cs475: CUDA 1 code

wim bohm

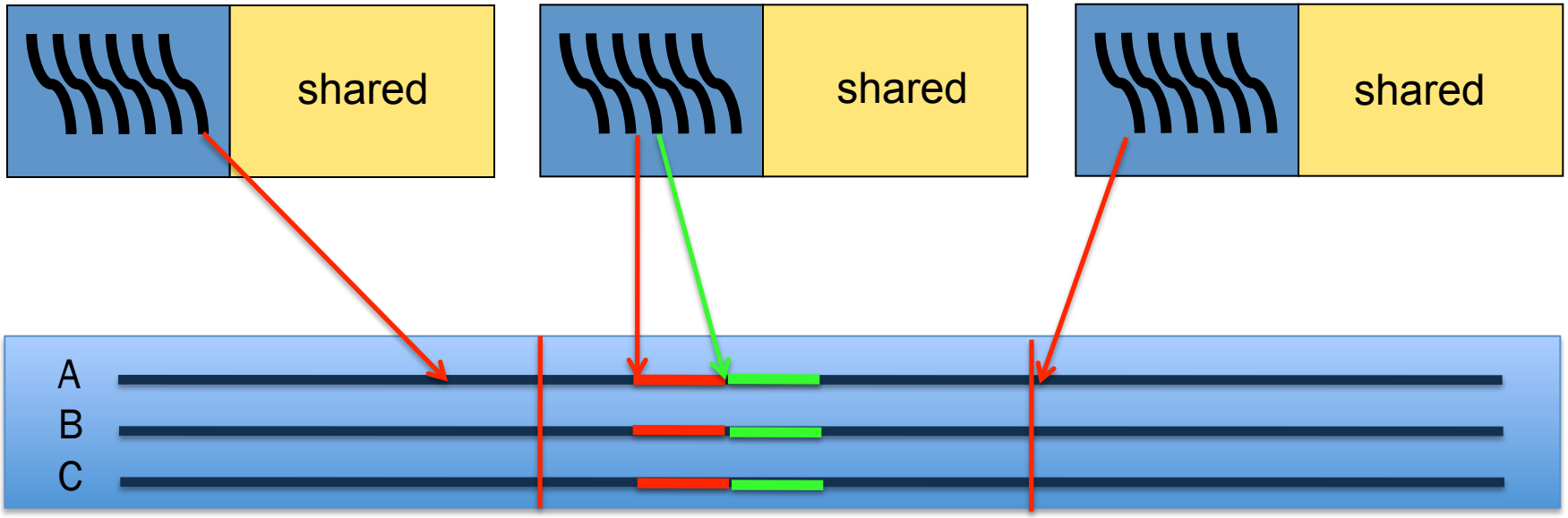
cs, csu

vector add

Thread blocks access contiguous partitions of A, B, and C

Threads access contiguous chunks in a partition

grid: 1D, threadBlock: 1D



Global Memory

host

- create host and device vectors
- memcpy input vectors to device
- invoke <<<gridDim,blockDim>>>kernel(params)
- do timing
- memcpy result vectors back to host
- check results

host: cmd line interface

```
// Defines
#define GridWidth 60
#define BlockWidth 128

// Host code performs setup and calls the kernel.
int main(int argc, char** argv) {
    int ValuesPerThread; // number of values per thread
    int N;                          // total Vector size
    sscanf(argv[1], "%d", &ValuesPerThread);
    N = ValuesPerThread * GridWidth * BlockWidth;
    size_t size = N * sizeof(float);   // number of bytes for a vector.
    dim3 dimGrid(GridWidth);        // grid dimensions
    dim3 dimBlock(BlockWidth);    // thread block dimensions
```

host: allocate

```
// Allocate input vectors h_A and h_B in host memory
h_A = (float*)malloc(size);
if (h_A == 0) Cleanup(false);
h_B = (float*)malloc(size);
if (h_B == 0) Cleanup(false);
h_C = (float*)malloc(size);
if (h_C == 0) Cleanup(false);

// Allocate vectors in device memory.
cudaError_t error;
error = cudaMalloc((void**)&d_A, size);
if (error != cudaSuccess) Cleanup(false);
error = cudaMalloc((void**)&d_B, size);
if (error != cudaSuccess) Cleanup(false);
error = cudaMalloc((void**)&d_C, size);
if (error != cudaSuccess) Cleanup(false);

// intialize host vectors h_A and h_B
```

