CSE 746 - Parallel and High Performance Computing Lecture 3 - Optimizing memory access on GPUs

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo
ppomorsk@sharcnet.ca
http://ppomorsk.sharcnet.ca/

Introduction to GPU Programming: CUDA

OPTIMIZATION STRATEGIES

Beyond the basics…

- Exposing parallelism
- Memory address coalescing
- Shared memory
- Thread synchronization

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 device

- 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

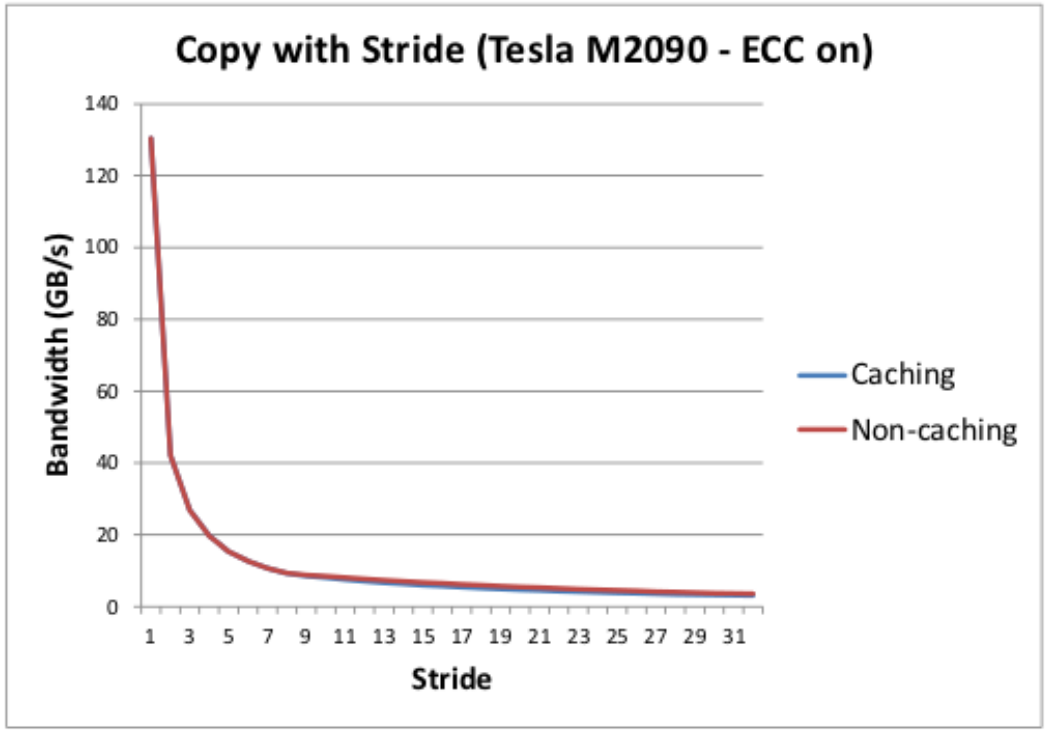
- A GPU has potentially high bandwidth for data transfer from global memory to cores. However, the latency for this transfer for any individual thread is also high (hundreds of cycles)
- Using many threads, latency can be overcome by hiding it among many threads.
  - group of threads requests some memory, 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

