Lecture 7

OPTIMIZATION STRATEGIES

CUDA code optimization

- Converting a code to CUDA can be considered an advanced exercise in code optimization
  - You should start profiling CUDA code from the very beginning, from the first kernel you write
  - You should start the conversion from the most CPU-intensive parts of the code
  - You often have to play with different approaches until you get the best performance in a given part of the code
- We will consider a few common optimization strategies

Kernels: how many?

- You have to start a new kernel every time there is a global (across multiple blocks) data dependence
  - Example: two-level binary reduction shown previously
  - Another example: you need a separate kernel to initialize variables used to store an atomic reduction result:

```
// In global device memory:
__device__  double d_sum;

// On host:
// Initializing d_sum to zero:
init_sum <<<1, 1>>> ();

// Here d_sum is used to store atomic summation result from multiple
blocks
compute_sum <<<Nblocks, BSIZE>>> ();
```

Kernels: how many? (cont.)

- You can try to split a kernel if it uses too many registers (*register pressure*)
  - It happens e.g. if the kernel has many complex algebraic expressions using lots of parameters
  - Register pressure can be identified when profiling the code with NVIDIA CUDA profilers (e.g., it will manifest itself via low *occupancy number*)
  - It is very non-intuitive: sometimes the register pressure can be decreased by making the kernel *longer* (presumably, because sometimes adding more lines of code gives CUDA compiler more flexibility to re-arrange register usage across the kernel)

CUDA code optimization

- Kernels: how many?
  - You **should** end a kernel when there is a device-host dependence
    - Example:

```
// On host:

kernel1 <<<N, M>>> (d_A);
cudaMemcpy (h_A, d_A, size, cudaMemcpyDeviceToHost);

// Host code dependent on kernel1 results:
library_function1 ();
```

Kernels: how many? (cont.)

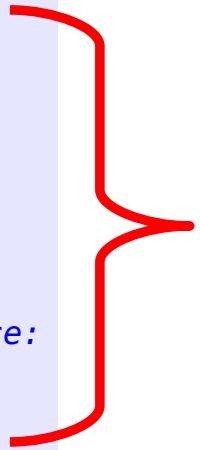
- Otherwise, you should try to make kernels as large as possible
  - Because each kernel launch has an overhead, in part because one has to store and then read the intermediate results from a slow (device or host) memory
  - You shouldn't worry that the kernel code won't fit on GPU: modern GPUs have large a limit of 512 million instructions per kernel
  - To improve readability, parts of the kernel can be modularized into device functions

What should be computed on GPU?

- You start with the obvious targets: CPU-intensive data-parallel parts of the code
- What should you do with the leftover code (not data-parallel and/or not very CPU-intensive)?
  - If not a lot of data needs to be copied from device to host and vice versa for the leftover code, it may be beneficial to leave these parts of the code on host.
  - If on the other hand the leftover code needs an access to a lot of intermediate results from CUDA kernels, then it may be more efficient to move everything to the GPU – even purely serial (single-thread) computations. This way, no intermediate (scratch) data will ever need to leave GPU.

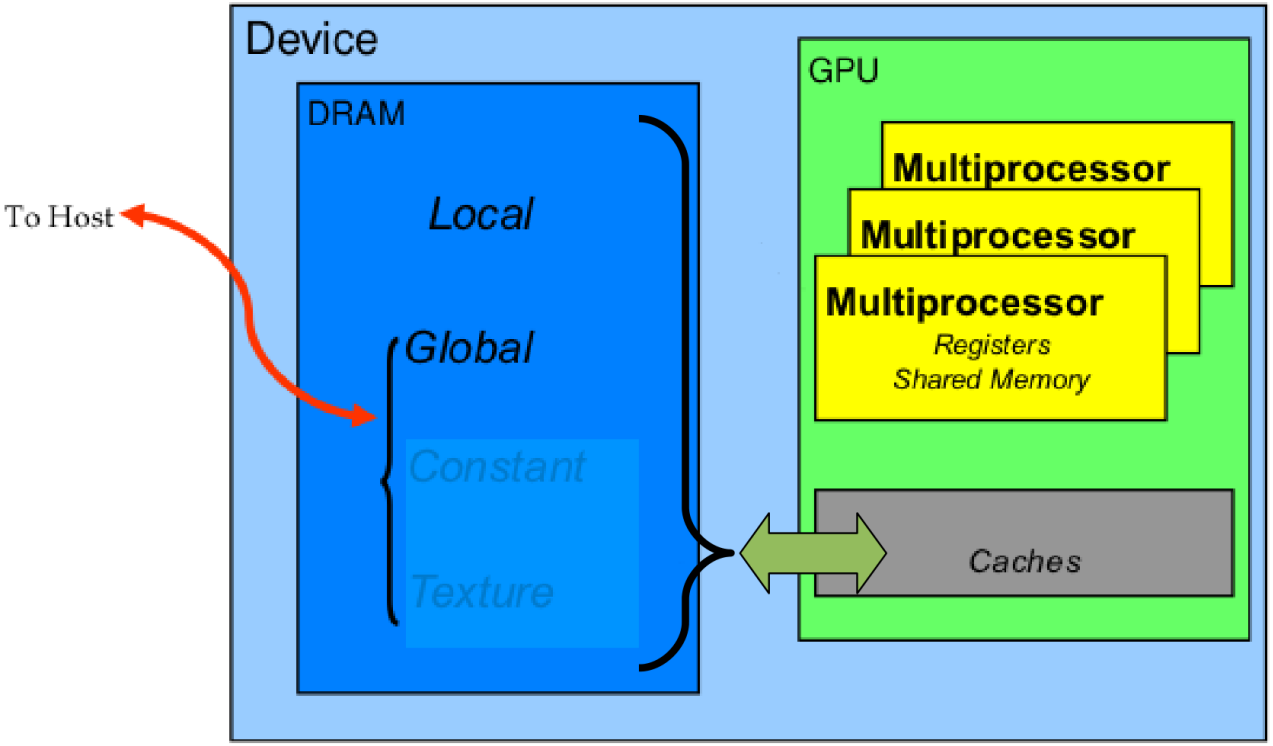
Moving leftover code to GPU

```
// On host:

// First chunk of data-parallel code goes here:
kernel1 <<<N, M>>> ();


// Copying kernel1 results to host:
cudaMemcpy (h_A, d_A, size,
    cudaMemcpyDeviceToHost);

// Non-parallelizable part of the code:
serial_computation (h_A, h_B);

// Copying serial_computation results to device:
cudaMemcpy (d_B, h_B,size,
    cudaMemcpyHostToDevice);



// Second chunk of data parallel code which
depends on d_B:
kernel2 <<<N, M>>>
```



```
// On host:

// First chunk of data-parallel code
goes here:
kernel1 <<<N, M>>> ();




// Now it is executed on GPU, serially:
serial_computation_kernel <<<1, 1>>> ();




// Second chunk of data parallel code
which depends on d_B:
kernel2 <<<N, M>>>
```

Memory spaces on GPU



Optimizing memory access on GPU

- Registers <-> “Local” memory are not under your direct control, making it harder to optimize
- Global and shared memory, on the other hand, are under direct programmer's control, so they are easier to optimize.
- Main strategies for optimization:
  - Global memory: coalescence of memory accesses
  - Shared memory: minimizing bank conflicts

Exploiting fully the parallelism of the problem

- A GPU has a large number of cores, to take full advantage of the GPU they must all be given something to do.
- It is hence beneficial to have the work to be done decomposed among a large number of threads.
  - GPU architecture can easily handle large numbers of threads without overhead (unlike CPU)
  - for this to work optimally threads belonging to the same block must be executing similar (ideally exactly the same) instructions, operating on different data
  - this means one must avoid divergent branches within a block
  - size of block should be multiple of 32 (warp size), must not exceed the maximum for device

Important caveat: is more threads always useful?