host: copy host data to device invoke kernel on device copy device data to host

```
// Copy host vectors h_A and h_B to device vectors d_A and d_B
error = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
if (error != cudaSuccess) Cleanup(false);
error = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
if (error != cudaSuccess) Cleanup(false);

// Invoke kernel
AddVectors<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, ValuesPerThread);
error = cudaGetLastError();
if (error != cudaSuccess) Cleanup(false);

// Copy result from device memory to host memory
error = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
if (error != cudaSuccess) Cleanup(false);
```

device: kernel

```
// Device code.
// N: values per thread
__global__ void AddVectors(const float* A, const float* B, float* C, int N)
{
    int blockStartIndex  = blockIdx.x * blockDim.x * N;
    int threadStartIndex = blockStartIndex + (threadIdx.x * N);
    int threadEndIndex   = threadStartIndex + N;
    int i;

    for( i=threadStartIndex; i<threadEndIndex; ++i ){
        C[i] = A[i] + B[i];
    }
}
```

vecadd: your job

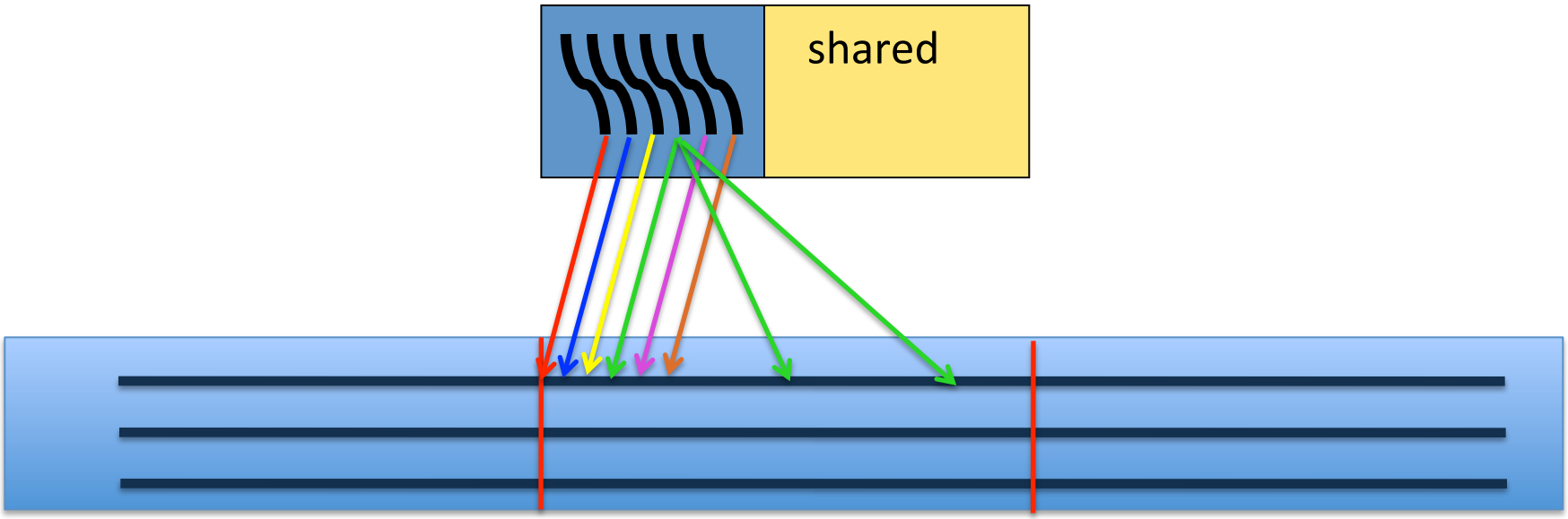
- The code described does not coalesc

  WHY?

- What do you need to do to make it coalesc?

  Consecutive threads read

  consecutive memory locations.



Shared Shared Matrix Multiply

- Generic case:

  Problem:

