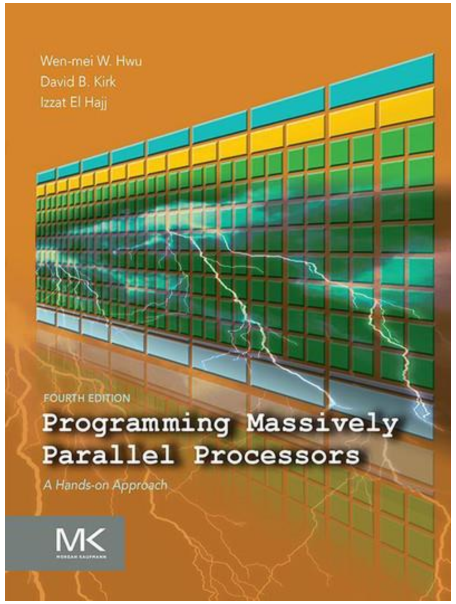
LLM Sys

GPU Acceleration

Lei Li

Language Technologies Institute | Carnegie Mellon University School of Computer Science



Programming Massively Parallel Processors (4th ed)
https://learning.oreilly.com/library/view/programming-massively-parallel/9780323984638/?sso_link=yes&sso_link_from=cmu-edu

CUDA Programming Guide (from Nvidia)
https://docs.nvidia.com/cuda/cuda-programming-guide

Outline: GPU Acceleration techniques

- Tiling (Chap 5)
- Memory parallelism (Chap 5 & 6)
- Sparse Matrix Multiplication
- cuBLAS

Memory Access Efficiency is Critical

A100 80G PCIe	
FP32	19.5 TFLOPS
Tensor Float 32	156 TFLOPS
GPU Memory Bandwidth	1,935 GB/s

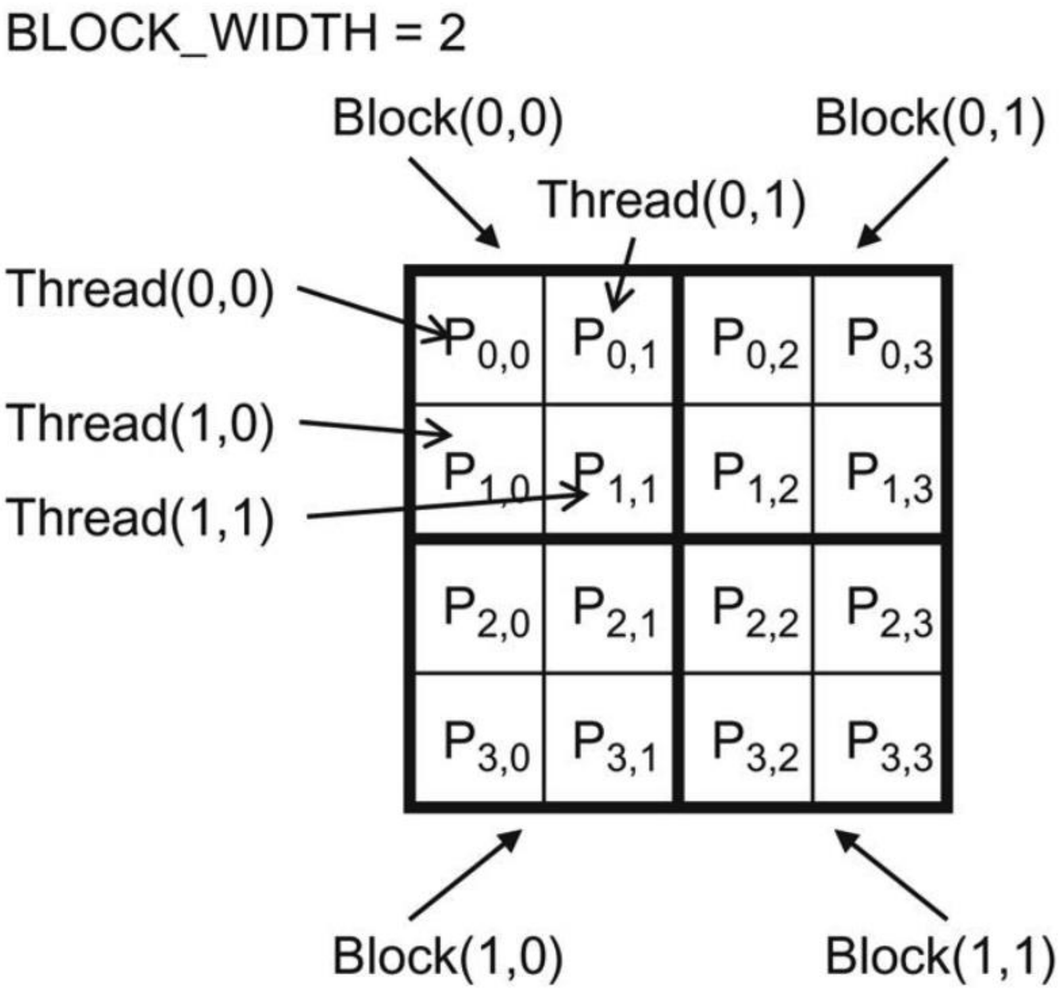
- How many FP32 operations per second
  - may also be bounded by memory load/store
- Compute-to-global-memory-access-ratio
  - the number of FLOPs performed for each byte access from the GPU global memory

Memory Access in Matrix Multiplication

- Grid
  - four thread blocks (2x2)
  - each block with 4 threads (2x2)

- Assuming every thread is responsible for calculating one element of result matrix P.

```
dim3 dimBlock(2, 2);
dim3 dimGrid(2, 2);
```



Example: simple matrix multiplication

```
__global__ void MatMulKernel(float *a, float *b, float *c, int N) {
    // Compute each thread's global row and col index -> output: (i, j)
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row >= N || col >= N) return;
    float Pvalue = 0.0;
    for (int k = 0; k < N; k++) {
        Pvalue += a[row * N + k] * b[k * N + col];
    }
    c[row * N + col] = Pvalue;
}
```

1 FP32 multiply
1 FP32 add

2 global memory access of FP32 (=4B)

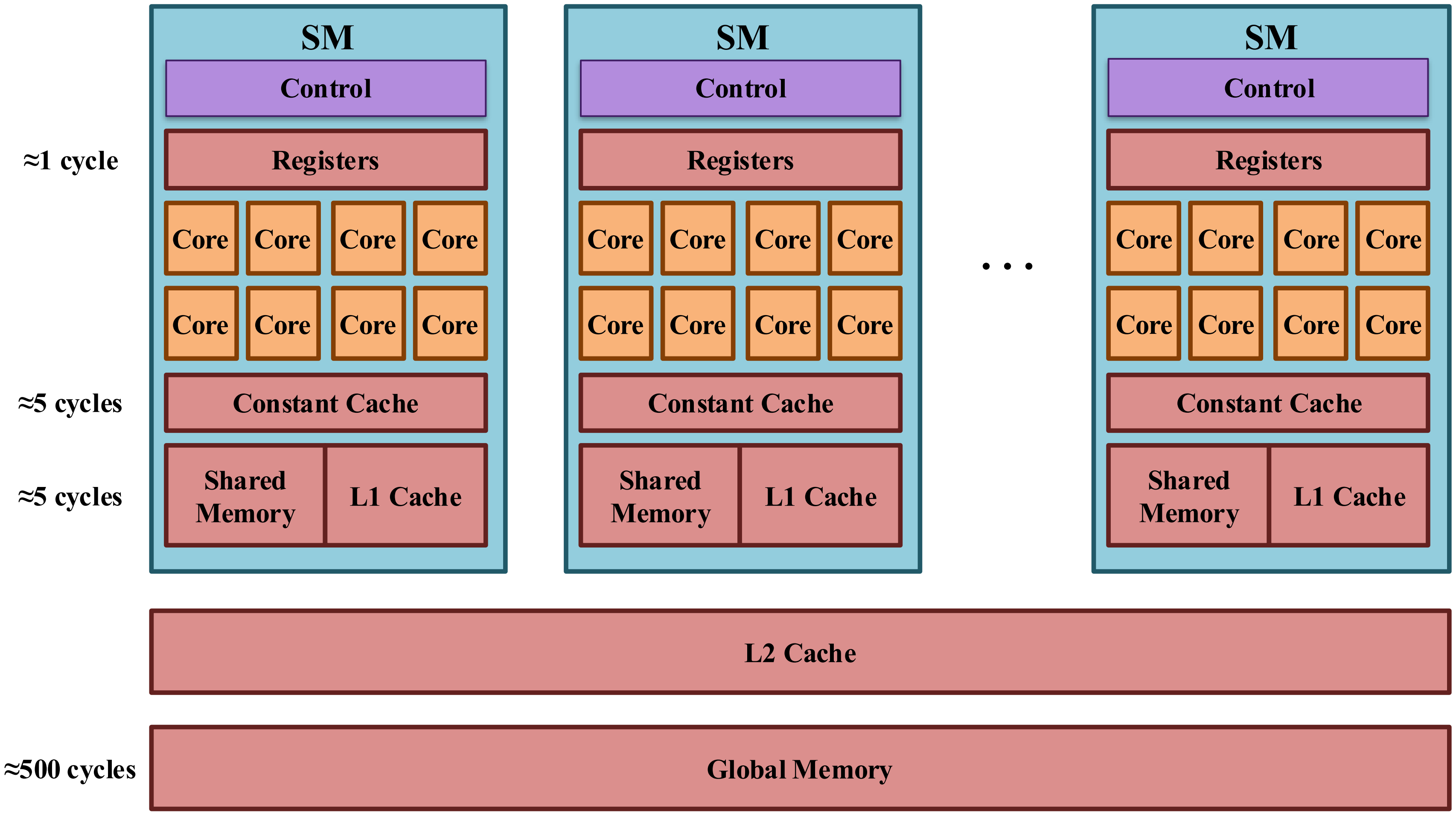
compute-to-global-memory-access: 2 / (2 * 4) = 0.25 FLOP/B

Peak Computation on A100

A100 80G PCIe	
FP32	19.5 TFLOPS
Tensor Float 32	156 TFLOPS
GPU Memory Bandwidth	1,935 GB/s

- How much computation is based on memory access?
  - 1935G * 0.25 FLOP/B = 483.75 GFLOPs
  - 483.75GFLOPs = 2.48% of peak FP32 ops
  - 483.75GFLOPs = 0.3% of peak Tensor FP32 ops
- Memory-bound program
  - computation limited by data transfer rate from memory

Memory Access Efficiency



Loading vs Computing

```
C[i] = A[i] + B[i];
```

GPU Instructions:

```
ld.global.f32 %f1, [%rd1]; // Load A[i]
ld.global.f32 %f2, [%rd2]; // Load B[i]

add.f32 %f3, %f1, %f2; // Perform fp32 addition

st.global.f32 [%rd3], %f3; // Store result
```

500 cycle

500 cycle

1 cycle

Loading data takes more time than actual computation!

