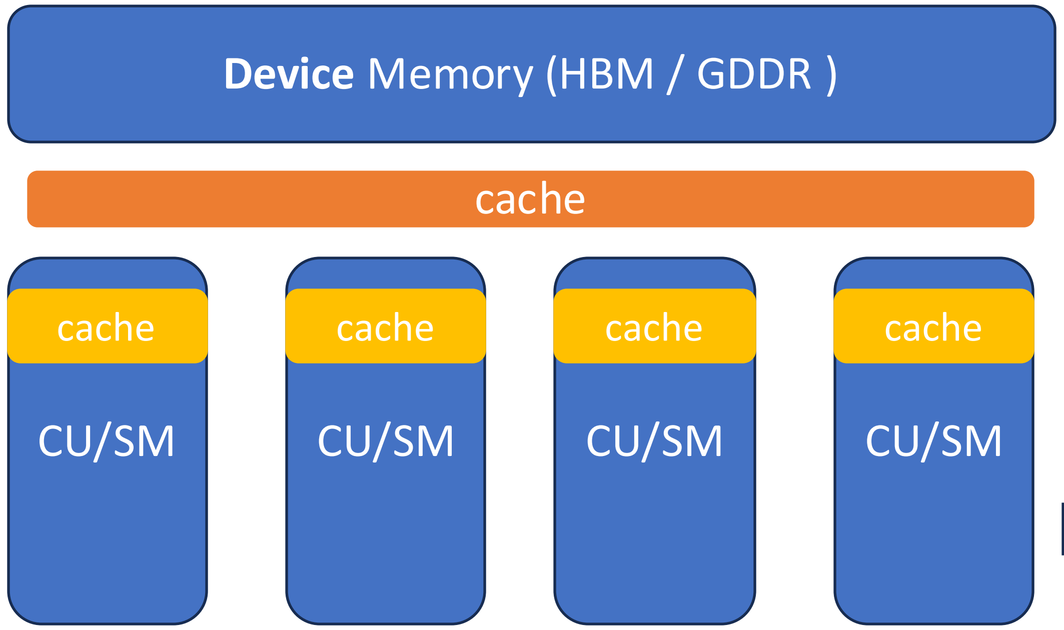
```
result = 0.0;
#pragma omp parallel for reduction(+:result)
for (size_t  i = 0; i < N; ++i)
   result += x[i]*x[i];
```

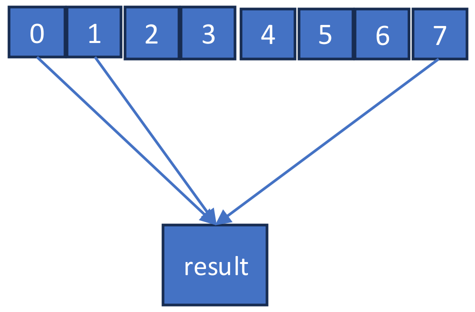
Parallel: shared and distributed memory

```
result_tmp = 0.0;
result_tmp = reduce_local(x,Nlocal);
MPI_Allreduce(&result_tmp,1,&result,…);
```