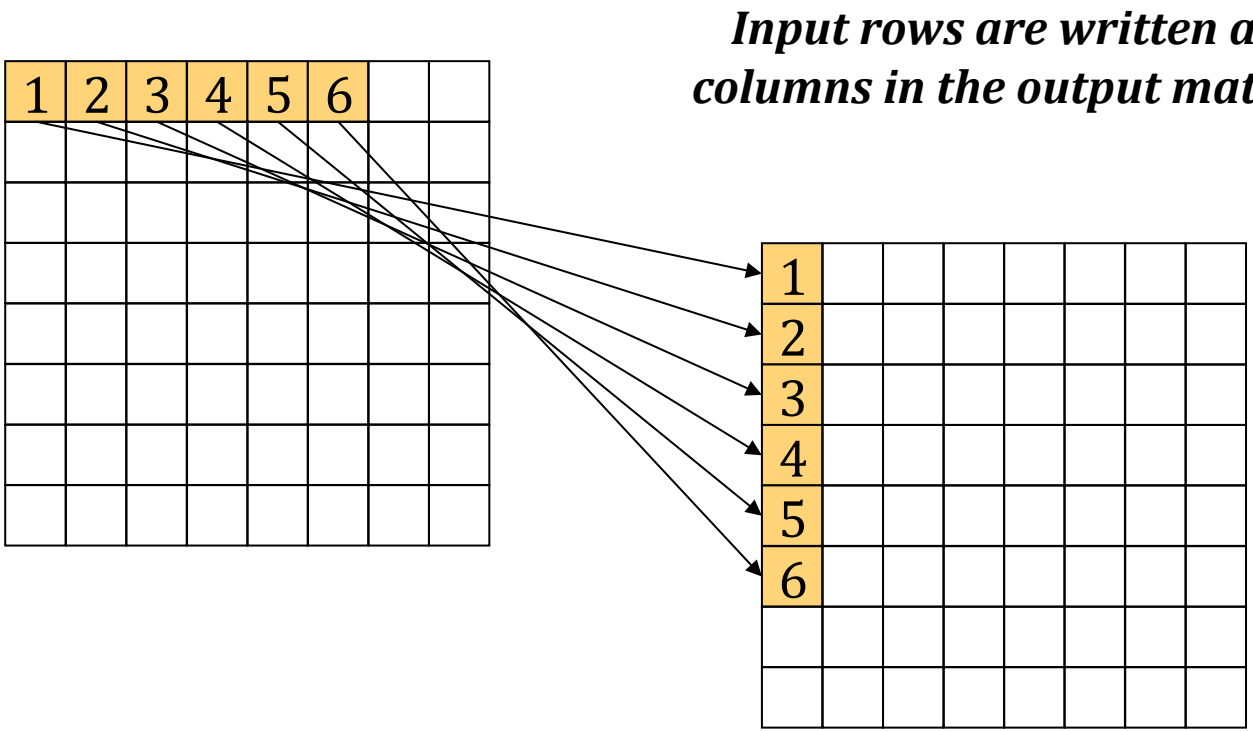
Coalesced access to memory is essential

- i=threadIdx.x; b[i]=a[i+stride]; a[i+stride]=c[i];
- performance worsens as stride increases



