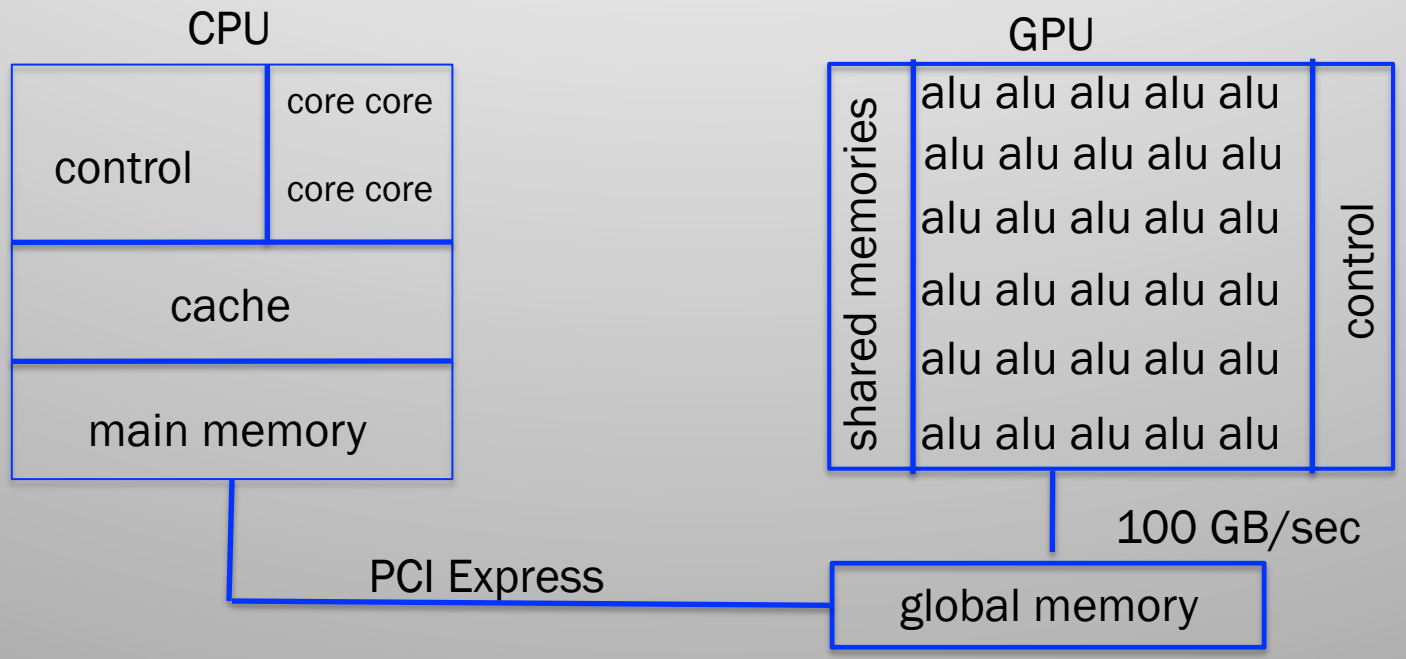
Cuda

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 (eg Fermi) there is also implicit cache

tesla 1060: a specific GPU

- 30 streaming multiprocessors (SMs)
  - each with 8 scalar processors (ALUs) and 2 special function units (sqrt and reciprocal)
  - each multiprocessor has 16 KB programmable cache called shared memory, and 16 KW registers, which are used for storing local program variables
- the GPU is connected to a 1 GB global memory by a 100 GB/sec interconnection network
  - this global memory is connected to the host memory by a PCI express bus.