- Each thread consumes some resources, mainly registers and shared memory. Given that these resources are limited, the number of threads "alive" at any one time (i.e. actively running on the hardware) is also limited.

- Hence the benefit of adding more threads tends to plateau.
  - one can optimize around the resources needed, especially registers, to improve performance

Avoiding transfers between GPU and CPU

- That is a huge bottleneck, but unavoidable since GPU has limited capabilities, most significantly no access to file system (note: AMD's APU Fusion avoids this problem)
- CPU essential because GPU cannot be independent. All kernels must be launched from the CPU which is the overall controller
  - changed on Kepler architecture released in late 2012 on which kernels can launch other kernels
- Using pinned memory helps a bit
- Using asynchronous transfers (overlapping computation and transfer) also helps

Optimizing access to global memory

- A GPU has a large number of cores with great computational power, but they must be “fed” with data from global memory

- If too little computation done on core relative to memory transfer, then it becomes the bottleneck.
  - most of the time is spent moving data in memory rather than number crunching
  - for many problems this is unavoidable

- Utilizing the memory architecture effectively tends to be the biggest challenge in CUDA-fying algorithms

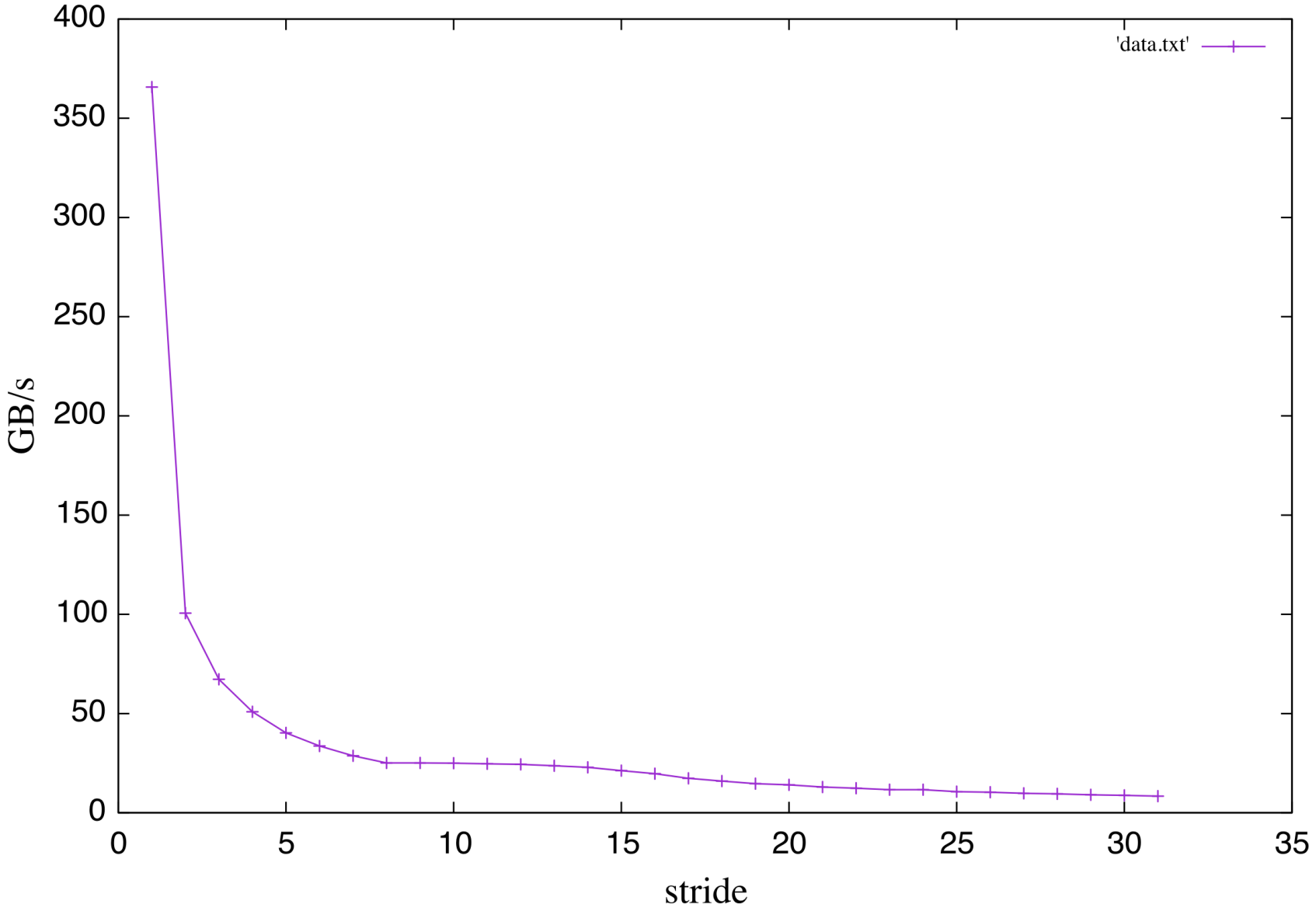
GPU memory is high bandwidth/high latency

- A GPU has potentially high bandwidth for data transfer from global memory to cores. However, the latency for this transfer for any individual thread is also high (hundreds of cycles)
- Using many threads, latency can be overcome by hiding it among many threads.
  - group of threads requests some memory, while it is waiting for it to arrive, another group is computing
  - the more threads you have, the better this works
- The pattern of global memory access is also very important, as cache size of the GPU is very limited.