For some problems coalesced access is hard

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

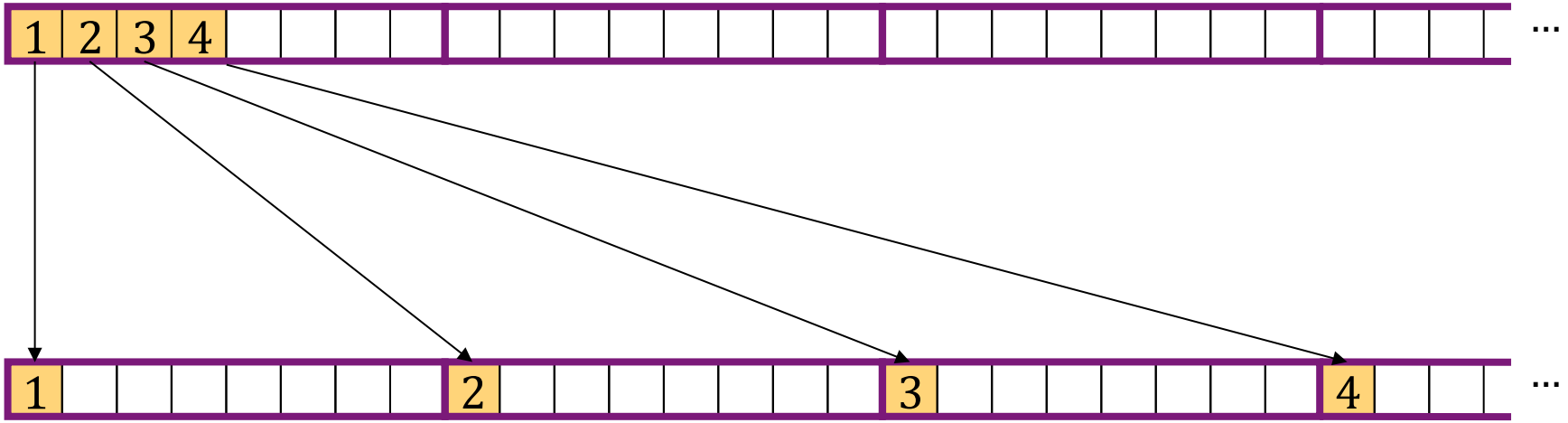


The naïve matrix transpose

```
__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];
    }
}
```

Naïve matrix transpose (cont.)

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



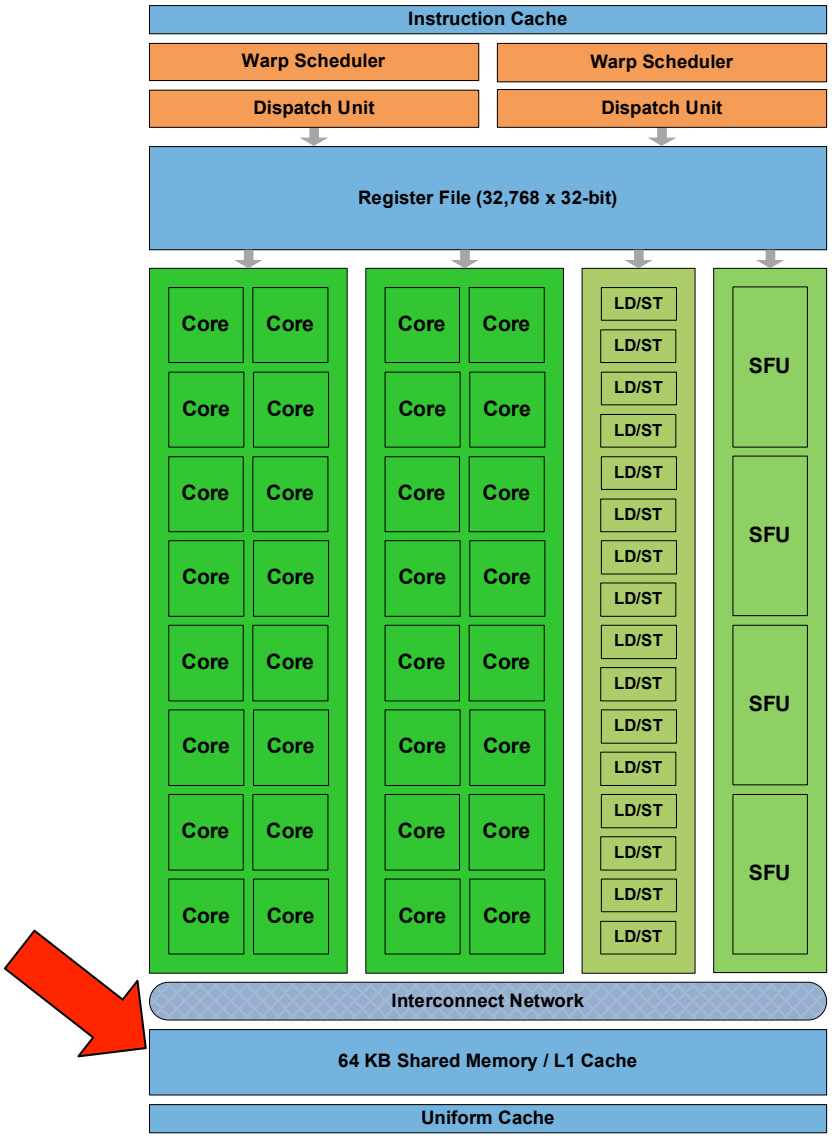
Can this problem be a avoided?