GPU programming model

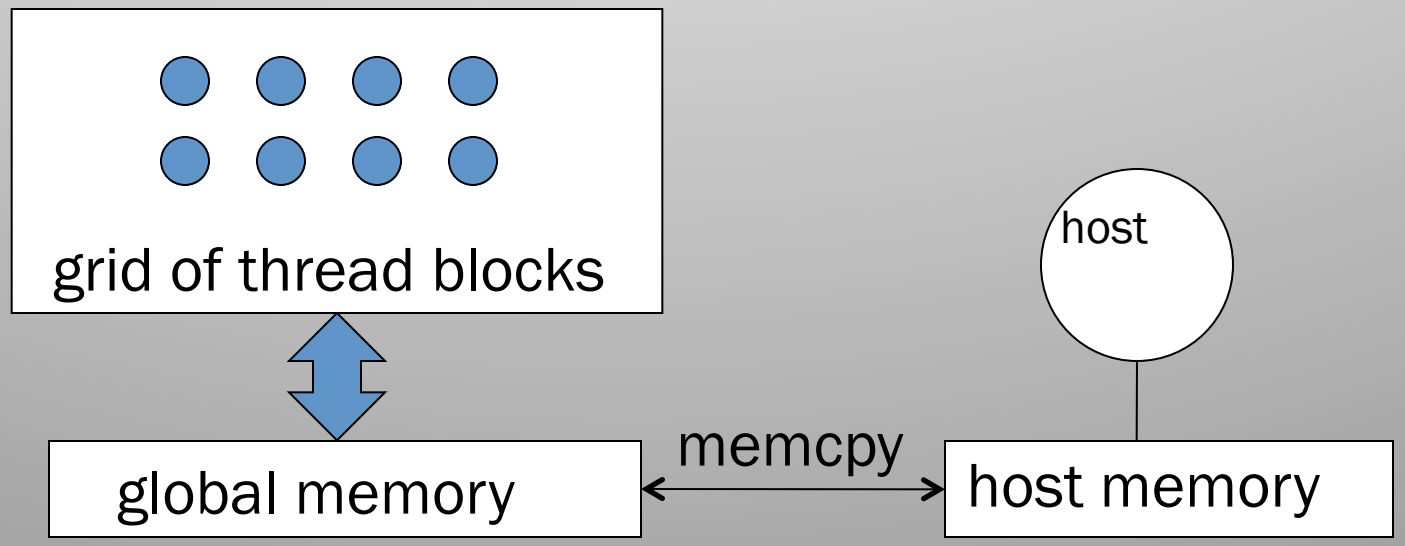
grid of thread blocks

- (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 **kernel**s on SMs

  kernel executed in SIMD fashion
- memcpy-s data out

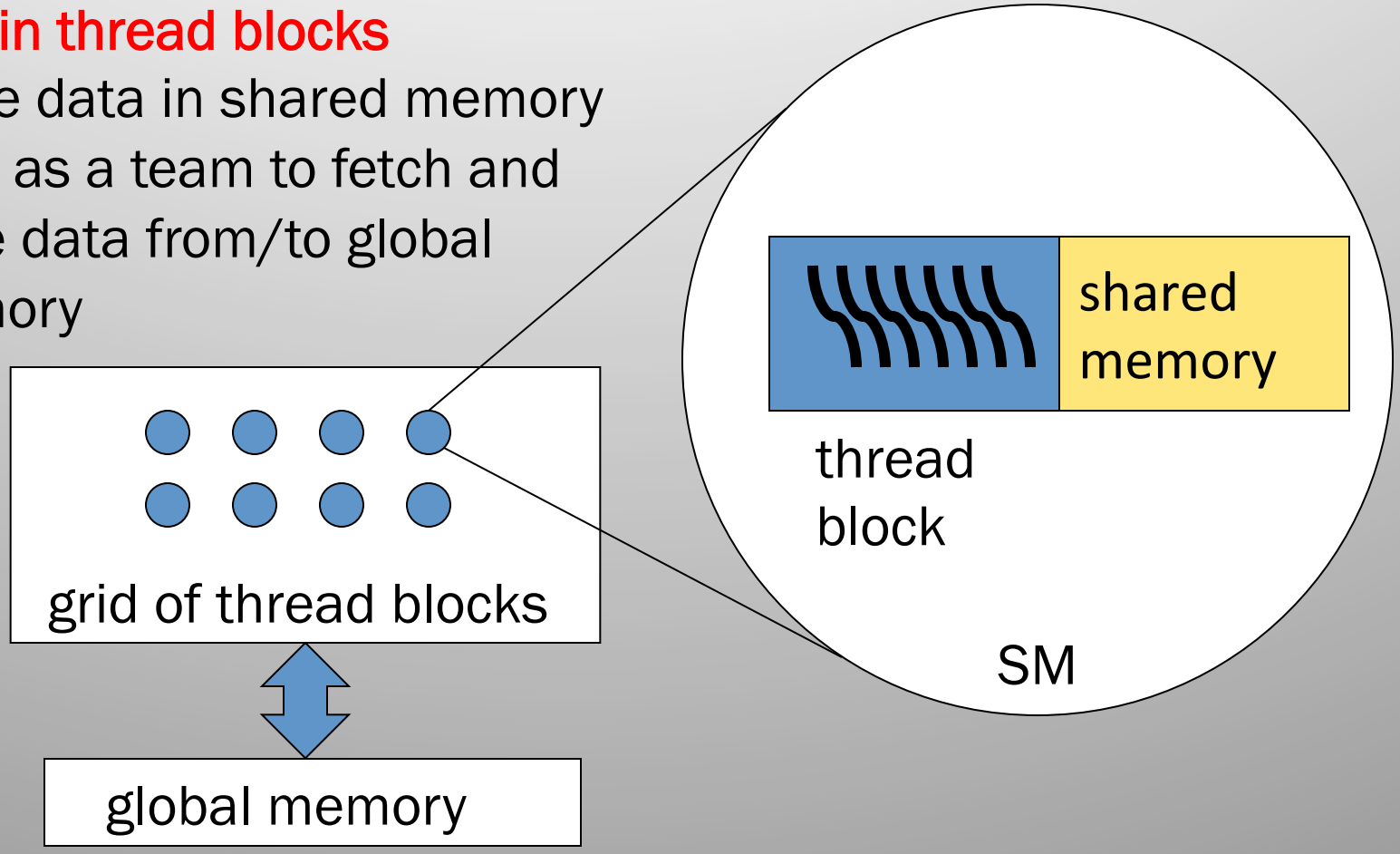


GPU programming model

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?

