CME213/ME339
Lecture 7

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

Shared Memory Transpose

Code Example 1/3

```
__global__
void fastTranspose(int *in, int *out,
                   int M, int N)
{
    const int numWarps = blockDim.y; //let's assume 4
    const int warpId   = threadIdx.y;
    const int lane     = threadIdx.x;
    const int smemRows = 32;

    __shared__ int smem[smemRows][warpSize];

    int blockCol = blockIdx.x;
    int blockRow = blockIdx.y;
```

Shared Memory Transpose

Code Example 2/3

```
//load 32x32 block into shared memory
for (int i = 0; i < smemRows / numWarps; ++i) {
    int gr = blockRow * warpSize + i * numWarps + warpId;
    int gc = blockCol * warpSize + lane;
    int row = i * numWarps + warpId;

    smem[row][lane] = in[gr * N + gc];
}

__syncthreads(); //<-- Important!
```

Shared Memory Transpose

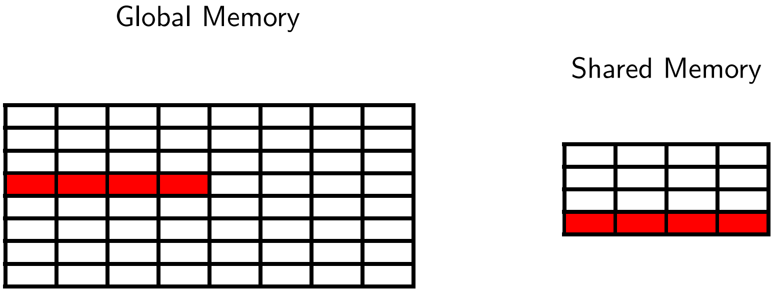
Code Example 3/3

```
//now we switch to each warp outputting a row, which will read
//from a column in the shared memory
//this way everything remains coalesced
for (int i = 0; i < smemRows / numWarps; ++i) {
    int gr = blockRow * warpSize + lane;
    int gc = blockCol * warpSize + i * numWarps + warpId;
    int row = i * numWarps + warpId;

    out[gc * M + gr] = block[lane][row];
}
```

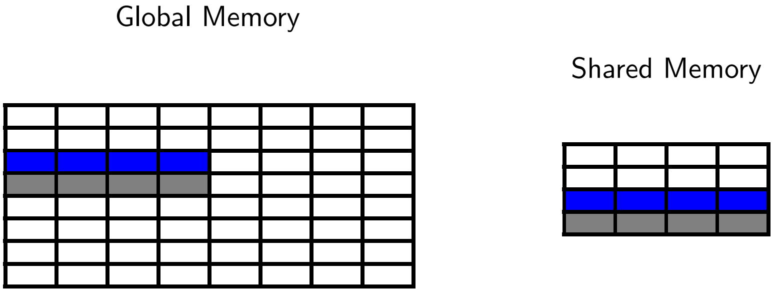
Block Synchronization

- `__syncthreads()` is a block level barrier
- All threads much reach this location before they are allowed to proceed
- Common use case is:
  - Load data
  - `__syncthreads()`
  - Perform computation
  - `__syncthreads()`
  - Write data