- 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 accesing same data multiple times, try to put it in shared memory.
- Unfortunately, it is very small (48 KB or 16KB)
- Must be managed by the programmer

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

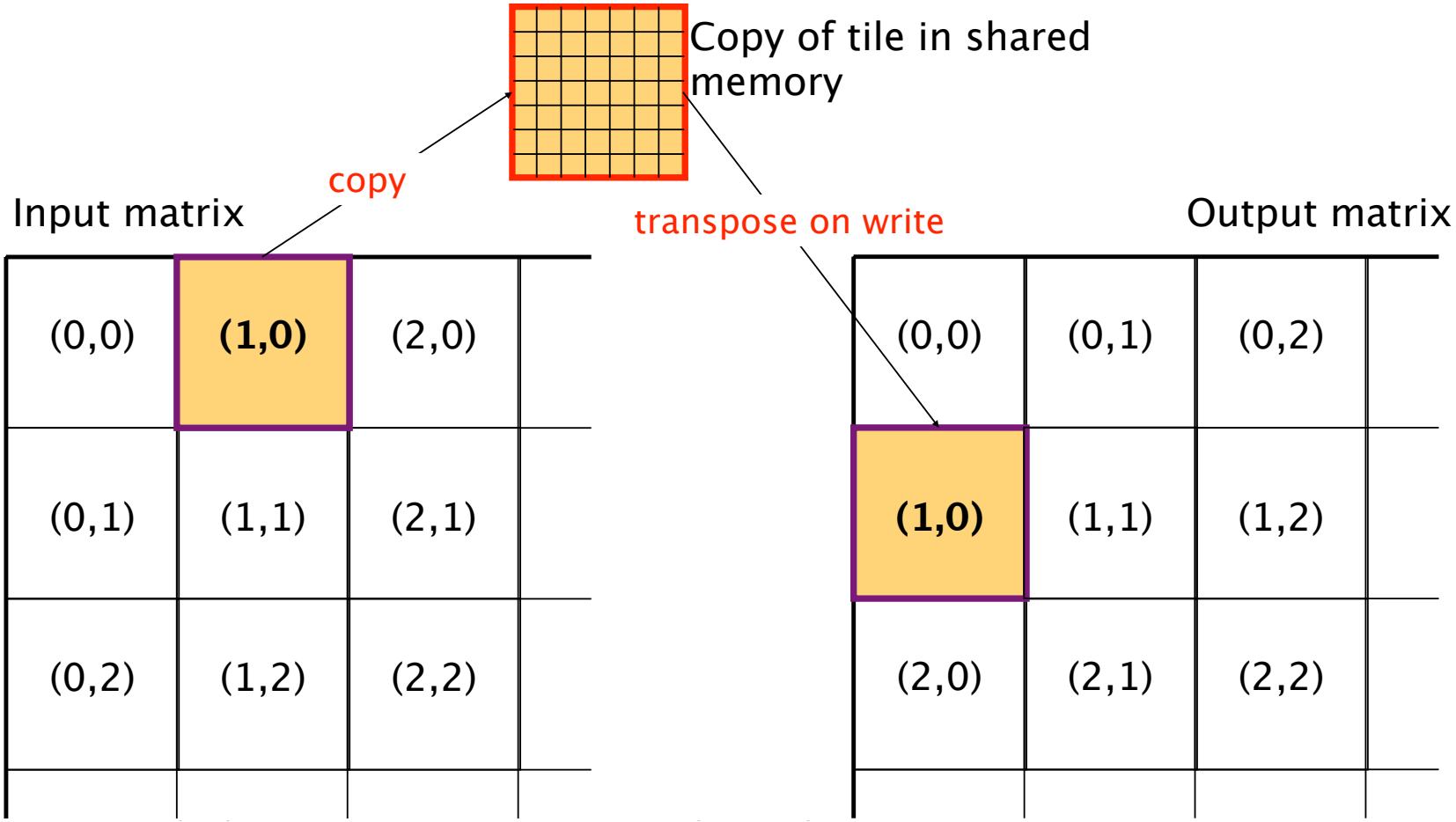
```
__syncthreads();
```



Using shared memory

- To 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 it from the input matrix using coalesced reads
- Each tile is then transposed as it is written out to its properly location in the output matrix
- The main difference here is that the tile is written out using coalesced writes