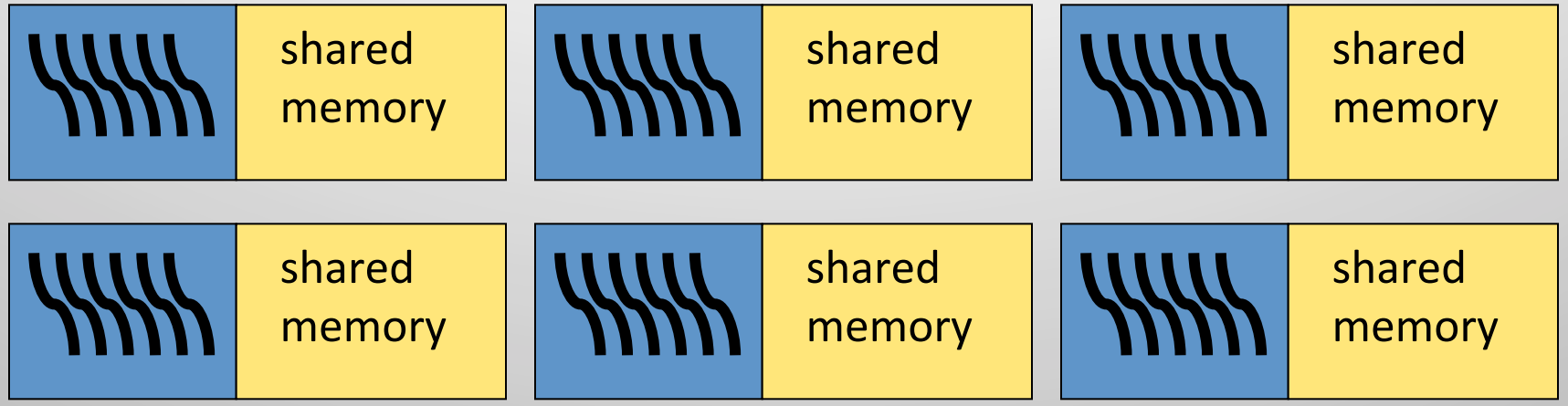
thread allocation

- A thread block can get allocated on any stream multiprocessor and thread blocks are independent of each other, ie 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 harder
- 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

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

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 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
- 16 (half a warp) global array elements are accessed at once: coalescing produces vectorized reads / writes that are much faster than element wise reads / writes
- This is very important for high speed GPU computing, and the subject of your CUDA PA 1a: vector add

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 for your PAs

Cuda programming assignment one

1a. Vector add

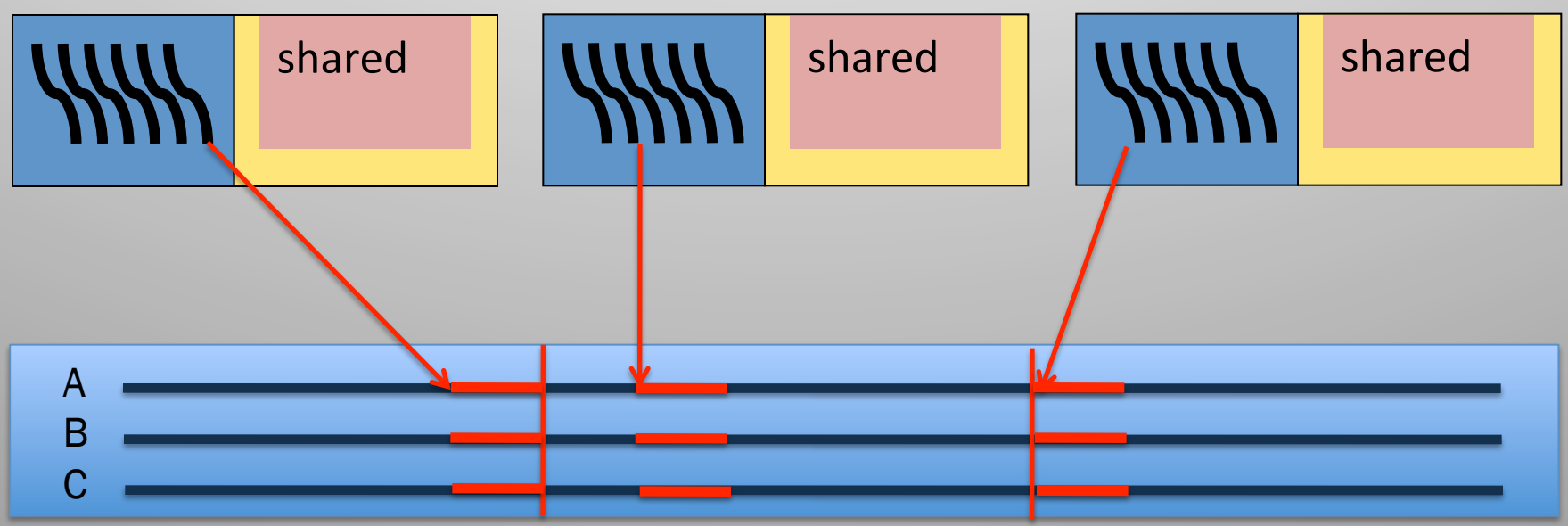
We will give you a non coalescing code, and you need improve and report its performance by turning it into a coalescing code

1b. 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)

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?