  A is an m.k matrix, B is a k.n matrix → C is an m.n matrix

  GridBlock:

  computes a block of C

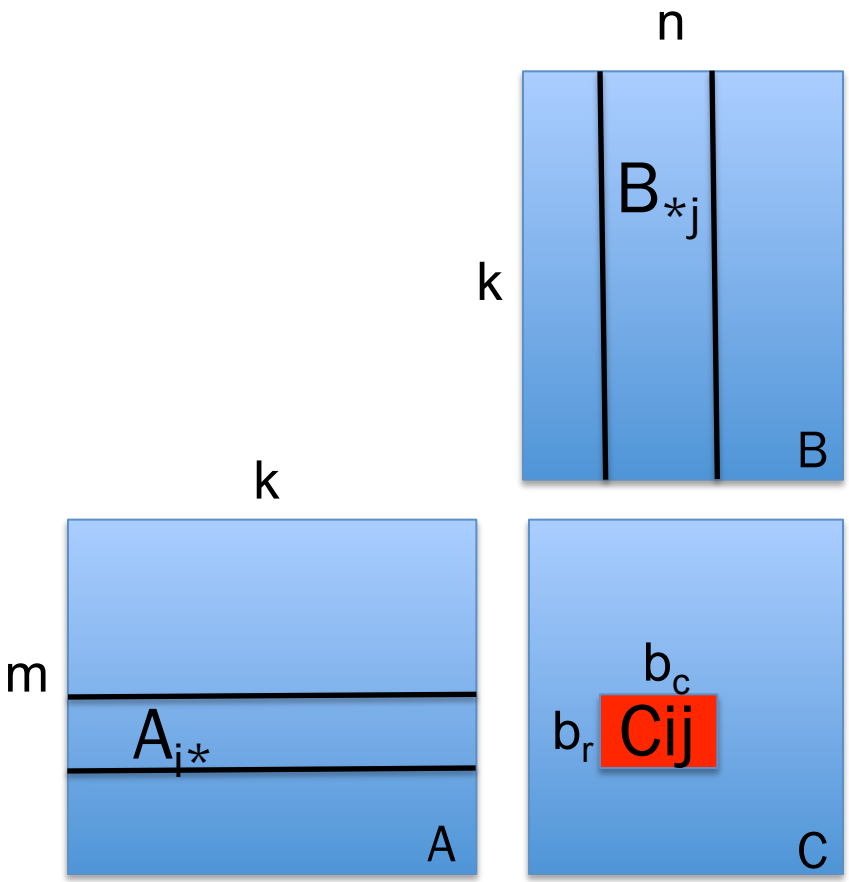
  with a certain "footprint" b_r . b_c

  using a thread block of t_r . t_c threads

- We cannot choose m, k, and n

  but we can choose b_r and b_c, and t_r and t_c

- **How does the complexity of the algorithm change with b_r, b_c, t_r, t_c ?**



Each thread block with blockIdx = (i,j) computes block Cij of size b_r . b_c

each thread block
- computes $k.b_r.b_c$ multiply adds
- communicates $k.b_r + k.b_c$ values

The whole computation takes
- $m.n / b_r.b_c$ C blocks
- computes m.k.n multiply adds
- communicates m.k.n $(b_r+b_c)/ b_r.b_c$ values

We cannot optimize the computes, but we can optimize the communicates.

Communication

- Communication takes $m.k.n\ (b_r+b_c) / b_r.b_c$

- Some extremes:
  - $b_r=m$ and $b_c=n$ (1 threadblock)

    comms = m.k + n.k (like sequential matmult)
  - $b_r=1$ and $b_c=1$ (fine grain)

    comms = 2.m.k.n (1 order of magnitude higher)

Optimizing Communication

- Communication takes $m.k.n\ (b_r+b_c) / b_r.b_c$
- What is the optimum Cij tile shape?

