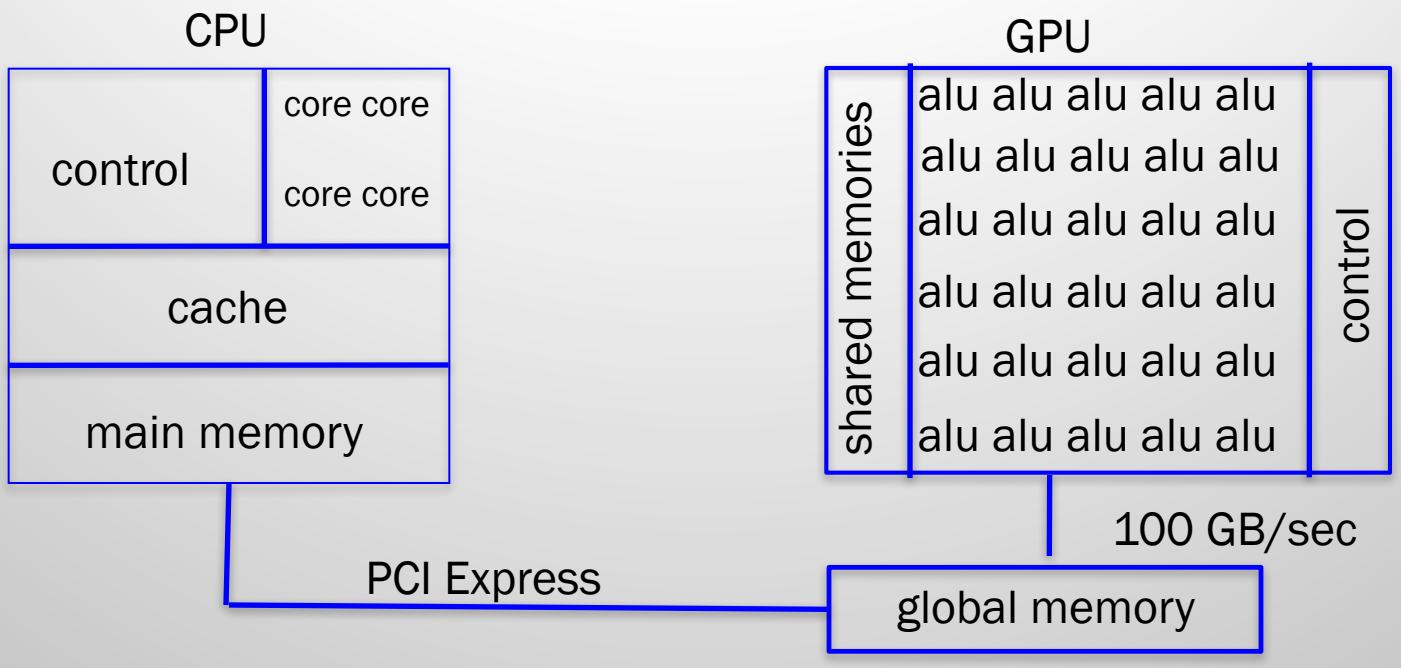
Cuda

Wim Bohm, CS CSU

CUDA Architecture



CPU versus GPU

- CPU
  - small number of cores, in our example 4
  - large amount of control to deal with Instruction Level Parallelism (instruction scheduling)
  - Cache (L1, L2, ..) and its control
  - small fraction of the CPU area is dedicated to compute resources (ALUs) .
- GPU
  - mainly ALUs (100s, varying per GPU type), some control
  - some user programmable cache (called "shared memory")
  - on some GPUs there is also implicit cache

GPU architecture

- Streaming Multiprocessors (SMs) have scalar processors (ALUs) and special function units (sqrt and reciprocal)
  - each SM has programmable (under control of the program) cache, called shared memory, and local memory: a large register file, used for storing local program variables
- the GPU is connected to a global memory by a high speed on-chip interconnection network
  - this global memory is connected to the host memory by a PCI express bus.

NVidia GPU


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

GPU programming model

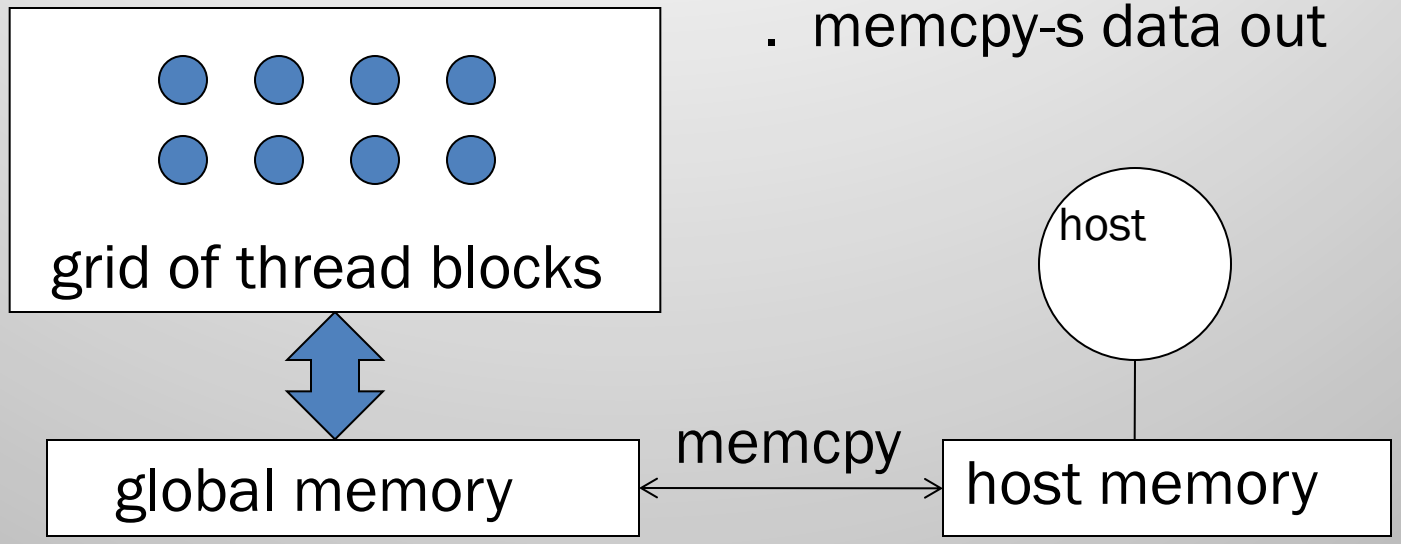
grid of thread blocks

- (potentially multiple) thread blocks run on an SM
- threads in a thread block share data in shared memory