Global Memory

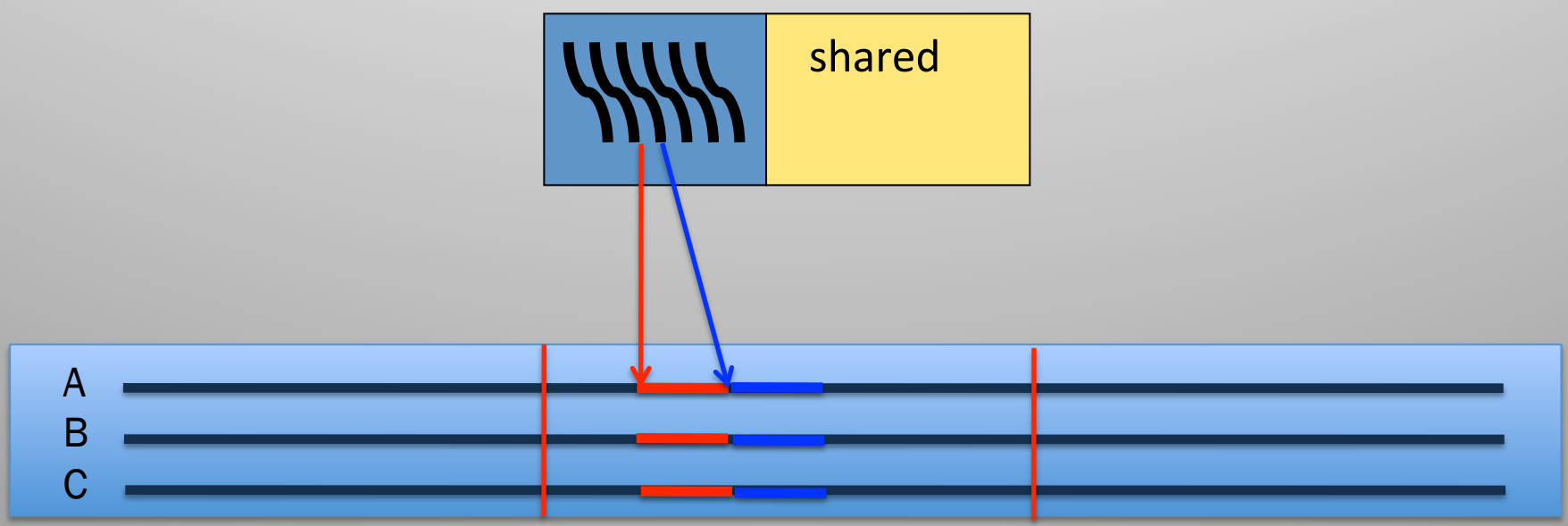
1a: vector add

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

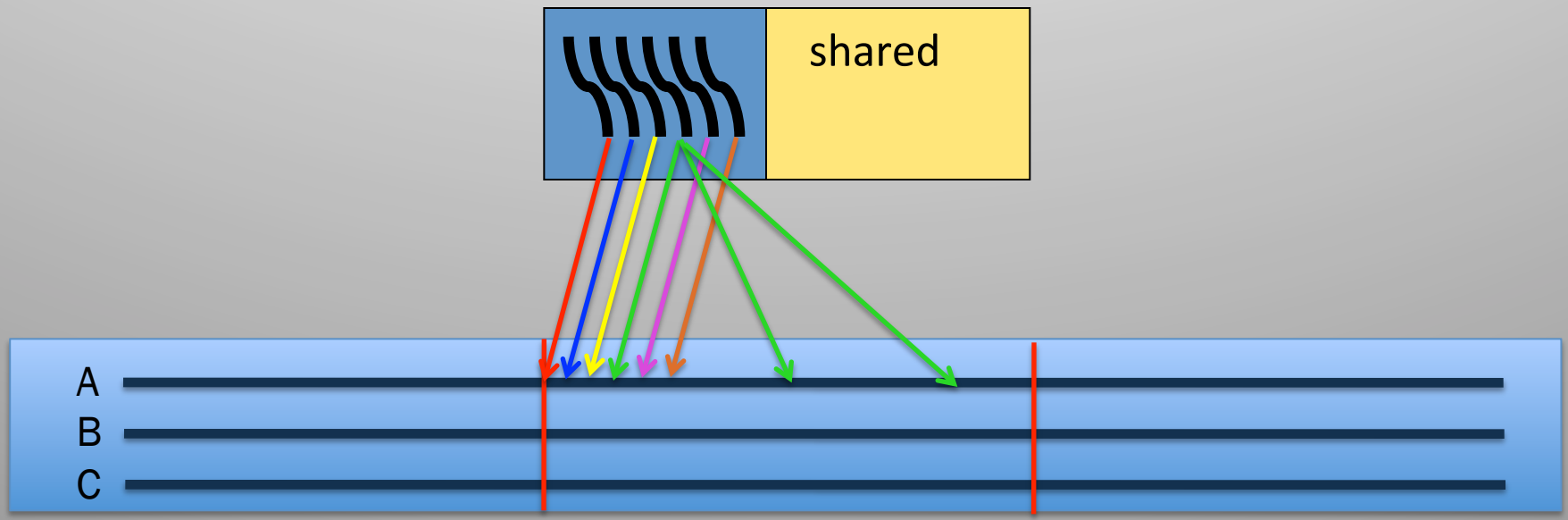