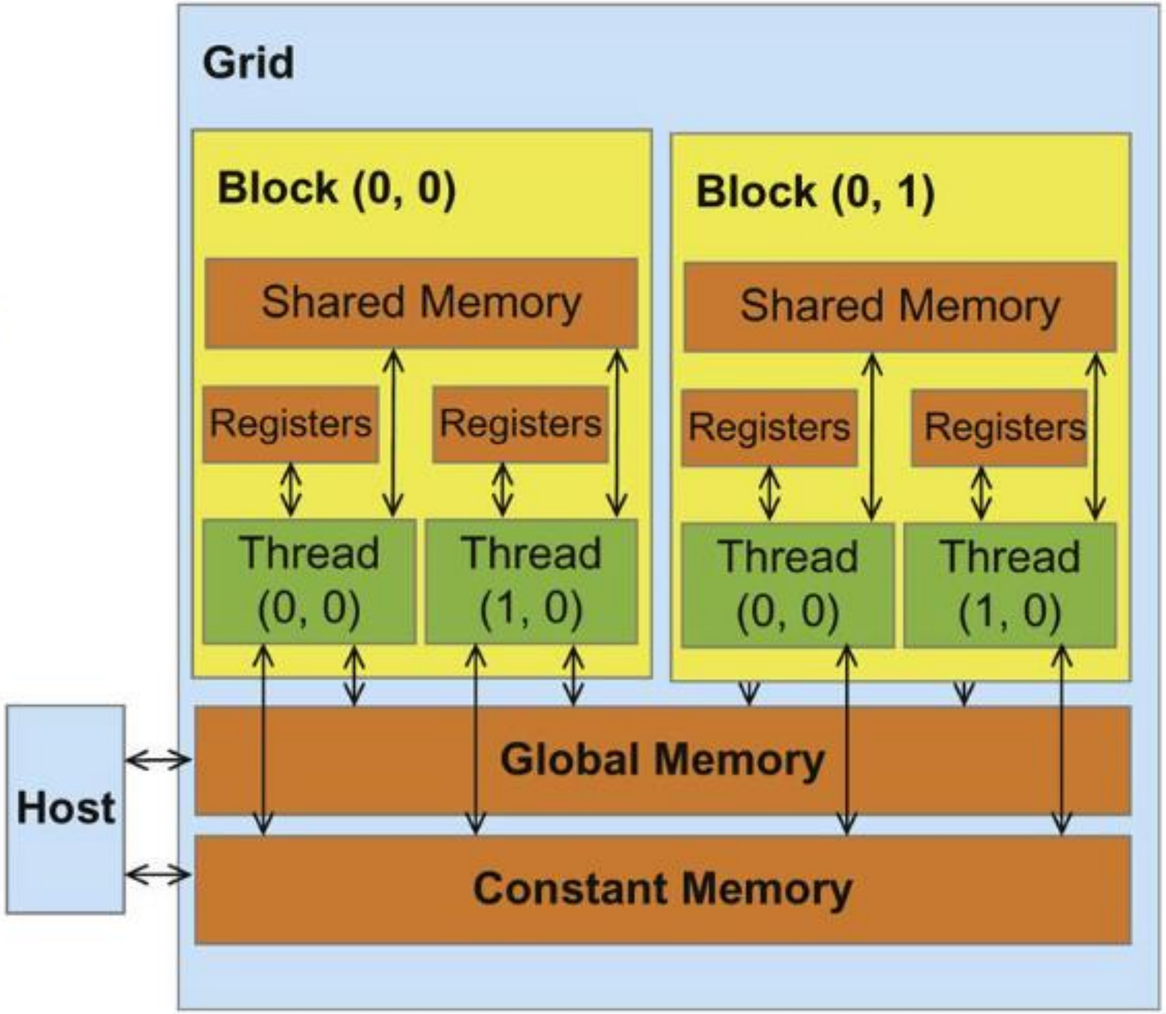
CUDA Device Memory Model

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories

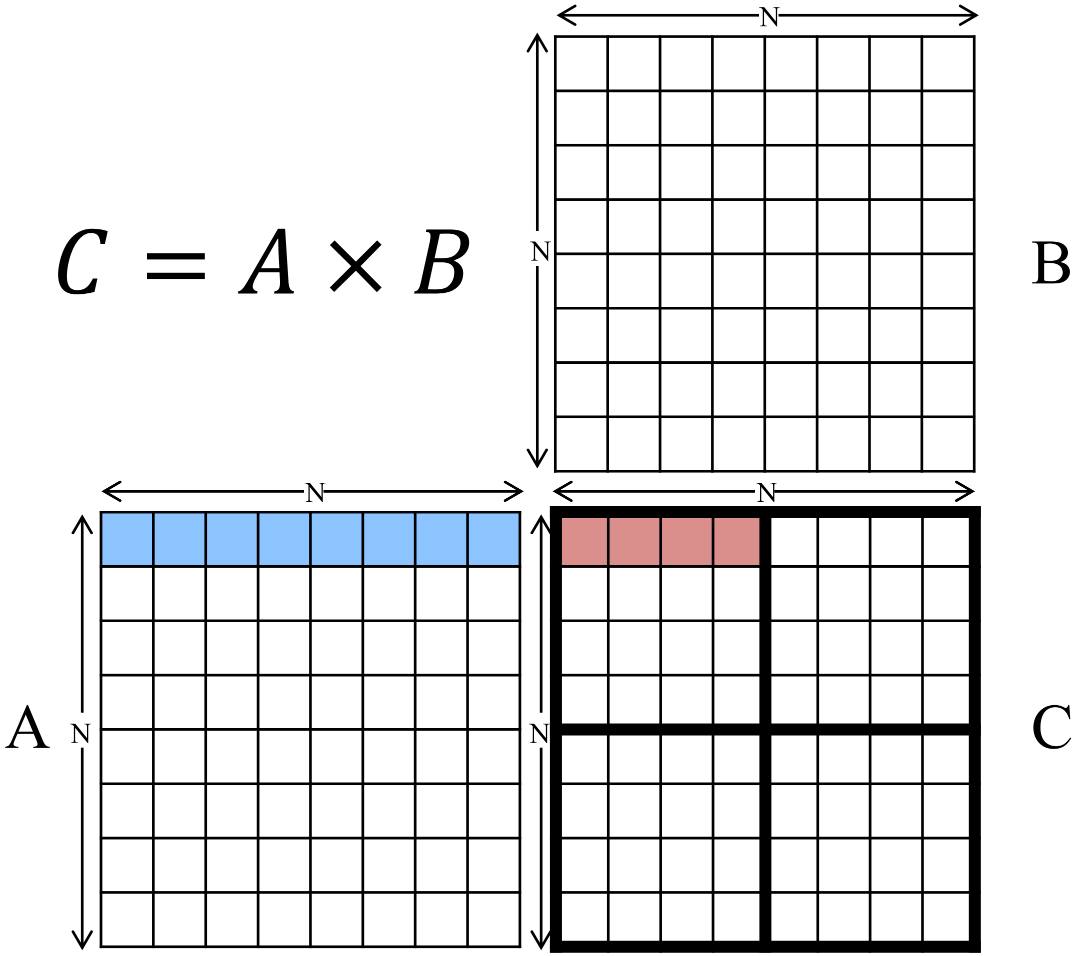


Access Device Memory

Variable declaration	Memory	Scope	Lifetime
`int var;`	Register	Thread	Grid
`int varArr[N];`	Local	Thread	Grid
`__device__ __shared__ int SharedVar;`	Shared	Block	Grid
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int constVar;`	Constant	Grid	Application