host treats GPU as co-processor

- memcpy-s data in
- launches grids of thread blocks of **kernel**s on SMs

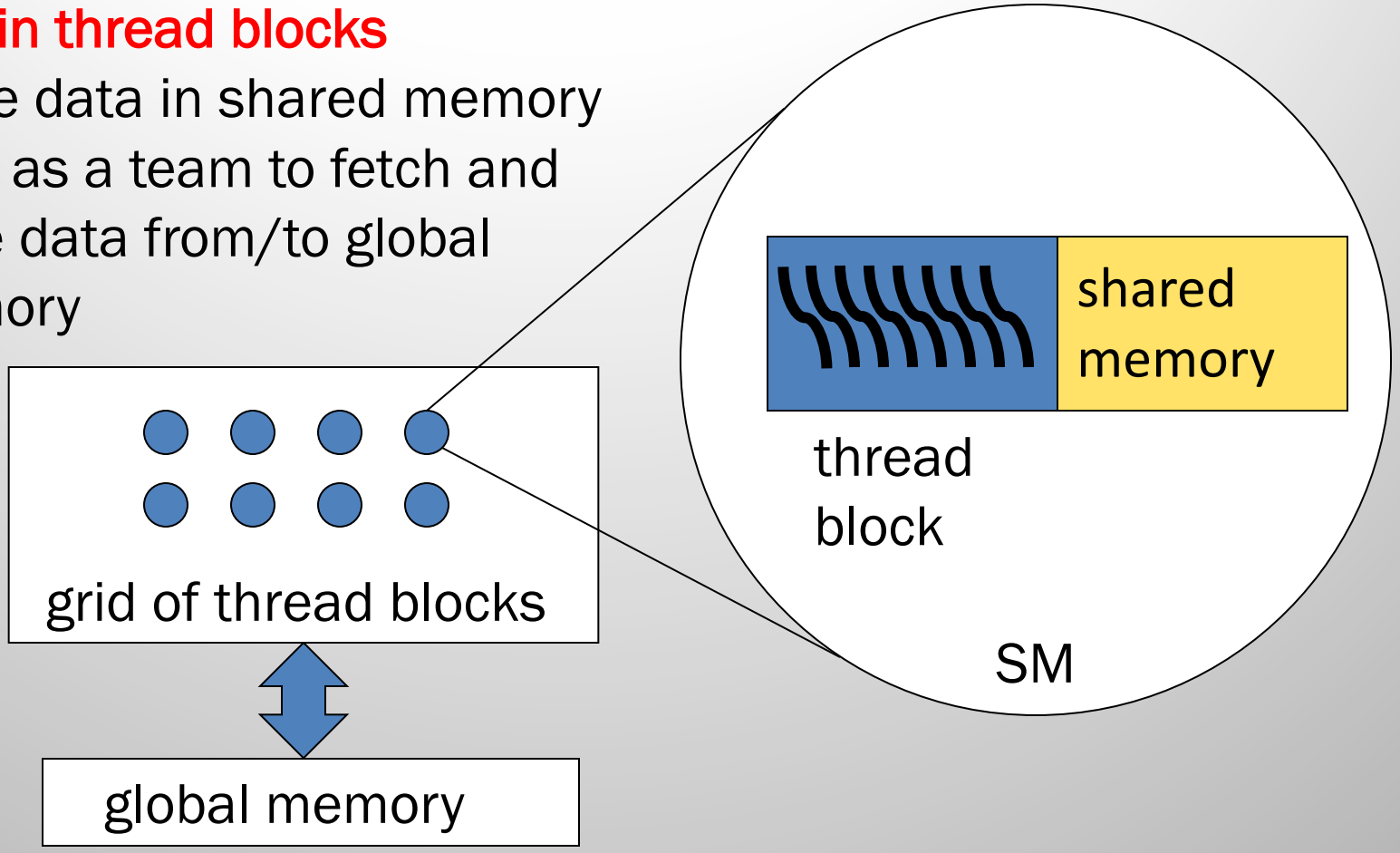
  kernel executed in SIMD fashion
- memcpy-s data out



GPU programming model

threads in thread blocks

- share data in shared memory
- work as a team to fetch and store data from/to global memory



questions…

- How do thread-blocks get allocated on stream multiprocessors?
- How do threads synchronize / communicate?
- How do thread blocks synchronize / communicate?
- How do threads disambiguate memory accesses?
  - which thread reads / writes which memory location?

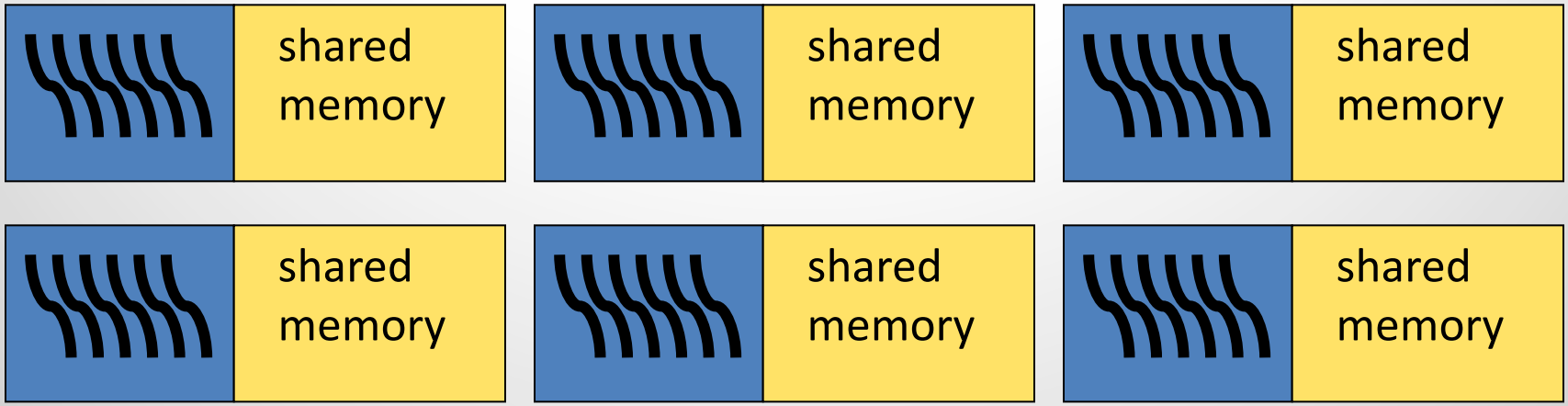
thread allocation

- A thread block can get allocated on **any** SM and thread blocks are independent of each other, i.e., they cannot communicate with each other at all.
  - **pro:** now the computation can run on any number of stream processors
  - **con**: this makes programming a GPU more restrictive
- multiple thread blocks can be scheduled on one multiprocessor, if resources allow it. They still are independent of each other.