coalesced vector add

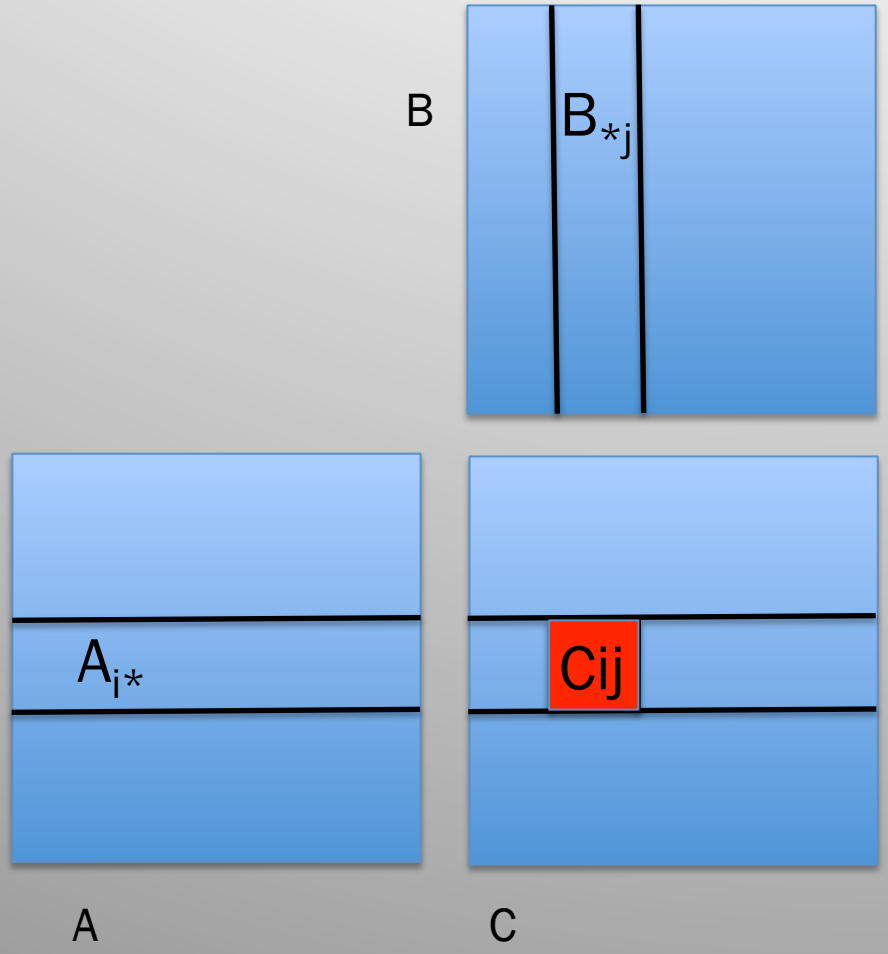
Thread blocks access contiguous partitions of A, B, and C
need for change from uncoalesced?

