CME213/ME339
Lecture 7

Erich Elsen

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

Spring 2013

- 2D/3D Blocks and Grids
- GPU vs CPU Philosophy
- GPU Architecture
- Warps and Memory Coalescing
- Matrix Transpose Example
- Shared Memory
- Matrix Transpose with Shared Memory

2D Arrays in Linear Memory

NOT This:

```
float **A = new float*[N];
for (int i = 0; i < N; ++i)
  A[i] = new float[N];

A[row][col] = 5.f;
```

This:

```
float *A = new float[N * N];
A[row * N + col] = 5.f;
```

- First version requires O(N) calls to new and delete vs 1 call
- First version requires *two* pointer dereferences for each item
- Chaining pointer dereferences like this is slow, you pay the memory access latency twice
- You can use the [] [] with statically declared arrays - where the bounds are given at compile time
- The compiler will automatically convert the first from into the second for you

Grid

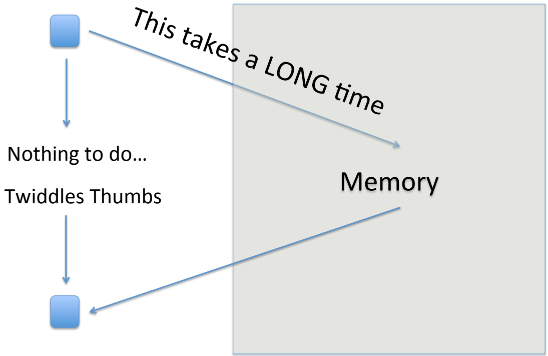
Blocks are organized into a 1D, 2D or 3D grid of thread blocks.

`threadIdx`: 1D, 2D or 3D index to identify a thread. `blockIdx`: 1D, 2D or 3D index to identify a block.

```
__global__ void MatAdd(const float* A, const float* B,
                        float* C, const int N) {
    int c = blockIdx.x * blockDim.x + threadIdx.x;
    int r = blockIdx.y * blockDim.y + threadIdx.y;
    if (r < N && c < N)
      C[r*N+c] = A[r*N+c] + B[r*N+c];
}
```

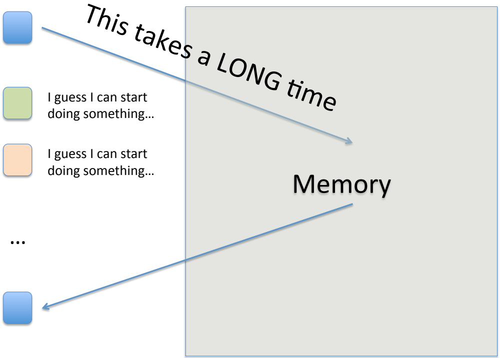
`blockDim`: dimension of the thread block
`gridDim`: dimension of the grid

CPU Philosophy



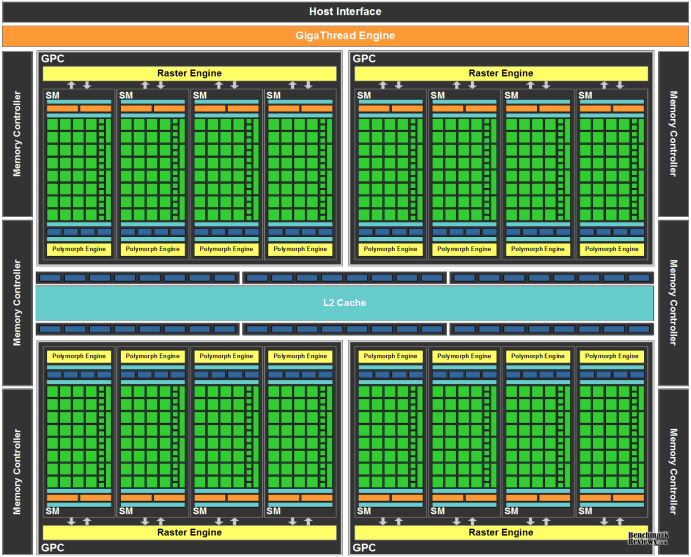
- CPUs aim to reduce latency
- Introduce many levels of large caches to decrease time to get memory
- Majority of transistors are actually cache, they don't actually do any "work"