Reduction on GPU

Reduction on GPU: dot product





Multiplication is trivially parallelizable

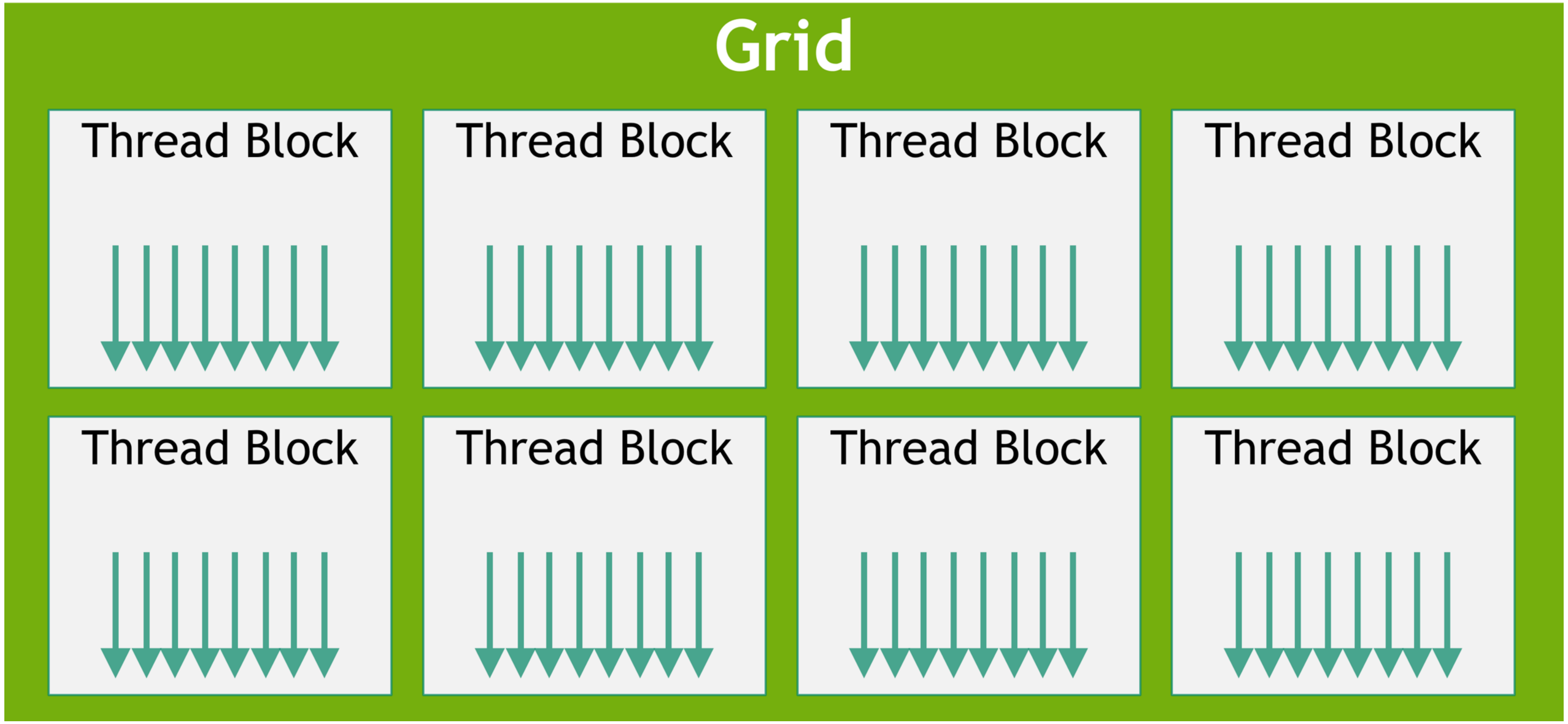
summation is not …

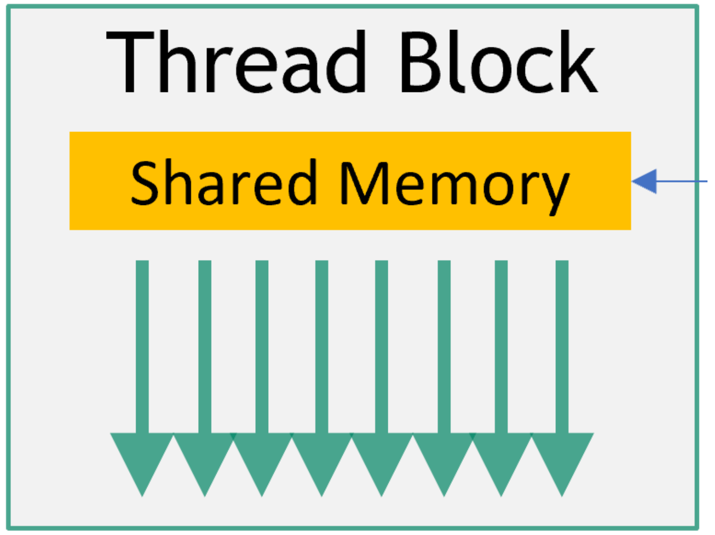
summation requires communication of data