Opportunity to speedup: Reuse loaded data

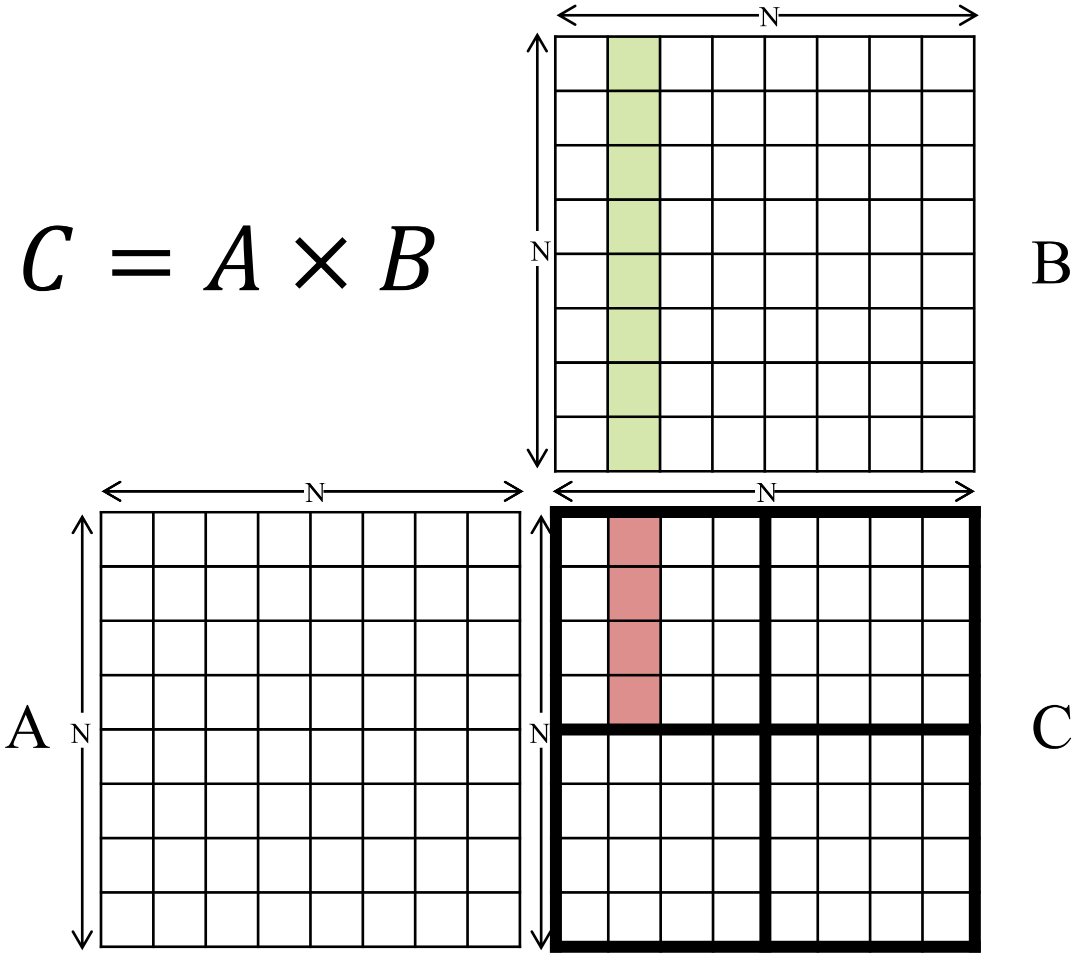
$C = A \times B$



threads in the thread block may use the same data

Opportunity to speedup: Reuse loaded data

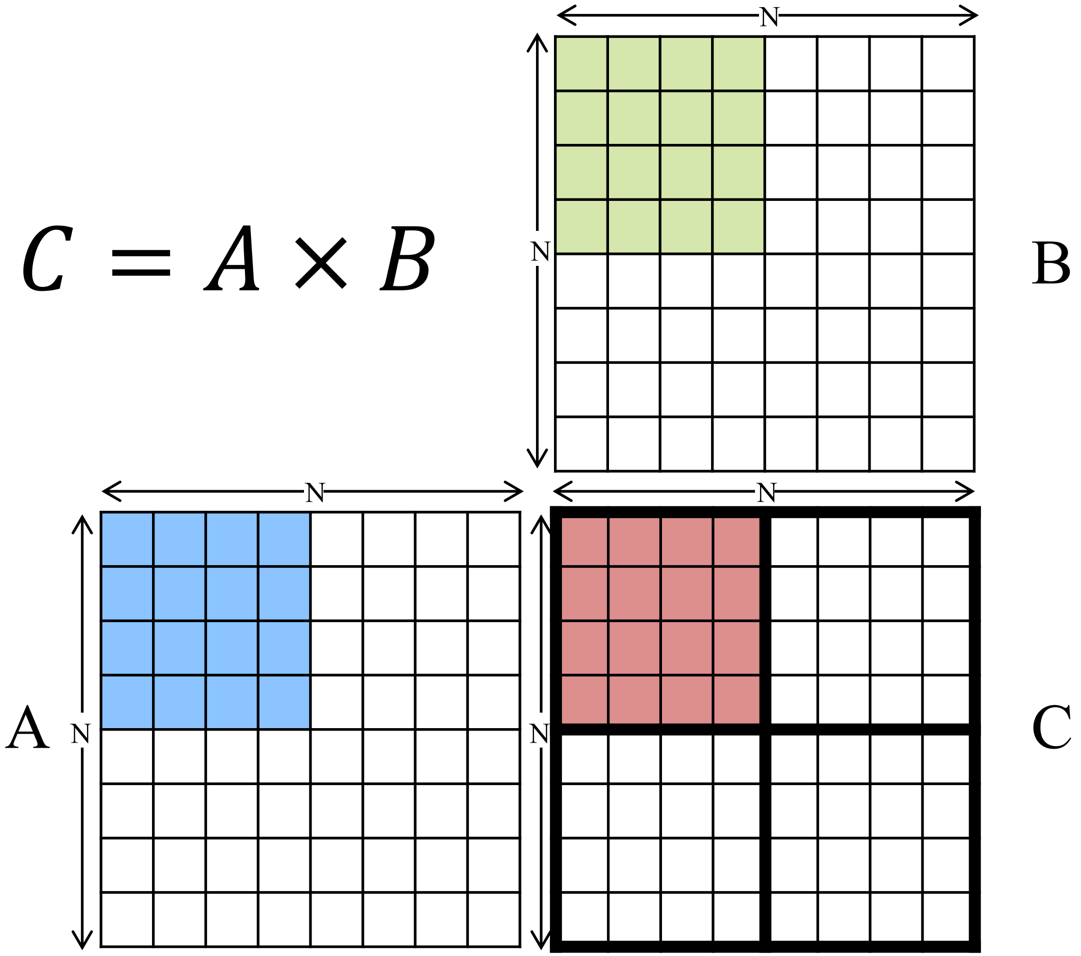
$$C = A \times B$$



threads in the thread block may use the same data

Tiling for Matrix Multiplication

$$C = A \times B$$



Step 1 (simultaneously)

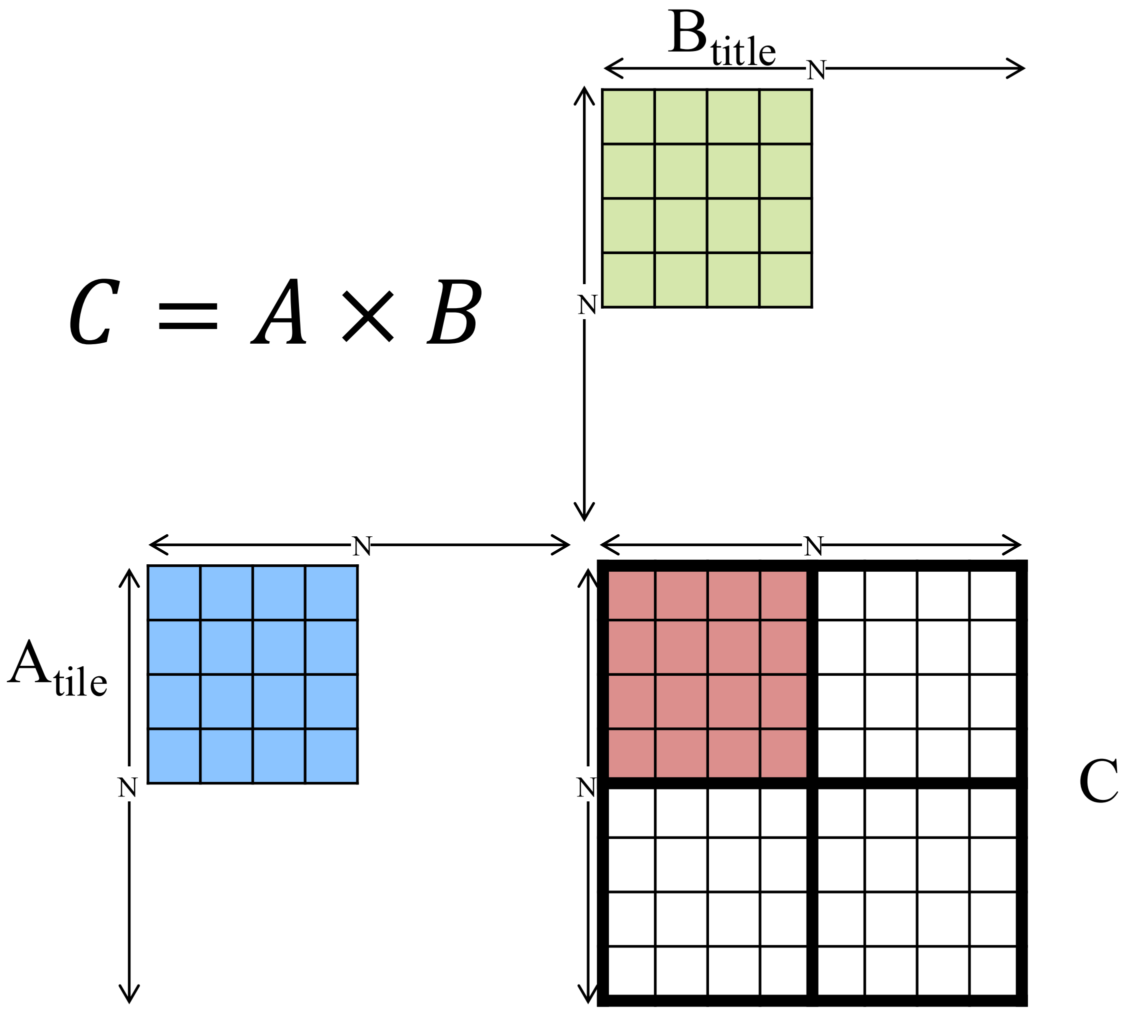
load the first tile of each input matrix to shared memory