Global memory access is fast when coalesced

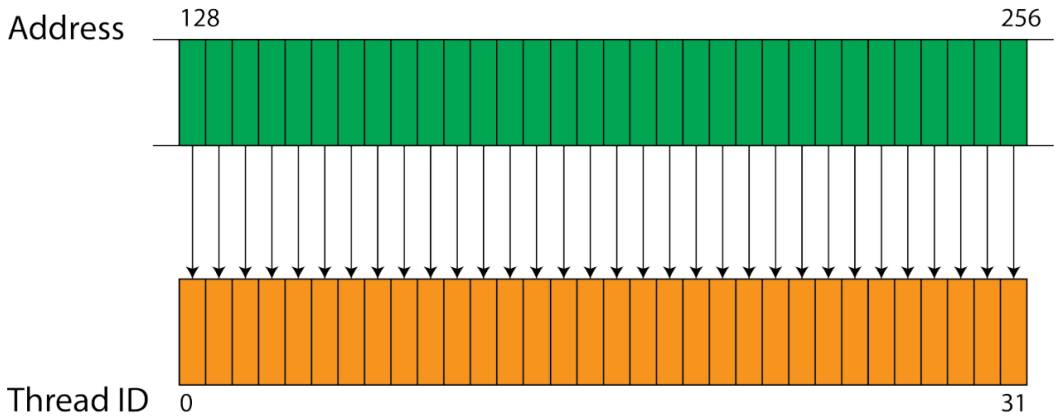
- It is best for adjacent threads belonging to the same warp (group of 32 threads) to be accessing locations adjacent in memory (or as close as possible)
- Good access pattern: thread `i` accesses global memory array member `a[i]`
- Inferior access pattern: thread `i` accesses global memory array member as `a[i*nstride]` where `nstride>1`
- Clearly, random access of memory is a particularly bad paradigm on the GPU

Memory bandwidth decreases as stride increases



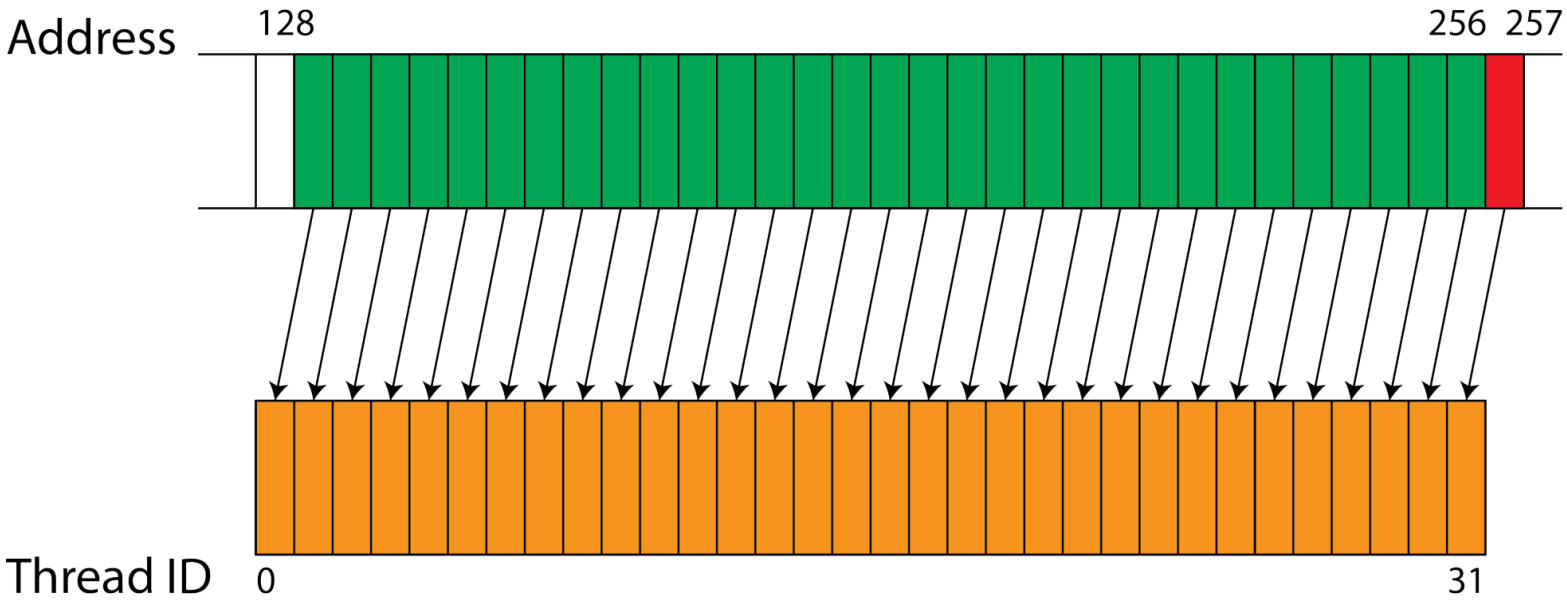
Global memory: coalescence of memory accesses

- Global memory loads and stores by threads of a warp are coalesced by the device into as few as one transaction when certain access requirements are met
- By default, all accesses are cached through L1 as 128-byte lines
- Coalescence is the best when accessing flat arrays (unit stride) consecutively.



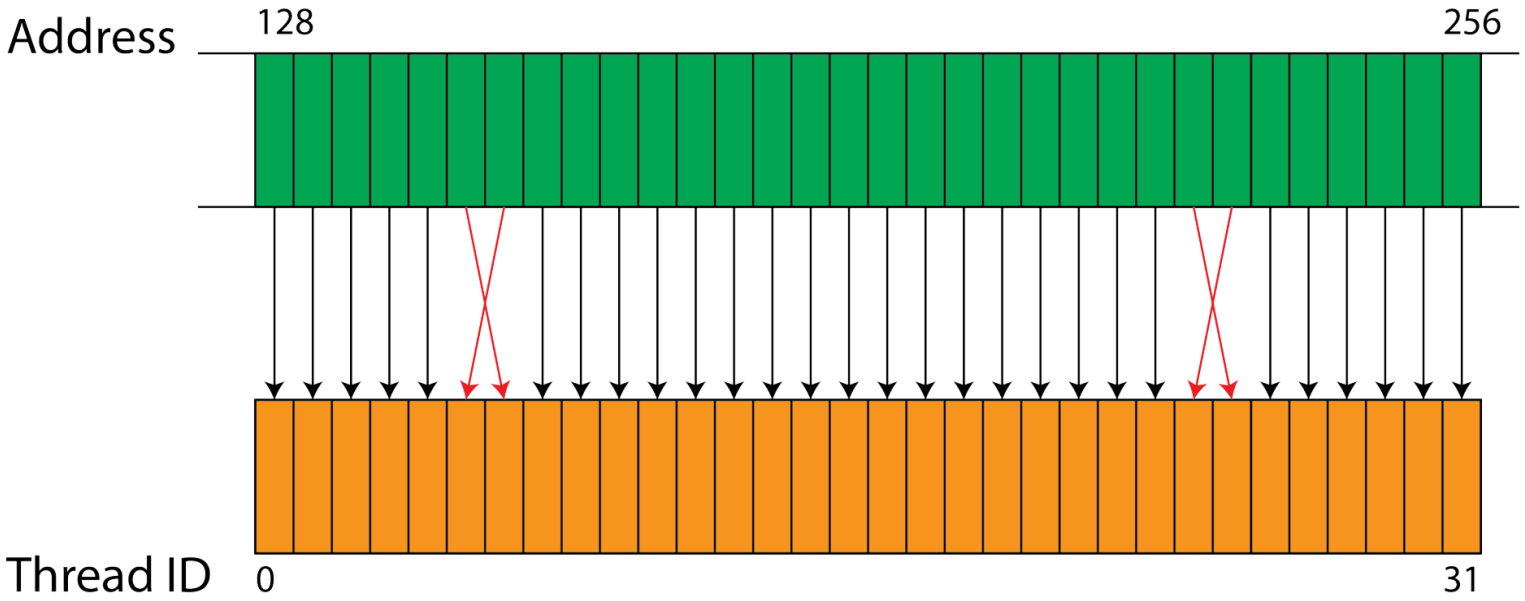
Global memory: coalescence of memory accesses (2)