GPU Philosophy

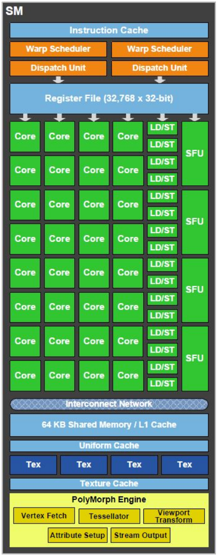


- GPUs aim to maximize throughput
- Devote more transistors to units that do "work"
- Hide memory latency with parallelism
- While some threads are waiting, swap others in
- Also requires a VERY large register set compared to CPUs

GPU Architecture Overview



SM Overview



- 32 Processing cores
- Shared Memory
- Register File
- Each block runs entirely on one SM

Warps and Memory Coalescing

- Warps are the smallest cooperative unit
- Since beginning of CUDA - 32 thread per warp
- Corresponds to the 32 processing cores in an SM
  - Could change someday
- Communication within warps very fast
- Warps cooperate to load memory
- For best performance:
  - Want the locations accessed by threads within a warp to be contiguous
  - Start at a 128-byte boundary
  - Each thread should read at least 4 bytes; 8 is better
  - But not more than 16 bytes

Warps and Divergence

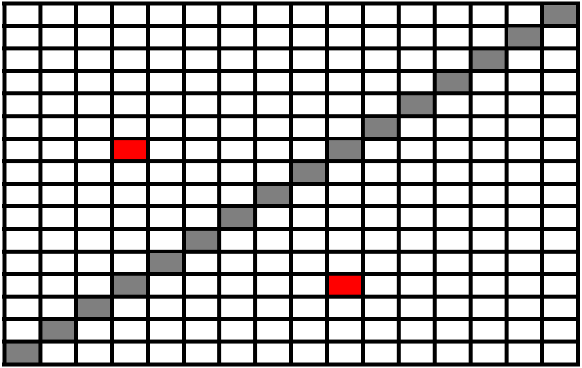
- All threads in a warp are executed simultaneously
    - Not exactly true on all hardware, but this should be your mental model
- Which means all threads must execute the same instruction
- If different threads want to execute different instructions, this is known as divergence
- Divergence is handled by the hardware for you - it will execute the warp as many times as needed to cover all branches
- This can lead to significant slow down due to many idle threads
- Loops that execute for a different number of iterations are also divergent

```
if (tid % 2)
  out[tid] = sqrtf(in[tid]);
else
  out[tid] = logf(in[tid]);
```

```
while (tid > 1) {
   if (tid % 2)
     tid /= 2;
   else
     tid = 3 * tid + 1;
}
```

Matrix Transposition

M rows; N columns $\rightarrow$ N rows; M columns



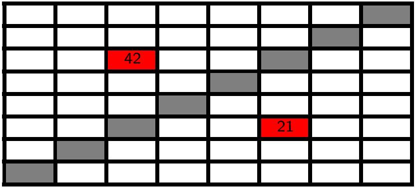
$$A(i,j) \rightarrow A(j,i)$$

Simple Implementation

```
__global__
void simpleTranspose(int *array_in, int *array_out,
                     int M, int N)
{
  const int tid = threadIdx.x + blockDim.x * blockIdx.x;

  int col = tid % N; //row, col in INPUT
  int row = tid / N;

  array_out[col * M + row] = array_in[row * N + col];
}
```

- We assume `M * N` is a multiple of `blockDim.x`
- Each block is 1D and the grid is also 1D
- Loc (n, m) in an array of size (M, N) is given by `m * N + n`

Indexing Example



- The output array is (N, M) and the transposed location is (m, n)
- Accounting for the formula `n * M + m`
- $\mathrm{M} = \mathrm{N} = 8$
- m: `21 / 8 = 2`
- n: `21 % 8 = 5`
- New Location: `5 * 8 + 2 = 42`

Performance

# Array Dimensions	GB/sec
(256, 256)	32
(512, 512)	38
(1024, 1024)	28
(2048, 2048)	22
(4096, 4096)	-

- Invalid configuration argument at the last size due to requesting 65536 blocks
- The max is 65535 — move to a 2D grid
- Theoretical peak is 152 GB/sec — why such poor performance?
- And why does performance drop with increasing grid size?