each thread in the thread block loads one element

wait for loading **__syncthreads()**

Tiling for Matrix Multiplication

$$C = A \times B$$



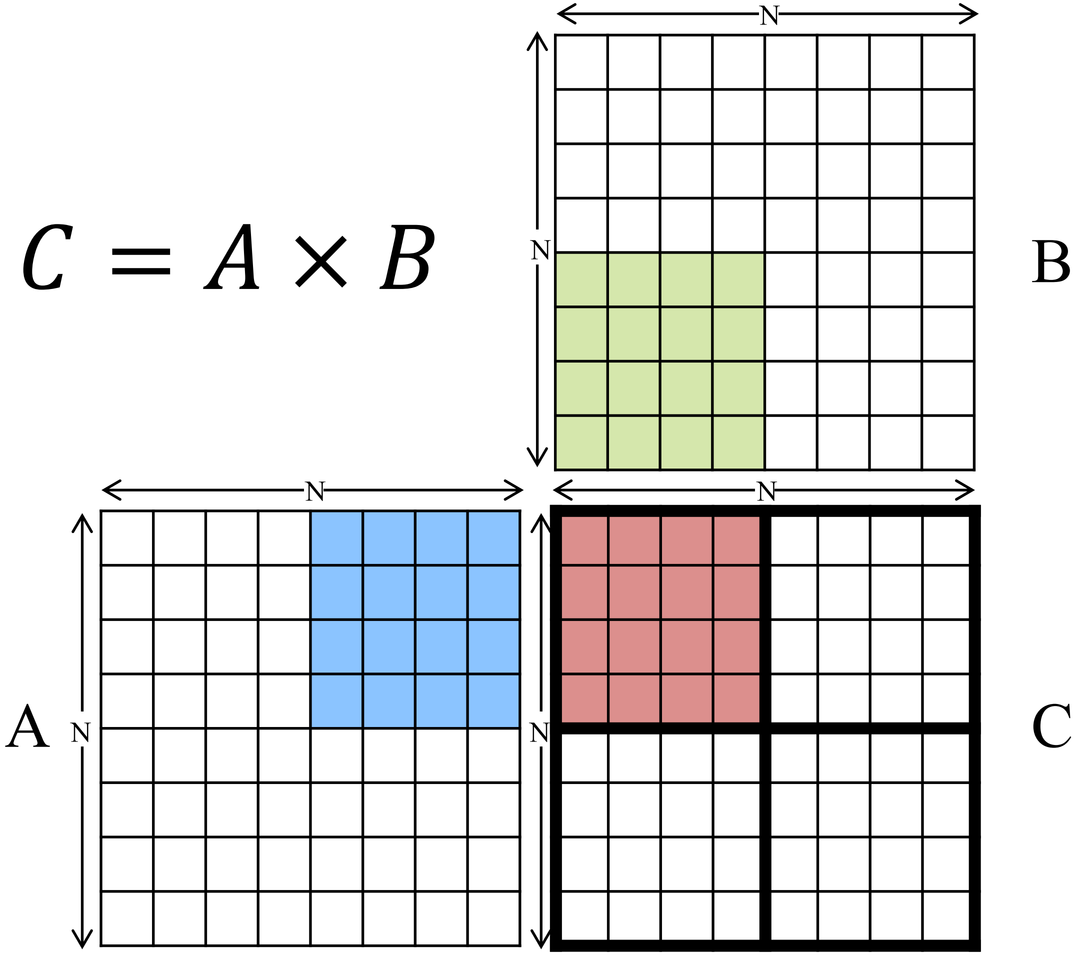
Step 2

each thread in the thread block computes the partial sum from the tiles in shared memory, threads wait for each other to finish

__syncthreads()

Tiling for Matrix Multiplication

$$C = A \times B$$



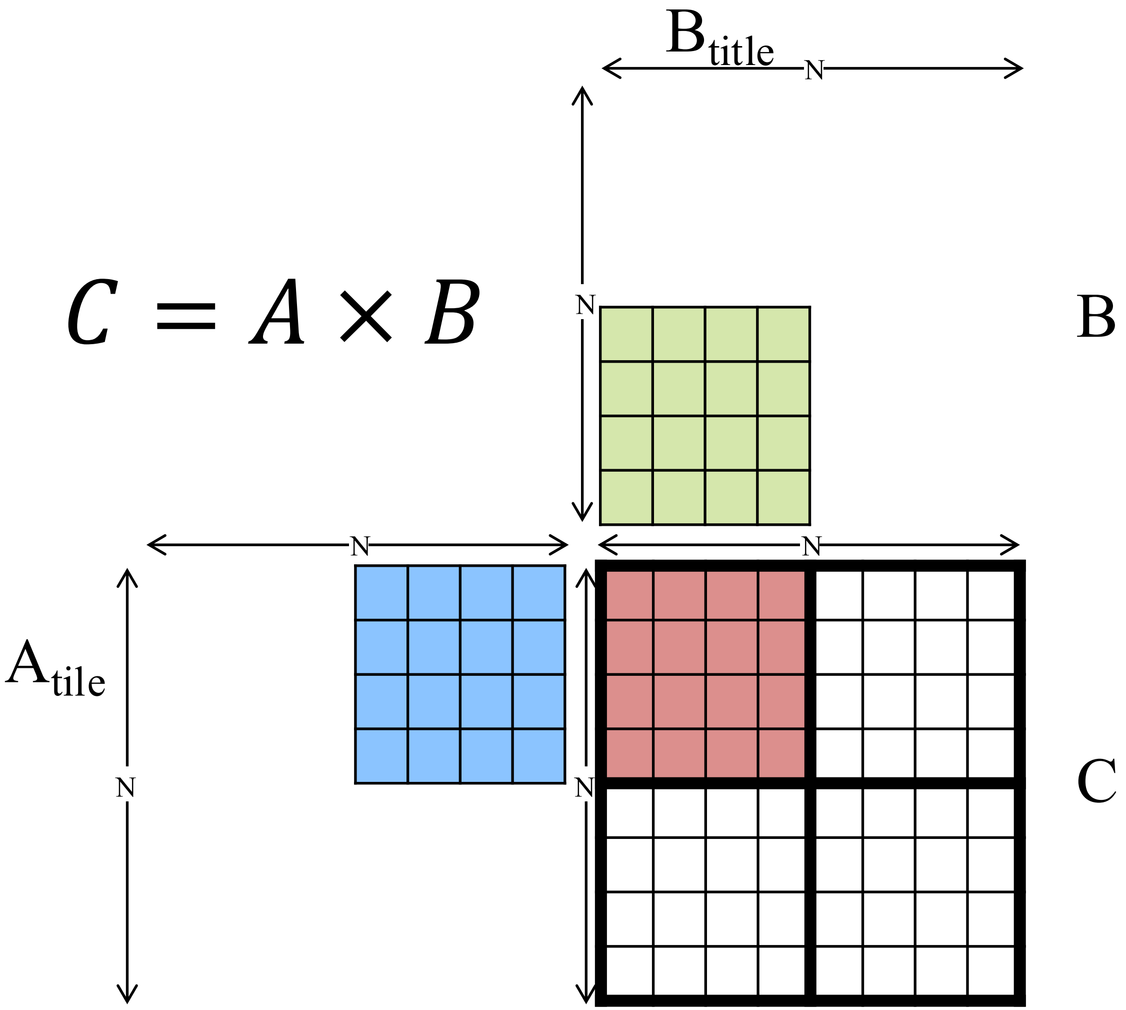
Step 3 (simultaneously)

load the next tile of each input matrix to shared memory

each thread in the thread block loads one element

Tiling for Matrix Multiplication

$$C = A \times B$$

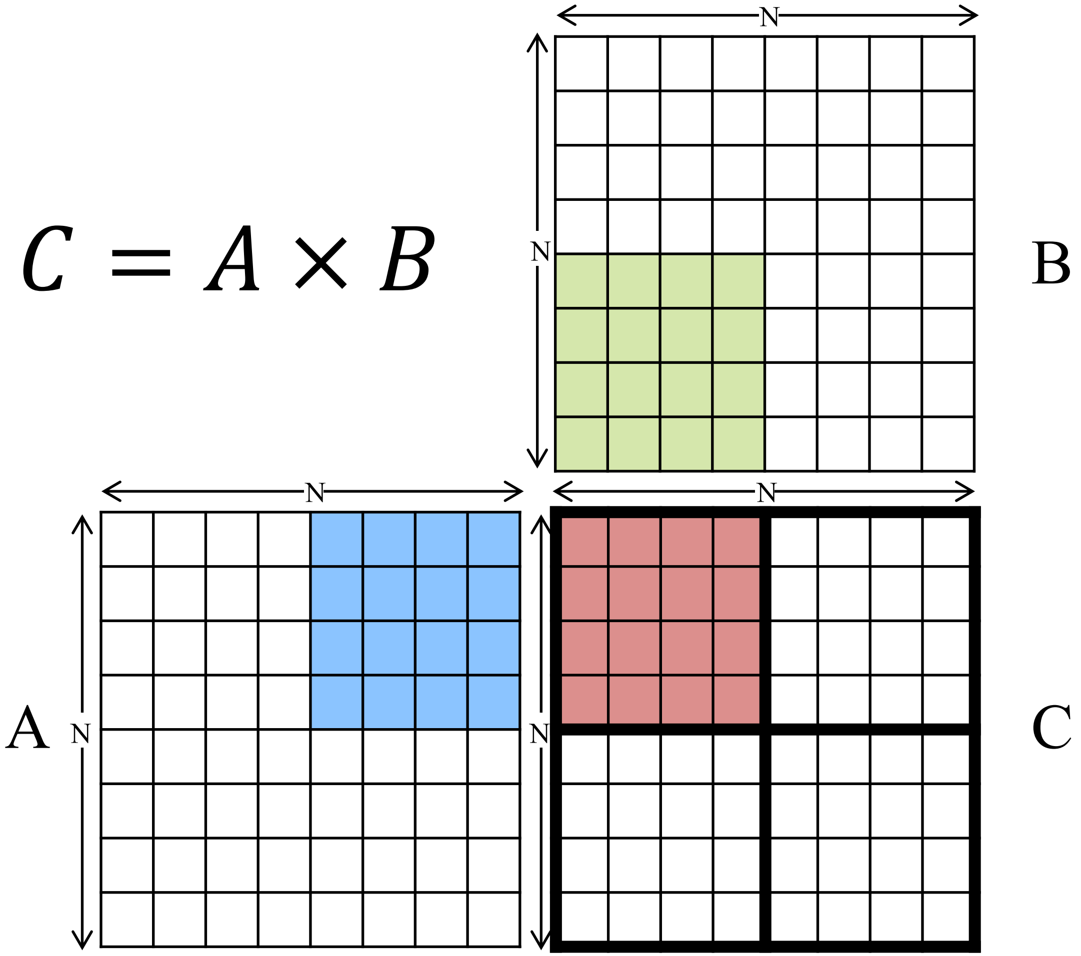


Step 4

each thread in the thread block updates the partial sum from the tiles in shared memory, threads wait for each other to finish