- Misaligned access degrades the performance, but not dramatically



Global memory: coalescence of memory accesses (3)

- Any kind of non-sequential memory access (not just stride>1) is bad*



Global memory: coalescence of memory accesses (4)

- The strategy with multi-D arrays is to either
  - flatten them yourself (the only way if >3 dimensions), or
  - use special CUDA functions cudaMallocPitch() and cudaMalloc3D() to allocate properly aligned 2D and 3D arrays, respectively, or
  - at the very least, convert row-major arrays to column-major ones

2D arrays

```
// Using cudaMallocPitch, 2D case
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;

cudaMallocPitch (&devPtr, &pitch, width * sizeof(float), height);
MyKernel <<<64, 64>>> (devPtr, pitch);

// Device code
__global__ void MyKernel (float* devPtr, size_t pitch)
{
  int ix = blockIdx.x;
  int iy = threadIdx.x;
  float* row = (float*)((char*)devPtr + ix * pitch);
  float element = row[iy];   // Coalesced access
}
```

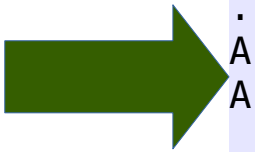
Multi-D arrays

```
// Flattened, good for any D; individual dimensions can be arbitrary
// On device:
#define N_TOTAL  N1*N2*N3*N4
__device__  float  d_A[N_TOTAL];
__global__  void  mykernel  (){
int i = threadIdx.x + blockDim.x * blockIdx.x;
if (i < N_TOTAL)   {
    d_A[i] = ...
    // You compute individual indexes only if they are needed for the
computations:
    int i1 = i % N1;  int m = i / N1;
    int i2 = m % N2;  m = m / N2;
    int i3 = m % N3;
    int i4 = m / N3;
   }
}
// On host:
int Nblocks = (N_TOTAL + BLOCK_SIZE - 1) / BLOCK_SIZE;
mykernel <<<Nblocks, BLOCK_SIZE>>> ();
```

Multi-D arrays (2)

- If you have to use non-flattened static multi-D arrays, transpose them to “column-major” if they are “row-major”:

```
// Row-major (non coalesced)

float A[N][30];
...
A[threadIdx.x][0]=...;
A[threadIdx.x][1]=...;
```



```
// Column-major (coalesced)

float A[30][N];
...
A[0][threadIdx.x]=...;
A[1][threadIdx.x]=...;
```

Structures of arrays

- For the same reason, use structures of arrays instead of arrays of structures (the latter results in a memory access with a large stride)

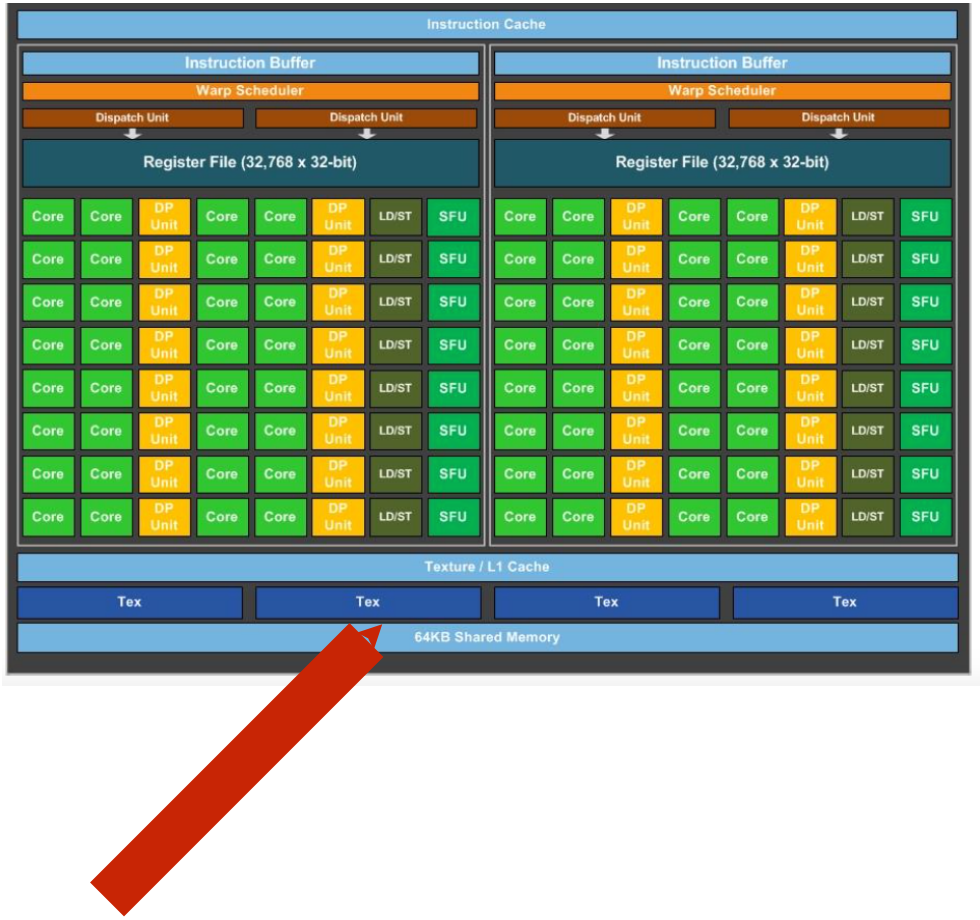
```
// Array of structures behaves like row major accesses (non coalesced)
struct Point { double x; double y; double z; double w; } A[N];
...
A[threadIdx.x].x = ...
```

```
// Structure of arrays behaves like column major accesses (coalesced)
struct PointList { double *x; double *y; double *z; double *w; } A;
...
A.x[threadIdx.x] = ...
```

Shared memory

- Each multiprocessor has some fast on-chip shared memory
- Threads within a thread block can communicate using the shared memory
- Each thread in a thread block has R/W access to all of the shared memory allocated to a block
- Threads can synchronize using the intrinsic

```
__syncthreads();
```



Using shared memory to optimize access to global memory

- Shared memory is much faster than global memory; also, access to shared memory doesn't need to be coalesced
- Shared memory can be viewed as a “user-managed cache for global memory”
  - One can store in shared memory frequently used global data
  - One can use shared memory to make reading data from global memory coalesced