a) between threads in a warp
b) Between warps in a thread-block
c) between thread-blocks
d) between GPUs
r) between servers

summation requires communication of data

a) between threads in a warp
b) Between warps in a thread-block
c) between thread-blocks





summation requires communication of data

a) between threads in a warp
b) between warps in a thread-block

WARP – is a collection of threads that

a) executed simultaneously on an SM/CU
b) have a specialized mechanism for communication/exchange of data

Communication of data within a WARP

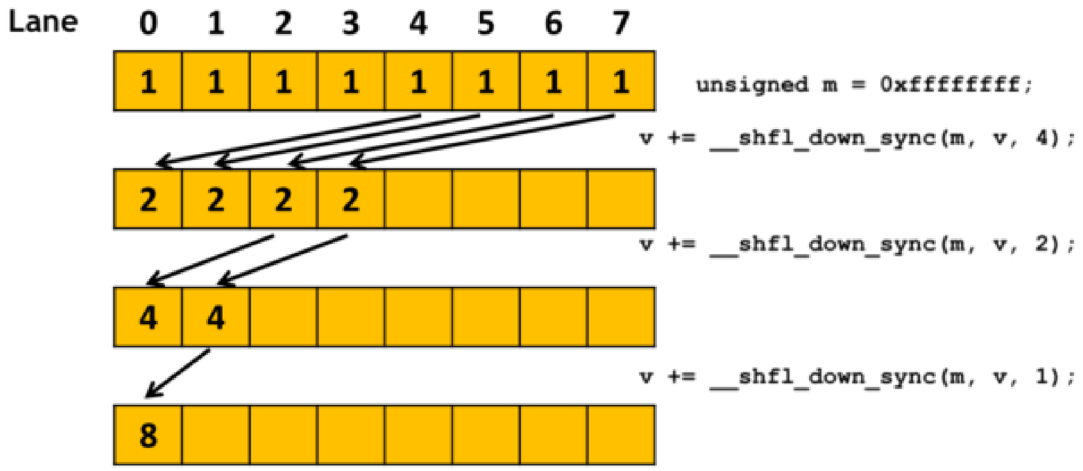
In general, we have 3 ways of communication of data between threads within a warp

1. Shuffle operations (fastest)
2. Use of shared memory within a block
3. Writing data into global memory and reading back (most likely data will still be in the cache)

There are also cross-lane operations (faster than shuffle)
https://gpuopen.com/learn/amd-gcn-assembly-cross-lane-operations/

Communication within warp

```
#define FULL_MASK 0xffffffff

int val =  threadIdx.x;

__syncwarp() ; //sync lanes within warp

for (int offset = WARP_SIZE/2; offset > 0; offset /= 2)
   val += __shfl_down_sync(FULL_MASK, val, offset);
```



```
unsigned m = 0xffffffff;
v += __shfl_down_sync(m, v, 4);
v += __shfl_down_sync(m, v, 2);
v += __shfl_down_sync(m, v, 1);
```

AMD GPUs: use

```
for (int offset = WARP_SIZE/2; offset > 0; offset /= 2)
   val += __shfl_down(val, offset);
```

There are additional types of shuffle operations, for more info:

https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

Shared memory

Shared memory is allocated per thread block → all threads in the block have access to the same shared memory.