Threads access memory in interleaved pattern,

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

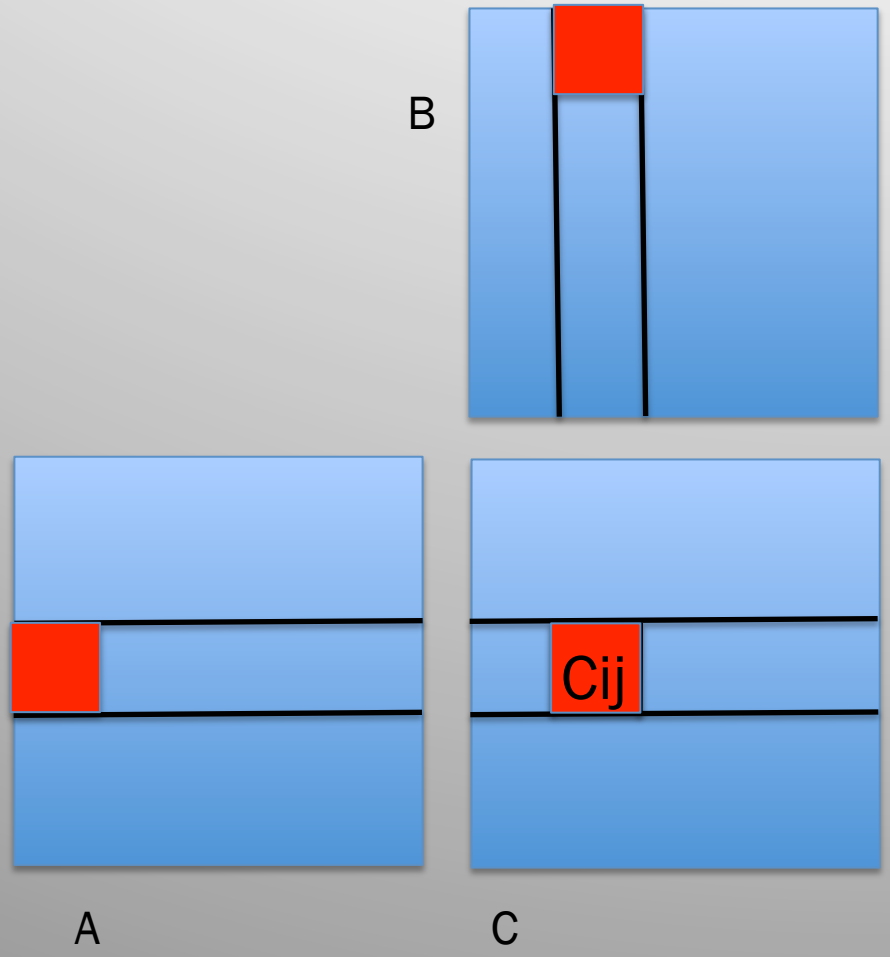


1b: Shared / shared matmult



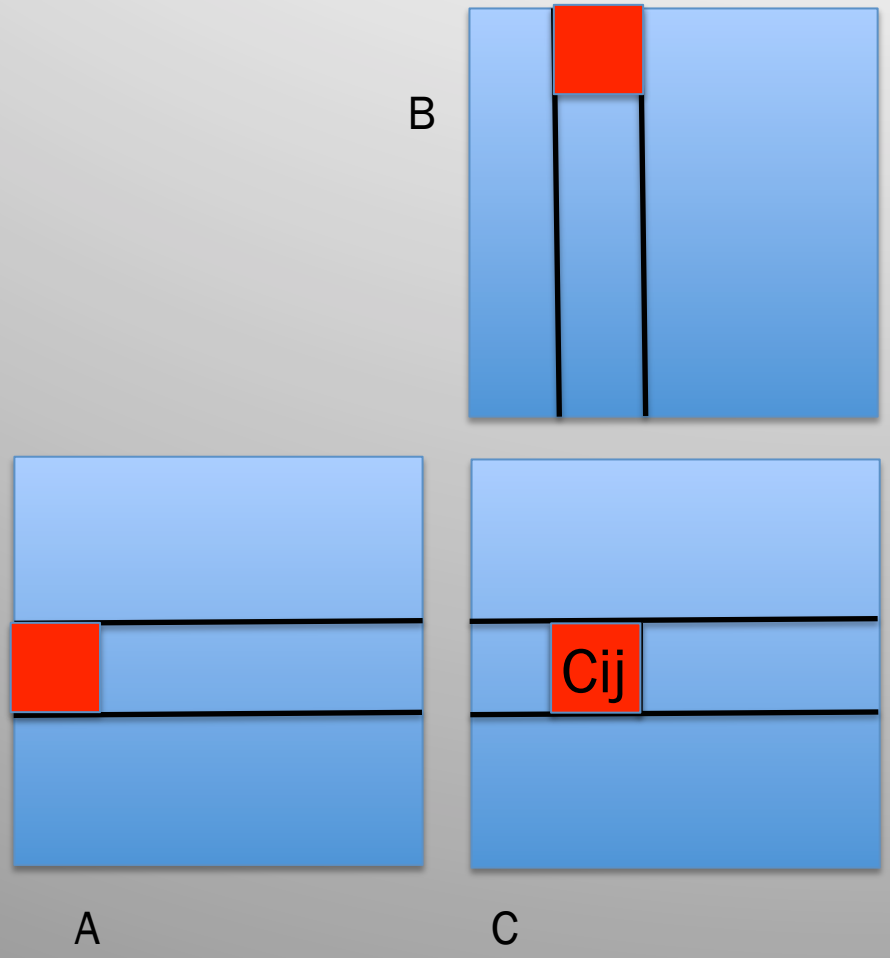
- A and B in global memory
- 2D grid of 2D thread blocks, each 16x16 thread block computes a 16x16 C block