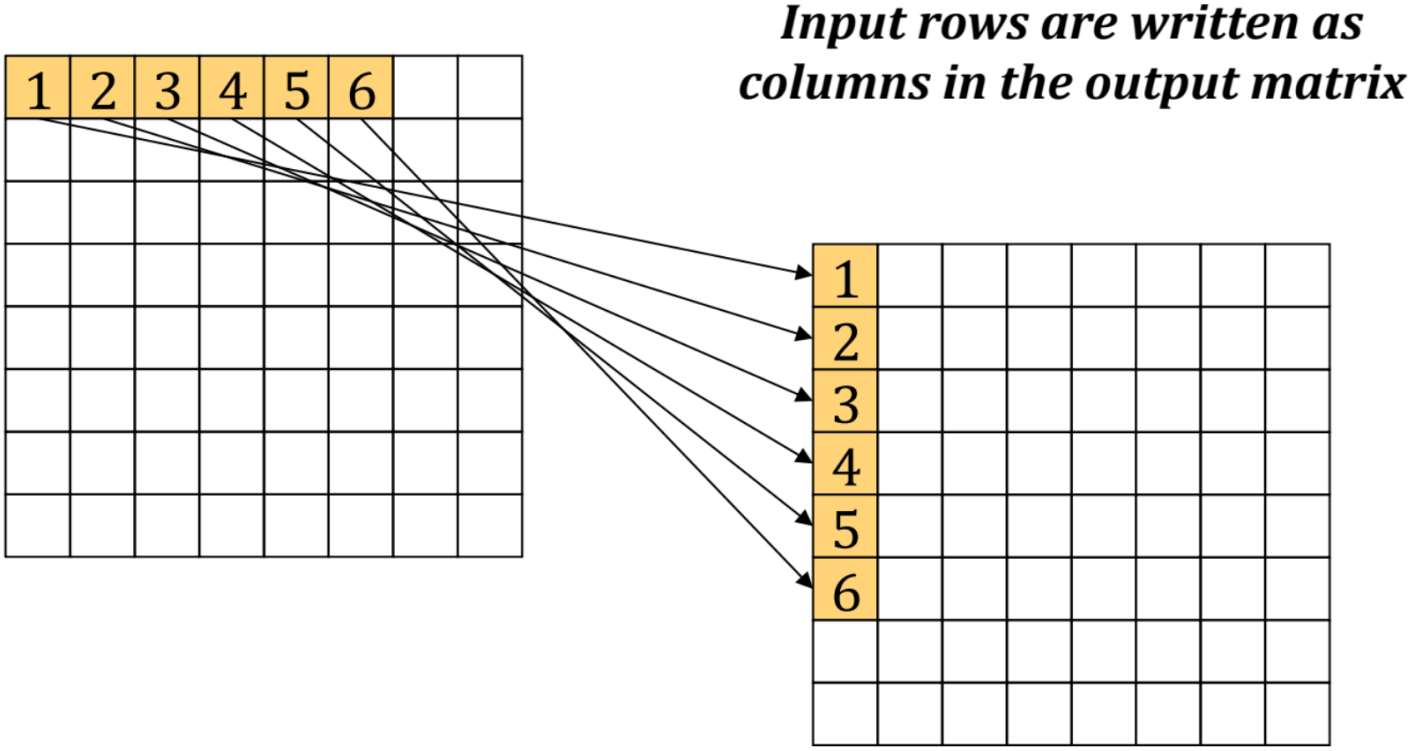
Linear algebra example

```
// Straightforward and inefficient way

__global__ void simpleMultiply(float *a,
float* b, float *c, int N)
  {
   int row = blockIdx.y * blockDim.y +
       threadIdx.y;
   int col = blockIdx.x * blockDim.x +
       threadIdx.x;
   float sum = 0.0f;
   for (int i = 0; i < TILE_DIM; i++) {
      sum += a[row*TILE_DIM+i] * b[i*N+col];
   }
   c[row*N+col] = sum;
  }
```

```
// Using shared memory to both store frequently used global
// data and to make the access coalesced – 2.3x faster on
K20

__global__ void sharedABMultiply(float *a, float* b, float
*c, int N)
  {
   __shared__ float aTile[TILE_DIM][TILE_DIM],
                                bTile[TILE_DIM][TILE_DIM];

   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   float sum = 0.0f;
   aTile[threadIdx.y][threadIdx.x] =
       a[row*TILE_DIM+threadIdx.x];
   bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
   __syncthreads();
   for (int i = 0; i < TILE_DIM; i++) {
       sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
   }
   c[row*N+col] = sum;
  }
```

For some problems coalesced access is hard

- Example: matrix transpose
- A bandwidth-limited problem that is dominated by memory access

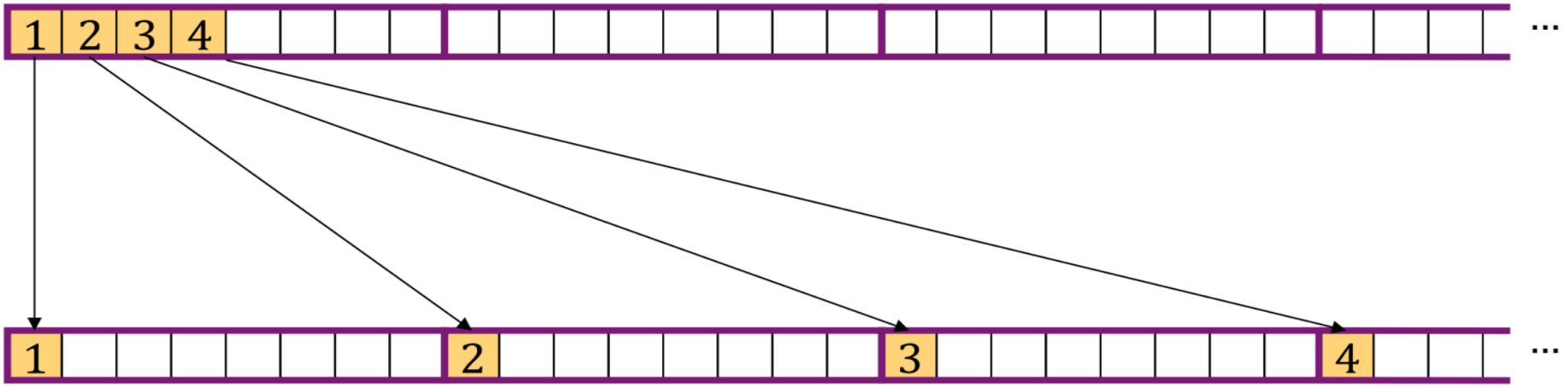


The naive matrix tranpose

```
__global__ void transpose_naive(float *odata, float *idata, int width,int height)
{
        int xIndex, yIndex, index_in, index_out;
        xIndex = blockDim.x * blockIdx.x + threadIdx.x;
        yIndex = blockDim.y * blockIdx.y + threadIdx.y;

        if (xIndex < width && yIndex < height)
        {
                index_in = xIndex + width * yIndex;
                index_out = yIndex + height * xIndex;
                odata[index_out] = idata[index_in];
        }
}
```

Naive matrix transpose (cont.)