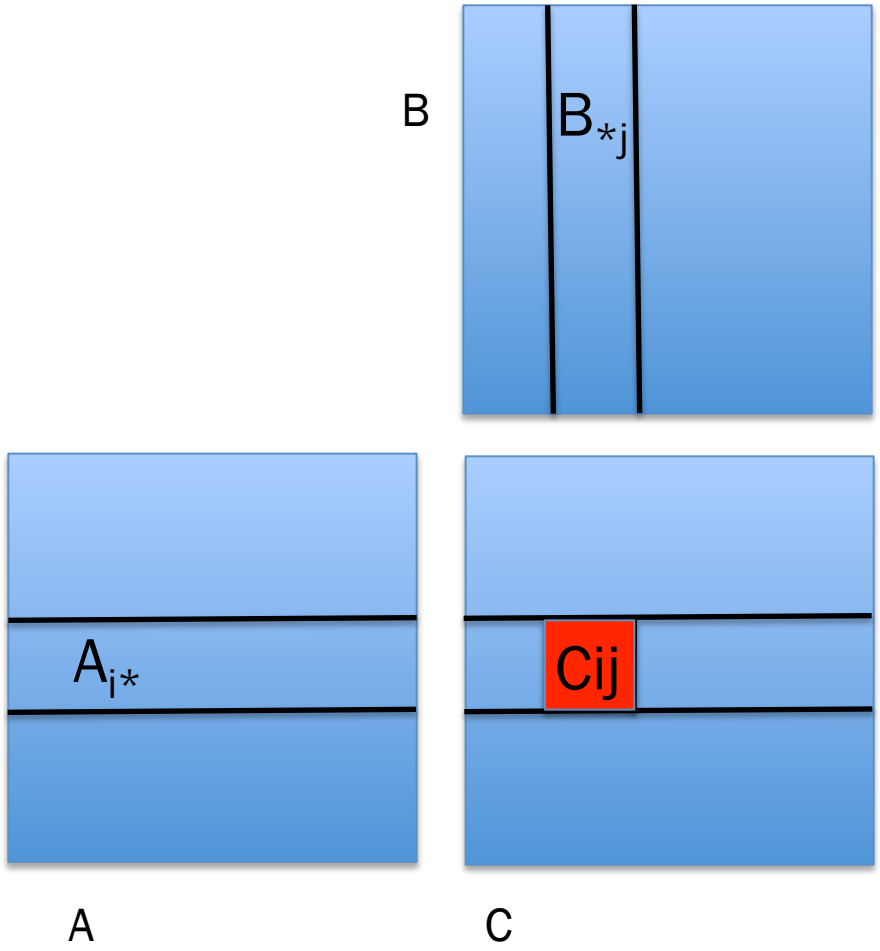
  minimize $(b_r+b_c) / b_r.b_c$

  $b_r=s+h \quad b_c=s-h$

  $b_r+b_c= 2.s \quad b_r.b_c = s^2-h^2$

  minimal when $h = 0$ → square tile !

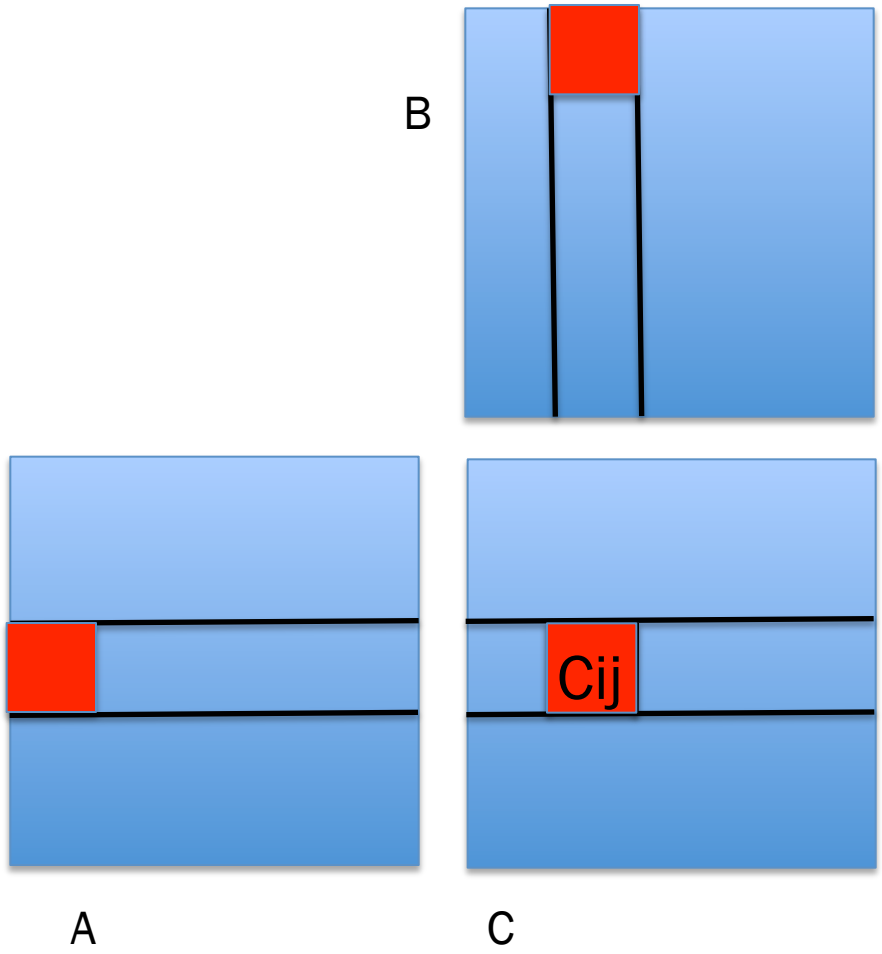
shared / shared matmult



- A and B in global memory
- blocks of A and B copied into shared memory
- 2D grid of 2D thread blocks, each 16x16 thread block computes a 16x16 C block

