CIS 431/531 Intro to Parallel Computing

CUDA

Quiz

1. Write a CUDA kernel to calculate Pi using the area of a circle method. The following OpenMP code is provided as a reference. (10 minutes).

You may also need to use `double atomicAdd(double* address, double val);`

```
double step, x, y, pi;
double sum = 0.0;
step = 1.0/(double) num_steps;
for (int i = 0; i < num_steps; i++) {
    x = i * step;
    y = sqrt(1 - x * x);
    sum = sum + y * step;
}
pi = 4 * sum;
return pi;
```

Quiz

2. If the memory bandwidth of a GPU is **100 GB/s (100 * 10^9 bytes/second)** and the time it takes to move data from the DRAM to the core is **200 ns (200 * 10^-9 seconds)**, how many in-flight memory requests of 4 bytes are needed to saturate the bandwidth? (5 minutes).

Previously

CUDA programming

- Thread hierarchy
- Memory hierarchy
- Synchronization

Performance tips

- Many threads should be created to increase latency hiding (both for instructions and data)
- Data should be accessed in a coalesced manner
- Shared memory should be used for
    - storing data that is reused frequently
    - accessing data in a non-coalesced manner in the shared memory so that access to DRAM can be done in a coalesced manner (e.g., matrix transpose)

Example

- In the matrix transpose example, using the shared memory properly can lead to > 3x speedup in performance

Reduction

$$s = \sum_{i=1}^{N} A_i$$

Commonly used algorithm in many applications

How do we parallelize it?

Reduction

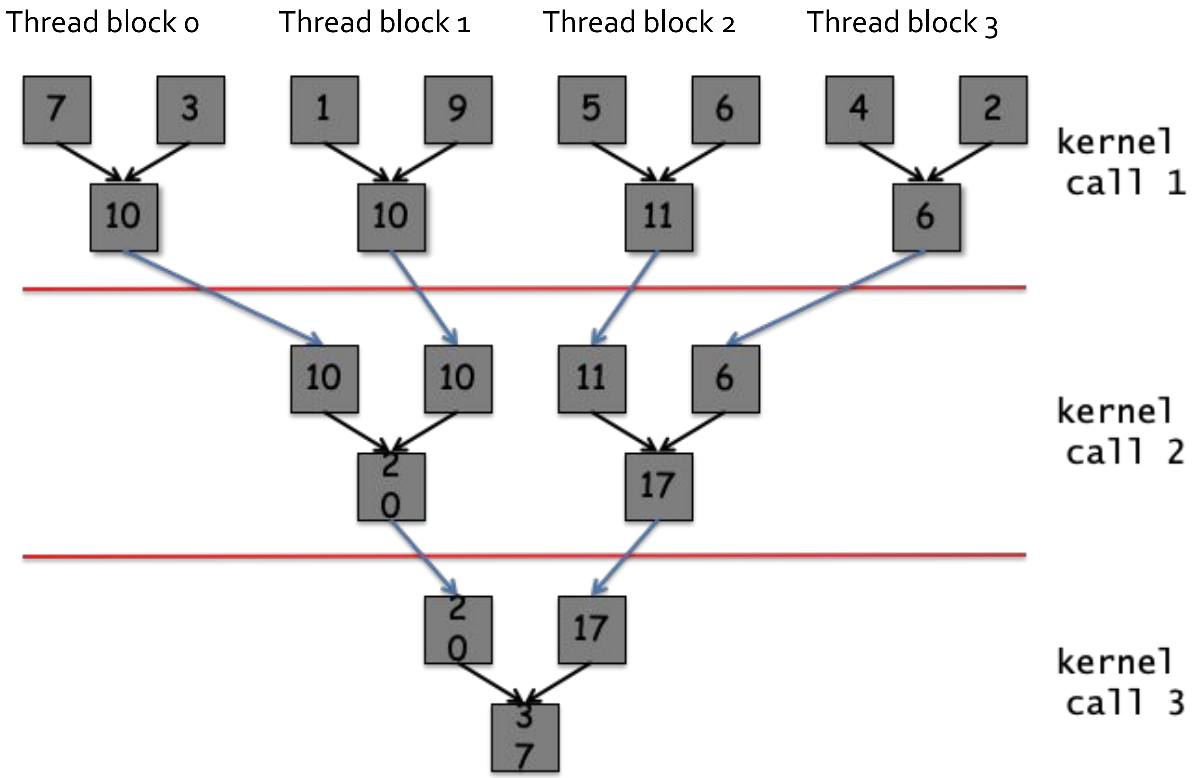
$$s = \sum_{i=1}^{N} A_i$$

Commonly used algorithm in many applications

How do we parallelize it?

- Tree-based approach
- Each thread reduces a portion
- Global synchronization is required to communicate partial results between thread blocks

Reduction


Thread block 0
Thread block 1
Thread block 2
Thread block 3
7
3
1
9
5
6
4
2
10
10
11
6
kernel call 1
10
10
11
6
20
17
kernel call 2
20
17
37
kernel call 3

Reduction

Performance Target?

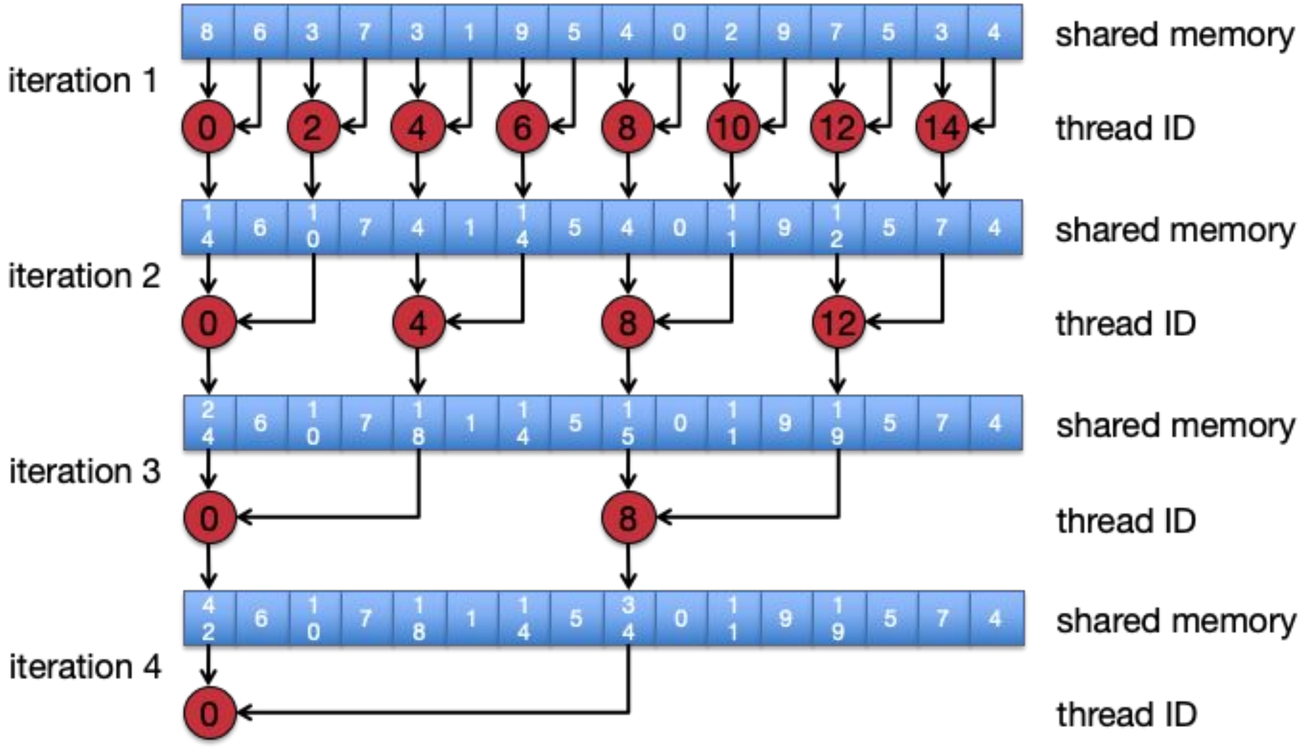
- Arithmetic Intensity = 1 flop / 4 bytes = 0.25
- Memory bandwidth-bound

Performance on Talaps?