Since the matrices are stored as 1D arrays, here's what is actually happening:



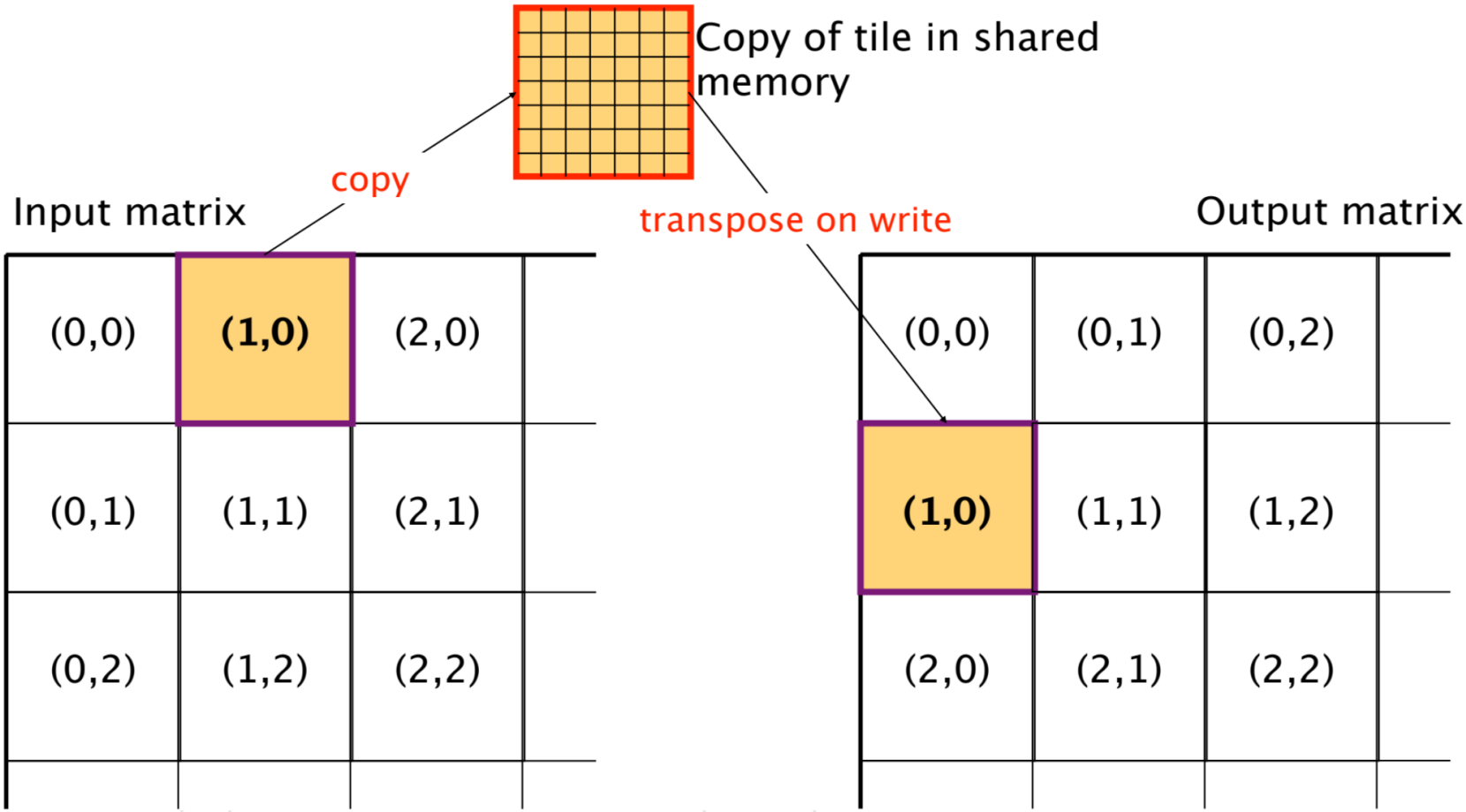
Can this problem be a avoided?

- Yes, by using a special memory which does not have a penalty when accessed in a non-coalesced way
- On the GPU this is the shared memory
- Shared memory accesses are faster than even coalesced global memory accesses. If accessing same data multiple times, try to put it in shared memory.
- Unfortunately, it is very small (64 KB)
- Must be managed by the programmer

Using shared memory

- To coalesce the writes, we will partition the matrix into 32x32 tiles, each processed by a different thread block
- A thread block will temporarily stage its tile in shared memory by copying ti from the input matrix using coalesced reads
- Each tile is then transposed as it is written out to its proper location in the output matrix
- The main difference here is that the tile is written out using coalesced writes