On NVIDIA GPUs shared memory is “carved out” of L1 cache

On AMD GPUs shared memory (also called Local Device Storage) is a separate from cache memory

reading:

https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

Allocation of shared memory

```
kernel_A<<<grid_dim,block_dim, 0, 0>>>(x,n);

__global__ void kernel_A(int *x, int n) {

   __shared__ int shared_mem_array[64];

..
}
```

Statically allocated shared memory

```
kernel_B<<<grid_dim,block_dim, Nbytes, 0>>>(x,n);

__global__ void kernel_B(int *x, int n) {

    extern __shared__ int shared_mem_array[];
    …

}
```

Dynamically allocated shared memory

```
kernel_C<<<grid_dim,block_dim, Nbytes, 0>>>(x,n);

__global__ void kernel_B(int *x, int n) {

    extern __shared__ int shared_mem_array[];
    __shared__  float shared_mem_array2[4][32];
    …

}
```

Statically and Dynamically allocated shared memory

Communication between warps

```
#define __WARP_SIZE__  32


__shared__ float s_mem[1024/ __WARP_SIZE__ ];   //max number of warps within a thread block

int nwarps = blockDim.x / __WARP_SIZE__ ;
int warp_ID = threadIdx.x / __WARP_SIZE__ ;
int laneID_in_warp = threadIdx.x % __WARP_SIZE__ ;

if (laneID_in_warp == 0)   s_mem[warp_ID ] = value;
__syncthreads();

if (threadIdx.x == 0) {
   for (int i = 1; i < nwarps; ++i)  value += s_mem[i];
}
```

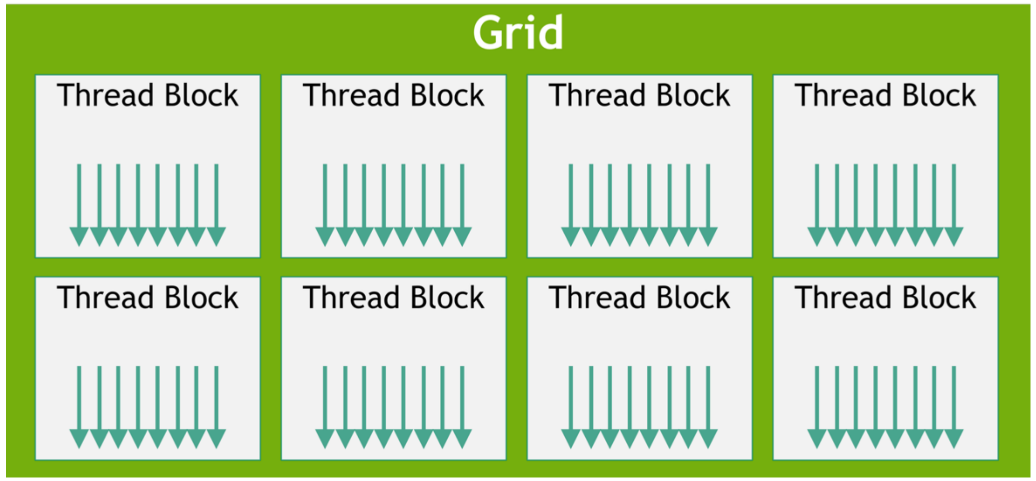
Task : write a kernel performing a dot product and using one thread-block

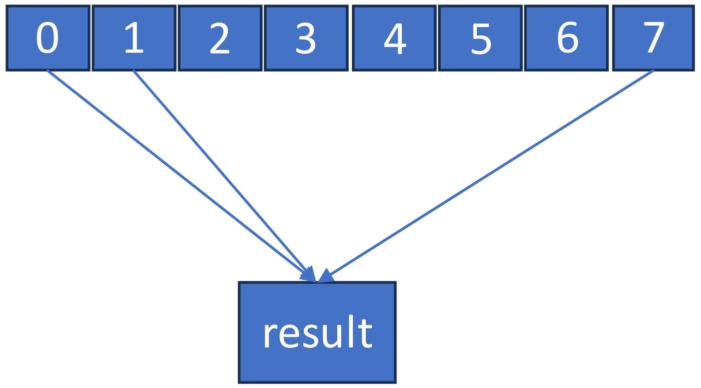
```
int main() {
  …
  double *X;
  X = new double[N];
  for (auto i=0; i<N; i++)
    X[i] = 0.01*i;

  double *x_d;
  cudaMalloc(&x_d,N*sizeoff(double));
  cudaMemcpy(x_d,X,…);

  double result_h = 0.0;

  gpu_func<<< … >>>(x_d, N, result_d);
  cudaDeviceSynchronize();


  cudaFree(x_d);

  delete[] X;

}
nvcc name.cu
```