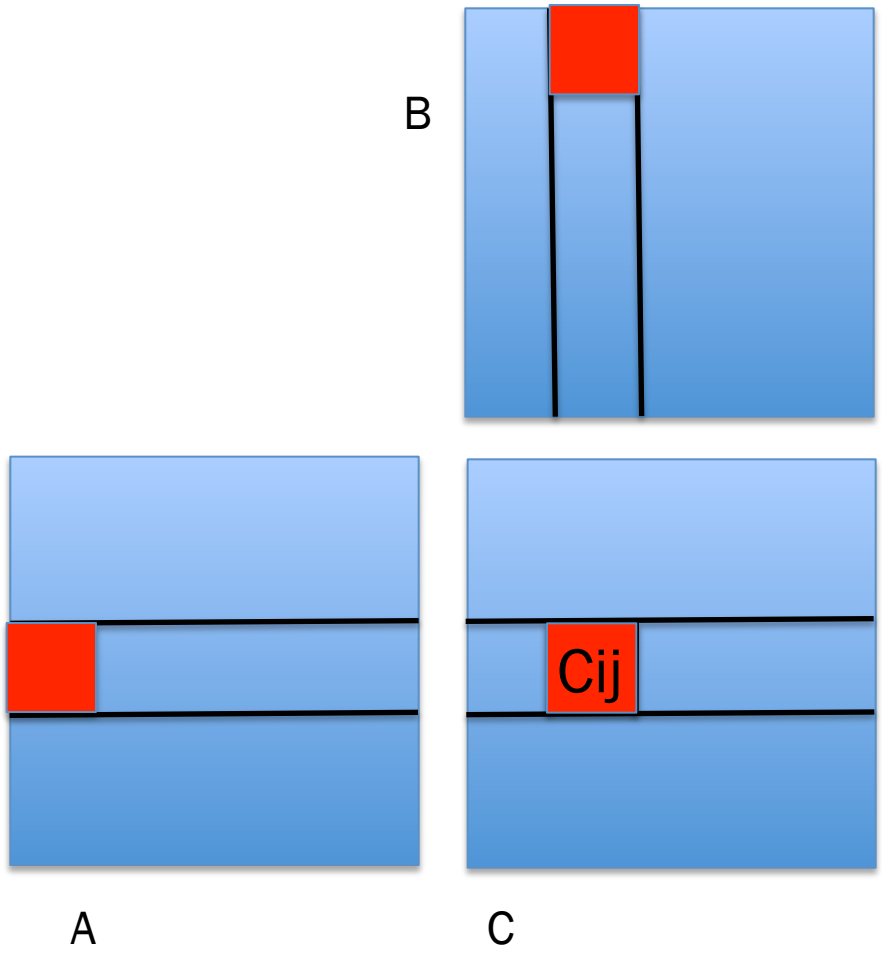
  ($t_r=b_r$, $t_c=b_c$)
- C elements: registers