Tiling for Matrix Multiplication

$$C = A \times B$$



Step 5, 6, 7, 8 …

continue for next tiles

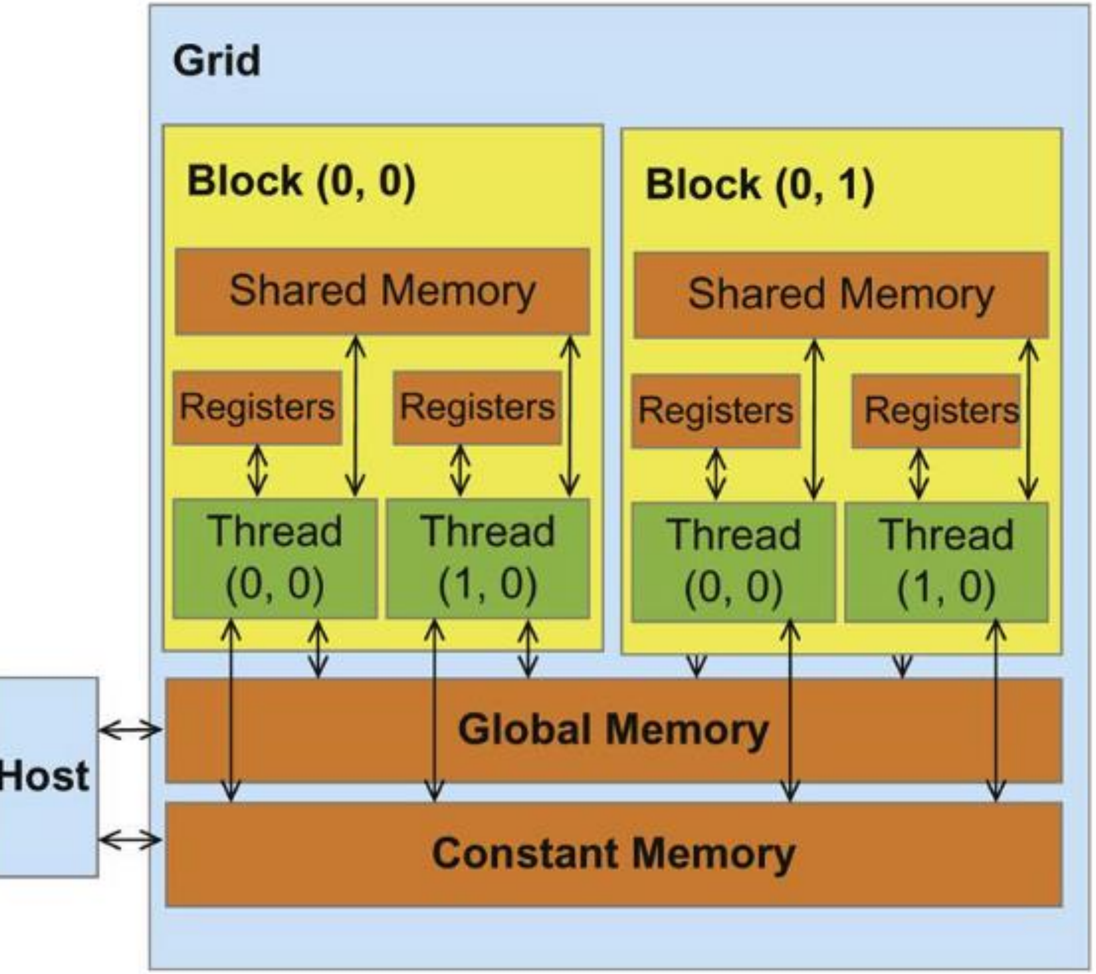
Outline: GPU Acceleration techniques

- Tiling (Chap 5)
- Memory parallelism (Chap 5 & 6)
- Sparse Matrix Multiplication
- cuBLAS

Memory Restriction

GPU	NVIDIA H100	NVIDIA A100
FP32	67 teraFLOPS	19.5 teraFLOPS
Memory	80GB HBM3	80GB HBM2e
Memory Bandwidth	3.35TB/s	2TB/s
SMs	132	104
Shared memory per SM	256K	192K
Registers per SM	64K	64K

Memory Restriction



If 1024 threads, 16384 registers

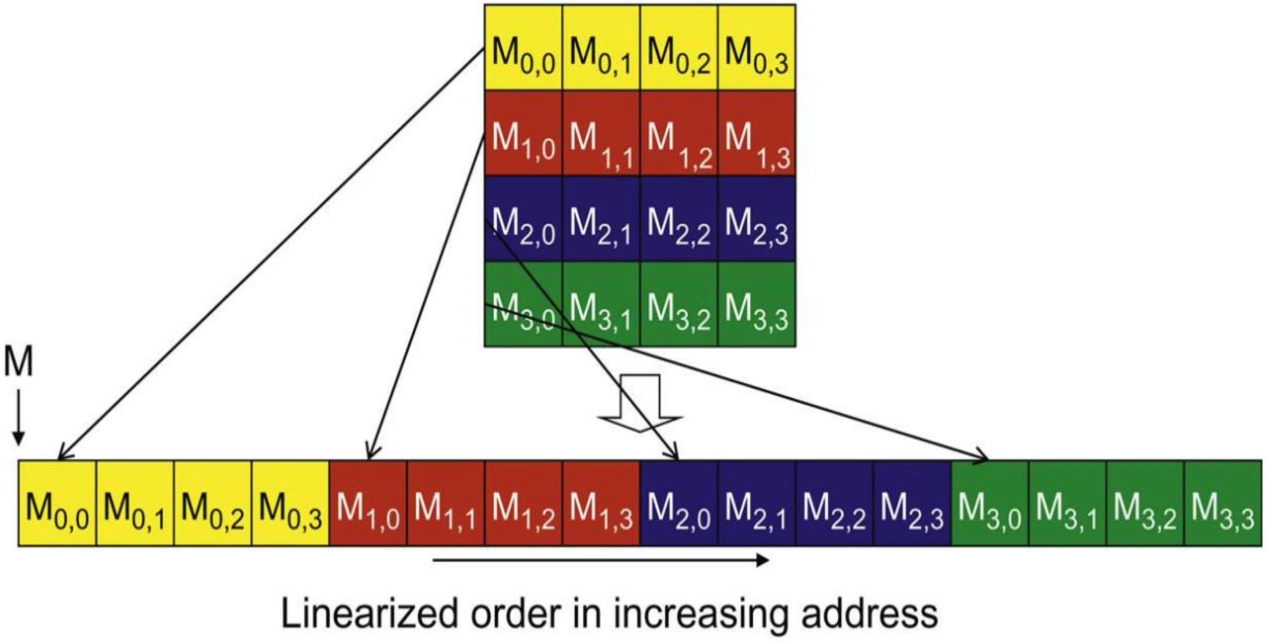
- Each thread can use only 16384/1024 = 16 registers

Each block can use up to 192 KB of shared memory

Tiled memory (e.g.):

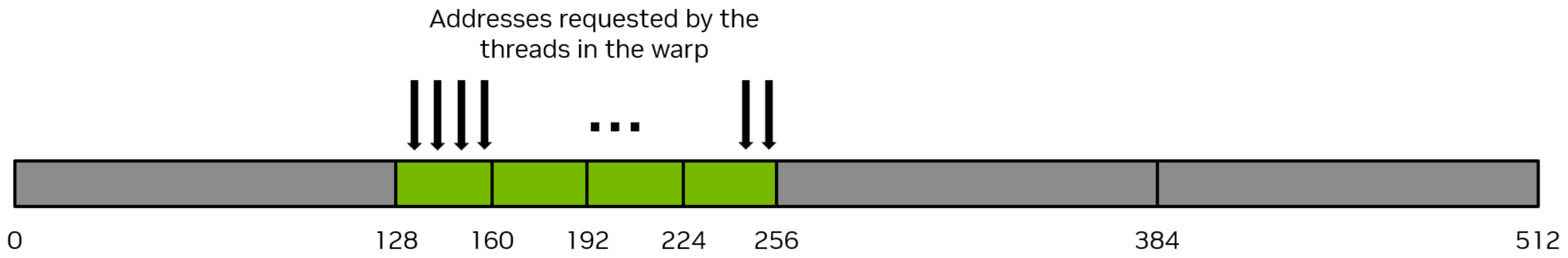
- As: 32 x 32 x 4 = 4KB
- Bs: 32 x 32 x 4 = 4KB

Locality / Bursts Organization



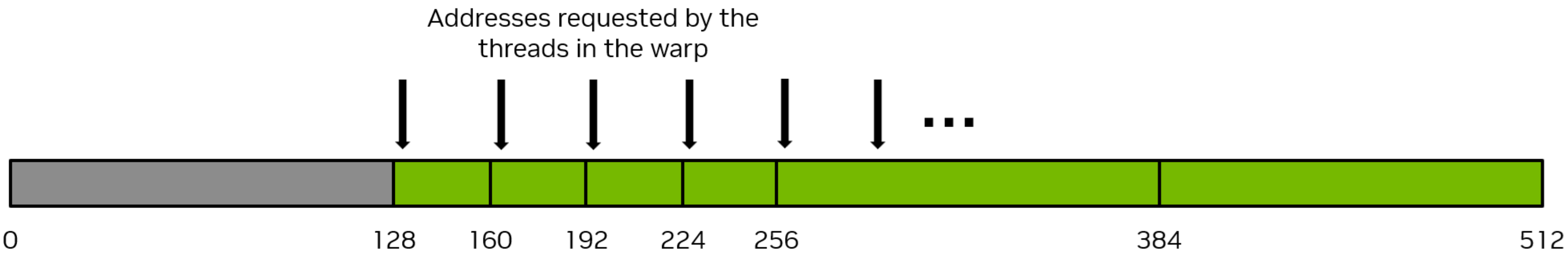
- access 32-byte each time
- Consecutive memory accesses in a warp are coalesced together.
- Row-major format to store multidimensional array in C and CUDA
- allows DRAM burst, faster than individual access

Coalesced Memory Access (fast)