Thread synchronization

- threads **inside** one thread block can synchronize
  - _syncthreads() command
  - Why would that be necessary?
- host can synchronize kernel calls
  - either explicitly through cudaThreadSynchronize()
  - or implicitly through memcpy()-s

threads and memory access



- each thread **block** has 2D (x,y) **block-indices** in the grid
- each **thread** has 3D (p,q,r) **thread-indices** in the block
- so each thread has **its own identity** **based on** (x,y,p,q,r)
  - and can therefore decide which memory locations to access (**responsibility of the programmer**)

Consequences

- There is no sharing or synchronization between thread blocks. So
  - the thread blocks can be scheduled in any (parallel or sequential) order
  - this allows for scalability: a program can be run on a GPU with any number of multiprocessors, at a price: the user is responsible for breaking the problem up in independent tasks

Programming CPU + GPU

- At CPU host level, the program is sequential with Grid kernel invocations to the GPU.
- A grid is a user definable 1D or 2D hierarchy of grid blocks, each grid block being a user definable 1D, 2D or 3D block of threads.
- Communication, via shared memory, and synchronization are only possible inside a user defined thread block.

Declaring Grid and block dimensions

- The host code does a kernel call. In this call it defines grid and thread block dimensions
  - **kernelName<<<gridDims,threadDims>>> (params)**

- Grid and block dimensions are declared using variables of predefined type **dim3**
  - with three fields: x, y and z
  - also used for lower dimensional cases

Built-in variables

- In the kernel a set of built-in variables specifies the grid and block dimensions (Dim) and indices (Idx).
- These can be used to determine the thread ID
  - **gridDim** contains .x and .y grid dimensions (sizes)
  - **blockIdx** contains block indices .x and .y in the grid
  - **blockDim** contains the thread block .x, .y, .z dimensions (sizes)
  - **threadIdx** contains .x, .y and .z thread block indices
- Think back of MPI (1D: size, rank). Now we have a 5D size, rank space.

thread-in-block ID (row major order)

- 1D thread block:

  ID = threadIdx.x

- 2D thread block:

  ID = threadIdx.x + threadIdx.y*blockDim.x

- 3D thread block:

  ID = threadIdx.x + threadIdx.y*blockDim.x +
  threadIdx.z*blockDim.x*blockDim.y

BUT

does this create a unique ID for each thread in the grid?

Example vecadd1:
1D grid, 1D thread Block, 1 add per thread

host:

```
vecAdd1<<<blocksPerGrid,threadsPerBlock>>>(A,B,C);
```

kernel: (each thread determines the C value it needs to compute)

```
__global__ void vecAdd1(float* A, float* B, float* C)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  C[i]=A[i]+B[i];
}
```

Executing a kernel: SIMD style

In thread blocks multiples of 32 threads form a warp.

- A warp consists of threads with consecutive thread IDs
- A warp is the unit of execution: one instruction of a warp is executed, then 1 instruction of a next warp is executed
- Because there are eight ALUs, a warp takes 4 cycles to execute. Shared memory access takes 2 to 4 cycles, so warp execution provides memory latency hiding
- In case of conditionals, branch divergence occurs:
  - then and else branches are executed sequentially
  - this occurs within a warp
  - different warps execute their conditionals independently
  - costly, so avoid conditionals as much as possible!

memory model: private memory

- each thread has private (or local) memory

  it is used for local variables of the thread
- private memory is first allocated in registers

  (there are 16K registers in a thread block, they are used for all the threads)
- if the threads need more private memory than there are registers, local memory is spilled **to global memory** with serious performance consequences
- hence the makefile in your PAs employs an option to show register use: be aware of register pressure

memory model: shared memory

- Threads in a thread block share a **shared memory (programmable cache)**. The program explicitly declares variables (usually arrays) to live in shared memory. Access to shared memory is faster than to global memory, but slower than to registers
- **Team work in thread block**:

  Different threads may read different elements into shared memory, but all threads can access all shared memory locations. We use this in e.g. matrix multiply.

Memory model: global memory

- The host memcpy-s data in and out of global memory
- All threads in all thread blocks can access all global memory locations
- Global memory is persistent across thread block activations
- Global memory is persistent across kernel calls
- There are other forms of global memory (constant, texture) that we will not discuss

Coalesced Global memory access

- Global memory is the slowest memory on the GPU
- Coalescing improves memory performance; it occurs when multiple (row major order) **consecutive** threads (IDs) read / write **consecutive** data items from / to global memory
- A number of global array elements can be accessed at once: coalescing produces vectorized accesses that are much faster than element wise accesses. See Programming Guide section 5.3.2: device memory accesses
- This is very important for high speed GPU computing, and the subject of your first CUDA lab and first PA.