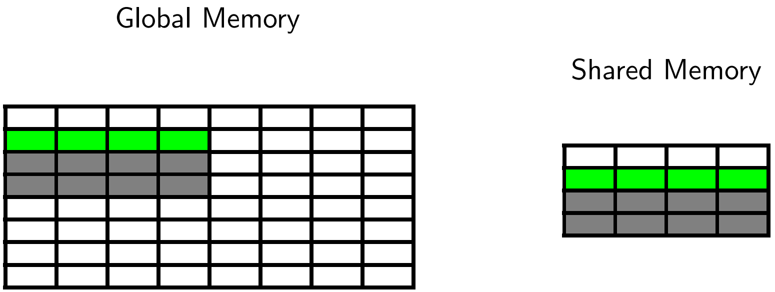
Read Memory Access Pattern



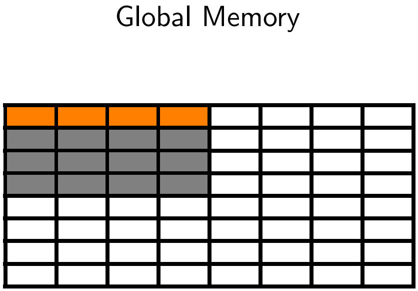
Read Memory Access Pattern

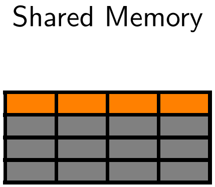


Read Memory Access Pattern

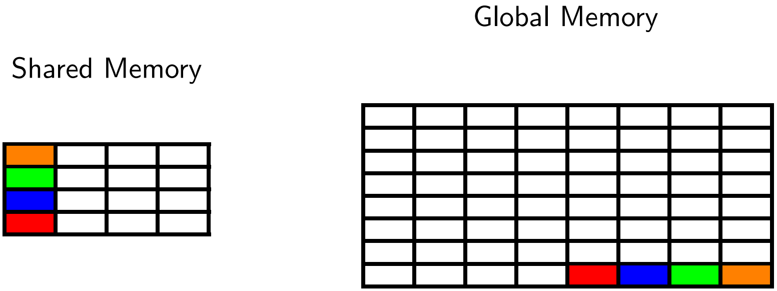


Read Memory Access Pattern

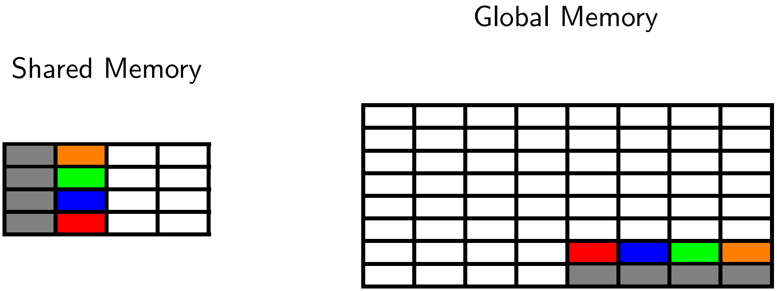




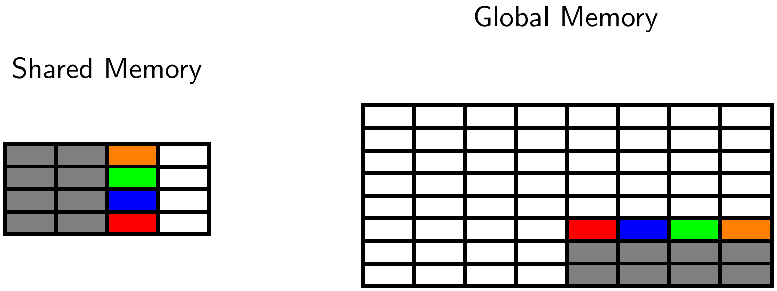
Write Memory Access Pattern



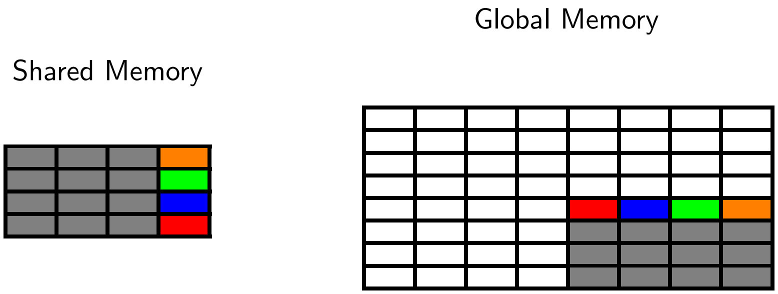
Write Memory Access Pattern



Write Memory Access Pattern



Write Memory Access Pattern



Performance

2D Version w/o Shared Memory

# Array Dimensions	GB/sec
(256, 256)	53
(512, 512)	63
(1024, 1024)	70
(2048, 2048)	69
(4096, 4096)	71

Shared Memory Version

# Array Dimensions	GB/sec
(256, 256)	22
(512, 512)	46
(1024, 1024)	57
(2048, 2048)	64
(4096, 4096)	68

It got worse?