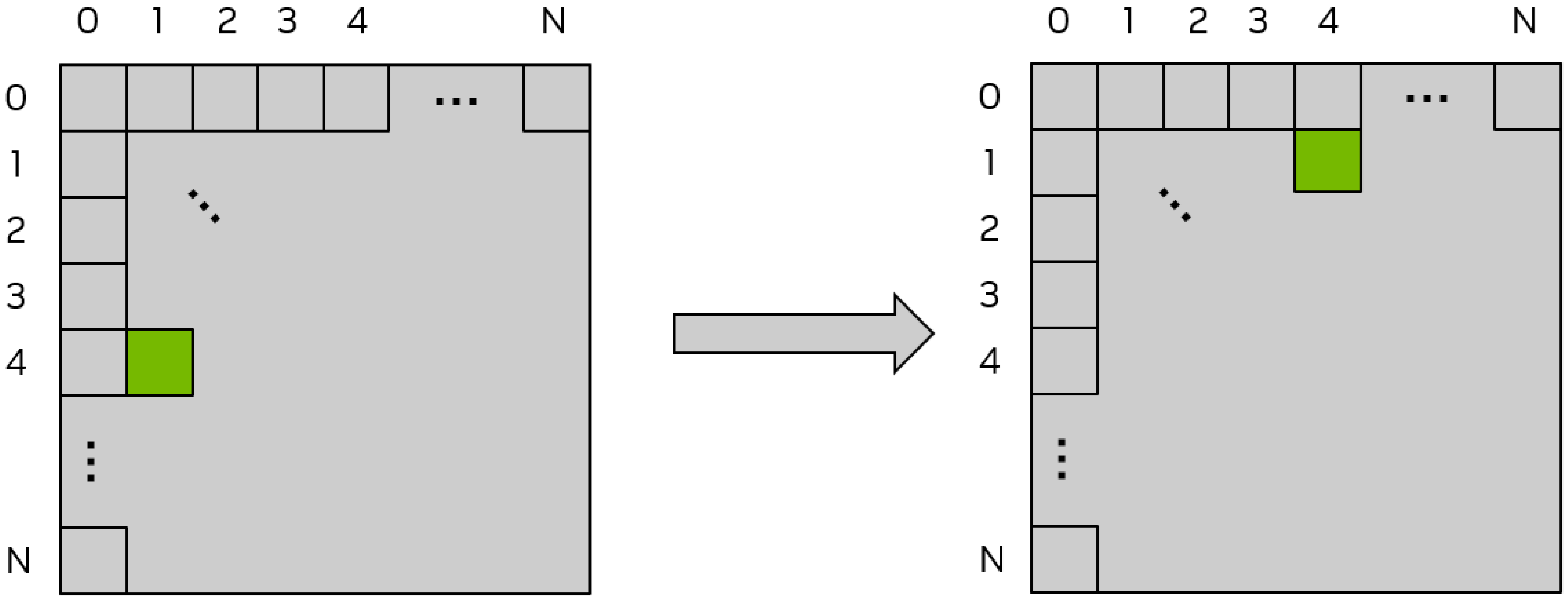
one memory load to serve multiple threads in a warp

Uncoalesced Memory Acess (slow)



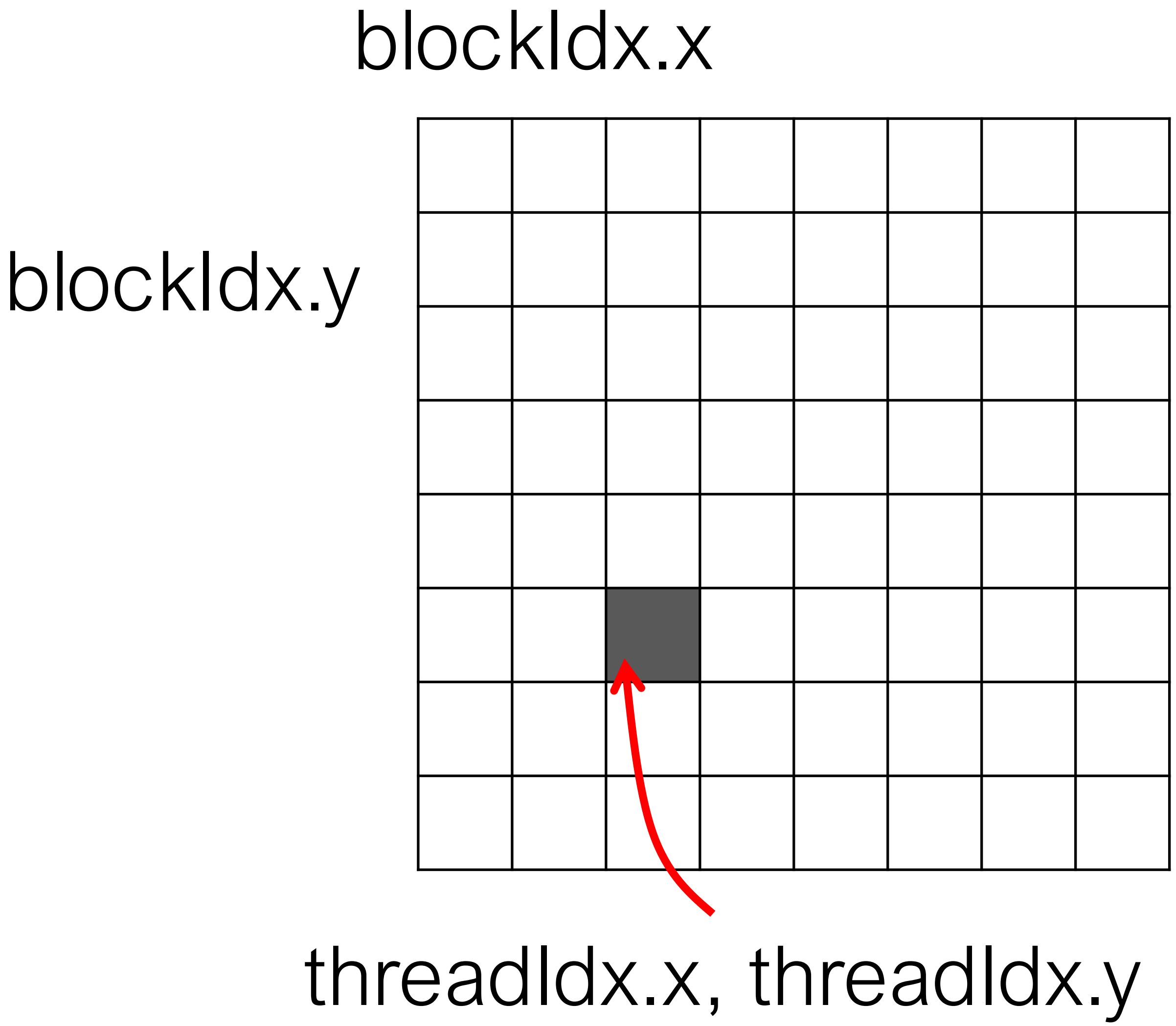
32 bytes apart, need two loads for two threads!

Matrix Transpose Example



Naïve Implementation of Matrix Transpose

```
/* macro to index a 1D memory array with 2D indices in row-major order */
/* ld is the leading dimension, i.e. the number of columns in the matrix */
#define INDX( row, col, ld ) ( ( (row) * (ld) ) + (col) ) __global__ void
naive_cuda_transpose(int m, float *a, float *c ) {
  int myCol = blockDim.x * blockIdx.x + threadIdx.x;
  int myRow = blockDim.y * blockIdx.y + threadIdx.y;
  if( myRow < m && myCol < m ) {
    c[INDX( myCol, myRow, m )] = a[INDX( myRow, myCol, m )];
  } /* end if */
  return;
}
```



how many rows before?
blockDim.y * blockIdx.y + threadIdx.y

how many columns before?
blockDim.x * blockIdx.x + threadIdx.x

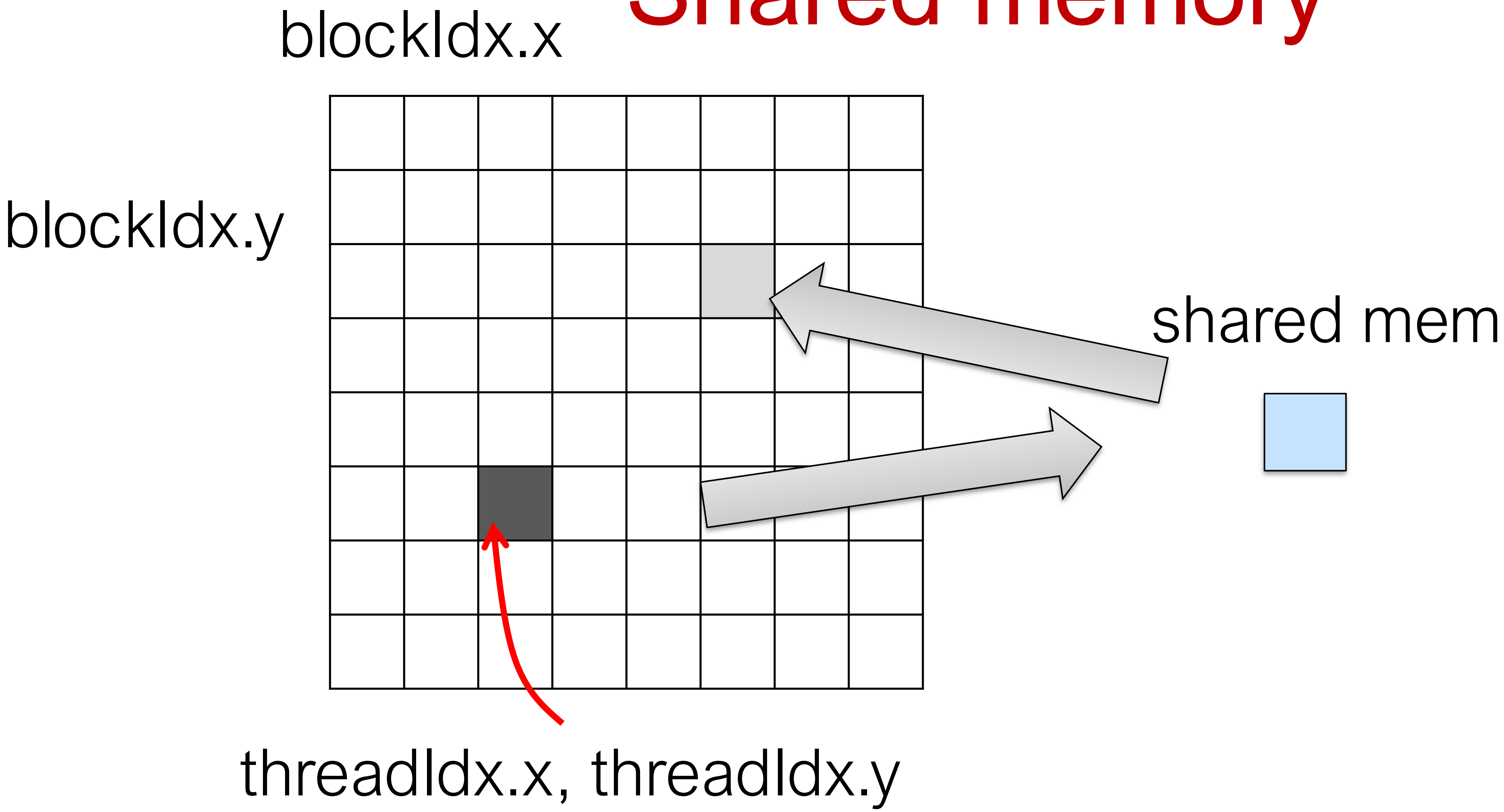
pos in original data array?
[ ] * #cols + [ ]

pos in transposed array?
[ ] * #cols + [ ]