Access patterns for coalescing

- The simplest access pattern: consecutive thread IDs access consecutive global memory locations. This is what we will concentrate on.
- Different GPU versions allow more or less complicated access patterns to be coalesced. (See the programming guide for this.)
- We don't expect you to need more complex access patterns, as, once data is read into shared memory, all threads can read all data

Machine dimensions

- Machine dimensions vary per GPU design
  - Grid / thread block dimensions
  - # SMs
    - # registers (local memory) per SM
    - Size of the shared memories
    - Shared memory delay
    - Warp size
  - Global memory size
- Our current GPUs (GeForce GTX 1060):
  - 3D grid of 3D thread blocks
  - 10 SMs
  - 64 K regs / SM
  - 96 KB shared memory, BUT <= 48 KB / thread block

Cuda lab exercise: coalescing

Vector add

We will give you a non coalescing code, and you need improve and report its performance by turning it into a coalescing code

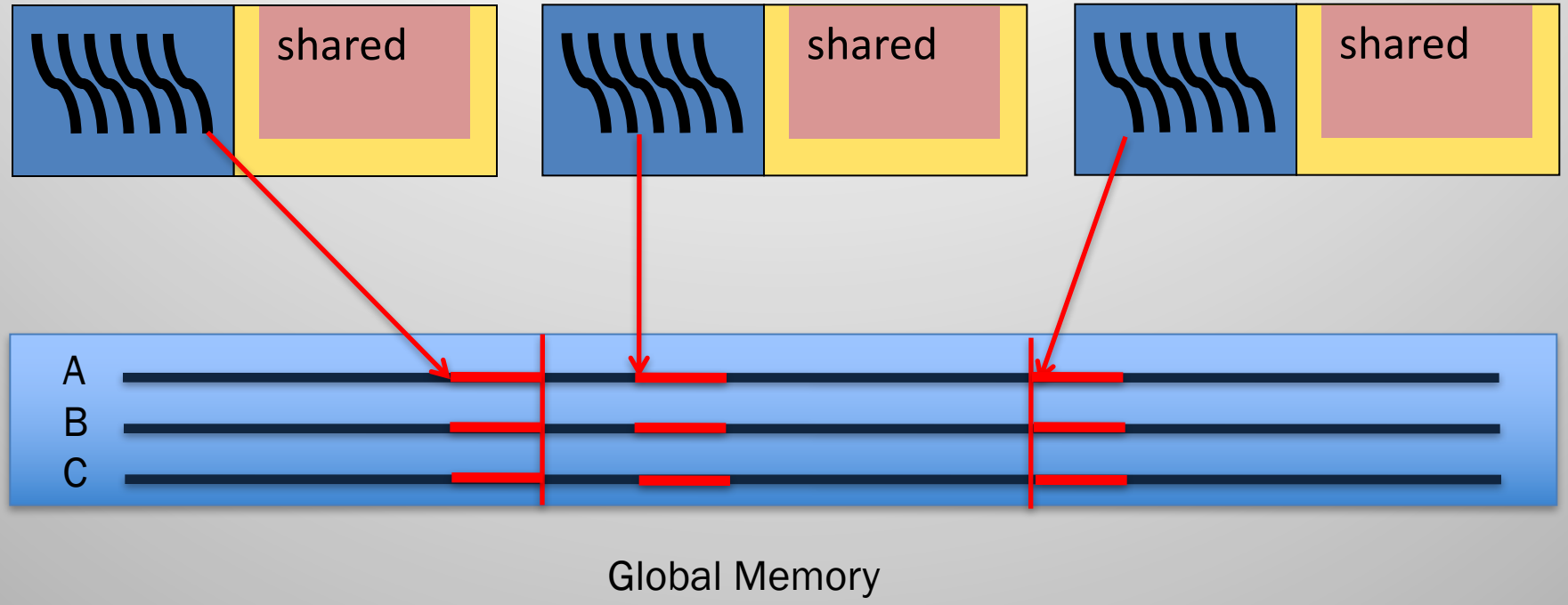
1a: vector add

Threads add a number of elements together

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

Threads access contiguous chunks in a partition

Does this coalesce? How do you make it coalesce?



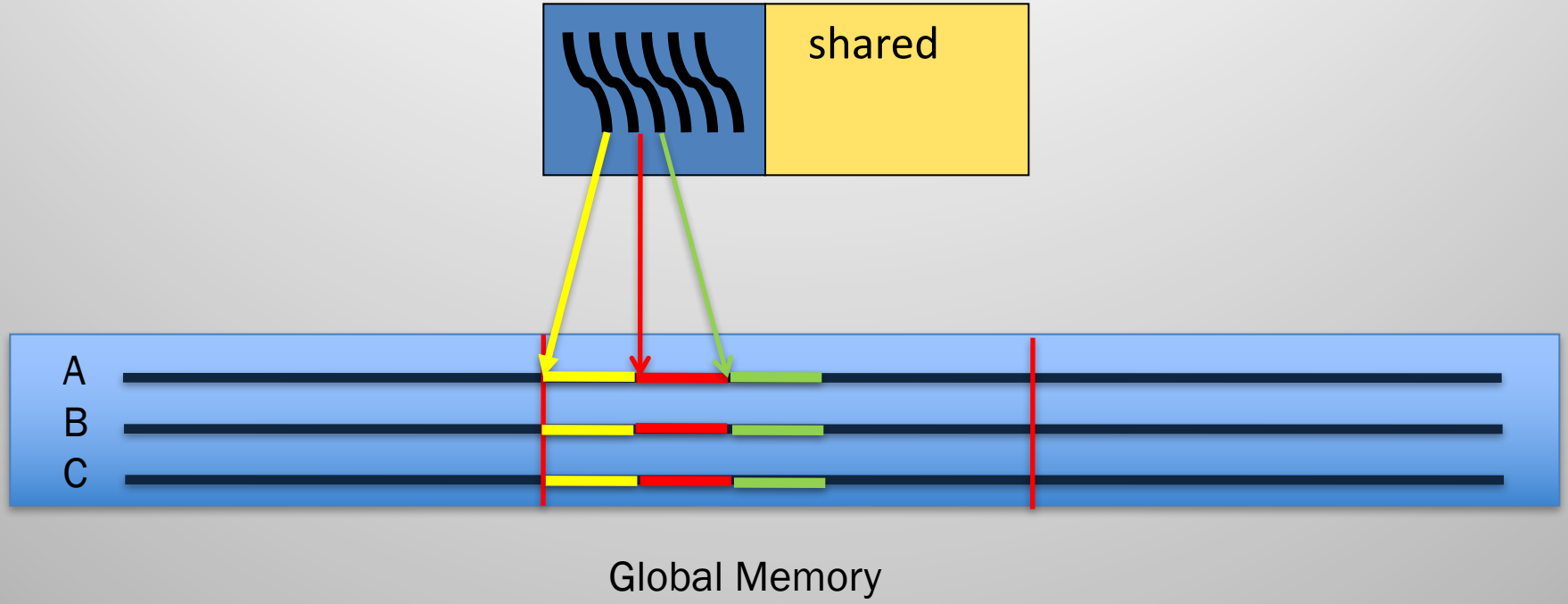
1a: vector add

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

Threads access contiguous chunks in a partition

Does this coalesce? How do you make it coalesce?