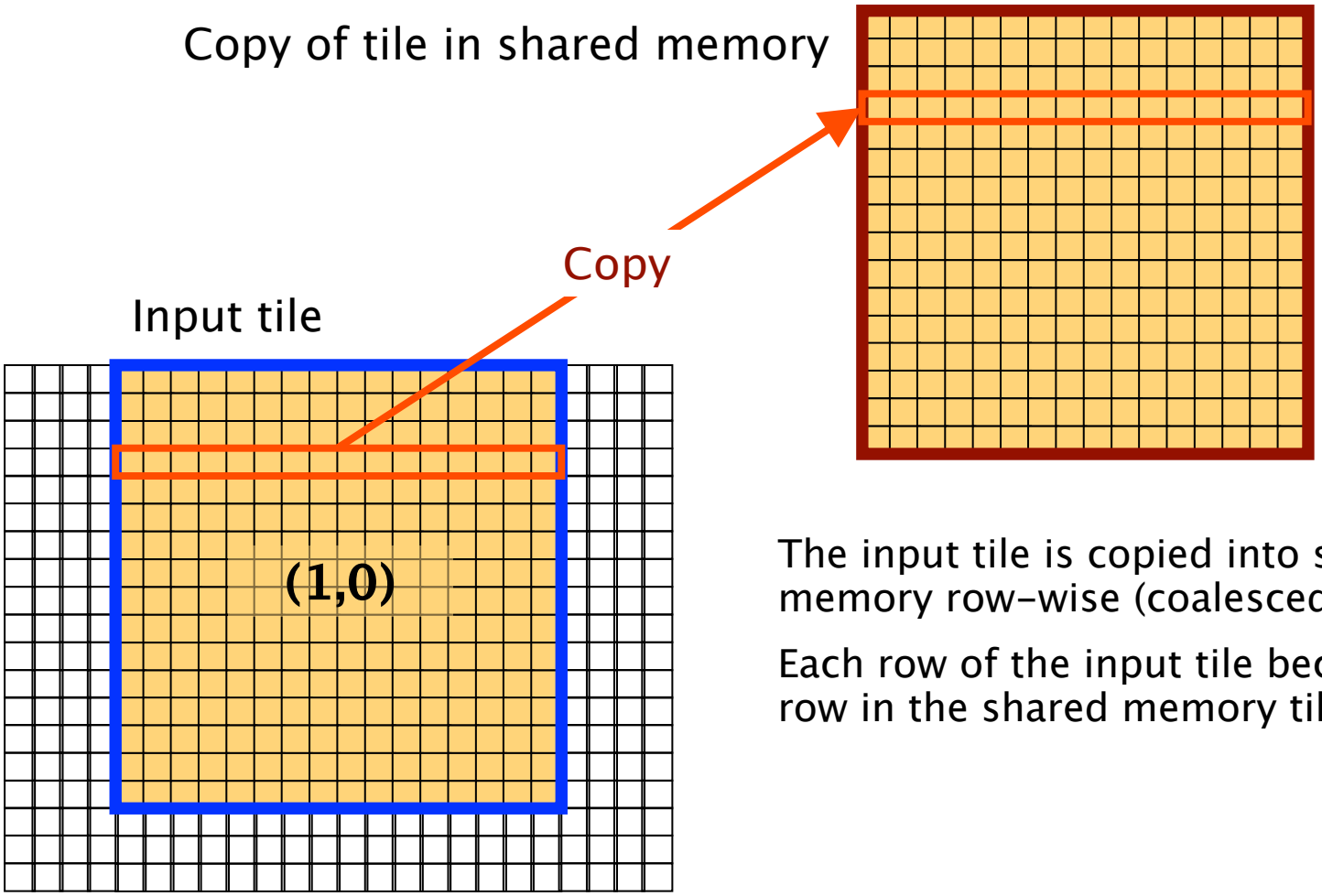
Optimized matrix transpose



Optimized matrix transpose (1)

```
__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];
    }
}
```