- ~170 GB/s

Version 1 - Interleaved addressing

Reduction



This is for a single thread block - each thread block generates a single reduced value
Many thread blocks are doing the same thing over their sequence of elements

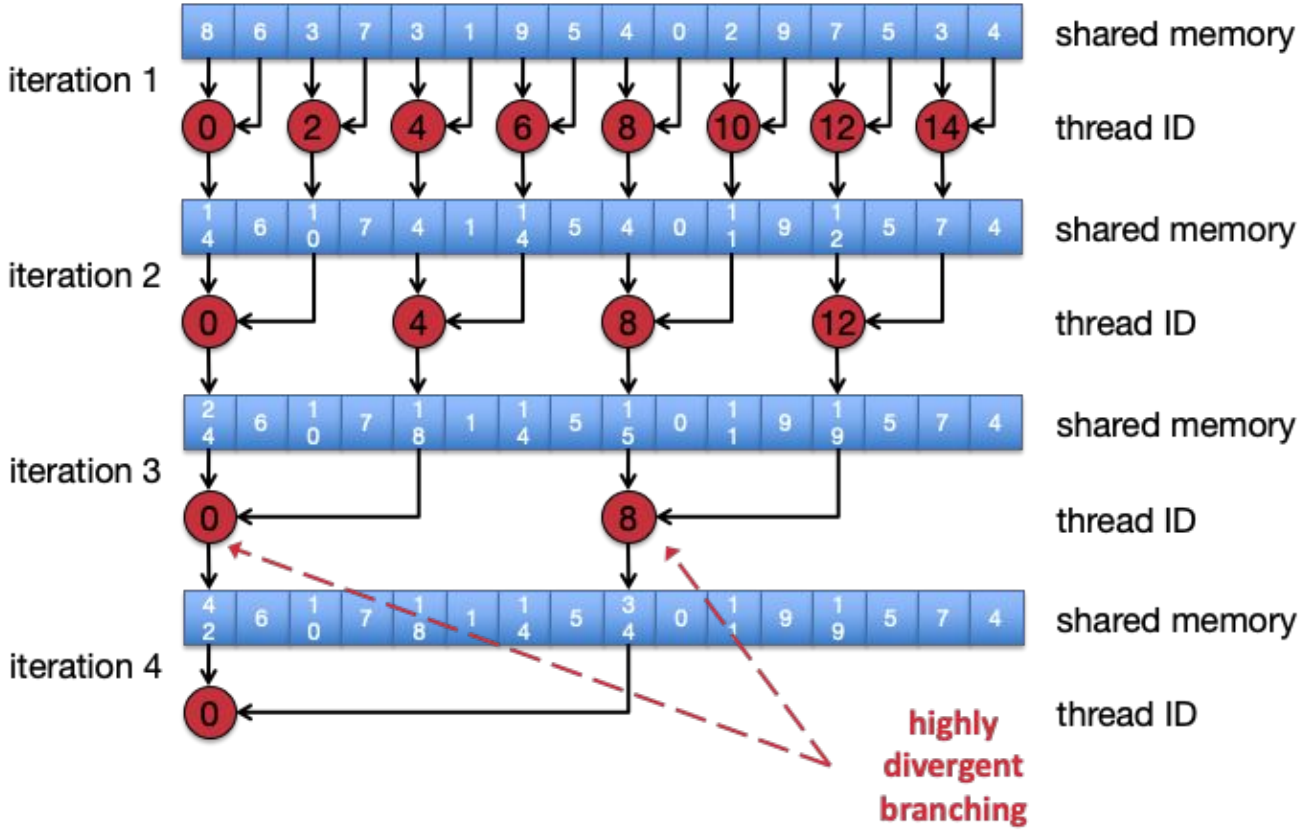
Reduction

```
__global__ void reduce (const int* In, int* Out)
{
  int tid = threadIdx.x;  // Local thread ID
  int i = blockIdx.x*blockDim.x + tid;  // Global index

  extern __shared__ int Local[];
  Local[tid] = In[i];   // Load into shared mem
  __syncthreads ();

  // Reduce data in shared memory within a thread block
  for (int s = 1; s < blockDim.x; s*=2) {
    if (tid % (2*s) == 0) //  Is multiple of s (2, 4, 8, …)
      Local[tid] += Local[tid + s];
    __syncthreads ();
  }

  if (tid == 0) Out[blockIdx.x] = Local[0];  // Each thread
block generates a single value
}
```

Reduction

Version	N	Bandwidth (GB/s)	Speedup
Interleaved addressing	16M	8.69	1.0

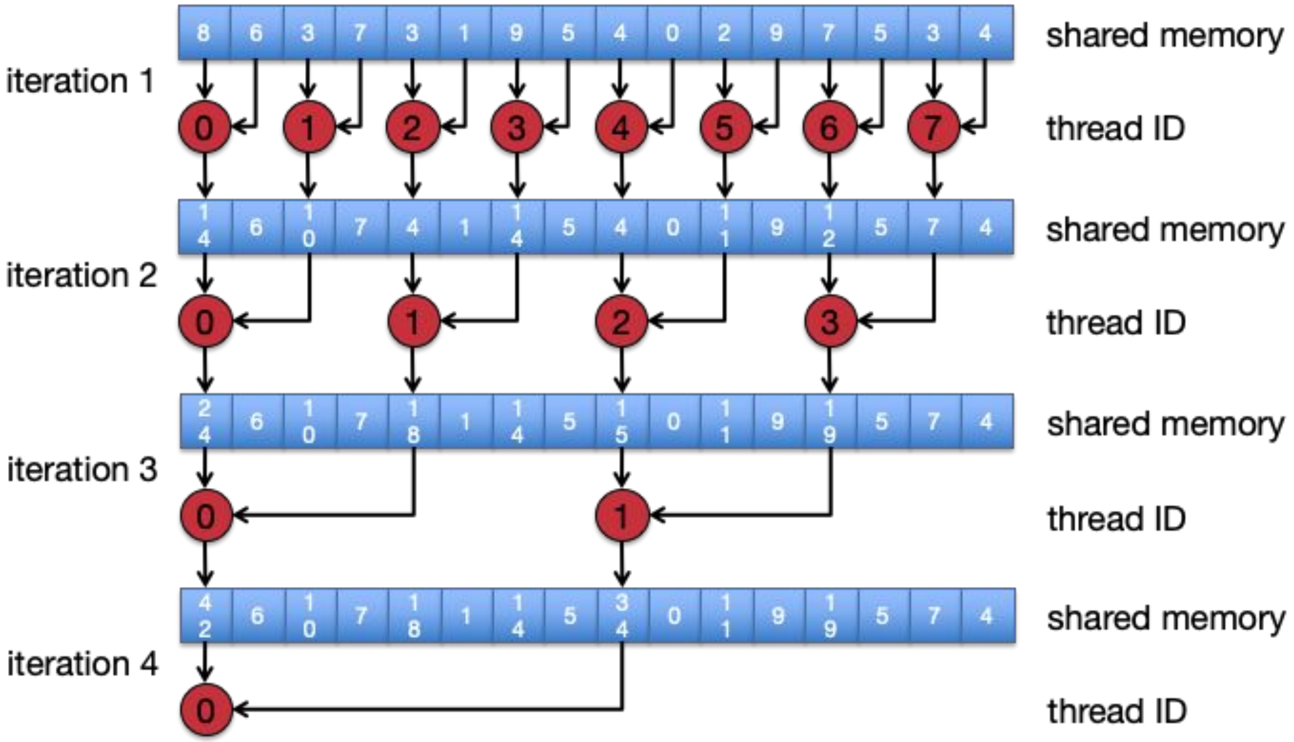
Reduction



shared memory
thread ID
iteration 1
iteration 2
iteration 3
iteration 4
highly divergent branching

Version 2 - Non-divergent branching

Reduction



Reduction

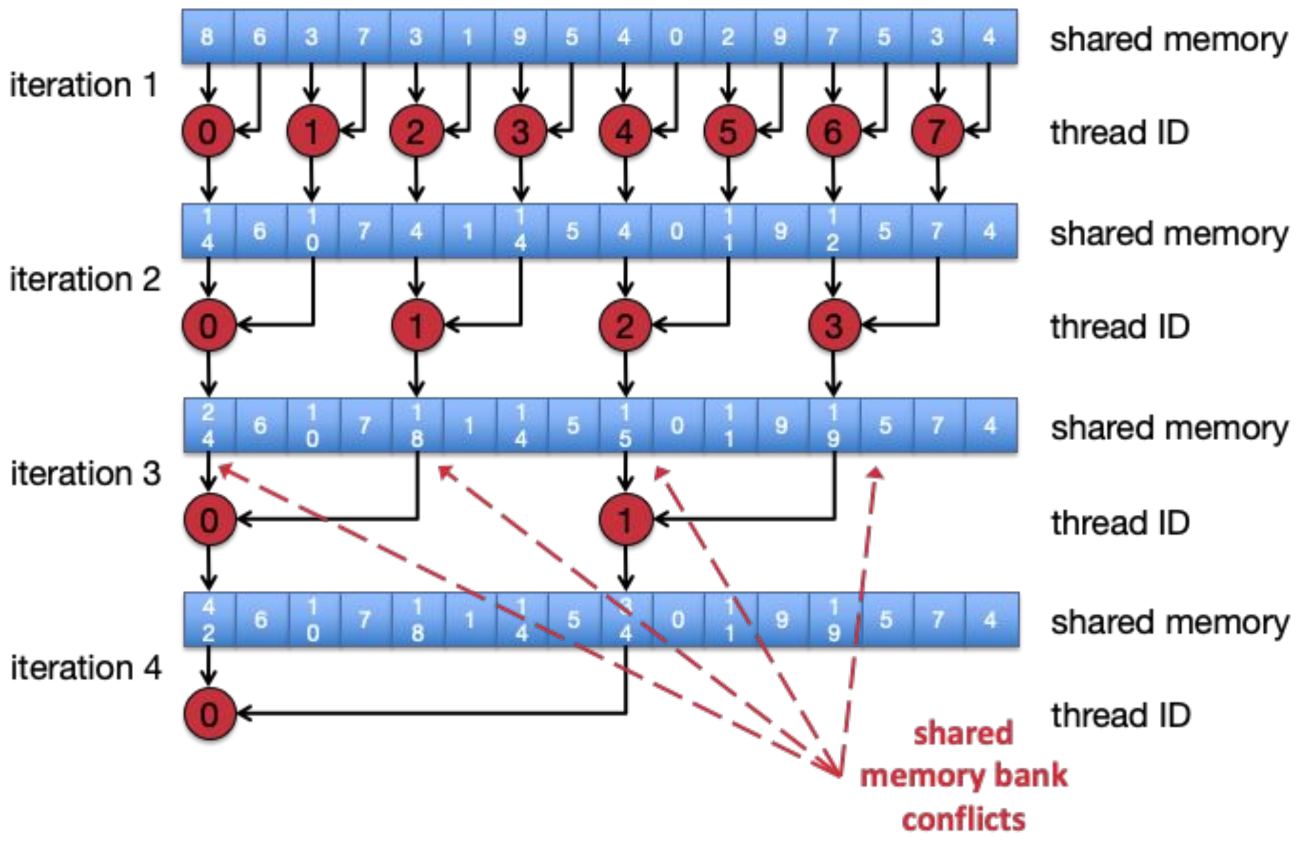
```
__global__ void reduce (const int* In, int* Out)
{
  int tid = threadIdx.x; //  Local thread ID
  int i = blockIdx.x*blockDim.x + tid; //  Global index

  extern __shared__ int Local[];
  Local[tid] = In[i];  //  Load into shared mem
  __syncthreads ();

for (int s = 1; s < blockDim.x; s*=2) { //  Element stride
    int index = 2*s*tid; //  Thread ID stride
    if (index < blockDim.x) {
      Local[tid] += Local[tid + s];
    }
    __syncthreads ();
}

  if (tid == 0) Out[blockIdx.x] = Local[0];
}
```