Let's go look at the code

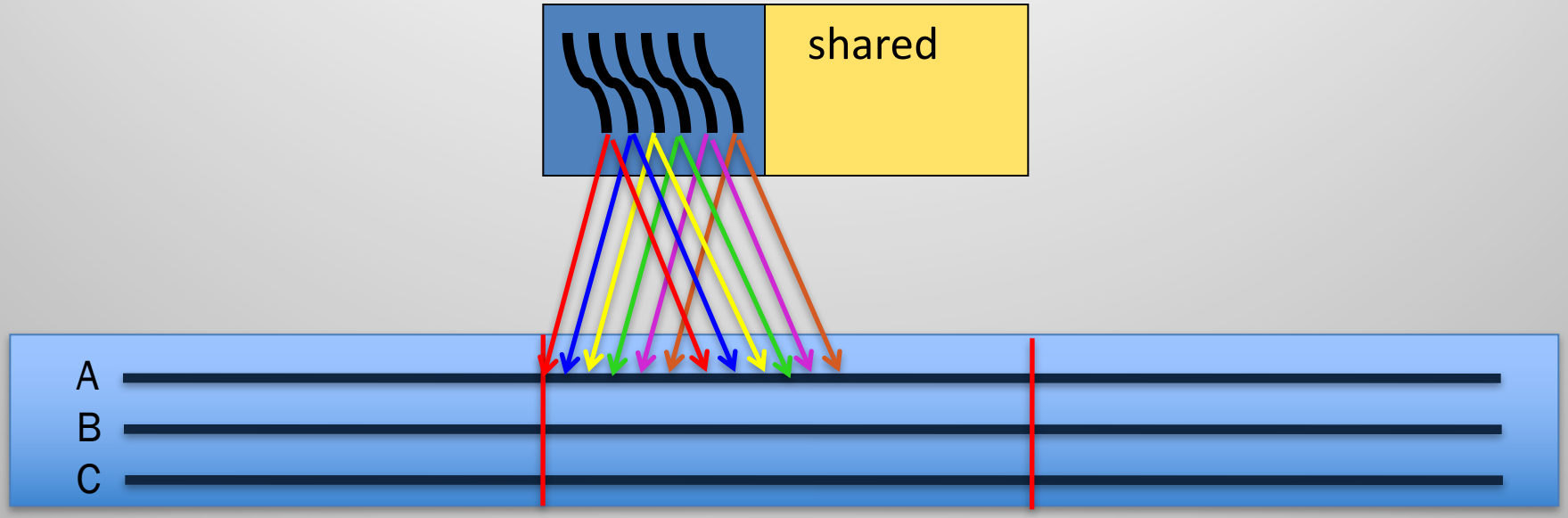


coalesced vector add

Thread blocks access contiguous partitions of A, B, and C
need for change from uncoalesced?

Threads access memory in interleaved pattern,

thread i+k accesses A[i+k*blockDim.x],… k = 0,1,…



Global Memory

Why is coalescing so important? Transaction size!

- In the coalesced case, 16 floats (of 4 bytes) are **perfectly packed** in one transaction of 64 bytes
- In the non coalesced case 1 float occupies a packet of 32 bytes, i.e., the total # bytes transferred is **8 fold what would be needed**

- eg: 384,000 loads of 64 = 24,576,000 bytes

  vs 6,144,000 loads of 32 = 196,608,000 bytes

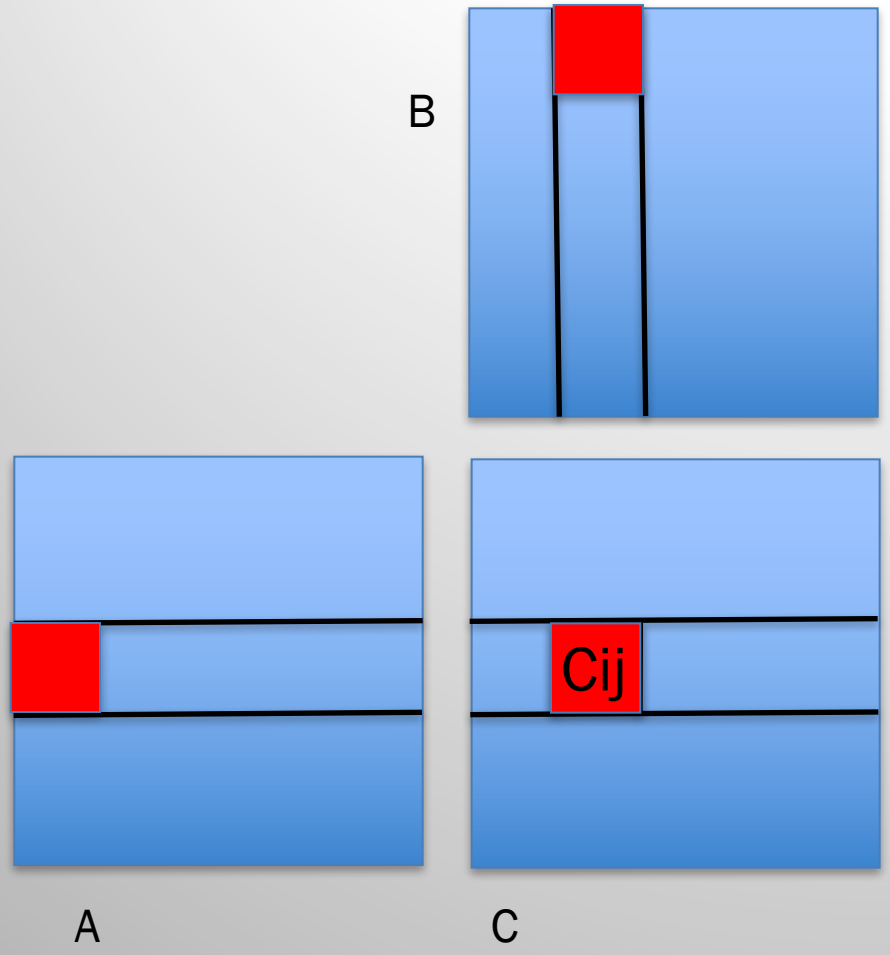
  not only 16x more transactions, but also 8x larger transaction volume.