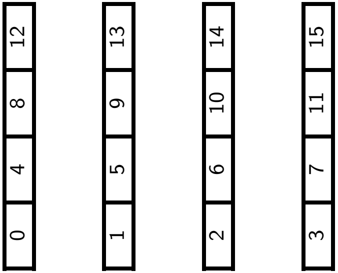
Shared Memory

Banks

- We need to understand the concept of banked memory to be able to explain why the performance actually went down
- The actual hardware implementation of shared memory unfortunately doesn't allow for constant time access to arbitrary locations
- Instead there are 32 banks
- Within each bank there is constant time access to arbitrary locations

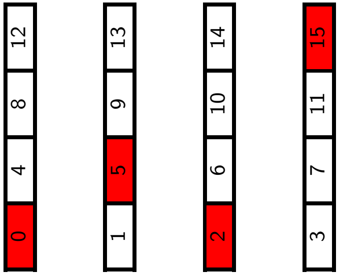
Banks

- Suppose we have 4 memory banks
- Columns correspond to banks
- Our warp (of 4 threads) wants to access locations: 0, 2, 5, 15



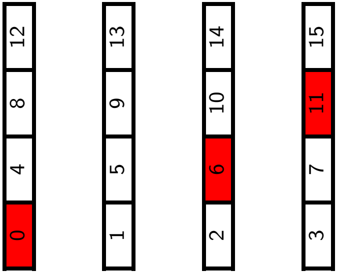
Banks

- This is OK
- Each bank needs one value
- No conflicts



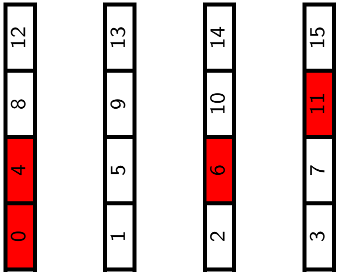
Banks

- 0, 0, 6, 11
- Each bank needs one value
- No conflicts
- Duplication not a problem



Banks

- 0, 4, 6, 11
- Bank 0 needs two values
- Conflict!
- Causes serialization — it takes twice as long to read these values



Back To The Code

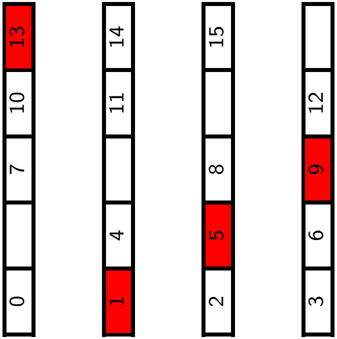
```
//Write out to global memory
for (int i = 0; i < smemRows / numWarps; ++i) {
    int gr = blockRow * warpSize + lane;
    int gc = blockCol * warpSize + i * numWarps + warpId;
    int row = i * numWarps + warpId;

    out[gc * M + gr] = block[lane][row];
}
```

- We can see that each warp reads from the same column of shared memory
- Each row is a multiple of the warp size
- Which means the warp is reading from the same bank

Solution

Make the number of columns coprime to the warp size — easiest is to just add one

```
__shared__ int smem[smemRows][warpSize + 1];
```



New Performance Numbers

2D Version w/o Shared Memory

# Array Dimensions	GB/sec
(256, 256)	53
(512, 512)	63
(1024, 1024)	70
(2048, 2048)	69
(4096, 4096)	71

Shared Memory Version — No Bank Conflicts

# Array Dimensions	GB/sec
(256, 256)	70
(512, 512)	88
(1024, 1024)	112
(2048, 2048)	123
(4096, 4096)	124

Not bad — but we can do even better!