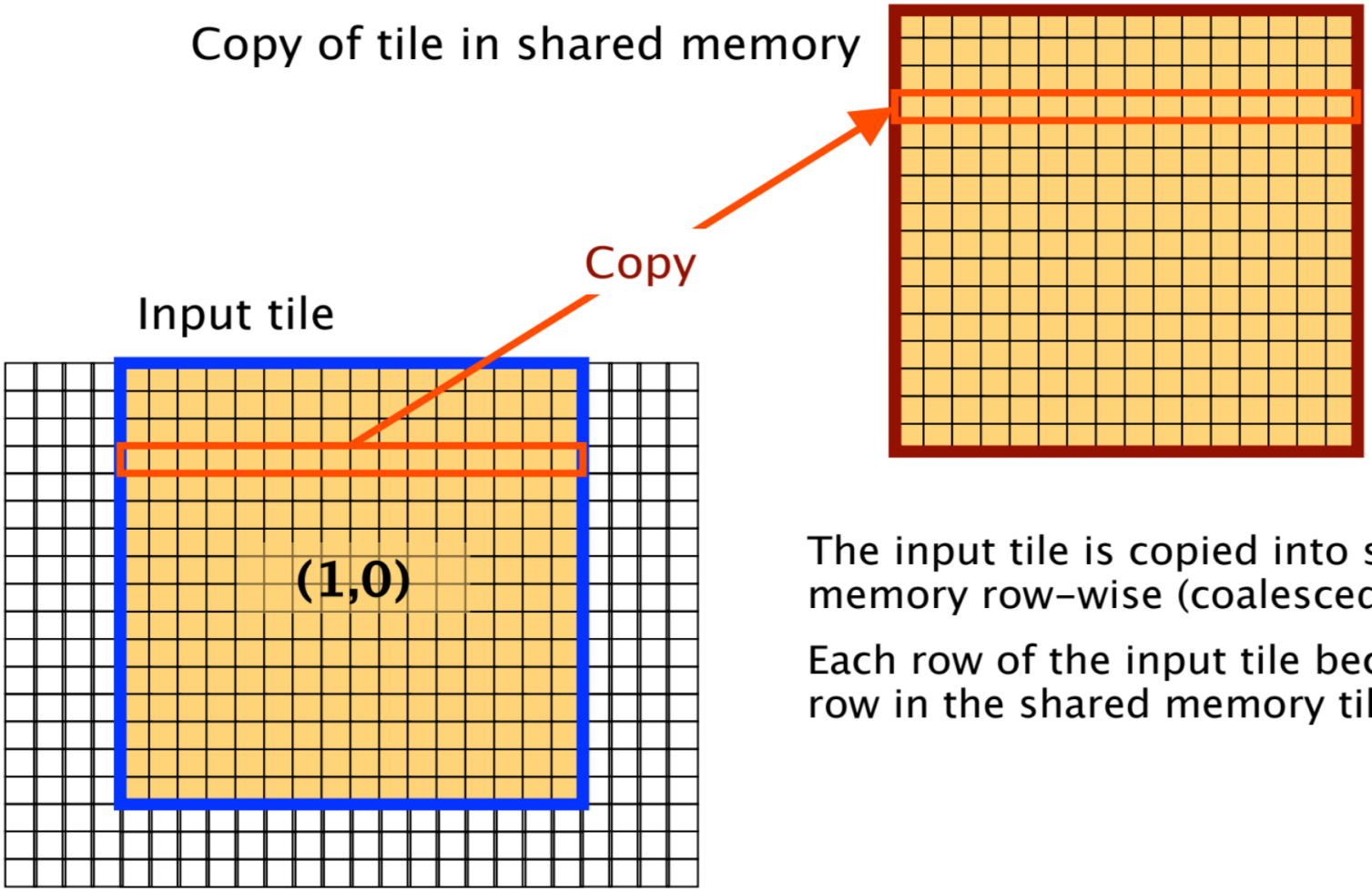
Optimized matrix transpose



Optimized matrix transpose (cont.)

```
__global__ void transpose(float *odata, float *idata,
                          int width, int height)
{
  __shared__ float block[BLOCK_DIM][BLOCK_DIM];
  unsigned int xIndex, yIndex, index_in, index_out;

  /* read the matrix tile into shared memory */
  xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
  yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
  if ((xIndex < width) && (yIndex < height))
  {
     index_in = yIndex * width + xIndex;
     block[threadIdx.y][threadIdx.x] = idata[index_in];
  }
  __syncthreads();
  /* write the transposed matrix tile to global memory */
  xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
  yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
  if ((xIndex < height) && (yIndex < width))
  {
     index_out = yIndex * height + xIndex;
     odata[index_out] = block[threadIdx.x][threadIdx.y];
  }
}
```

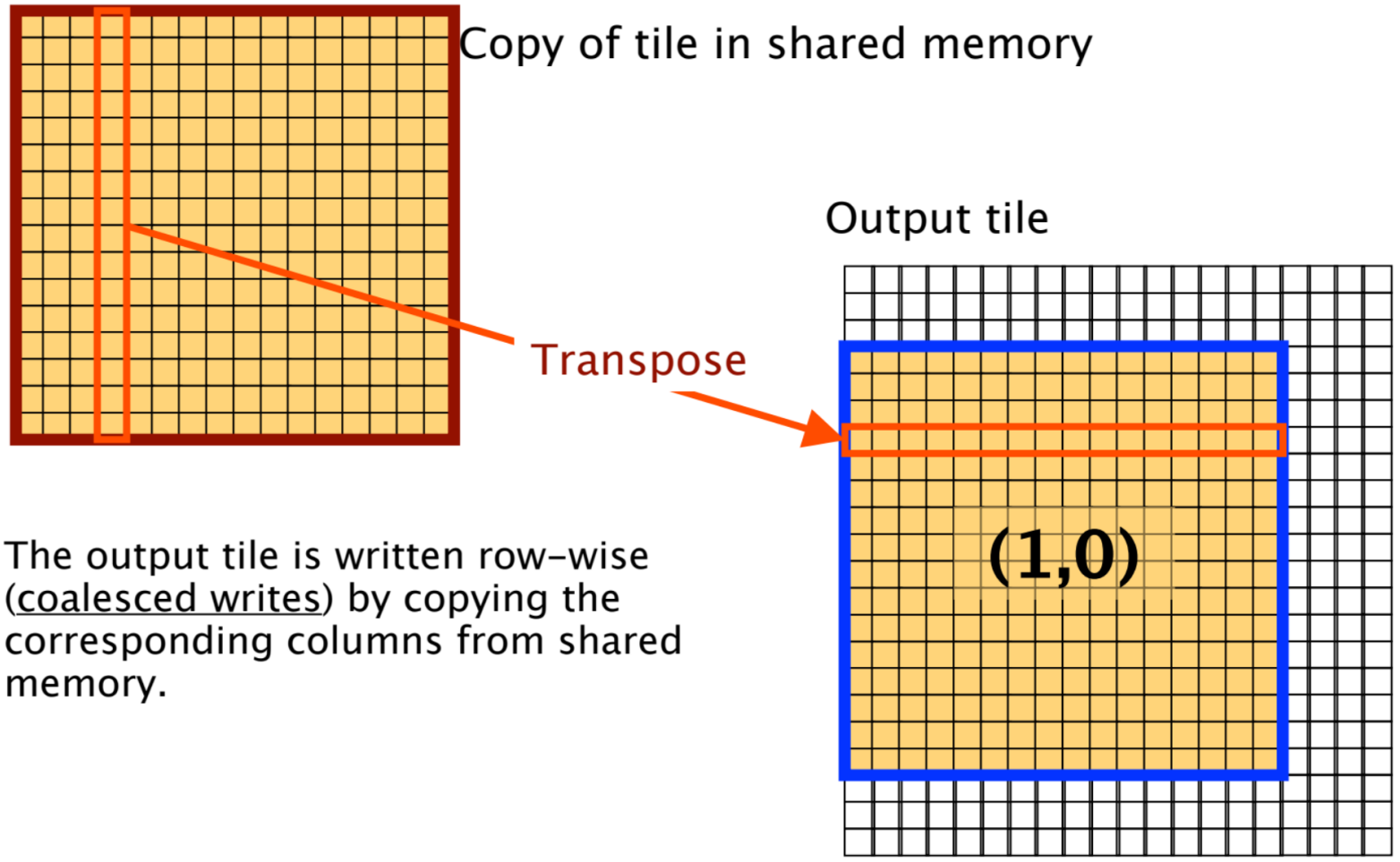
Optimized matrix transpose (cont.)



The input tile is copied into shared memory row-wise (coalesced reads).

Each row of the input tile becomes a row in the shared memory tile.

Optimized matrix transpose (cont.)



The output tile is written row-wise (coalesced writes) by copying the corresponding columns from shared memory.

One additional complication: bank conflicts

- Not a big concern but something to keep in mind
- Shared memory ***bank conflicts*** occur when the tile in shared memory is accessed column-wise
- Illustration of the need to really know the hardware when coding for GPU
- Bank conflicts matter only in highly optimised code where other sources of inefficiency have been eliminated

Shared memory banks

- To facilitate high memory bandwidth, the shared memory on each multiprocessor is organized into equally-sized ***banks*** which can be accessed simultaneously

- However, if more than one thread tries to access the same bank, the accesses must be serialized, causing delays
  - this situation is called a ***bank conflict***

- The banks are organized such that consecutive 32-bit words are assigned to consecutive banks

Shared memory banks (cont.)

- There are 32 banks, thus:

```
bank# = (index in floating point array) % 32
```

- The number of shared memory banks is equal to warp size

Bank conflict solution

- In the matrix transpose example, bank conflicts occur when the shared memory is accessed column-wise as the tile is being written

- The threads in each warp access addresses which are offset from each other by `BLOCK_DIM` elements (with `BLOCK_DIM = 32`)

- Given 32 shared memory banks, that means that all accesses hit the same bank!