1b Shared / shared matmult



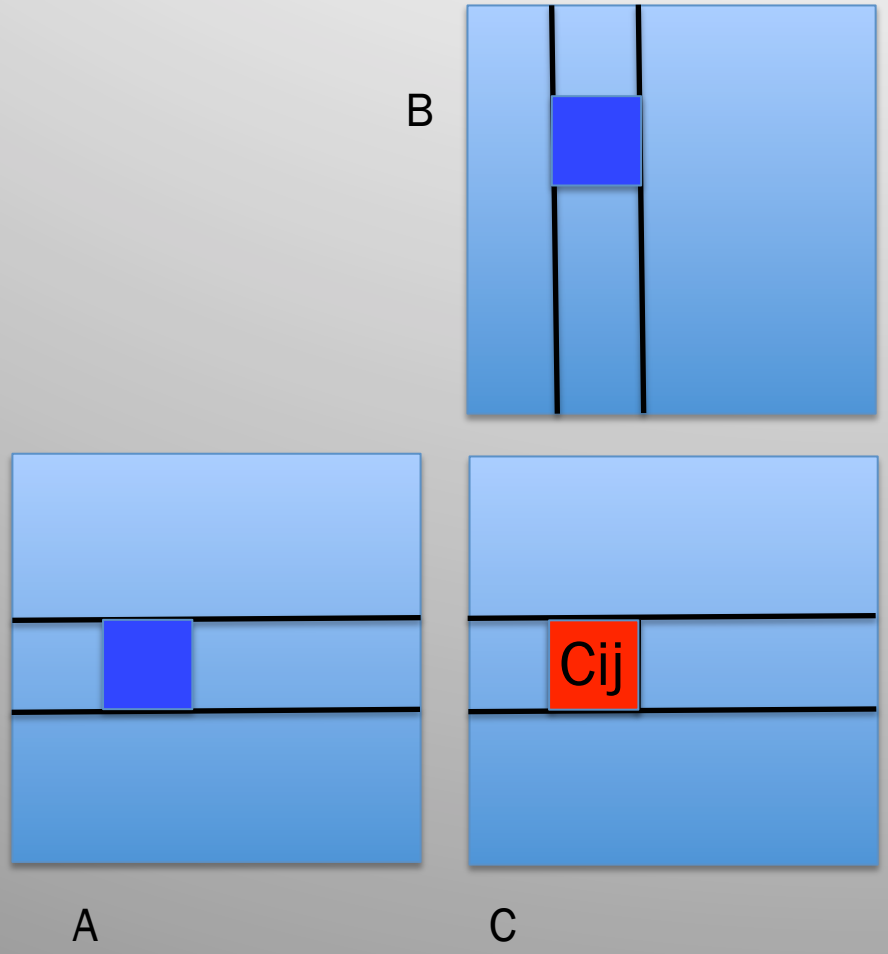
- A and B in global memory
- 2D grid, each 16x16 thread block computes a 16x16 C block
  - coalesced fetch a 16x16 A block into shared memory
  - coalesced fetch a 16x16 B block into shared memory

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
- let's go look at the code

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)

C foot-print and memory traffic

- The larger k, the larger footprint, the less traffic
- Is the shape of the foot print important?
  - Yes!
  - Square footprint optimizes global to shared memory traffic
- Are there other constraints than memory traffic?

C foot-print and memory traffic

- The larger k, the less traffic
- Are there other constraints than memory traffic?
  - parallelism (extreme (k=n) exploits 1 thread block)

C foot-print and memory traffic

- The larger k, the less traffic
- Are there other constraints than memory traffic?
  - parallelism

    (extreme (k=n) exploits 1 streaming multi-processor)
  - shared memory capacity (16KB)

    do two 32x32 blocks fit in 1 shared memory?

C foot-print and memory traffic

– The larger k, the less traffic

– Are there other constraints than memory traffic?

- parallelism (extreme (k=n) exploits 1 thread block)
- shared memory capacity (16KB)

  do two 32x32 blocks fit in 1 shared memory?

  2 KW = 8 KB OK

  do two 48x48 blocks fit?

C foot-print and memory traffic

- The larger k, the less traffic
- Are there other constraints than memory traffic?
  - parallelism (extreme (k=n) exploits 1 thread block)
  - shared memory capacity (16KB)

    do two 32x32 blocks fit in 1 shared memory?

    2 KW = 8 KB OK

    do two 48x48 blocks fit?

    no

C foot-print and memory traffic

- The larger k, the less traffic
- Are there other constraints than memory traffic?
  - parallelism (extreme (k=n) exploits 1 thread block)
  - shared memory capacity (16KB)

    do two 32x32 blocks fit in 1 shared memory?

    2 KW = 8 KB OK
- If we have a 16x16 thread-block and a 16x16 foot-print, how many shared memory reads per *+ ?
- If we have a 16x16 thread-block and a 32x32 foot-print, how many shared memory reads per *+ ?

A second look…

Inner product

Determine the performance difference of computing an inner product with both operands from shared memory, versus an inner product with one operand from shared memory and one from a register

Improved matrix multiply

Allocate one set of operands in shared memory and one in registers, still making sure to only read in coalescing style.

inner product: a micro-benchmark

- Just like vector add, inner product is a **micro-benchmark**: it isolates two approaches to a problem and measures their difference in behavior
- It is important to measure **only one phenomenon**, ie a micro-benchmark should do a comparison between two codes that only differ in the one aspect you try to understand

inner product

Determine the performance difference of computing an inner product with both operands from shared memory, versus one operand from shared memory and one from a register,
making sure that the codes are otherwise identical

This should determine the performance difference of shared/register vs. shared/shared