Simply remapping threads to work

Reduction

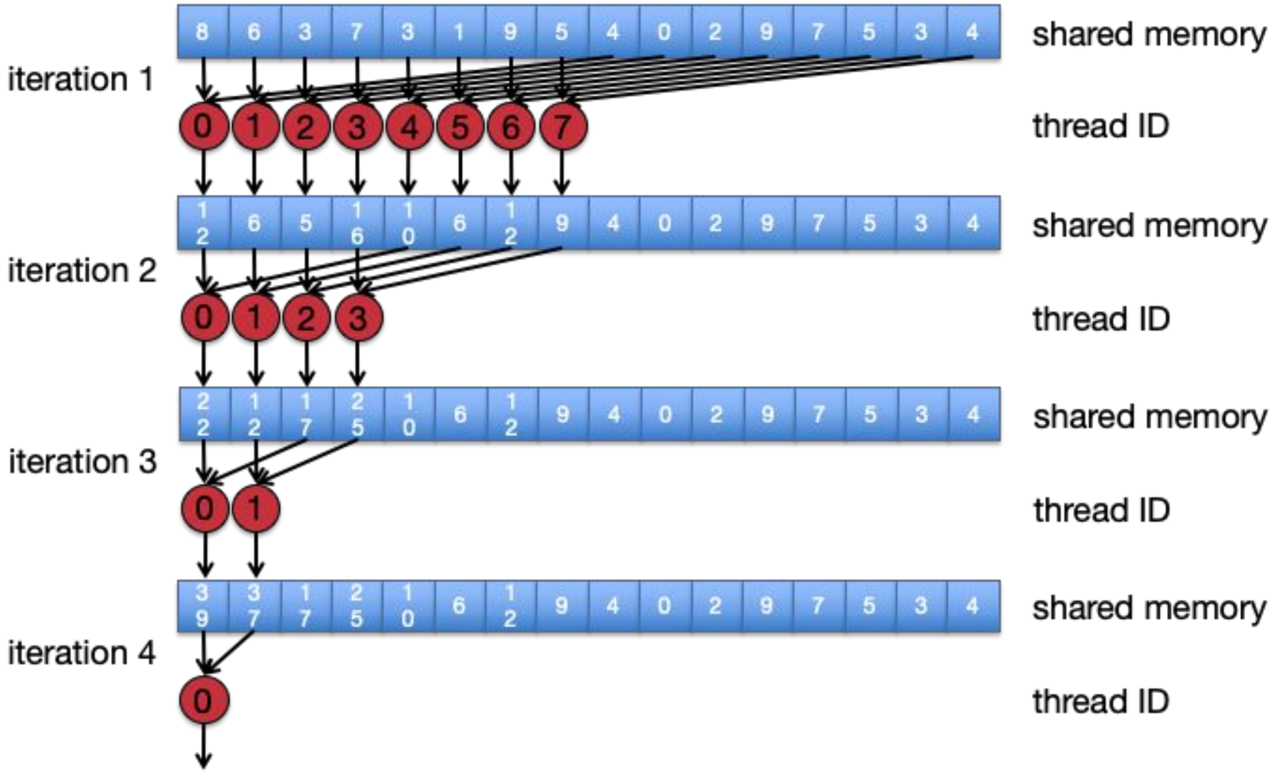
Version	N	Bandwidth (GB/s)	Speedup
Interleaved addressing (divergent branching)	16M	8.69	1.0
Non-divergent branching	16M	12.29	1.4

Reduction


8 6 3 7 3 1 9 5 4 0 2 9 7 5 3 4
shared memory
iteration 1
0 1 2 3 4 5 6 7
thread ID
14 6 10 7 4 1 14 5 4 0 11 9 12 5 7 4
shared memory
iteration 2
0 1 2 3
thread ID
24 6 10 7 18 1 14 5 15 0 11 9 19 5 7 4
shared memory
iteration 3
0 1
thread ID
42 6 10 7 18 1 14 5 34 0 11 9 19 5 7 4
shared memory
iteration 4
0
thread ID
shared memory bank conflicts

Version 3 - Sequential addressing

Reduction



Reduction

```
__global__ void reduce (const int* In, int* Out)
{
  int tid = threadIdx.x; //  Local thread ID
  int i = blockIdx.x*blockDim.x + tid; //  Global index

  extern __shared__ int Local[];
  Local[tid] = In[i];  //  Load into shared mem
  __syncthreads ();

  for (int s = blockDim.x / 2; s > 0; s>>=1) {
    if (tid < s) {
      Local[tid] += Local[tid + s];
    }
    __syncthreads ();
  }

  if (tid == 0) Out[blockIdx.x] = Local[0];
}
```

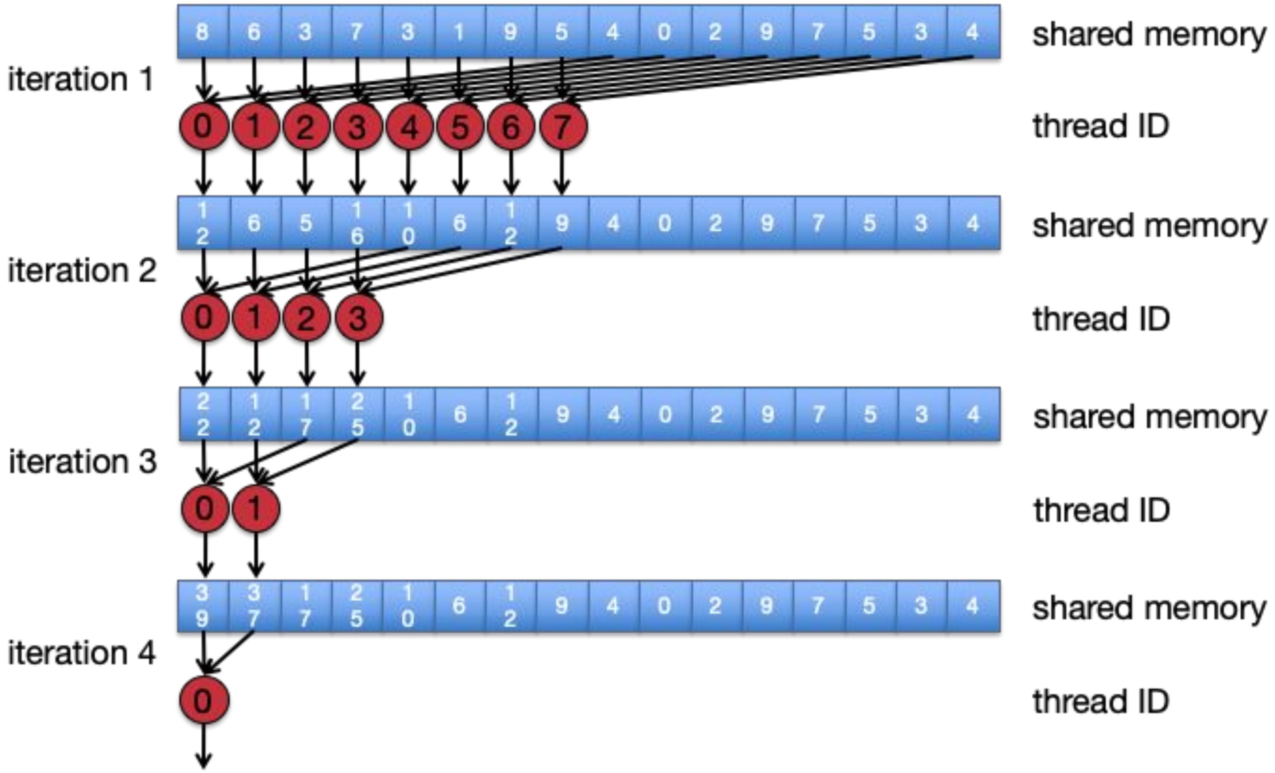
Simply change which elements are being added

Reduction

Version	N	Bandwidth (GB/s)	Speedup
Interleaved addressing (divergent branching)	16M	8.69	1.0
Non-divergent branching (bank conflicts)	16M	12.29	1.4
Sequential addressing	16M	16.81	1.9

Reduction

Only half the threads are "working"



Reduction

First add during load

- While loading the data, add the numbers
  - Reduces the number of threads by half
  - Double the number of memory requests
- Fewer threads are “wasted.”