Is memory access coalesced?

```
/* macro to index a 1D memory array with 2D indices in row-major order */
/* ld is the leading dimension, i.e. the number of columns in the matrix */
#define INDX( row, col, ld ) ( ( (row) * (ld) ) + (col) ) __global__ void
naive_cuda_transpose(int m, float *a, float *c ) {
  int myCol = blockDim.x * blockIdx.x + threadIdx.x;
  int myRow = blockDim.y * blockIdx.y + threadIdx.y;
  if( myRow < m && myCol < m ) {
    c[INDX( myCol, myRow, m )] = a[INDX( myRow, myCol, m )];
  } /* end if */
  return;
}
```

Key idea to improve memory efficiency: Use Shared memory

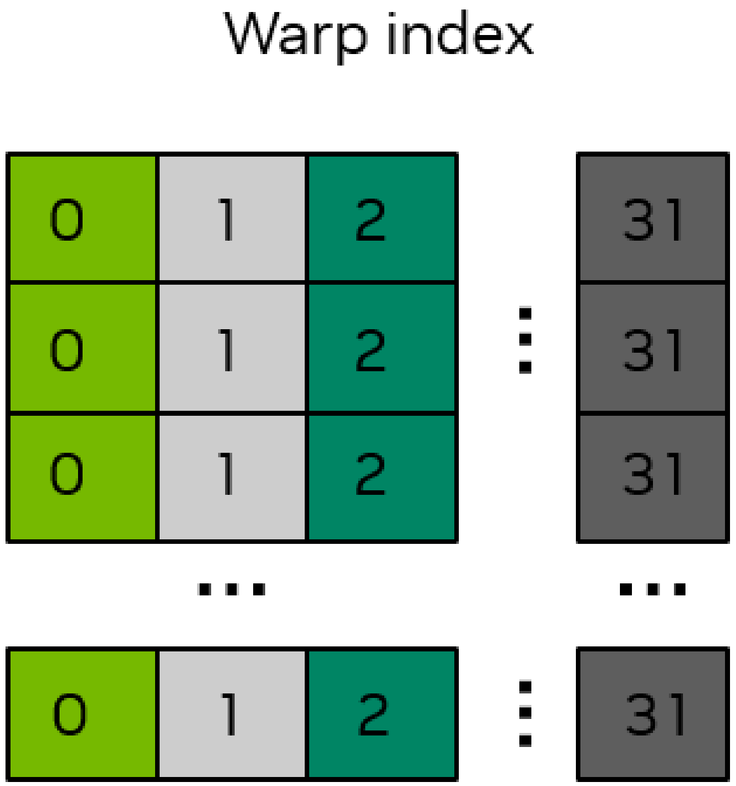


Matrix Transpose (coalesced)

```
__global__ void smem_cuda_transpose(int m, float *a, float *c ) {
  __shared__ float smemArray[THREADS_PER_BLOCK_X][THREADS_PER_BLOCK_Y];
  const int tileCol = blockDim.x * blockIdx.x;
  const int tileRow = blockDim.y * blockIdx.y;

  smemArray[threadIdx.x][threadIdx.y] = a[INDX( tileRow + threadIdx.y, tileCol + threadIdx.x, m )];

  __syncthreads();

  c[INDX( tileCol + threadIdx.y, tileRow + threadIdx.x, m )] = smemArray[threadIdx.y][threadIdx.x];
  return;
}
```

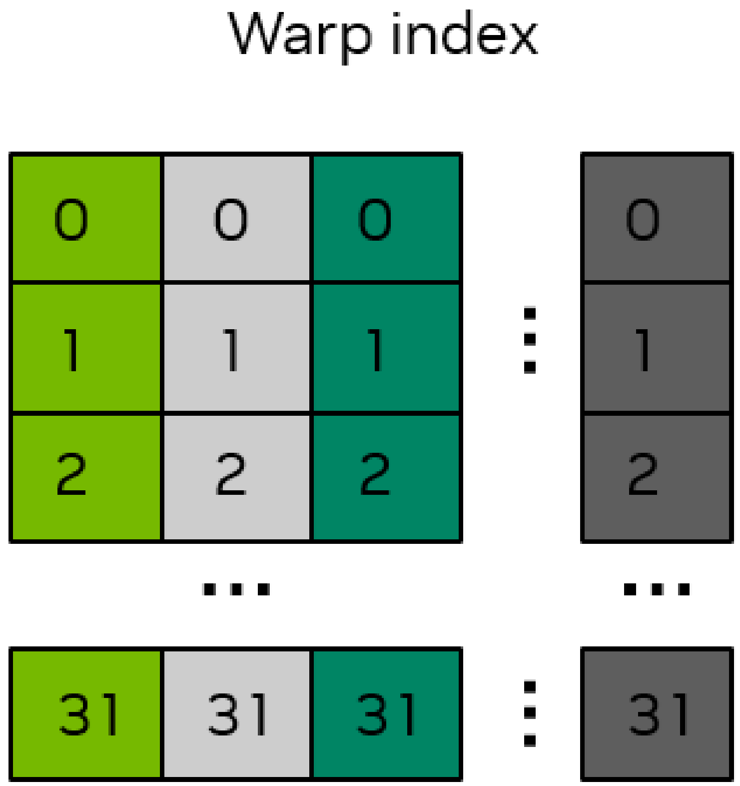
Shared Memory Bank Conflict

Shared memory has 32 banks, each bank stores 4 bytes (32 bits), access to same bank's different element will be sequential



smemArray[threadIdx.x][threadIdx.y]

Bank Conflict, slower



smemArray[threadIdx.y][threadIdx.x]

No Bank Conflict, faster

Matrix Transpose (coalesced)

```
__global__ void smem_cuda_transpose(int m, float *a, float *c ) {
  __shared__ float smemArray[THREADS_PER_BLOCK_X][THREADS_PER_BLOCK_Y];
  const int tileCol = blockDim.x * blockIdx.x;
  const int tileRow = blockDim.y * blockIdx.y;

  smemArray[threadIdx.x][threadIdx.y] = a[INDX( tileRow + threadIdx.y, tileCol + threadIdx.x, m )];

  __syncthreads();

  c[INDX( tileCol + threadIdx.y, tileRow + threadIdx.x, m )] = smemArray[threadIdx.y][threadIdx.x];
  return;
}
```

bank conflict

no bank conflict

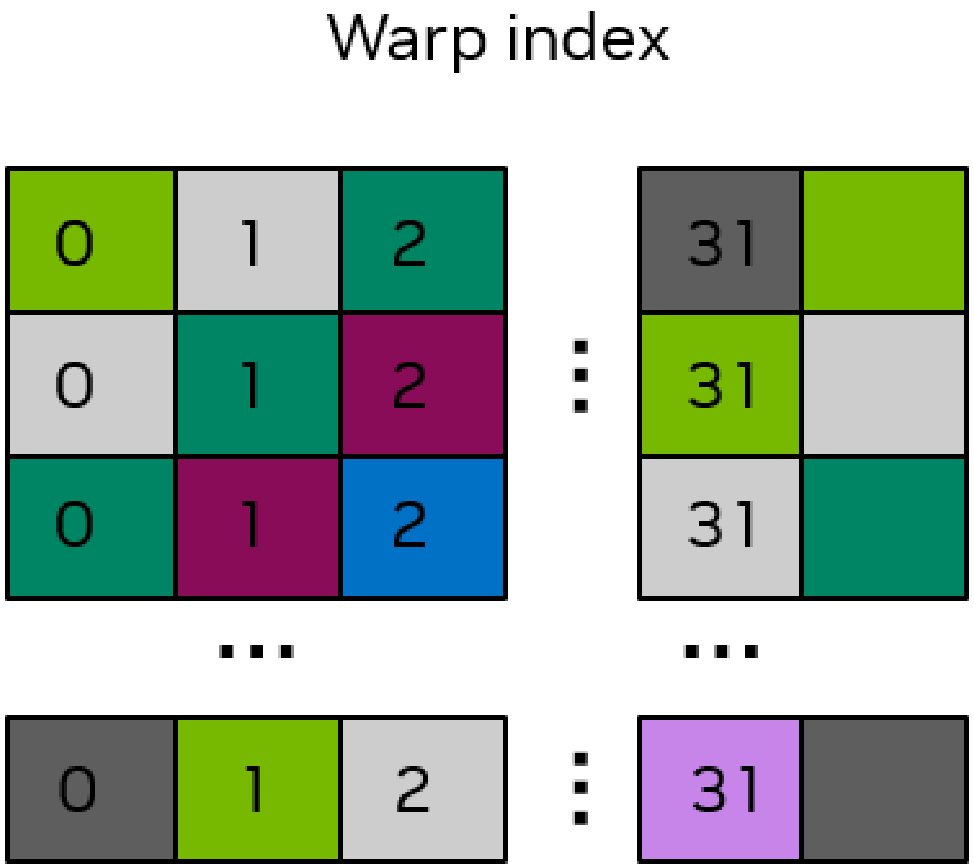
Matrix Transpose (coalesced, no bank conflict)

```
__global__ void smem_cuda_transpose(int m, float *a, float *c ) {
  __shared__ float smemArray[THREADS_PER_BLOCK_X][THREADS_PER_BLOCK_Y+1];
  const int tileCol = blockDim.x * blockIdx.x;
  const int tileRow = blockDim.y * blockIdx.y;

  smemArray[threadIdx.x][threadIdx.y] = a[INDX( tileRow + threadIdx.y, tileCol + threadIdx.x, m )];

  __syncthreads();

  c[INDX( tileCol + threadIdx.y, tileRow + threadIdx.x, m )] = smemArray[threadIdx.y][threadIdx.x];
  return;
}
```