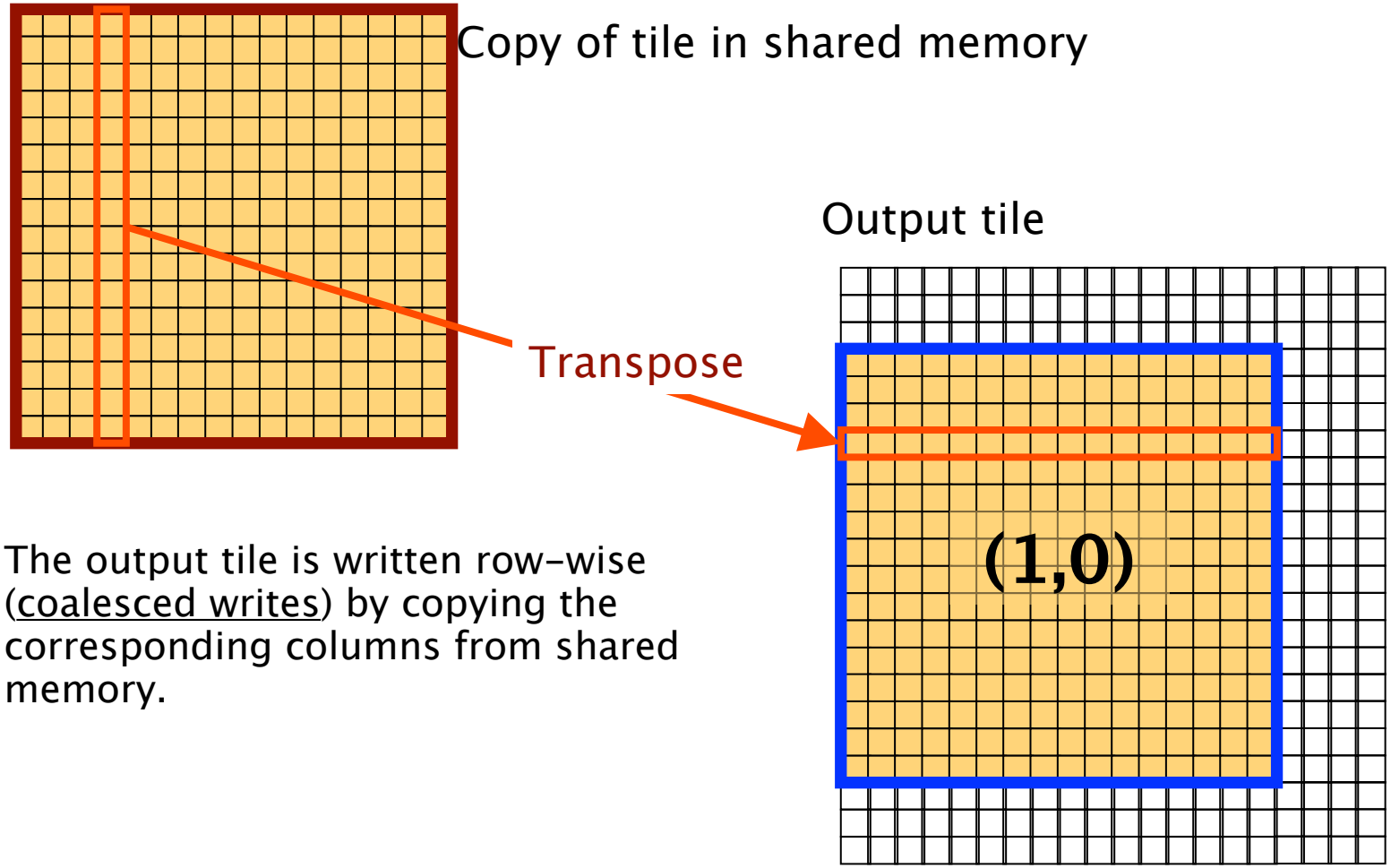
Optimized matrix transpose (cont.)