Reduction

```
__global__ void reduce (const int* In, int* Out)
{
  int tid = threadIdx.x; //  Local thread ID
  int i = blockIdx.x*blockDim.x + tid; //  Global index

  extern __shared__ int Local[];
  Local[tid] = In[i] + In[i + blockDim.x];
  __syncthreads ();

  for (int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
      Local[tid] += Local[tid + s];
    }
    __syncthreads ();
  }

  if (tid == 0) Out[blockIdx.x] = Local[0];
}
```

Reduction

Version	N	Bandwidth (GB/s)	Speedup
Interleaved addressing (divergent branching)	16M	8.69	1.0
Non-divergent branching (bank conflicts)	16M	12.29	1.4
Sequential addressing (wasted threads)	16M	16.81	1.9
First add during load	16M	31.25	3.6

Questions?

So far, we have

- Reduce branching
- Bank conflicts
- Reduce # of non-working threads

What else can we do?

Reduction

With each iteration, the number of working threads (**t**) halves

When t < 32, only 1 warp is left

- We do not need __syncthreads()
    - Granularity of execution is a warp - threads in a warp are executing in a lock-step manner (i.e., already synchronized)
- We do not need conditionals

Unroll the loop when t <= 32

Reduction

```
for (unsigned int s=blockDim.x / 2; s > 32; s>>=1) {
     if (tid < s) {
          sdata[tid] += sdata[tid + s];
     }
     __syncthreads();
}
if (tid < 32) {
     sdata[tid] += sdata[tid + 32];
     sdata[tid] += sdata[tid + 16];
     sdata[tid] += sdata[tid + 8];
     sdata[tid] += sdata[tid + 4];
     sdata[tid] += sdata[tid + 2];
     sdata[tid] += sdata[tid + 1];
}
```

Reduction

Version	N	Bandwidth (GB/s)	Speedup
Interleaved addressing (divergent branching)	16M	8.69	1.0
Non-divergent branching (bank conflicts)	16M	12.29	1.4
Sequential addressing (wasted threads)	16M	16.81	1.9
First add during load	16M	31.25	3.6
Unroll the last loop	16M	50.65	5.8

Reduction

If we know the # of iterations at compile time, we could try unrolling the loop completely

- Block sizes are typically power of 2
- Limit your block size to 512 (it used to be max, but now it's 1024)
- Use C++ templates

Reduction

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)

if (blockSize >= 512) {
      if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
__syncthreads();
}
if (blockSize >= 256) {
      if (tid < 128) { sdata[tid] += sdata[tid + 128]; }
__syncthreads();
}
if (blockSize >= 128) {
      if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
}
if (tid < 32) {
      if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
      if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
      if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
      if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
      if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
      if (blockSize >= sdata[tid] += sdata[tid + 1];
}
```

Reduction

Version	N	Bandwidth (GB/s)	Speedup
Interleaved addressing (divergent branching)	16M	8.69	1.0
Non-divergent branching (bank conflicts)	16M	12.29	1.4
Sequential addressing (wasted threads)	16M	16.81	1.9
First add during load	16M	31.25	3.6
Unroll the last loop	16M	50.65	5.8
Complete unroll	16M	53.7	6.2

Reduction

Increase the number of adds per thread (we’ve seen this with transpose, where each thread was responsible for 4 elements)

- Reduces thread block scheduling overhead
- Increases the number of memory requests

Reduction

```
__global__ void reduce (const int* In, int* Out)
{
  int tid = threadIdx.x; //  Local thread ID
  int i = blockIdx.x*blockDim.x + tid; //  Global index
  Int gridSize = gridDim.x * blockDim.x; // total number of threads

  extern __shared__ int Local[];
  Local[tid] = 0;
  while (i < n) {
    Local[tid] += In[i] + In[i+blockSize];
    i += gridSize;
  }
  __syncthreads();

  for (int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
      Local[tid] += Local[tid + s];
    }
    __syncthreads ();
  }

  if (tid == 0) Out[blockIdx.x] = Local[0];
}
```

Reduction

Version	N	Bandwidth (GB/s)	Speedup
Interleaved addressing (divergent branching)	16M	8.69	1.0
Non-divergent branching (bank conflicts)	16M	12.29	1.4
Sequential addressing (wasted threads)	16M	16.81	1.9
First add during load	16M	31.25	3.6
Unroll the last loop	16M	50.65	5.8
Complete unroll	16M	53.7	6.2
More work per thread	16M	102.74	11.9

It is difficult to get the full bandwidth since you are writing & reading from intermediate arrays multiple times

Questions?

Stencil

Found in many scientific applications

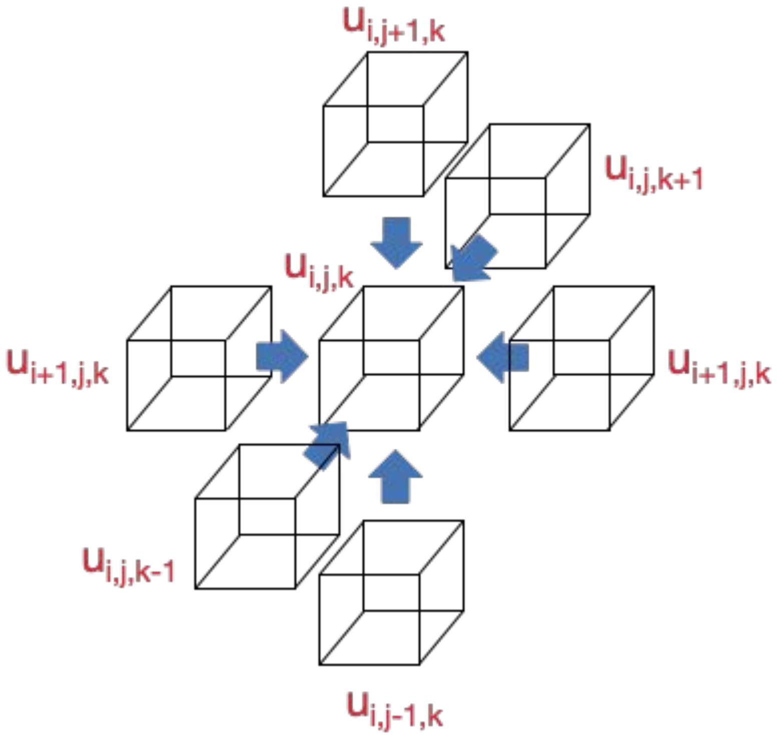
- PDE, systems of linear equations (e.g., Jacobi, Gauss-Siedel), convolution filter, etc.

7-point (3-D) stencil

- For each point (i, j, k) in a NxNxN grid

$$\begin{aligned}\hat{\mathbf{u}}_{\mathbf{i,j,k}} = {} & \mathbf{C_0} \cdot \mathbf{u_{i,j,k}} \\ & + \mathbf{C_1} \cdot (\mathbf{u_{i-1,j,k}} + \mathbf{u_{i+1,j,k}} \\ & \quad + \mathbf{u_{i,j-1,k}} + \mathbf{u_{i,j+1,k}} + \mathbf{u_{i,j,k-1}} + \mathbf{u_{i,j,k+1}})\end{aligned}$$

Stencil



Stencil

Performance

- Naively, each point requires
  - 7 reads and 1 write
  - 8 flops
  - I = 8 / (8 * 4) = 0.25
  - Memory bandwidth-bound

Optimally

- Load data only once per point (and reuse it as many times as needed)
- 1 read and 1 write (vs. 7 reads and 1 write)
- Up to ~4x speedup over a naive version (theoretically)

Stencil

Naive

- 1 thread per point
- 3-D thread blocks and grid (but the data is in 1-D)

Stencil

```
int tidx, tidy, tidz, gid;
tidx = threadIdx.x + blockDim.x * blockIdx.x;
tidy = threadIdx.y + blockDim.y * blockIdx.y;
tidz = threadIdx.z + blockDim.z * blockIdx.z;
gid = tidz * N * N + tidy * N + tidx; // data is laid out in 1-D