Matrix Multiply

Shared / shared memory matrix multiply

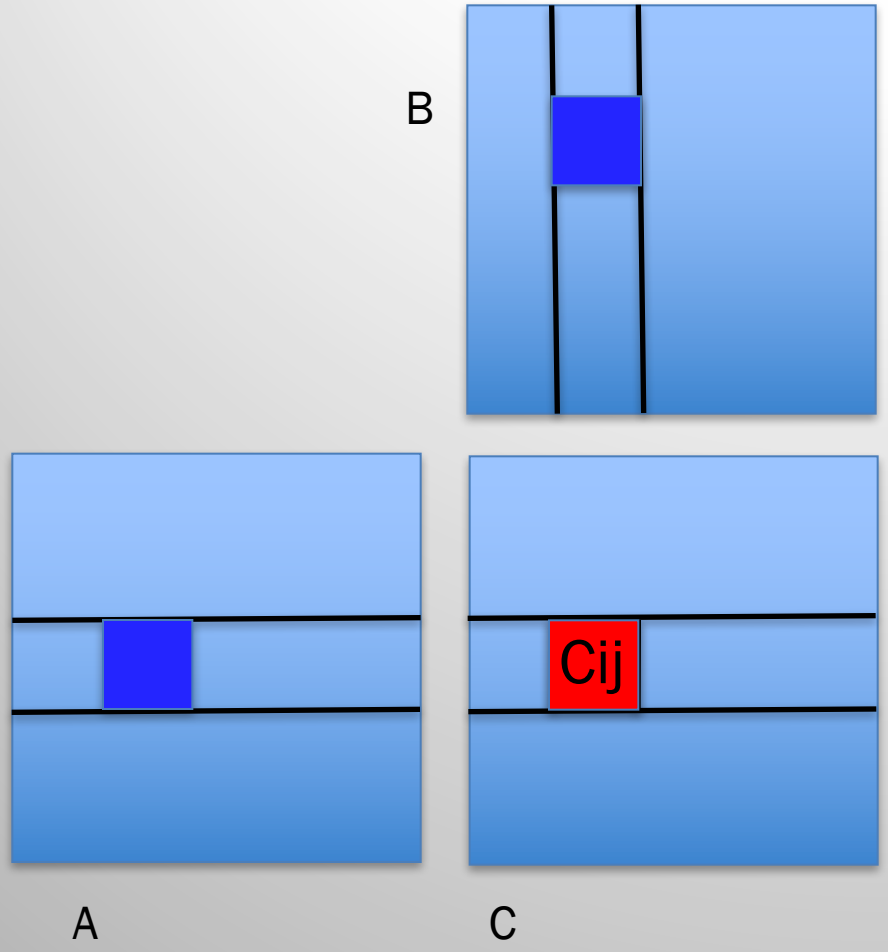
We will give you the matrix multiply code from the Programming Guide plus a driver, and you need to improve its performance by increasing the size of the C block each thread block computes (we call this the C footprint of a thread block)

1b 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

1b Shared / shared matmult



– etcetera

C foot-print and memory traffic

- If every thread block computes a kxk C block in a nxn matrix multiply (k divides n), what is the **global → shared** (block copies of A and B) **traffic volume**?
  - Grid Dimensions?

C foot-print and memory traffic

- If every thread block computes a kxk C block in a nxn matrix multiply (k divides n), what is the **global → shared** (block copies of A and B) traffic volume?
  - Grid Dimensions?

    n/k * n/k
  - Global shared memory traffic per thread block?

C foot-print and memory traffic

- If every thread block computes a kxk C block in a nxn matrix multiply (k divides n), what is the **global → shared** (block copies of A and B) traffic volume?
  - Grid Dimensions?

    n/k * n/k
  - Global shared memory traffic per thread block?

    2kn
  - Total traffic?

C foot-print and memory traffic

- If every thread block computes a kxk C block in a nxn matrix multiply (k divides n), what is the **global → shared** (block copies of A and B) traffic volume?
  - Grid Dimensions?

    $n/k * n/k$
  - Global shared memory traffic per thread block?

    $2kn$
  - Total traffic?

    $2n^3/k$

  What does this mean?

C foot-print and memory traffic

- If every thread block computes a kxk C block in a nxn matrix multiply (k divides n), what is the **global → shared** (block copies of A and B) traffic volume?
  - Grid Dimensions?

    $n/k * n/k$
  - Global shared memory traffic per thread block?

    $2kn$
  - Total traffic?

    $2n^3/k$

  The larger k, the less traffic (check extremes: k=1, k=n)

Shared Shared Matrix Multiply

- General case:

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

  Grid Block computes a block of C

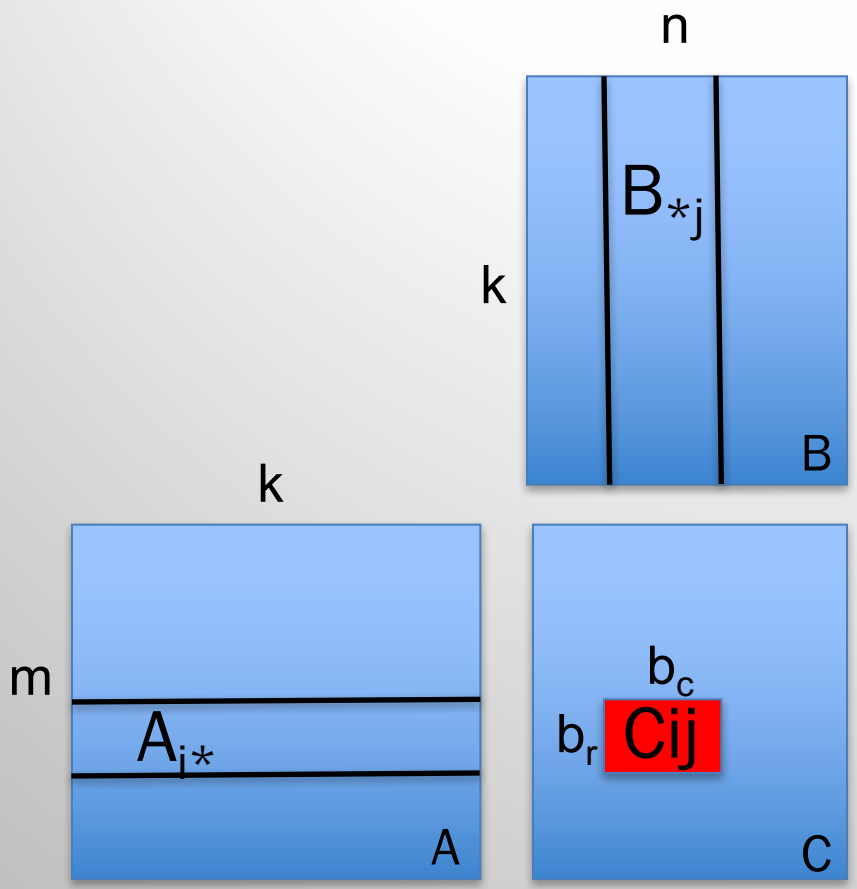
  with a certain "footprint" b_r . b_c

  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 $b_r \cdot b_c$

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?

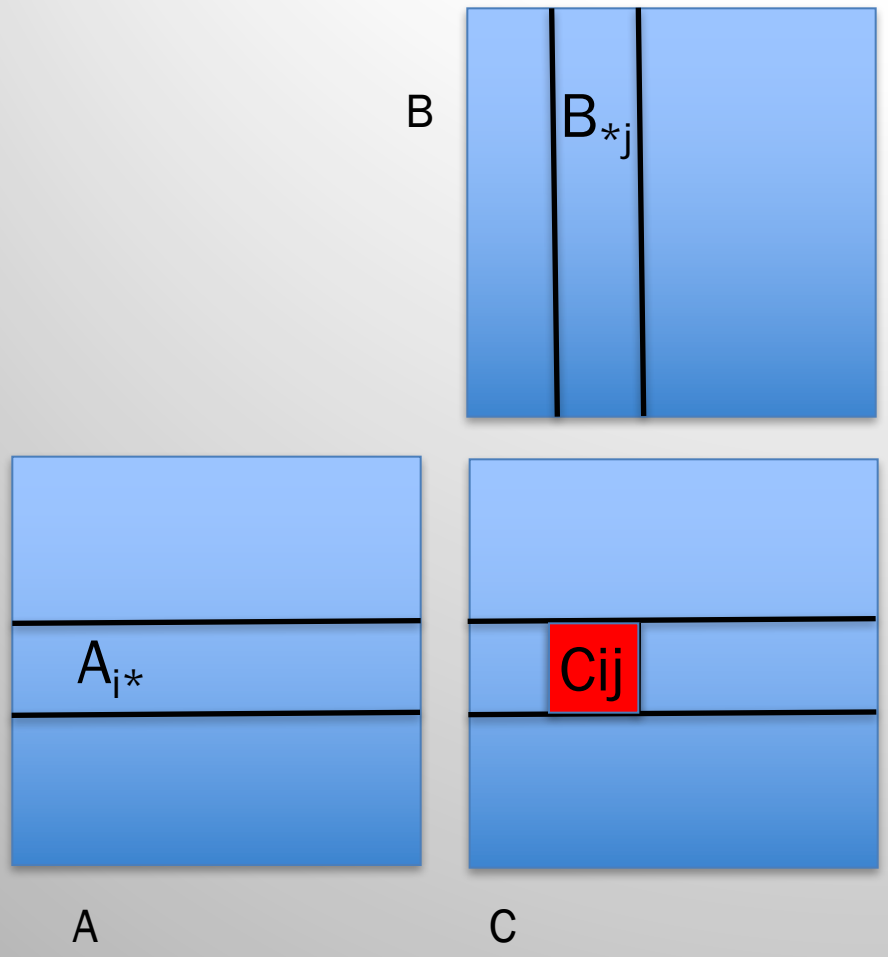
  minimize $(b_r+b_c) / b_r.b_c$

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

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

minimal when $h = 0$ → square tile !