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# = address % 32
```

- The number of shared memory banks is closely tied to the warp size
- Shared memory accesses are serviced such that the threads in the first half of a warp and the threads in the second half of the warp will not cause conflicts
- Thus we have `NUM_BANKS = 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!

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

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];
    }
}
```

Exercise - study transpose sample from SDK

Install your own copy of SDK via:

cp -R /opt/sharcnet/cuda/6.0.37/samples /scratch/$USER/gpucomputingsdk

cd /scratch/$USER/gpucomputingsdk/

make (if you want to compile all samples, will take a while)

To compile only the Transpose sample:

cd 6_Advanced/transpose/
make
./transpose (this will run the executable you created)

You can experiment by modifying transpose.cu and running make again.
Useful documentation is in the *doc* subdirectory.
Can compare CC 2.0 (monk) and other machines (cat.sharcnet.ca)

Exercise - study transpose sample from SDK

[ppomorsk@mon54:/scratch/ppomorsk/gpucomputingsdk/6_Advanced/transpose] ./transpose

Transpose Starting...

Matrix size: 1024x1024 (64x64 tiles), tile size: 16x16, block size: 16x16

transpose simple copy , Throughput = 75.0985 GB/s, Time = 0.10403 ms, Size = 1048576 fp32 elements, NumDevsUsed = 1, Workgroup = 256

transpose shared memory copy, Throughput = 72.6449 GB/s, Time = 0.10754 ms, Size = 1048576 fp32 elements, NumDevsUsed = 1, Workgroup = 256

transpose naive , Throughput = 44.9562 GB/s, Time = 0.17378 ms, Size = 1048576 fp32 elements, NumDevsUsed = 1, Workgroup = 256

transpose coalesced , Throughput = 64.1410 GB/s, Time = 0.12180 ms, Size = 1048576 fp32 elements, NumDevsUsed = 1, Workgroup = 256

transpose optimized , Throughput = 65.4518 GB/s, Time = 0.11936 ms, Size = 1048576 fp32 elements, NumDevsUsed = 1, Workgroup = 256

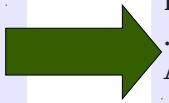
transpose coarse-grained , Throughput = 65.6251 GB/s, Time = 0.11905 ms, Size = 1048576 fp32 elements, NumDevsUsed = 1, Workgroup = 256

transpose fine-grained , Throughput = 71.0186 GB/s, Time = 0.11001 ms, Size = 1048576 fp32 elements, NumDevsUsed = 1, Workgroup = 256

transpose diagonal , Throughput = 59.0445 GB/s, Time = 0.13232 ms, Size = 1048576 fp32 elements, NumDevsUsed = 1, Workgroup = 256

Higher dimensional data needs coalesced access

```
// non coalesced

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



```
// coalesced

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

Static arrays on GPU should be declared with __device__ keyword

For copying between static arrays, can use

cudaMemcpyToSymbol
cudaMemcpyFromSymbol

Exercise - improve access pattern

Can be found in:

/home/ppomorsk/CSE746_lec3/coalesce

CUDA - Concurrent execution and streams

Concurrency (parallel execution) between GPU and CPU is either a default, or easily enabled behaviour

Kernel launches are always asynchronous with regards to the host code; one has to use explicit device-host synchronization any time kernel needs to be synchronized with the host:

CudaDeviceSynchronize ()

CUDA - Concurrent execution and streams

The default behaviour of GPU<->CPU memory copy operations is asynchronous for small transfers, and synchronous otherwise. But one can enforce any memory copying to be asynchronous by adding Async suffix, e.g.:

cudaMemcpyAsync()
cudaMemcpyToSymbolAsync()

For debugging purposes, one can enforce everything to be synchronous by setting the CUDA_LAUNCH_BLOCKING environment variable to 1.