if(tidx < N && tidy < N && tidz < N) {
  tmp0 = c0 * In[gid];
  tmp1 = 0.0;
  if((tidx - 1) >= 0) tmp1 += In[tidz * N * N + tidy * N + (tidx - 1)];
  if((tidx + 1) < N)  tmp1 += In[tidz * N * N + tidy * N + (tidx + 1)];
  if((tidy - 1) >= 0) tmp1 += In[tidz * N * N + (tidy - 1) * N + tidx];
  if((tidy + 1) < N)  tmp1 += In[tidz * N * N + (tidy + 1) * N + tidx];
  if((tidz - 1) >= 0) tmp1 += In[(tidz - 1) * N * N + tidy * N + tidx];
  if((tidz + 1) < N)  tmp1 += In[(tidz + 1) * N * N + tidy * N + tidx];
  Out[gid] = tmp0 + c1 * tmp1;
}
```

$$\begin{aligned}\hat{u}_{i,j,k} = {} & C_0 \cdot u_{i,j,k} \\ & + C_1 \cdot (u_{i-1,j,k} + u_{i+1,j,k} \\ & \qquad + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1})\end{aligned}$$

Stencil

Version	N	Performance (GP/s)	Speedup
Naive	512	2.00	1.0

Stencil

Issues

- Data reuse occurs only through the (small) L1 and L2 cache
- Non-coalesced memory access
- Thread divergence

Which is the biggest problem?

Stencil

Issues

Data reuse occurs only through the (small) L1 and L2 cache

Non-coalesced memory access

Thread divergence

Which is the biggest problem?

- Increasing data reuse can increase performance by up to **4×**
- Non-coalesced access only occurs when accessing **(x + 1)** and **(x – 1)** neighbors but not when accessing the **y** or **z** neighbors
- Thread divergence only occurs at the volume surface (all threads must still pay the penalty of checking its tidx, tidy, and tidz values)

Stencil

Issues

Data reuse occurs only through the L2 (very small) cache

Non-coalesced memory access

Thread divergence

Which is the biggest problem?

- Increasing data reuse can increase performance by up to **4×**
- Non-coalesced access only occurs when accessing **(x + 1)** and **(x – 1)** neighbors but not when accessing the **y** or **z** neighbors
- Thread divergence only occurs at the volume surface (all threads must still pay the penalty of checking its tidx, tidy, and tidz values)

Stencil

Issues

Data reuse occurs only through the L2 (very small) cache

Non-coalesced memory access

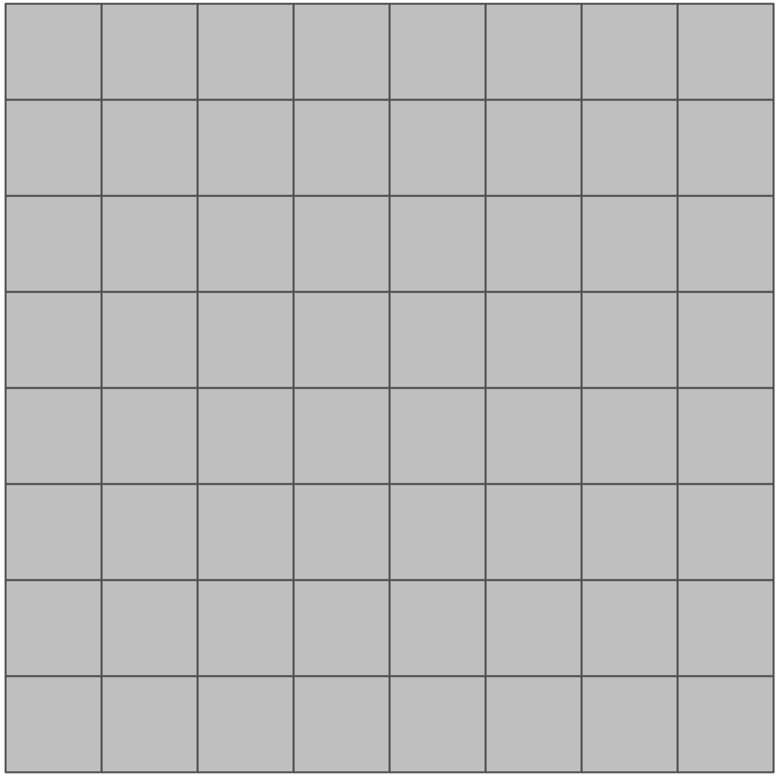
Thread divergence

Which is the biggest problem?

- Increasing data reuse can increase performance by up to **4×**
- **Non-coalesced access only occurs when accessing (x + 1) and (x – 1) neighbors but not when accessing the y or z neighbors**
- Thread divergence only occurs at the volume surface (all threads must still pay the penalty of checking its tidx, tidy, and tidz values)

Stencil

If cache line is (8 bytes x 8) = 64 bytes



Cache line

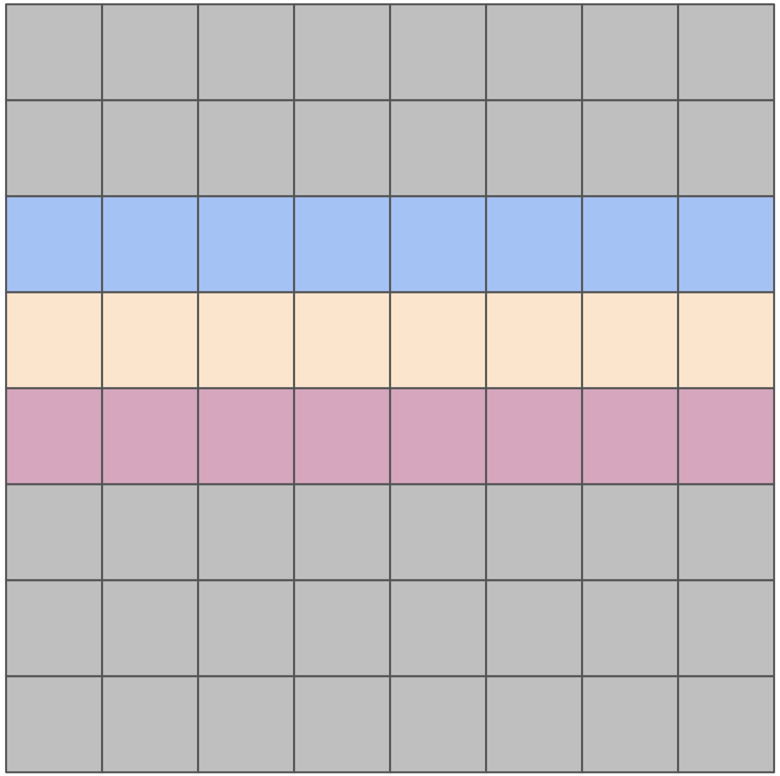
Cache line

...

Cache line

If cache line is (8 bytes x 8) = 64 bytes

Stencil



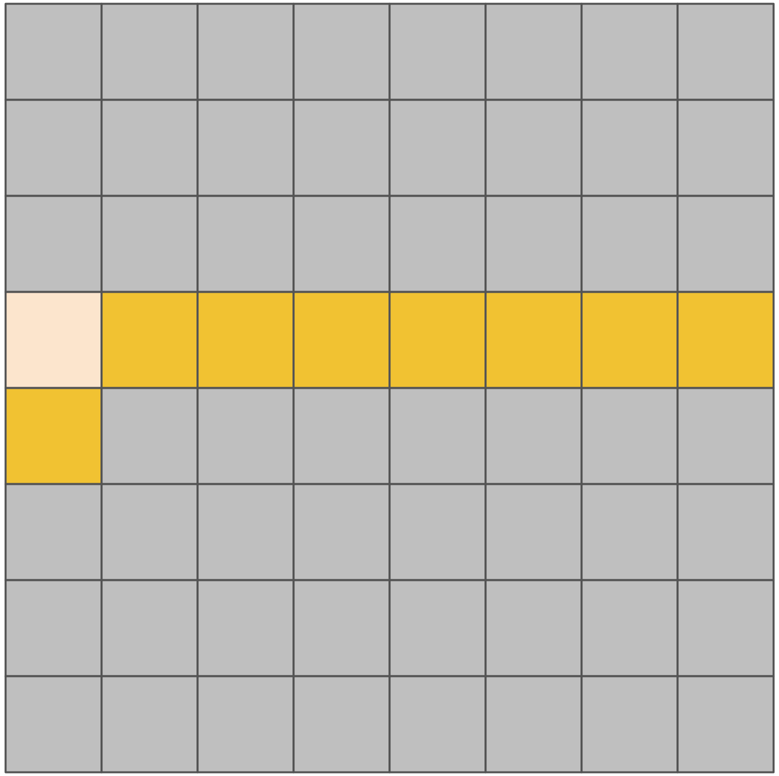
y - 1 neighbors

target

y + 1 neighbors

If cache line is (8 x 8) = 64 bytes

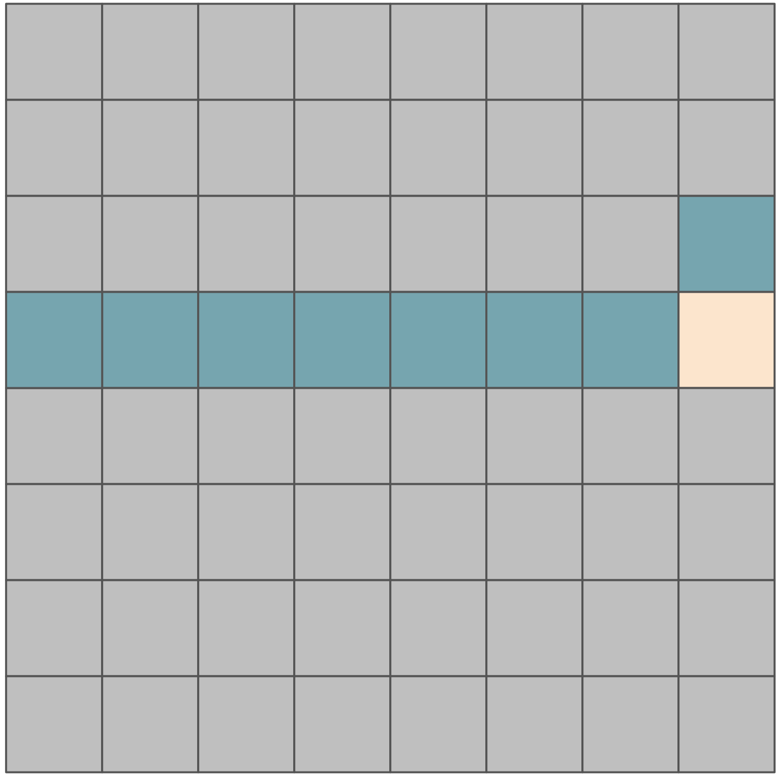
Stencil



x + 1 neighbor
(2 cache lines)

If cache line is (8 x 8) = 64 bytes

Stencil



x - 1 neighbor
(2 cache lines)

Stencil

Increasing data reuse can increase performance by up to **4×**

Solution?

Stencil

Increasing data reuse can increase performance by up to **4×**

Solution?

- Use **shared memory** to reuse the data
- Store a sub-volume of the data in shared memory, along with the elements in the boundary (a.k.a. *halo region*)
- Halo region is still being loaded **redundantly** (i.e., neighboring tiles are also loading the halo region)
    - Which is better - bigger tiles or smaller tiles?

Stencil

Increasing data reuse can increase performance by up to **4×**

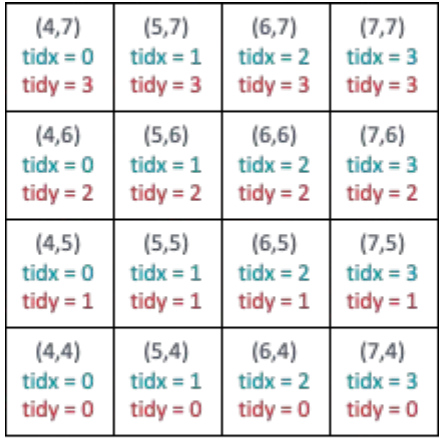
Solution?

- Use **shared memory** to reuse the data
- Store a sub-volume of the data in shared memory, along with the elements in the boundary (a.k.a. halo region)
- Halo region is still being loaded **redundantly** (i.e., neighboring tiles are also loading the halo region)
    - Which is better - bigger tiles or smaller tiles?
    - Bigger tiles are better - smaller surface-to-volume ratio (i.e., compared to the required data, redundant data volume is smaller)

Stencil

Using code for 2-D stencil for simplicity