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

Bank conflict solution

- The solution is surprisingly simple – instead of allocating a `BLOCK_DIM x BLOCK_DIM` shared memory tile, we allocate a `BLOCK_DIM x (BLOCK_DIM+1)` tile

- The extra padding breaks the pattern and forces concurrent threads to access different banks of shared memory
  - the columns are no longer aligned on 32-word offsets
  - no additional changes to the device code are needed

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01
02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02
03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03
04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04
05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05
06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06
07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07
08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08
09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10
11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11
12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13
14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Optimized matrix transpose (2)

```
__global__ void transpose(float *odata, float *idata,
                          int width, int height)
{
  __shared__ float block[BLOCK_DIM][BLOCK_DIM + 1];
  unsigned int xIndex, yIndex, index_in, index_out;

  /* read the matrix tile into shared memory */
  xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
  yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
  if ((xIndex < width) && (yIndex < height))
  {
     index_in = yIndex * width + xIndex;
     block[threadIdx.y][threadIdx.x] = idata[index_in];
  }
  __syncthreads();
  /* write the transposed matrix tile to global memory */
  xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
  yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
  if ((xIndex < height) && (yIndex < width))
  {
     index_out = yIndex * height + xIndex;
     odata[index_out] = block[threadIdx.x][threadIdx.y];
  }
}
```

Performance

- Tesla P100 GPU on graham, transpose of 8192x8192 matrix of SP floats
- Averaged over 100 runs:

Size	time (ms)	Speedup
simple memory copy	1.30	—
simple memory copy +shared	1.32	—
naive	7.51	x 1.0
coalesced	1.77	x 4.2
coalesced, bank optimized	1.61	x 4.7
CUBLAS	1.46	x 5.1

- timings don’t include data transfers!!!

Cache effects

- Access to main memory is relatively slow
- Computers get around this by trying to put variables that are used multiple times into cache, a small but very fast region of memory
- If a program tries to access a variable, it first tries to find it in the cache. If it does not find it there, it gets it from main memory which takes much longer. That is called a cache miss.
- Minimizing cache misses is crucial to improving program performance, both on GPUs and CPUs

Cache hierarchy

Cache is fast but expensive. The faster it is, the less you have.

Cache hierarchy

L1 - right on multiprocessor, very fast but very small (24 KB on P100)

L2 - shared between multiprocessors (4096 KB on P100)

Random numbers in CUDA

Random numbers in parallel

- Pseudo random number generation is not trivial in parallel.
- The best approach is to instruct each thread
  - to start with a unique initial state
    - In the simplest case, one can just set the initial “seed” number to the global thread ID
  - to continue along the chosen (unique) sequence by means of saving/restoring the current thread-specific state.

cuRAND

- In CUDA, all these tasks are handled by a special library cuRAND, bundled with CUDA.
- It has all the components one would need for parallel pseudo-random number generation:
  - Parallel initialization of unique (per-thread) random number generator states, via `curand_init()`
  - Ability for threads to read (usually at the beginning of a kernel) and write (usually at the end of the kernel) the current state of the generator, which is thread specific.
  - A number of functions to generate random numbers following different distributions (uniform, normal, poisson etc.)

cuRAND (2)

- To use the library, one has to include the relevant header file:

```
#include <curand_kernel.h>
```

- Load the “cuda” module:

```
$ module load cuda
```

- If you compile the code with `nvcc`, there is no need to provide library specific `-I`, `-L` and `-l` flags:

```
$ nvcc -O3 your_curand_code.cu
```

Random states vector

- The first step is to allocate a vector which will contain the states of the random number generator
  - Number of elements is equal to the total number of threads which will be generating random numbers (typically all threads in a kernel)

```
// Initializing the device random number generator:
curandState* d_states;
cudaMalloc ( &d_states, N_BLOCKS*BSIZE*sizeof( curandState ) );
```

  Here `N_BLOCKS` is the number of blocks, and `BSIZE` is the block size (number of threads in each block).

- You can skip this step if in your code random number generation takes place only in one kernel.

Initializing the states

- Next, we need to initialize all the random states. This can be accomplished in a separate kernel:

```
__global__ void initialize_states ( curandState * globalState, int seed)
{
    // Global thread index:
    unsigned long long id = blockIdx.x*blockDim.x + threadIdx.x;

    // Generating initial states for all threads in a kernel:
    curand_init ( (unsigned long long)seed, id, 0, &globalState[id] );
    return;
}
```

Generating random numbers

- Now we can write one or more kernels which will generate random numbers
- Each kernel will look like this:

```
__global__ void my_kernel ( curandState* globalState)
{
    // Global thread index:
    int id = threadIdx.x + blockDim.x*blockIdx.x;
    // Reading global states from device memory at the start of the kernel:
    curandState localState = globalState[id];
    …
    // Generating random numbers (as many times as needed):
    float r = curand_uniform(&localState);
    …
    // Writing the global states to device memory at the end of the kernel:
    globalState[id] = localState;
    return;
}
```

Distributions

```
unsigned int curand (curandState_t *state)

float curand_uniform (curandState_t *state)

float curand_normal (curandState_t *state)

float curand_log_normal (curandState_t *state)

unsigned int curand_poisson (curandState_t *state, double lambda)

double curand_uniform_double (curandState_t *state)

double curand_normal_double (curandState_t *state)

double curand_log_normal_double (curandState_t *state)
```

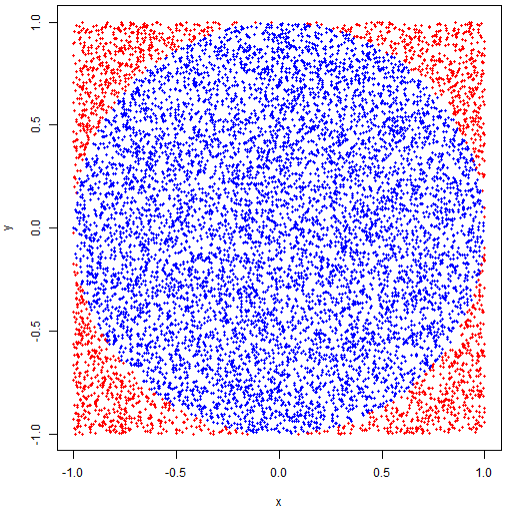
Single kernel example

- If only one kernel needs to generate random numbers, we can fit everything into that kernel:

```
__global__ void my_kernel ( int seed )
{
   // Global thread index:
   int id = threadIdx.x + blockDim.x*blockIdx.x;
   curandState localState;
   // Generating initial per-thread state:
   curand_init ( (unsigned long long)seed, id, 0, &localState );
   ...
   // Generating random numbers (as many times as needed):
   float r = curand_uniform(&localState);
   ...
   return;
}
```

In-class exercise

- Write from scratch a single-kernel CUDA code to compute π number using random numbers.
- Use 56 blocks (for P100) of 256 threads, with each thread generating NLOOP=10^5 random points sequentially.
- Each computation consists of generating random `x` and `y` (`double` type) in the interval [-1,1]. If `(x^2+y^2)<1`, we increment the "inner points" counter, `N_inner`.
- At the end of the kernel, use `atomicAdd(` to compute the total `N_inner`, which we copy to the host (cudaMemcpyFromSymbc
- Then we compute π as:
  `Pi = 4*N_inner/(56*256*NLOOP)`



Tips

- Header:
  - `#include <stdio.h>`
  - `#include <cuda.h>`
  - `#include <curand_kernel.h>`
- Inside the (only) kernel, use
  - `curand_init`
  - `curand_uniform_double`
  - `atomicAdd`
- In the host code, use
  - `cudaMemcpyFromSymbol`
  - `cudaDeviceSynchronize`