Concurrent execution and streams

Concurrency between different device operations (kernels and/or memory copying) is a completely different story

On a hardware level, modern GPUs are capable of running multiple kernels and memory transfers both to and from the device concurrently

By default, everything on device is done serially (no concurrency)

To make use of the device concurrency features, one has to start using multiple streams in the CUDA code

But even with multiple streams, there are some limitations to concurrency on GPU

Concurrent execution and streams

- A stream is a sequence of commands (possibly issued by different host threads) that execute in order
- If stream ID is omitted, it is assumed to be “0” (default) stream. For non-default streams, the IDs have to be used explicitly:

```
mykernel <<<Nblocks, Nthreads, 0, ID>>> ();
cudaMemcpyAsync (d_A, h_A, size, cudaMemcpyHostToDevice, ID);
```

- Before using, streams have to be created.
- At the end, they have to be destroyed

```
// Host code

cudaStream_t ID[2];

// Creating streams:
for (int i = 0; i < 2; ++i) cudaStreamCreate (&ID[i]);

// These two commands will run concurrently on GPU:

mykernel <<<Nblocks, Nthreads, 0, ID[0]>>> ();
cudaMemcpyAsync (d_A, h_A, size, cudaMemcpyHostToDevice, ID[1]);

// Destroying streams:
for (int i = 0; i < 2; ++i) cudaStreamDestroy (ID[i]);
```

Limitations

For memory copying operations to run concurrently with any other device operation (kernel or an opposite direction memory copying operation), the host memory has to be page-locked (or pinned; allocated with **cudaMallocHost** instead of malloc; static variables can be made pinned using **cudaHostRegister**)

Up to 16 kernels can run concurrently

Concurrency on GPU is not guaranteed (e.g., if kernels use too much local resources, they will not run concurrently)

Other stream-related commands

- cudaDeviceSynchronize() : global synchronization (across all the streams and the host);
- cudaStreamSynchronize (ID) : synchronize stream ID with the host;
- cudaStreamQuery (ID) : tests if the stream ID has finished running.

Streams and memory transfers

- streams can sometimes help with memory transfers between device and host
- one possible scenario: overlap memory transfer with host computation

```
// On host:

// This memory copying will be asynchronous only in regards to the host code:
cudaMemcpyAsync (d_a, h_a, size, cudaMemcpyHostToDevice, 0);

// This host code will be executed in parallel with memory copying:
serial_computation ();

// The kernel will be executed after copying and serial code is done:
kernel <<<N, M>>> (d_a);
```

Streams and memory transfers

- another scenario: overlap memory transfer with kernel
- one possible scenario: overlap memory transfer with host computation

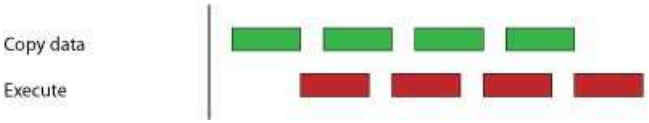
```
// This memory copying will be asynchronous in regards to the host and stream ID[1]:
cudaMemcpyAsync (d_a, h_a, size, cudaMemcpyHostToDevice, ID[0]);

// The kernel doesn't need d_a, and will run concurrently:
kernel1 <<<N, M, 0, ID[1]>>> ();

// This kernel needs d_a, but doesn't need the result of kernel1; it will run after the
// Memcpy operation, and concurrently with kernel1:

kernel2 <<<N, M, 0, ID[0]>>> ();
```

Staged copy using streams

- can use two streams to move data in stages and compute successively

One stream scenario:

You need two streams for this:



Exercise

- use starting program and convert to staged copy

  Found in:

  /home/ppomorsk/CSE746_lec3/staged