```
int tidx, tidy, gidx, gidy;
__shared__ float smem[NT + 2][NT + 2];
tidx = threadIdx.x; tidy = threadIdx.y;
gidx = tidx + blockDim.x * blockIdx.x;
gidy = tidy + blockDim.y * blockIdx.y;
gid = gidy * N + gidx; // data laid out in 1-D
smem[tidy + 1][tidx + 1] = In[gid];
```



(4,7)
tidx = 0
tidy = 3
	(5,7)
	tidx = 1
	tidy = 3
		(6,7)
		tidx = 2
		tidy = 3
			(7,7)
			tidx = 3
			tidy = 3

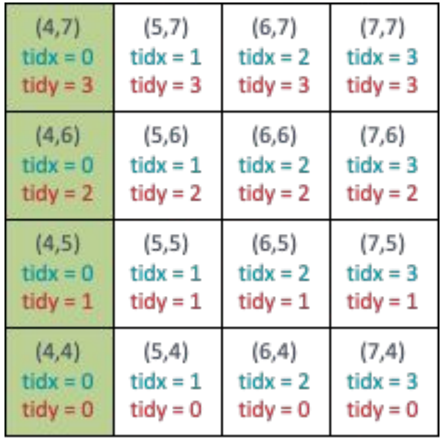
(4,6)
tidx = 0
tidy = 2
	(5,6)
	tidx = 1
	tidy = 2
		(6,6)
		tidx = 2
		tidy = 2
			(7,6)
			tidx = 3
			tidy = 2

(4,5)
tidx = 0
tidy = 1
	(5,5)
	tidx = 1
	tidy = 1
		(6,5)
		tidx = 2
		tidy = 1
			(7,5)
			tidx = 3
			tidy = 1

(4,4)
tidx = 0
tidy = 0
	(5,4)
	tidx = 1
	tidy = 0
		(6,4)
		tidx = 2
		tidy = 0
			(7,4)
			tidx = 3
			tidy = 0

Stencil

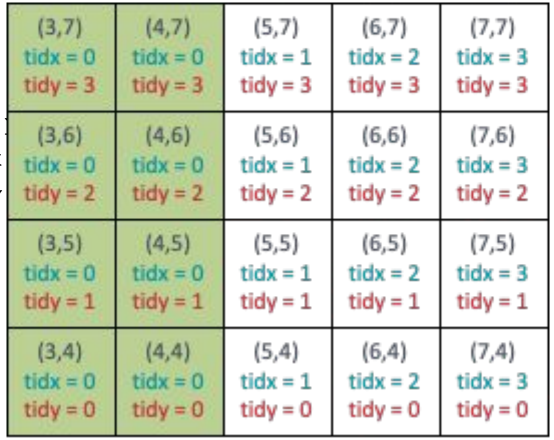
```
int tidx, tidy, gidx, gidy;
__shared__ float smem[NT + 2][NT + 2];
tidx = threadIdx.x; tidy = threadIdx.y;
gidx = tidx + blockDim.x * blockIdx.x;
gidy = tidy + blockDim.y * blockIdx.y;
gid = gidy * N + gidx;
smem[tidy + 1][tidx + 1] = In[gid];
if(tidx == 0)
```



(4,7) tidx = 0 tidy = 3	(5,7) tidx = 1 tidy = 3	(6,7) tidx = 2 tidy = 3	(7,7) tidx = 3 tidy = 3
(4,6) tidx = 0 tidy = 2	(5,6) tidx = 1 tidy = 2	(6,6) tidx = 2 tidy = 2	(7,6) tidx = 3 tidy = 2
(4,5) tidx = 0 tidy = 1	(5,5) tidx = 1 tidy = 1	(6,5) tidx = 2 tidy = 1	(7,5) tidx = 3 tidy = 1
(4,4) tidx = 0 tidy = 0	(5,4) tidx = 1 tidy = 0	(6,4) tidx = 2 tidy = 0	(7,4) tidx = 3 tidy = 0

Stencil

```
int tidx, tidy, gidx, gidy;
__shared__ float smem[NT + 2][NT + 2]
tidx = threadIdx.x; tidy = threadIdx.
gidx = tidx + blockDim.x * blockIdx.x
gidy = tidy + blockDim.y * blockIdx.y
gid = gidy * N + gidx;
smem[tidy + 1][tidx + 1] = In[gid];
if(tidx == 0)
  smem[tidy+1][0]=In[gidy*N+(gidx-1)]
```



(3,7)
tidx = 0
tidy = 3
	(4,7)
	tidx = 0
	tidy = 3
		(5,7)
		tidx = 1
		tidy = 3
			(6,7)
			tidx = 2
			tidy = 3
				(7,7)
				tidx = 3
				tidy = 3

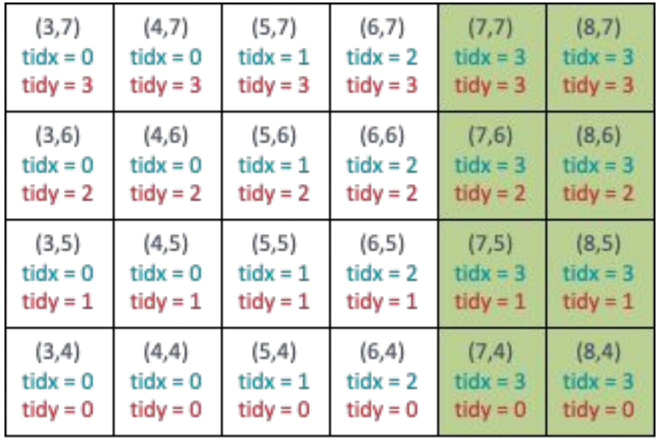
(3,6)
tidx = 0
tidy = 2
	(4,6)
	tidx = 0
	tidy = 2
		(5,6)
		tidx = 1
		tidy = 2
			(6,6)
			tidx = 2
			tidy = 2
				(7,6)
				tidx = 3
				tidy = 2

(3,5)
tidx = 0
tidy = 1
	(4,5)
	tidx = 0
	tidy = 1
		(5,5)
		tidx = 1
		tidy = 1
			(6,5)
			tidx = 2
			tidy = 1
				(7,5)
				tidx = 3
				tidy = 1

(3,4)
tidx = 0
tidy = 0
	(4,4)
	tidx = 0
	tidy = 0
		(5,4)
		tidx = 1
		tidy = 0
			(6,4)
			tidx = 2
			tidy = 0
				(7,4)
				tidx = 3
				tidy = 0

Stencil

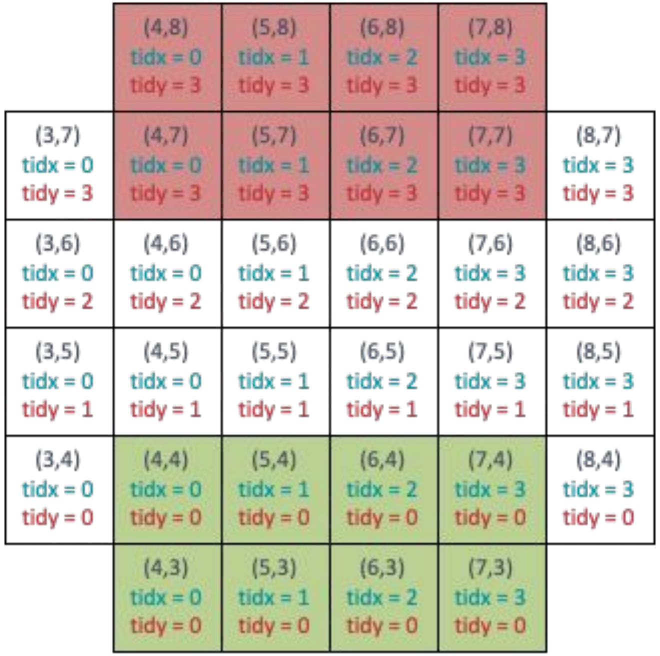
```
int tidx, tidy, gidx, gidy;
__shared__ float smem[NT + 2][NT + 2];
tidx = threadIdx.x; tidy = threadIdx.y;
gidx = tidx + blockDim.x * blockIdx.x;
gidy = tidy + blockDim.y * blockIdx.y;
gid = gidy * N + gidx;
smem[tidy + 1][tidx + 1] = In[gid];
if(tidx == 0)
  smem[tidy+1][0]=In[gidy*N+(gidx-1)];
if(tidx == (blockDim.x - 1))
  smem[tidy+1][NT+1]=In[gidy*N+(gidx+1)];
```



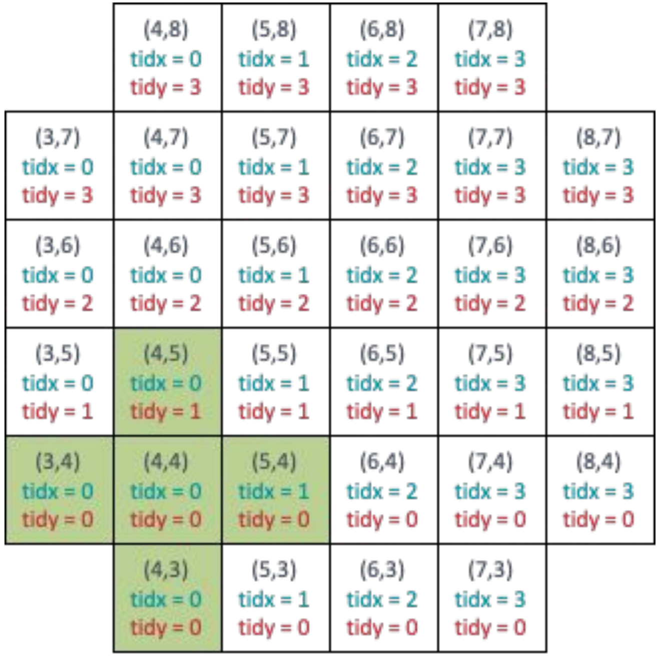
(3,7) tidx = 0 tidy = 3	(4,7) tidx = 0 tidy = 3	(5,7) tidx = 1 tidy = 3	(6,7) tidx = 2 tidy = 3	(7,7) tidx = 3 tidy = 3	(8,7) tidx = 3 tidy = 3
(3,6) tidx = 0 tidy = 2	(4,6) tidx = 0 tidy = 2	(5,6) tidx = 1 tidy = 2	(6,6) tidx = 2 tidy = 2	(7,6) tidx = 3 tidy = 2	(8,6) tidx = 3 tidy = 2
(3,5) tidx = 0 tidy = 1	(4,5) tidx = 0 tidy = 1	(5,5) tidx = 1 tidy = 1	(6,5) tidx = 2 tidy = 1	(7,5) tidx = 3 tidy = 1	(8,5) tidx = 3 tidy = 1
(3,4) tidx = 0 tidy = 0	(4,4) tidx = 0 tidy = 0	(5,4) tidx = 1 tidy = 0	(6,4) tidx = 2 tidy = 0	(7,4) tidx = 3 tidy = 0	(8,4) tidx = 3 tidy = 0

Stencil