Dot-product: communication of data between threadblocks

1) One thread from each block performs an atomic operation. [atomicAdd(ptr, value);]

2) One thread from each block writes into global memory into a different address





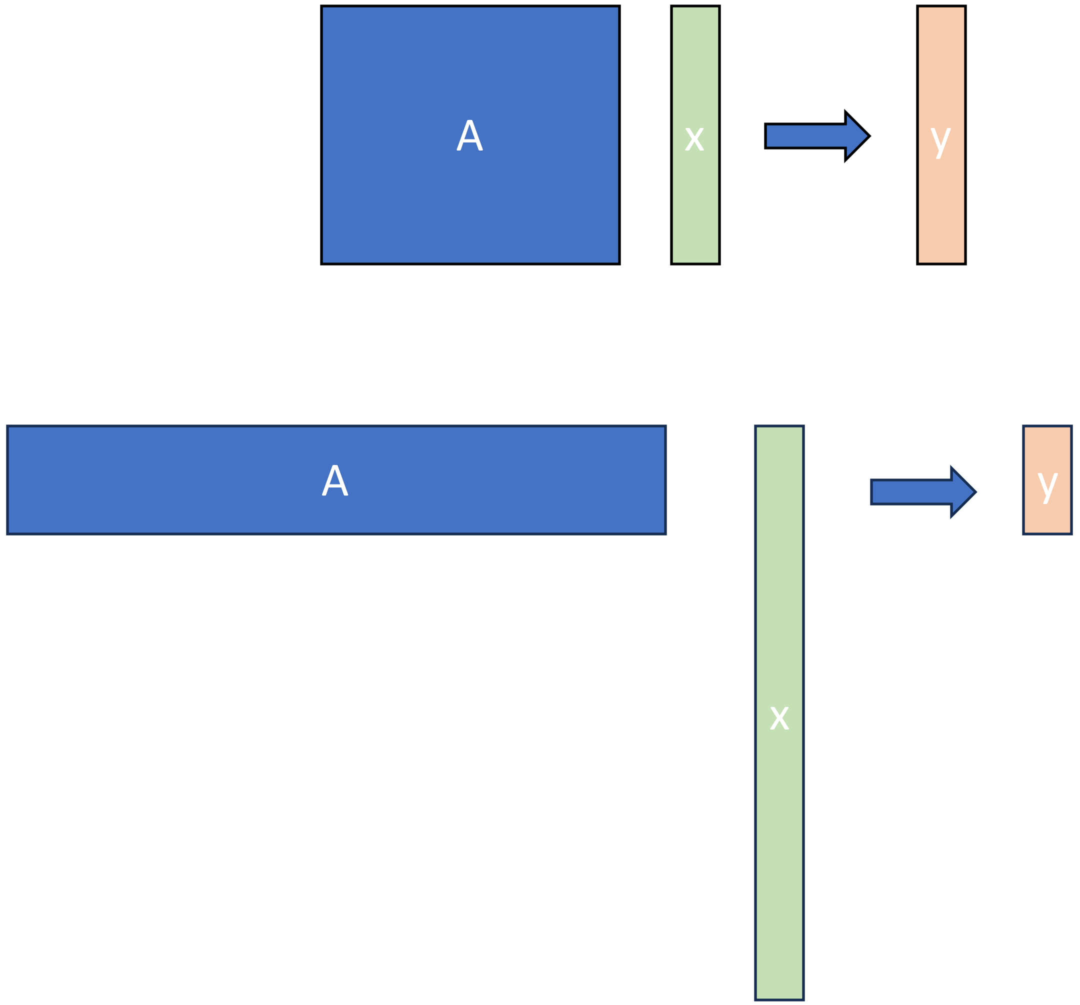
After a thread from a block writes into global memory, additional operations are required.
Options: a) last block performs a final reduction
b) launch additional kernel with 1 block to perform a final reduction
c) CPU performs a final reduction