Move to 2D

```
  __global__
void simpleTranspose2D(int *array_in, int *array_out,
                       int M, int N)
{
  const int n = threadIdx.x + blockDim.x * blockIdx.x;
  const int m = threadIdx.y + blockDim.y * blockIdx.y;

  array_out[n * M + m] = array_in[m * N + n];
}
```

- Here we are assuming that M is a multiple of `blockDim.y` and N is a multiple of `blockDim.x`
- Superficially similar, we trade a modulo and division for a multiply and add

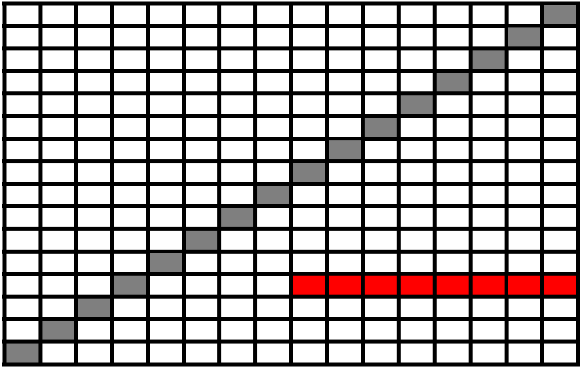
Performance

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

- Performance curve behaves as expected
- Increases with problem size until a plateau is reached
- Performance level has more than doubled
- Why such a difference?

Memory Access Pattern #1

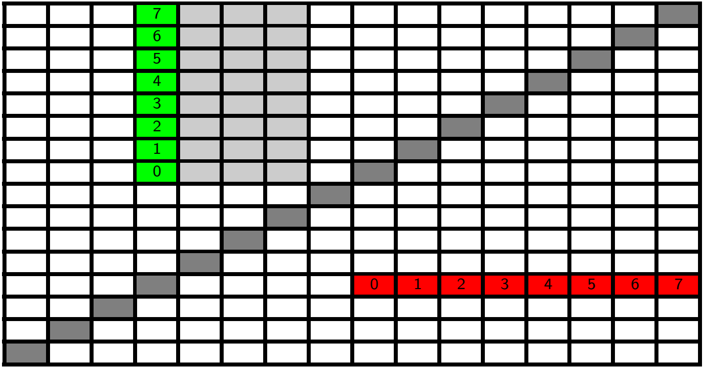
Warp Size of 8



Reads are coalesced

Memory Access Pattern #1

Warp Size of 8

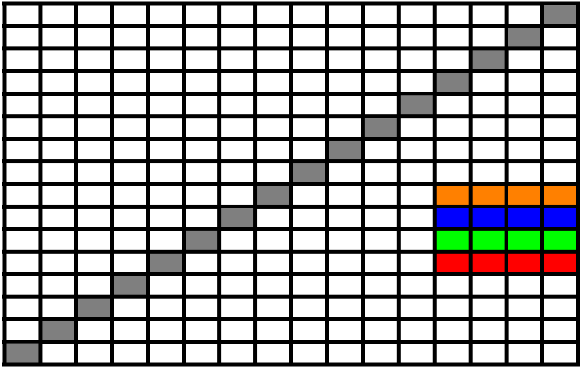


Writes are NOT!

Memory Access Pattern #1

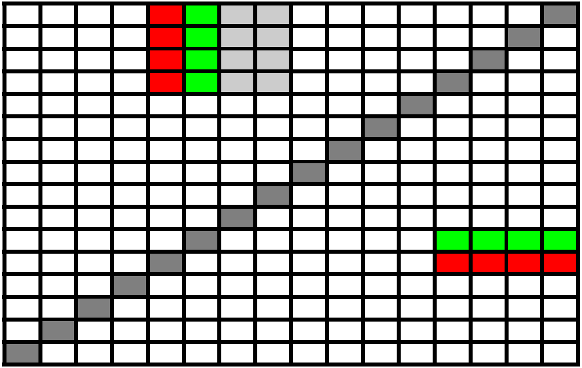
- The reads are fully coalesced
- However, the writes are completely uncoalesced
- For each item that is written a complete 128 byte transaction is made and 124 bytes are wasted
- Explains the low performance, but not why performance drops with increasing size

Memory Access Pattern #2



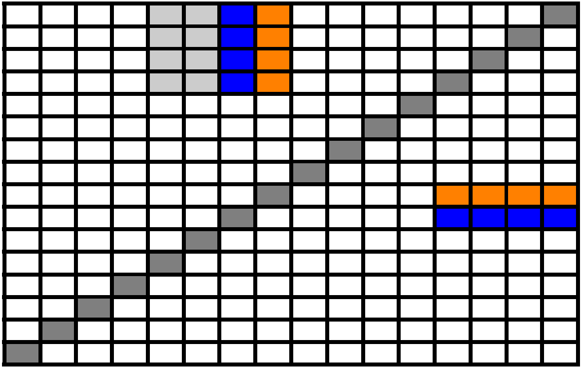
- Memory access pattern is 2D
- Reads might actually be less coalesced depending on dimensions of block
- Previous measurements were taken for 16×16, so each read was only half used but the performance still went up