```
int tidx, tidy, gidx, gidy;
__shared__ float smem[NT + 2][NT + 2];
tidx = threadIdx.x; tidy = threadIdx.y;
gidx = tidx + blockDim.x * blockIdx.x;
gidy = tidy + blockDim.y * blockIdx.y;
gid = gidy * N + gidx;
smem[tidy + 1][tidx + 1] = In[gid];
if(tidx == 0)
  smem[tidy+1][0]=In[gidy*N+(gidx-1)];
if(tidx == (blockDim.x - 1))
  smem[tidy+1][NT+1]=In[gidy*N+(gidx+1)];
if(tidy == 0)
  smem[0][tidx+1]=In[(gidy-1)*N+gidx];
if(tidy == (blockDim.y - 1))
  smem[NT+1][tidx+1]=In[(gidy+1)*N+gidx];
__syncthreads();
```

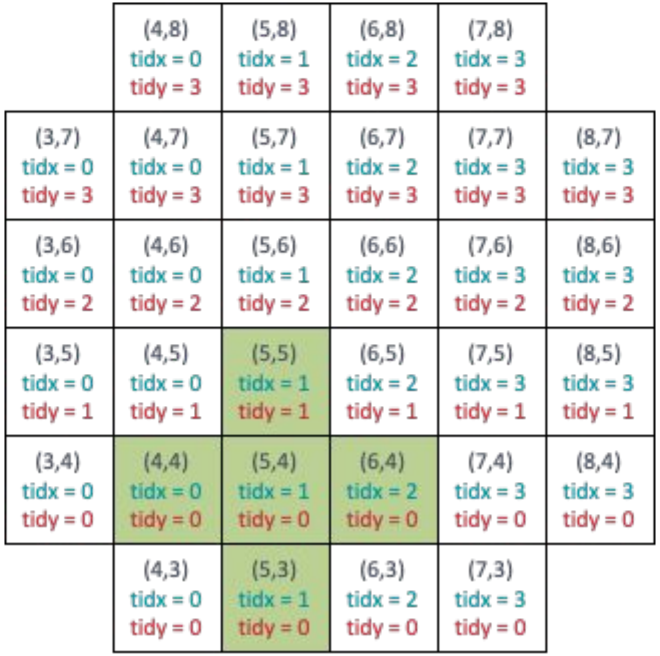


Stencil



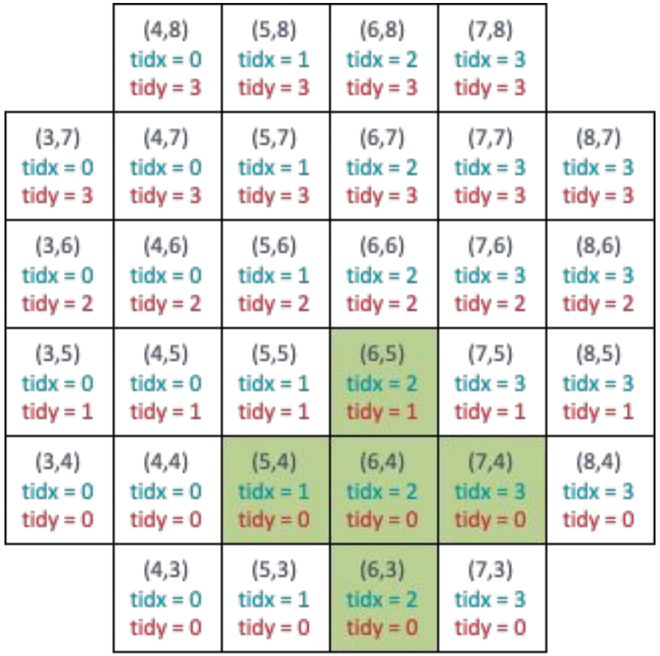
	(4,8) tidx = 0 tidy = 3	(5,8) tidx = 1 tidy = 3	(6,8) tidx = 2 tidy = 3	(7,8) tidx = 3 tidy = 3	
(3,7) tidx = 0 tidy = 3	(4,7) tidx = 0 tidy = 3	(5,7) tidx = 1 tidy = 3	(6,7) tidx = 2 tidy = 3	(7,7) tidx = 3 tidy = 3	(8,7) tidx = 3 tidy = 3
(3,6) tidx = 0 tidy = 2	(4,6) tidx = 0 tidy = 2	(5,6) tidx = 1 tidy = 2	(6,6) tidx = 2 tidy = 2	(7,6) tidx = 3 tidy = 2	(8,6) tidx = 3 tidy = 2
(3,5) tidx = 0 tidy = 1	(4,5) tidx = 0 tidy = 1	(5,5) tidx = 1 tidy = 1	(6,5) tidx = 2 tidy = 1	(7,5) tidx = 3 tidy = 1	(8,5) tidx = 3 tidy = 1
(3,4) tidx = 0 tidy = 0	(4,4) tidx = 0 tidy = 0	(5,4) tidx = 1 tidy = 0	(6,4) tidx = 2 tidy = 0	(7,4) tidx = 3 tidy = 0	(8,4) tidx = 3 tidy = 0
	(4,3) tidx = 0 tidy = 0	(5,3) tidx = 1 tidy = 0	(6,3) tidx = 2 tidy = 0	(7,3) tidx = 3 tidy = 0	

Stencil



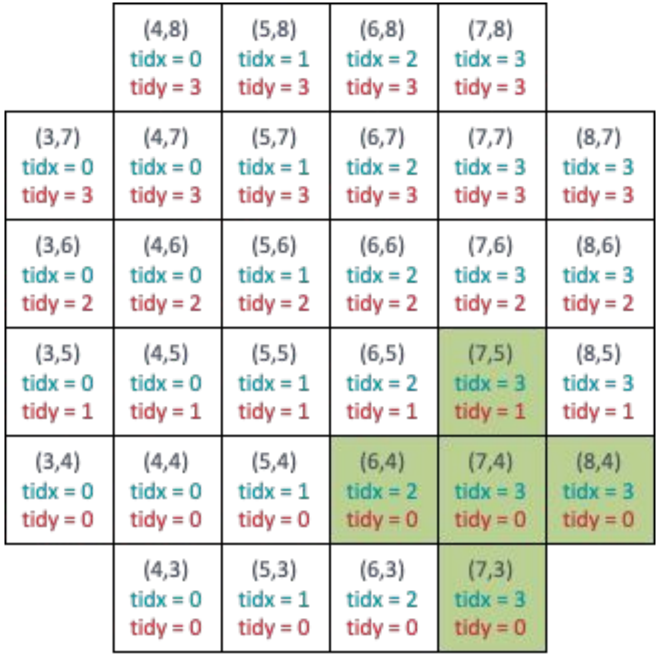
	(4,8) tidx = 0 tidy = 3	(5,8) tidx = 1 tidy = 3	(6,8) tidx = 2 tidy = 3	(7,8) tidx = 3 tidy = 3	
(3,7) tidx = 0 tidy = 3	(4,7) tidx = 0 tidy = 3	(5,7) tidx = 1 tidy = 3	(6,7) tidx = 2 tidy = 3	(7,7) tidx = 3 tidy = 3	(8,7) tidx = 3 tidy = 3
(3,6) tidx = 0 tidy = 2	(4,6) tidx = 0 tidy = 2	(5,6) tidx = 1 tidy = 2	(6,6) tidx = 2 tidy = 2	(7,6) tidx = 3 tidy = 2	(8,6) tidx = 3 tidy = 2
(3,5) tidx = 0 tidy = 1	(4,5) tidx = 0 tidy = 1	🟩 (5,5) tidx = 1 tidy = 1	(6,5) tidx = 2 tidy = 1	(7,5) tidx = 3 tidy = 1	(8,5) tidx = 3 tidy = 1
(3,4) tidx = 0 tidy = 0	🟩 (4,4) tidx = 0 tidy = 0	🟩 (5,4) tidx = 1 tidy = 0	🟩 (6,4) tidx = 2 tidy = 0	(7,4) tidx = 3 tidy = 0	(8,4) tidx = 3 tidy = 0
	(4,3) tidx = 0 tidy = 0	🟩 (5,3) tidx = 1 tidy = 0	(6,3) tidx = 2 tidy = 0	(7,3) tidx = 3 tidy = 0	

Stencil



	(4,8) tidx = 0 tidy = 3	(5,8) tidx = 1 tidy = 3	(6,8) tidx = 2 tidy = 3	(7,8) tidx = 3 tidy = 3	