improved Matrix multiply

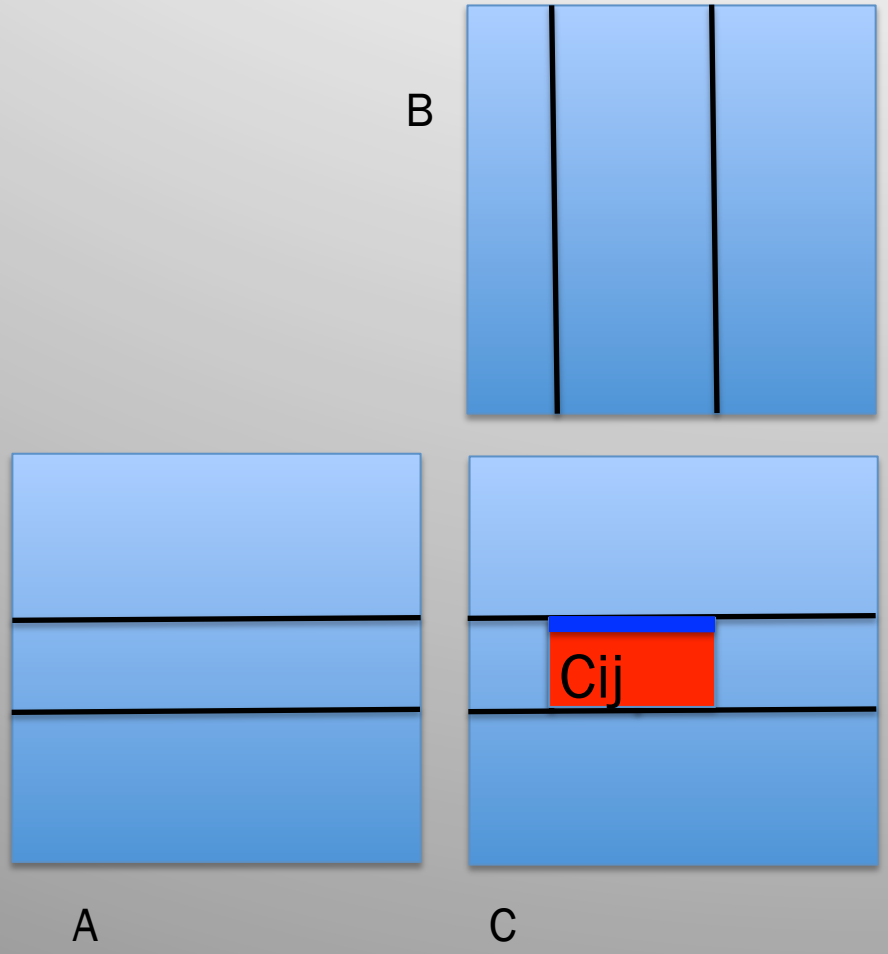
- In 1b a 16x16 thread block fetched two kxk blocks (k multiple of 16) into shared memory and then did a block matrix multiply on them

  do we need square A and B blocks?

  do the A and B blocks need to have the same shape?

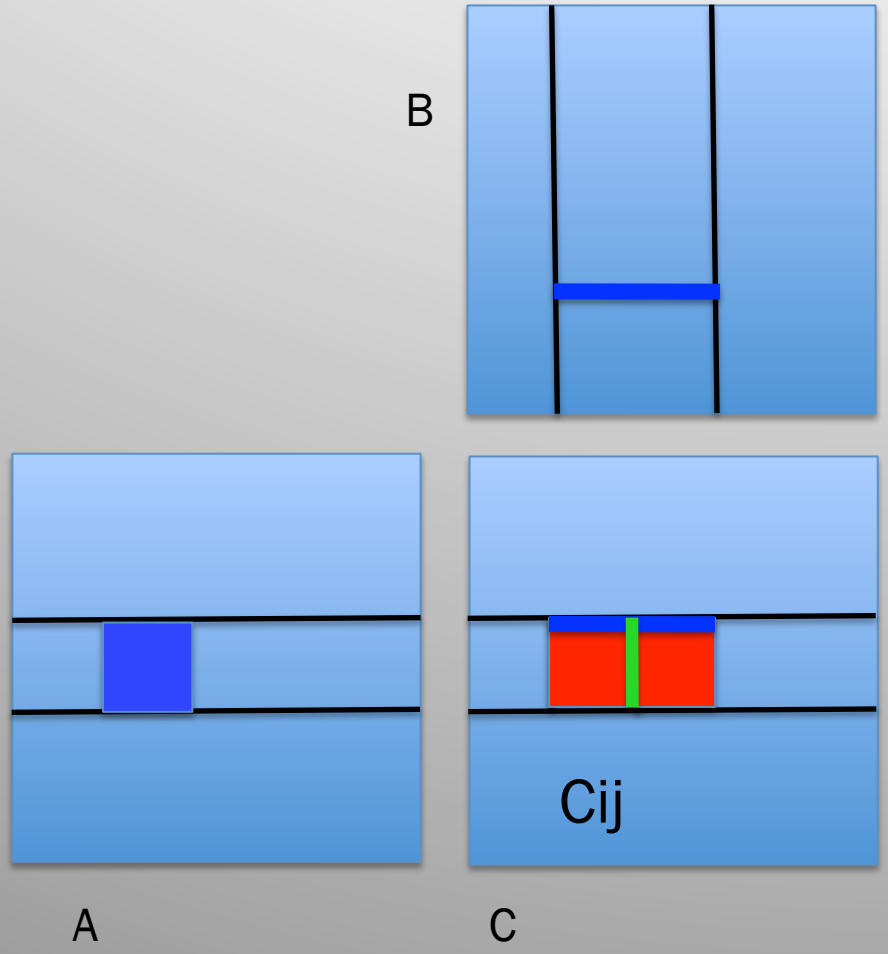
  do we need a 2D thread block?

Shared / register matmult



- A and B in global memory
- 2D grid of 1D thread blocks
- eg, each1x64 thread block computes a 16x64 C block

Shared / register matmult



- each thread of the thread block computes a column of the C block
- the thread block fetches an A block into shared memory, exploiting coalescing
- then for each column in the A block each thread fetches a B value into a register and performs a multiply add into the appropriate C elements