Memory Access Pattern #2



- Notice that we have now written *two* values in each transaction

Memory Access Pattern #2



- Caching!
- The writes of the second warp occur to memory that has been cached because of the first warp's transaction

Cache Hierarchy

- One 768KB L2 cache that serves ALL the SMs
- In the first case then L2 cache is able to hold relevant data for later warps before being evicted
- As the size increases most data gets evicted before it is used again
- In the 2D case we enforce a memory access pattern that ensures we can use data in the L2 before it is evicted

Shared Memory

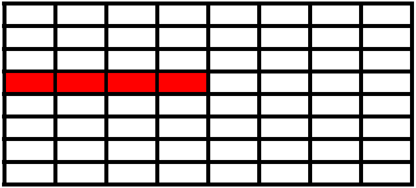
- Each block can declare memory that is shared amongst its threads
- Each thread can read and write to any location in this memory
- Allows for communication within a block
- Introduces need for synchronization - `__syncthreads()`
- Common pattern:
  - Everyone Read/Write
  - `__syncthreads()`
  - Everyone Write/Read
- Declare like `__shared__ int smemArray[256];`
- Smem is a limited resource, maximum of 48Kb available to each block
- Much faster than global memory

Using Smem for Transpose

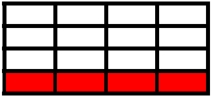
- Main idea is to use smem to allow us to make both reads and writes contiguous
- We can read columns from smem and write rows to global memory