C++ Templates

- Templates *can* be used with CUDA
- We will use function templates although class templates can be used as well
- If you are completely unfamiliar with templates I would recommend reading the tutorial at `http://www.cplusplus.com/doc/tutorial/templates`

Template Example

```cpp
template<typename T>
T add(const T &x, const T &y) {
    return x + y;
}

float  a, b; add<float>(a, b);
int    c, d; add<int>(c, d);
```

```cpp
template<int N>
int add(const int &x) {
    return x + N;
}

int x;
add<3>(x);
add<5>(x);
```

Templates and CUDA

- Most common usage is to specify the number of threads per block — `blockDim`
- Can also be used for things like the dimensions of shared memory

Shared Memory Transpose

Template Code Example 1/3

```
template<int numWarps> //assume 4
__global__
void fastTranspose(int *in, int *out,
                   int M, int N)
{
    const int warpId   = threadIdx.y;
    const int lane     = threadIdx.x;
    const int smemRows = 32;

    __shared__ int smem[smemRows][warpSize + 1];

    int blockCol = blockIdx.x;
    int blockRow = blockIdx.y;
```

Shared Memory Transpose

Template Code Example 2/3

```
//load 32x32 block into shared memory
#pragma unroll
for (int i = 0; i < smemRows / numWarps; ++i) {
    int gr = blockRow * warpSize + i * numWarps + warpId;
    int gc = blockCol * warpSize + lane;
    int row = i * numWarps + warpId;

    smem[row][lane] = in[gr * N + gc];
}

__syncthreads(); //<-- Important!
```

- Notice that now `smemRows / numWarps` can be determined at *compile* time to be 8.
- This allows use to use `#pragma unroll` to unroll the loop 8 times.

Shared Memory Transpose

Template Code Example 3/3

```
#pragma unroll
for (int i = 0; i < smemRows / numWarps; ++i) {
    int gr = blockRow * warpSize + lane;
    int gc = blockCol * warpSize + i * numWarps + warpId;
    int row = i * numWarps + warpId;

    out[gc * M + gr] = block[lane][row];
}
```

- Unrolling the loop reduces the loop counter overhead
- And more importantly allows the compiler to have more freedom to reorder instructions

Final Performance Numbers

Shared Memory Version — No Bank Conflicts

# Array Dimensions	GB/sec
(256, 256)	70
(512, 512)	88
(1024, 1024)	112
(2048, 2048)	123
(4096, 4096)	124

Shared Memory Version w/ Unrolling

# Array Dimensions	GB/sec
(256, 256)	78
(512, 512)	113
(1024, 1024)	124
(2048, 2048)	128
(4096, 4096)	130

Peak is 152 GB/sec (for this particular card) but 130 GB/sec is usually the maximum seen in practice.

Homework 2

Solving the 2D Heat Diffusion Equation

Assume a temperature field $u(t, x, y)$. A simple heat diffusion in an homogeneous isotropic medium can be modeled using the heat equation:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

This corresponds for example to a plate that has been heated and that progressively cools down.

We solve it using a numerical technique called finite differences:

$$\begin{aligned} u_{i,j,t+1} = u_{i,j,t} &+ \mathit{xcfl}\,(u_{i-1,j,t} - 2\, u_{i,j,t} + u_{i+1,j,t}) \\ &+ \mathit{ycfl}\,(u_{i,j-1,t} - 2\, u_{i,j,t} + u_{i,j+1,t}) \end{aligned}$$

Homework 2

Finite Difference Solution

- In this homework the region we are simulating will always be a rectangle
- Basic idea is to break the rectangle up into many grid points and solve the previous equation at each point
- We need boundary conditions — for the homework we simply specify a fixed temperature for each of the four sides of the rectangle
- And we need initial conditions — the temperature everywhere in the interior at time 0

	0	0	0	0	0	0	
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
	0	0	0	0	0	0	

Homework 2

Finite Difference Solution

- Once we have our starting grid we advance to the next time by applying the formula from Slide 30.

Input

	0	0	0	0	0	0	
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
	0	0	0	0	0	0	

Output

	0	0	0	0	0	0	
10							10
10							10
10							10
10							10
10							10
10	5						10
	0	0	0	0	0	0	

xcfl = ycfl = .5

Homework 2

Finite Difference Solution

- Once we have our starting grid we advance to the next time by applying the formula from Slide 30.