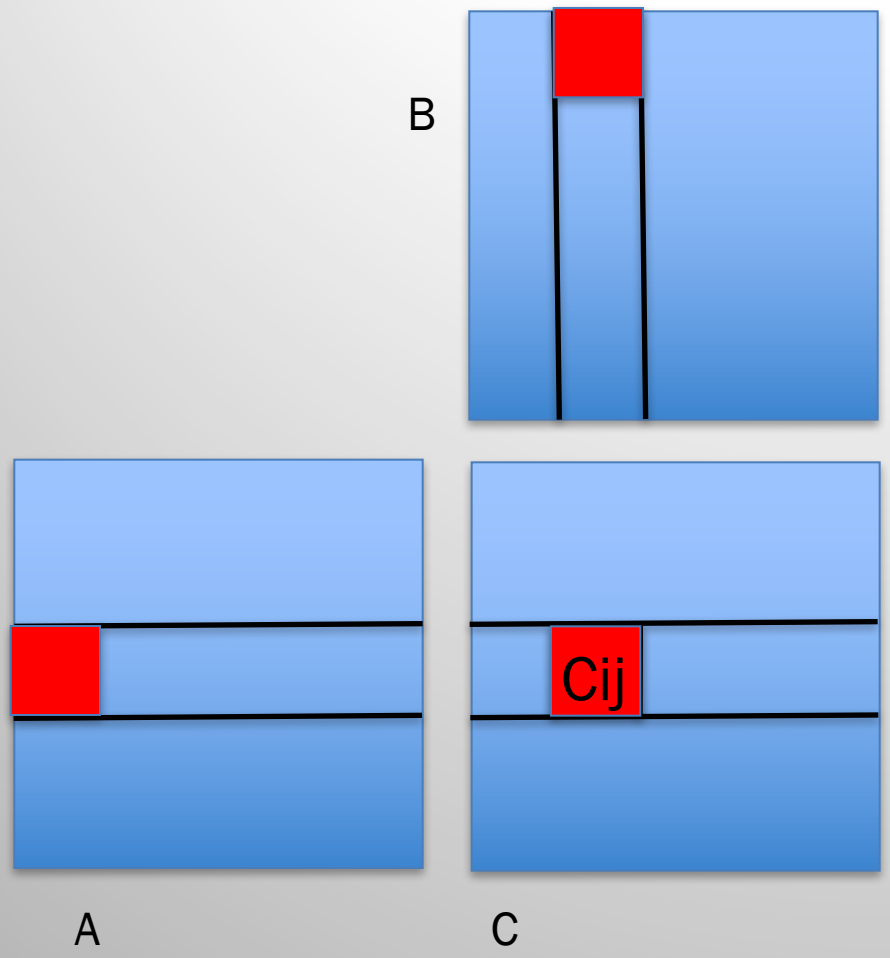
shared / shared matmult



- 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

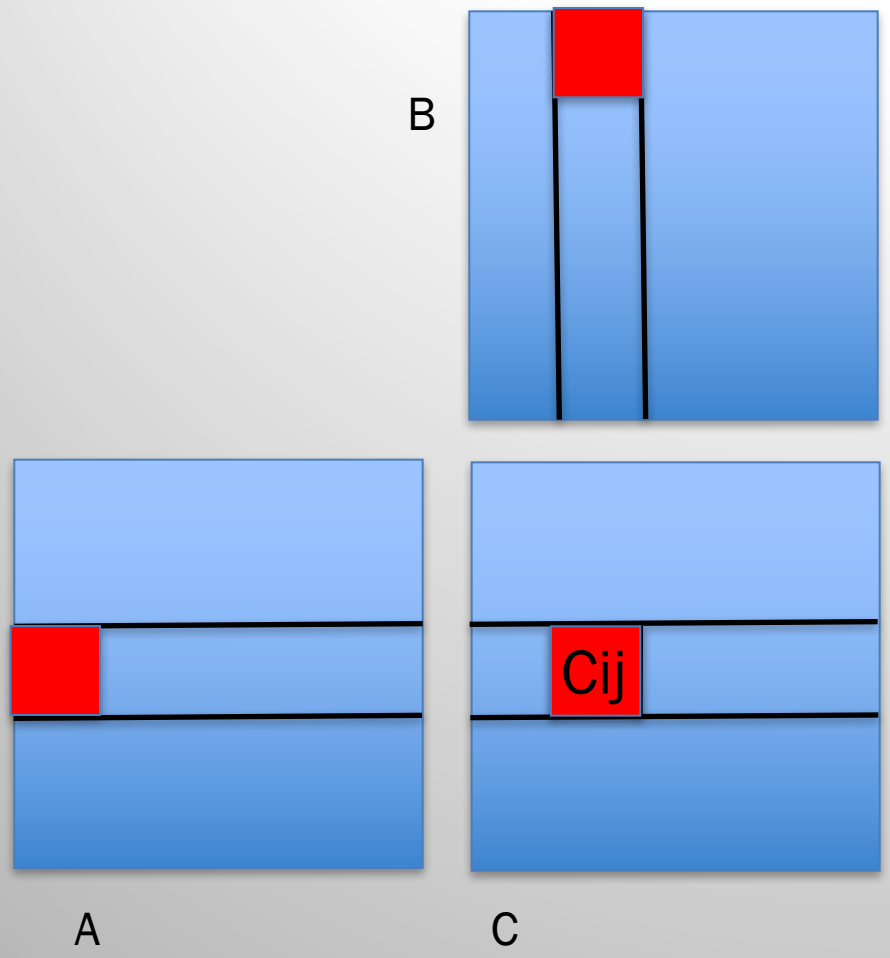
  ($t_r=b_r$, $t_c=b_c$)
- C elements: registers

shared / shared matmult



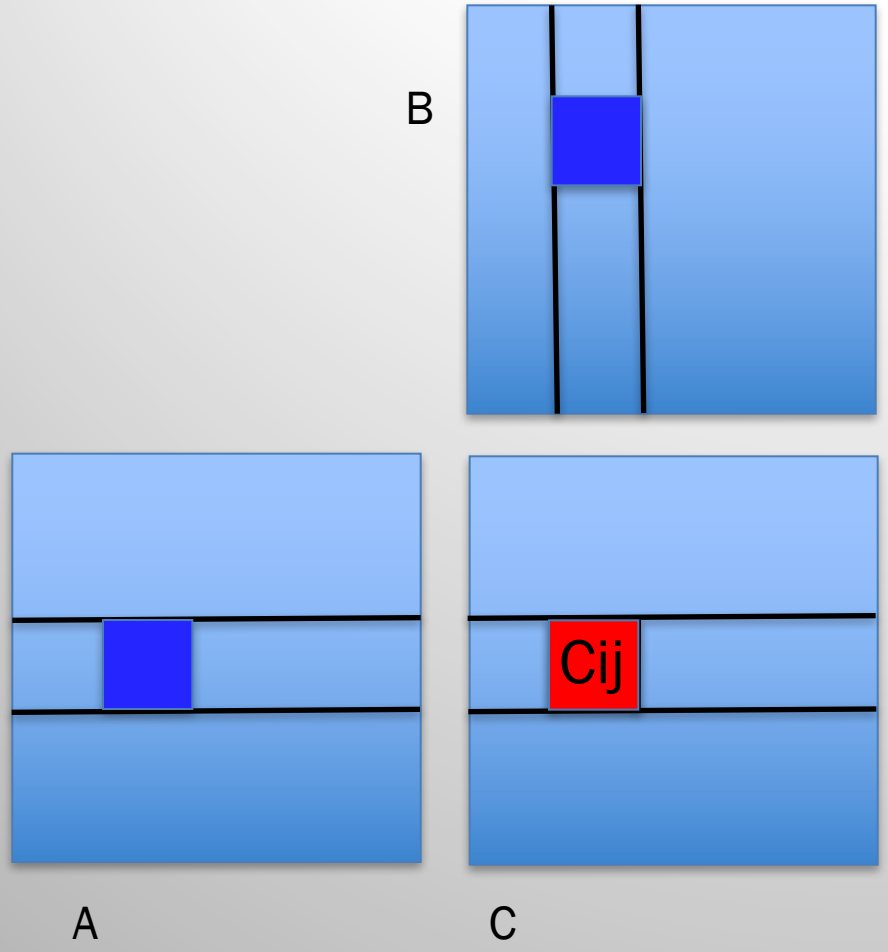
- A and B in global memory
- 2D grid, each 16x16 thread block computes a 16x16 C block
  - coalesced fetch a 16x16 A block into shared memory
  - coalesced fetch a 16x16 B block into shared memory

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

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

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