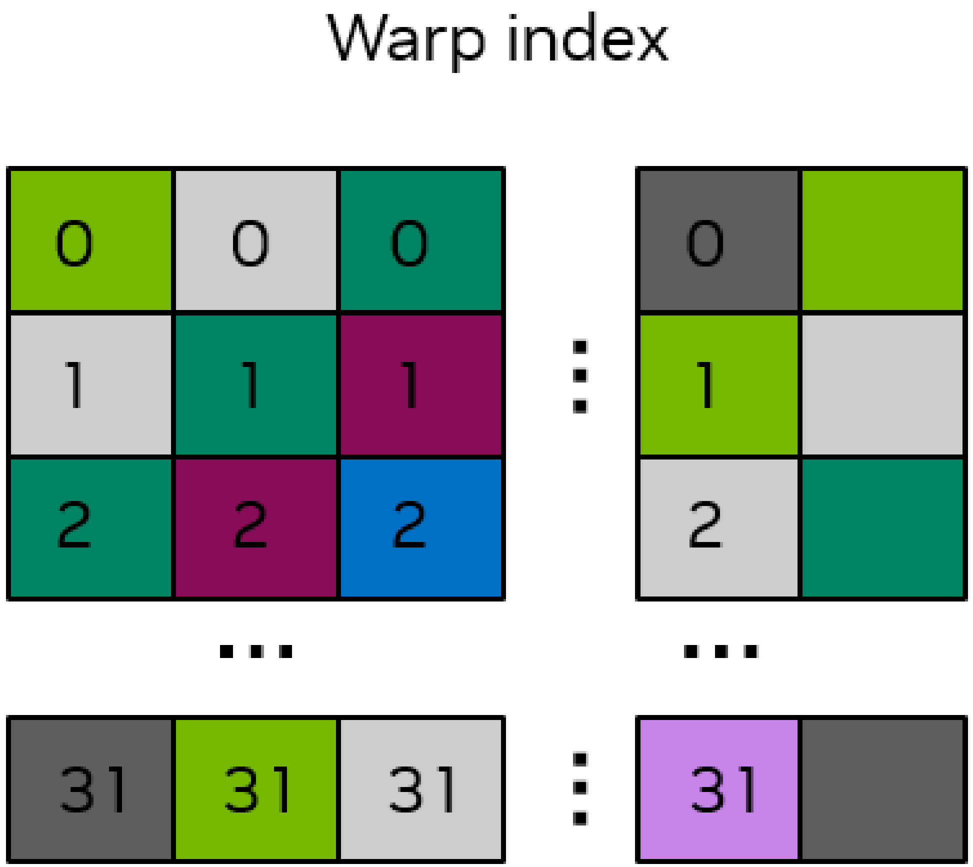
no bank conflict

no bank conflict

New Bank Structure



smemArray[threadIdx.x][threadIdx.y]



smemArray[threadIdx.y][threadIdx.x]

No Bank Conflict, faster

Live Coding Session: Tiled Matrix Multiplication

- Implement a kernel for tiled matrix multiplication

https://github.com/llmsystem/llmsys_code_examples/blob/main/cuda_acceleration_demo/matmul_tile.cu

Tiled Matrix Multiplication

```
#define TILE_WIDTH 2
__global__ void MatMulTiledKernel(float* d_A, float* d_B, float* d_C, int N) {
        __shared__ float As[TILE_WIDTH][TILE_WIDTH];
        __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];

        // Determine the row and col of the P element to be calculated for the thread
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;
        float Cvalue = 0;
        for(int ph = 0; ph < N/TILE_WIDTH; ++ph) {
                As[threadIdx.y][threadIdx.x] = d_A[row * N + ph * TILE_WIDTH + threadIdx.x];
                Bs[threadIdx.y][threadIdx.x] = d_B[(ph * TILE_WIDTH + threadIdx.y) * N + col];
                __syncthreads();
                for(int k = 0; k < TILE_WIDTH; ++k) {
                        Cvalue += As[threadIdx.y][k] * Bs[k][threadIdx.x];
                }
                __syncthreads();
        }
        d_C[row * N + col] = Cvalue;
}
```

Outline: GPU Acceleration techniques

- Tiling (Chap 5)
- Memory parallelism (Chap 5 & 6)
- Sparse Matrix Multiplication
- cuBLAS

Sparse Matrix - CSR

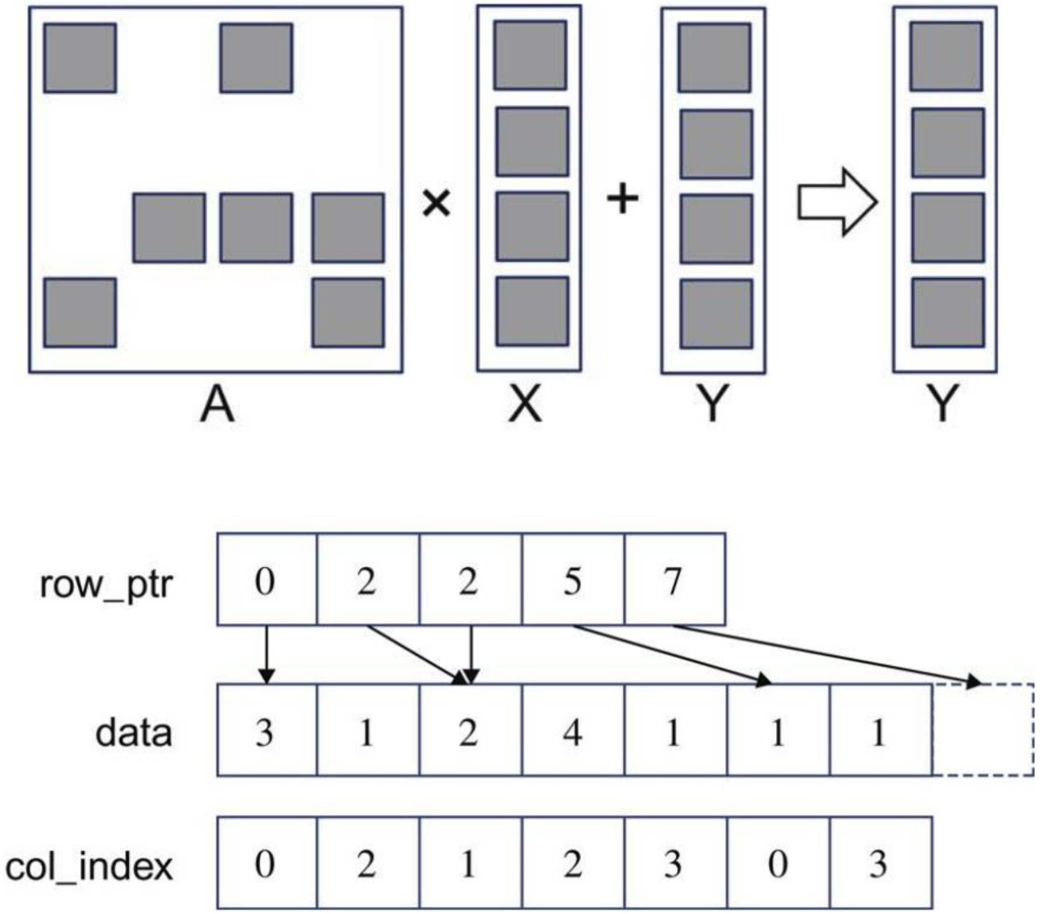
Sparse matrix

	0	1	2	3
0	a		b	c
1		d		
2			e	f
3				g

Compressed Sparse Row (CSR)

Row pointers	0	3	4	6	7		
Column offsets	0	2	3	1	2	3	3
Data	a	b	c	d	e	f	g

Sparse Matrix-Vector Multiplication




```
for(int row = 0; row < n; row++) {
        float dot = 0;
        int row_start = row_ptr[row];
        int row_end = row_ptr[row + 1];
        for(int el = row_start; el < row_end; el++)
        {
                dot += x[col_index[el]] * data[el];
        }
        y[row] += dot;
}
```

Sparse Matrix-Vector Multiplication

```
__global__ void SpMVCSRKernel(float *data, int *col_index, int *row_ptr, float *x, float *y, int
num_rows) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if(row < num_rows) {
        float dot = 0;
        int row_start = row_ptr[row];
        int row_end = row_ptr[row + 1];
        for(int elem = row_start; elem < row_end; elem++) {
            dot += x[col_index[elem]] * data[elem];
        }
        y[row] += dot;
    }
}
```

Outline: GPU Acceleration techniques

- Tiling (Chap 5)
- Memory parallelism (Chap 5 & 6)
- Sparse Matrix Multiplication
- cuBLAS

cuBLAS

- CUDA Basic Linear Algebra Subroutine library
- a lightweight library dedicated to GEneral Matrix-to-matrix Multiply (GEMM) operations

cuBLAS APIs

- must call before:

```
cublasStatus_t cublasCreate(cublasHandle_t *handle)
```

- must call after:

```
cublasStatus_t cublasDestroy(cublasHandle_t handle)
```

- float vector dot product

```
cublasStatus_t cublasSdot (cublasHandle_t handle, int n,
    const float *x, int incx,
    const float *y, int incy,
    float *result)
```

cuBLAS APIs

- Matrix vector product $y = \alpha A \cdot x + \beta y$

```
cublasStatus_t cublasSgemv(cublasHandle_t handle,
    cublasOperation_t trans,
    int m, int n,
    const float *alpha,
    const float *A, int lda,
    const float *x, int incx,
    const float *beta,
    float *y, int incy)
```

cuBLAS APIs

- Matrix matrix multiplication: $C = \alpha A \cdot B + \beta C$

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
  cublasOperation_t transa,
  cublasOperation_t transb,
  int m, int n, int k, const float *alpha,
  const float *A, int lda,
  const float *B, int ldb,
  const float *beta,
  float *C, int ldc)
```

Summary of GPU Acceleration

- Tiling for efficient matrix computation
- Coalesced memory access
- Sparse matrix representation and multiplication
- cuBLAS
  - readily available vector, matrix-vector, matrix-matrix operations

Quiz

- https://canvas.cmu.edu/courses/51352/quizzes/155251