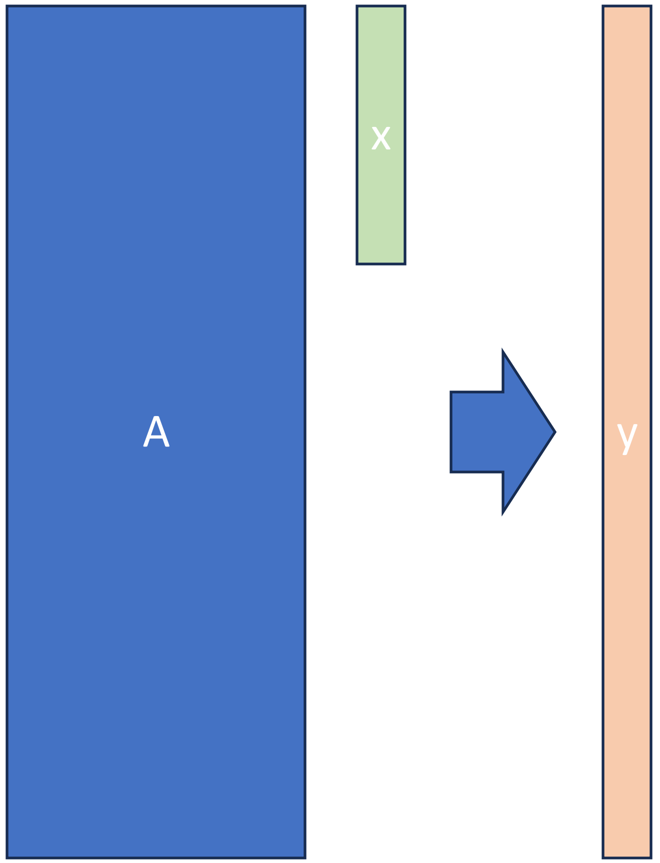
Task : write a kernel performing a dot product and using many thread-blocks

```
int main() {
  …
  double *X;
  X = new double[N];
  for (auto i=0; i<N; i++)
    X[i] = 0.01*i;

  double *x_d;
  cudaMalloc(&x_d,N*sizeoff(double));
  cudaMemcpy(x_d,X,…);

  double result_h = 0.0;

  gpu_func<<< … >>>(x_d, N, result_d);
  cudaDeviceSynchronize();


  cudaFree(x_d);

  delete[] X;

}
nvcc name.cu
```

Task: write a kernel for matrix vector multiplication on GPU

Matrix – vector multiplication on GPUs: how to express parallelism?





Matrix – vector multiplication on GPUs: how to express parallelism?

Considerations:

how many thread-blocks per each row?

how many threads [warps] per each thread-block?

how many rows each thread-blocks should process ?

2D grid vs 1D grid ?
2D thread-blocks vs 1D thread-blocks?

Matrix – vector multiplication on GPUs: where to start?

1. Re-use your code for matrix-vector multiplications on CPU
2. Start with 1D grid and 1 thread per row
3. 1D grid and 1 thread-block per row. (row_index == blockIdx.x)
4. 1D grid and multiple rows per thread-block.
   (row_index = blockId.x*blockDim.y + threadIdx.y;
   column_index = threadIdx.x)

…

N. General case with 2D grid and 2D blocks

Suggested reading

https://www.cise.ufl.edu/~sahni/papers/gpuMatrixMultiply.pdf

https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf

https://escholarship.org/content/qt1wb7f3h4/qt1wb7f3h4_noSplash_1e32f64125997ee6afa303a150338054.pdf