shared / shared matmult



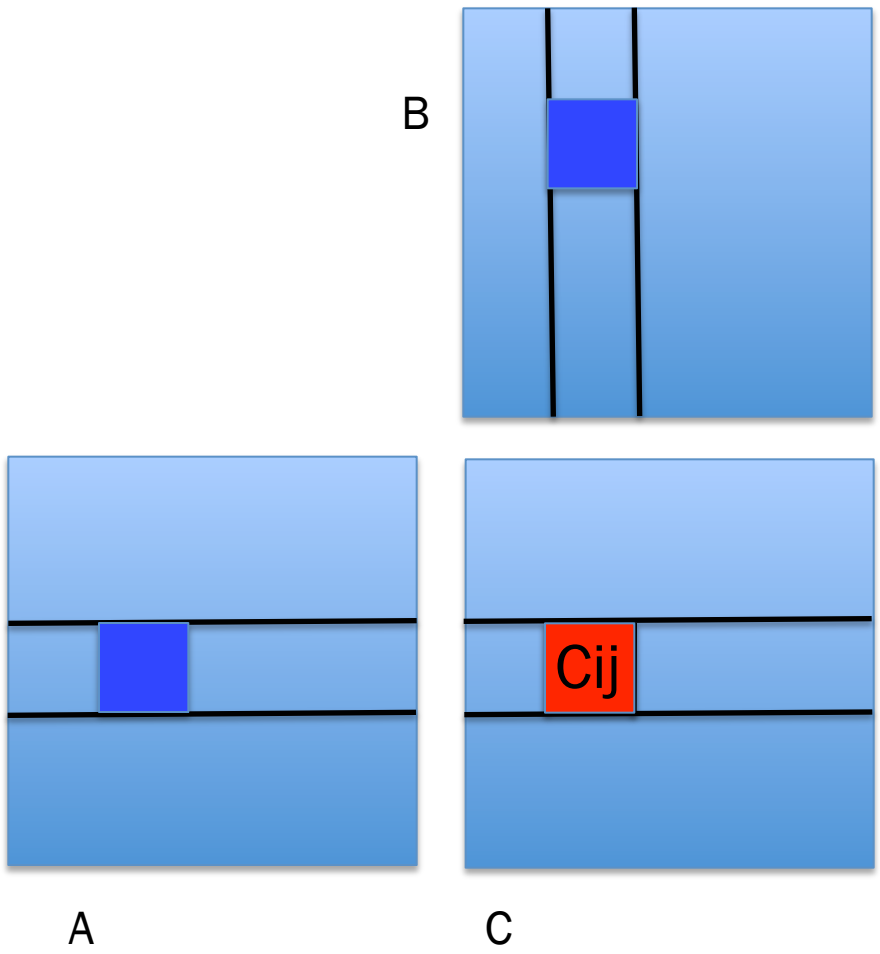
- A and B in global memory
- 2D grid, each 16x16 thread block computes a 16x16 C block
  - coalesced fetch a 16x16 A block into shared memory
  - coalesced fetch a 16x16 B block into shared memory

shared / shared matmult



- A and B in global memory
- 2D grid, each 16x16 thread block computes a 16x16 C block
  - coalesced fetch a 16x16 A block into shared memory
  - coalesced fetch a 16x16 B block into shared memory
  - each thread computes one inner product adding it to the one C element it is responsible for

shared / shared matmult



- etcetera
- initial code from CUDA Programming Guide

struct Matrix

```
// Matrices are stored in row major order:
// M[row,col] = *(M.elements + row * M.stride + col)
// A sub matrix takes the stride of the total matrix avoiding copies.
// Matrix is really a MATRIX DESCRIPTOR. Stride is the number of bytes
// from one element of the matrix to the element in the same column but
// one row down. It is present so that we may pad the rows in a matrix
// with additional bytes in order to control word boundary alignment for
// the elements of the matrix.

typedef struct {
  int width;
  int height;
  int stride;
  float* elements;
} Matrix;
```

host

- create host matrices h_A, h_B, h_C
- create device matrices d_A, d_B, d_C

  memcpy h_A → d_A

  memcpy h_B → d_B
- call device kernel
- do timing
- memcpy d_C → h_C
- check results

device kernel code

```
for (int m = 0;  m < (A.width / BLOCK_SIZE); ++m){
   // Get Asub and Bsub
      …
   // Notice: every thread declares shared_A and shared_B in shared memory
   //         even though a thread block has only one shared_A and one shared_B
   __shared__ float shared_A[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ float shared_B[BLOCK_SIZE][BLOCK_SIZE];

  // Each thread copies just one element of shared_A and one element of shared_B
   shared_A[thread_row][thread_col] = GetElement(Asub, thread_row, thread_col);
   shared_B[thread_row][thread_col] = GetElement(Bsub, thread_row, thread_col);
  // Synchronize to ensure all elements are read
   __syncthreads();

  //   do an inproduct into Cvalue
  //   a thread can use all shared_A row values and shared_B col values it needs
   __syncthreads();
}
write Cvalue back to global memory
```

device kernel: inproduct

```
    // Do an inproduct of one row of shared_A and one col of shared_B
    // computing one Cvalue by accumulation
#pragma unroll
    for(int e=0; e<BLOCK_SIZE; ++e)
       Cvalue += shared_A[thread_row][e] * shared_B[e][thread_col];
```

your job: double b_r and b_c

- $b_r=b_c=32$ but keep $t_r=t_c=16$
  (less thread overhead)
- each thread is now responsible for 4 C values
  e.g. $thread_{i,j}$ computes

$$C_{i,j} \quad , \quad C_{i,j+16}$$
$$C_{i+16,j} \, , \quad C_{i+16,j+16}$$

- Do this in one in-product loop (less loop overhead)
  (what are the loop bounds now?)