Smem Transpose Memory Read Access Pattern

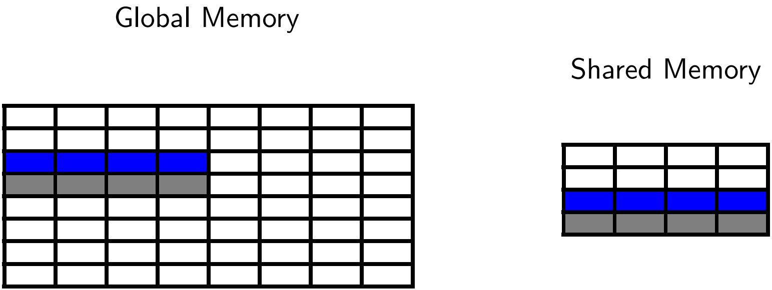
Global Memory



Shared Memory

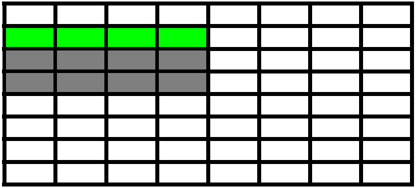


Read Memory Access Pattern

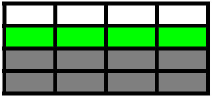


Read Memory Access Pattern

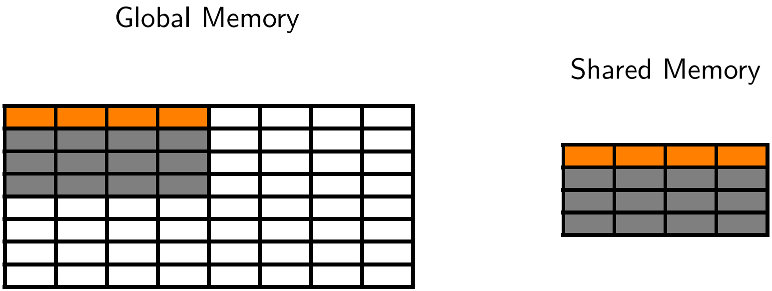
Global Memory



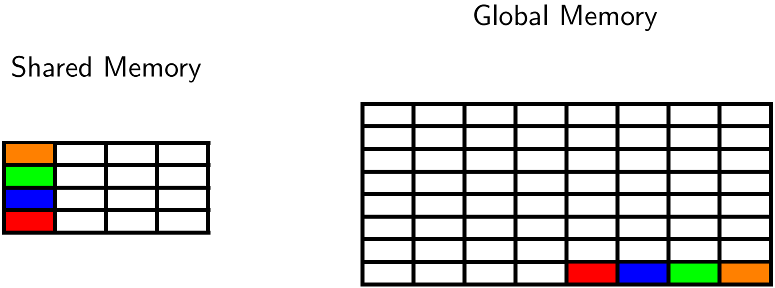
Shared Memory



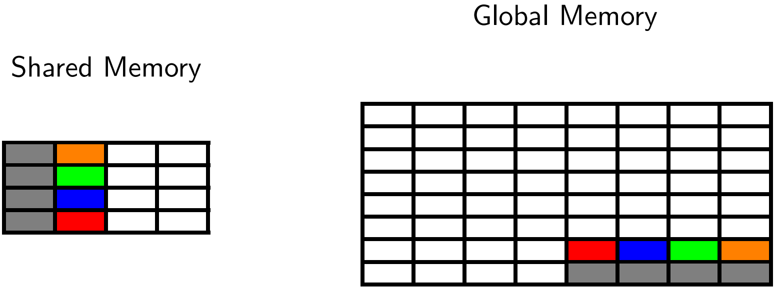
Read Memory Access Pattern



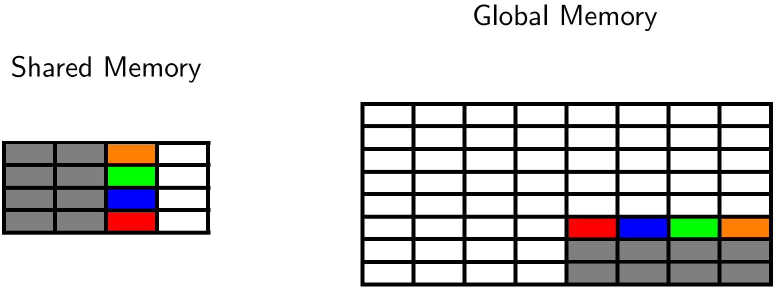
Write Memory Access Pattern



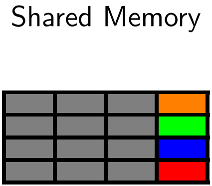
Write Memory Access Pattern

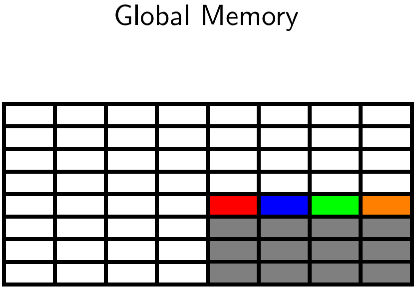


Write Memory Access Pattern



Write Memory Access Pattern





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 * smemRows + 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 * smemRows + i * numWarps + warpId;
    int row = i * numWarps + warpId;

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

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?

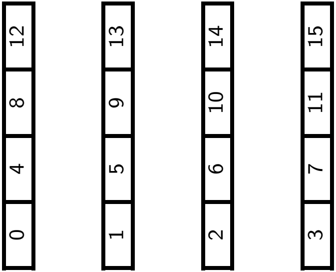
Shared Memory

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

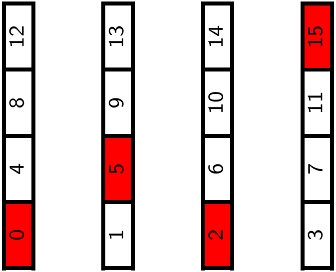
Banks

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



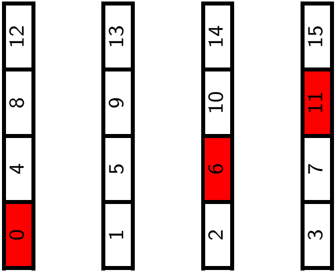
Banks

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



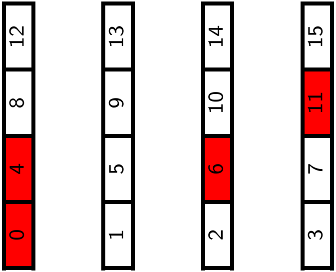
Banks

- 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

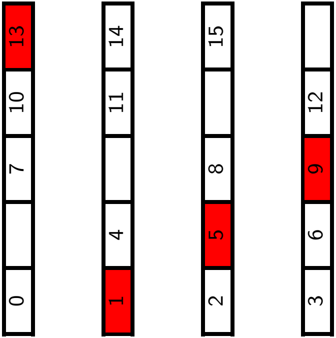
```
//Write out to global memory
for (int i = 0; i < smemRows / numWarps; ++i) {
    int gr = blockRow * warpSize + lane;
    int gc = blockCol * warpSize + i * numWarps + warpId;
    int row = i * numWarps + warpId;

    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 - next time!

Shared memory and HW3

- You will use smem to increase locality
- By bringing a 2D region into smem, you can compute many grid points without going to global memory
- Reducing global memory traffic should increase performance