Input

	0	0	0	0	0	0	
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
	0	0	0	0	0	0	

Output

	0	0	0	0	0	0	
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	2.5	5	5	5	5	10
	0	0	0	0	0	0	

$$\text{xcfl} = \text{ycfl} = .5$$

Homework 2

Orders and Precision

- The formula and stencil you've just seen is 2^{nd} order
- You also need to implement 4^{th} and 8^{th} order stencils
- They are wider — they use 2 and 4 points in each direction respectively
- See the cpu reference code for the coefficients you need
- There are always two grids `prev` which is at time t and `curr` which is at $t + 1$
- So we are always reading from `prev` and writing to `curr`

Halo cells

You may notice that the formula we provided:

$$u_{i,j,t+1} = u_{i,j,t} + xcfl\ (u_{i-1,j,t} - 2\ u_{i,j,t} + u_{i+1,j,t}) \\ + ycfl\ (u_{i,j-1,t} - 2\ u_{i,j,t} + u_{i,j+1,t})$$

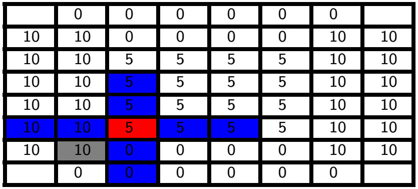
only allows updating nodes if all four values around the node are known. In particular it cannot be used to update a node right on the boundary.

The nodes around (i, j) that are required to update $u_{i,j,t+1}$ are called halo cells.

	0	0	0	0	0	0	
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
10	5	5	5	5	5	5	10
	0	0	0	0	0	0	

For a 2nd order method, this is fine since the boundary condition fixes the value of nodes on the boundary.

However for the 4th order scheme, the halo cell region is larger and includes 2 nodes in each direction.



	0	0	0	0	0	0	
10	10	0	0	0	0	10	10
10	10	5	5	5	5	10	10
10	10	5	5	5	5	10	10
10	10	5	5	5	5	10	10
10	10	5	5	5	5	10	10
10	10	0	0	0	0	10	10
	0	0	0	0	0	0	

As a result the formula given in the code for homework 2 does not allow updating nodes that are near the boundary.

In practice a special stencil would be used for these nodes.

In this homework, we simplified the problem and decided that we would not update these nodes at all. We therefore use the same stencil everywhere.

	0	0	0	0	0	0	
10	10	0	0	0	0	10	10
10	10	5	5	5	5	10	10
10	10	5	5	5	5	10	10
10	10	5	5	5	5	10	10
10	10	5	5	5	5	10	10
10	10	0	0	0	0	10	10
	0	0	0	0	0	0	

Make sure to pay close attention to this issue of halo cells. This is critical to get a correct implementation.

In the shared memory version you will need to correctly read all the values required by the calculation.

Homework 2

Templates to avoid conditionals

To handle the different orders we have a couple of options:

- We can create completely different kernels for each order
- We can pass the order in as a parameter and have an `if` statement inside the kernel
- We can pass the order in as a template parameter

The first option is preferred when no code will be duplicated between the kernels. This is the case for the global memory versions you will write.

When code will be duplicated the last option is preferred.

Homework 2

Templates to avoid conditionals

Not this

```
__global__
void myKernel(..., int order, ...) {
    //load data into shared memory

    if (order == 2) {
        //...
    }
    else if (order == 4) {
        //...
    }

    //write data out to global memory
```

Homework 2

Templates to avoid conditionals

This

```
template<int order>
__global__
void myKernel(...) {
    //load data into shared memory

    if (order == 2) {
        //...
    }
    else if (order == 4) {
        //...
    }

    //write data out to global memory
```

Get Started Now

- Recommend going through and trying to understand the cpu code that is there as soon as possible
- Try writing the global memory versions first; it will help you write the shared memory kernels, which are more complicated.
- Use a piece of paper and draw out how you are going to use shared memory, it really helps!
- The code you are given will write out to text files the initial grid, the cpu solution and your solution
- For debugging try setting the number of iterations to 1