(3,7) tidx = 0 tidy = 3	(4,7) tidx = 0 tidy = 3	(5,7) tidx = 1 tidy = 3	(6,7) tidx = 2 tidy = 3	(7,7) tidx = 3 tidy = 3	(8,7) tidx = 3 tidy = 3
(3,6) tidx = 0 tidy = 2	(4,6) tidx = 0 tidy = 2	(5,6) tidx = 1 tidy = 2	(6,6) tidx = 2 tidy = 2	(7,6) tidx = 3 tidy = 2	(8,6) tidx = 3 tidy = 2
(3,5) tidx = 0 tidy = 1	(4,5) tidx = 0 tidy = 1	(5,5) tidx = 1 tidy = 1	(6,5) tidx = 2 tidy = 1	(7,5) tidx = 3 tidy = 1	(8,5) tidx = 3 tidy = 1
(3,4) tidx = 0 tidy = 0	(4,4) tidx = 0 tidy = 0	(5,4) tidx = 1 tidy = 0	(6,4) tidx = 2 tidy = 0	(7,4) tidx = 3 tidy = 0	(8,4) tidx = 3 tidy = 0
	(4,3) tidx = 0 tidy = 0	(5,3) tidx = 1 tidy = 0	(6,3) tidx = 2 tidy = 0	(7,3) tidx = 3 tidy = 0	

Stencil



	(4,8) tidx = 0 tidy = 3	(5,8) tidx = 1 tidy = 3	(6,8) tidx = 2 tidy = 3	(7,8) tidx = 3 tidy = 3	
(3,7) tidx = 0 tidy = 3	(4,7) tidx = 0 tidy = 3	(5,7) tidx = 1 tidy = 3	(6,7) tidx = 2 tidy = 3	(7,7) tidx = 3 tidy = 3	(8,7) tidx = 3 tidy = 3
(3,6) tidx = 0 tidy = 2	(4,6) tidx = 0 tidy = 2	(5,6) tidx = 1 tidy = 2	(6,6) tidx = 2 tidy = 2	(7,6) tidx = 3 tidy = 2	(8,6) tidx = 3 tidy = 2
(3,5) tidx = 0 tidy = 1	(4,5) tidx = 0 tidy = 1	(5,5) tidx = 1 tidy = 1	(6,5) tidx = 2 tidy = 1	(7,5) tidx = 3 tidy = 1	(8,5) tidx = 3 tidy = 1
(3,4) tidx = 0 tidy = 0	(4,4) tidx = 0 tidy = 0	(5,4) tidx = 1 tidy = 0	(6,4) tidx = 2 tidy = 0	(7,4) tidx = 3 tidy = 0	(8,4) tidx = 3 tidy = 0
	(4,3) tidx = 0 tidy = 0	(5,3) tidx = 1 tidy = 0	(6,3) tidx = 2 tidy = 0	(7,3) tidx = 3 tidy = 0	

Stencil

Version	N	Performance (GP/s)	Speedup
Naive	512	2.00	1.0
Shared memory	512	2.79	1.4

Stencil

Issues?

Stencil

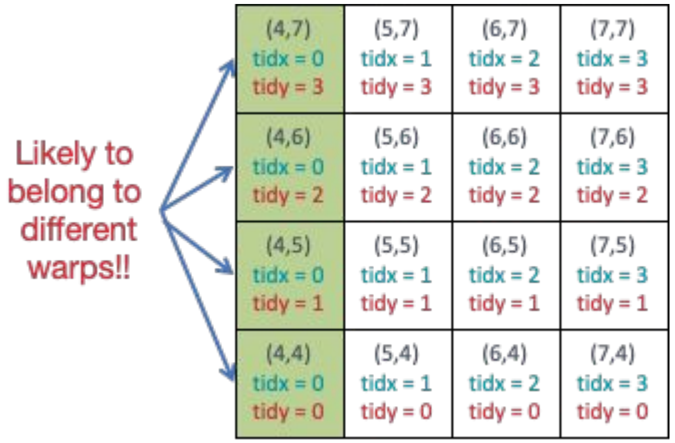
Issues

- Halo region is (still) redundant
- Thread divergence

Stencil

Issues

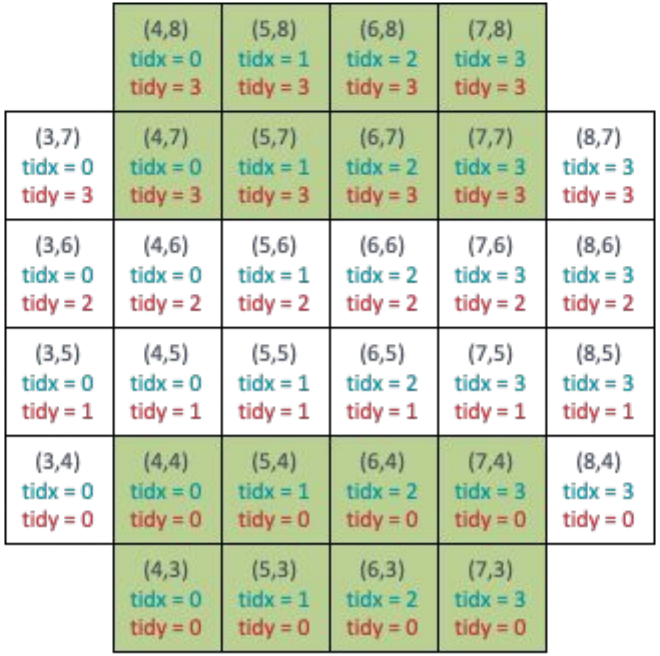
- Halo region is (still) redundant
- Thread divergence



(4,7) tidx = 0 tidy = 3	(5,7) tidx = 1 tidy = 3	(6,7) tidx = 2 tidy = 3	(7,7) tidx = 3 tidy = 3
(4,6) tidx = 0 tidy = 2	(5,6) tidx = 1 tidy = 2	(6,6) tidx = 2 tidy = 2	(7,6) tidx = 3 tidy = 2
(4,5) tidx = 0 tidy = 1	(5,5) tidx = 1 tidy = 1	(6,5) tidx = 2 tidy = 1	(7,5) tidx = 3 tidy = 1
(4,4) tidx = 0 tidy = 0	(5,4) tidx = 1 tidy = 0	(6,4) tidx = 2 tidy = 0	(7,4) tidx = 3 tidy = 0

Stencil

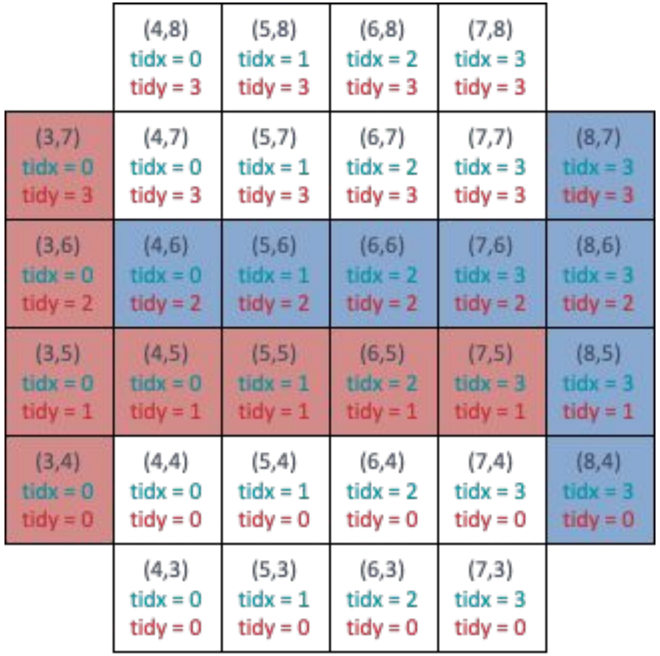
Have threads in the same warp load halo to shared memory



	(4,8) tidx = 0 tidy = 3	(5,8) tidx = 1 tidy = 3	(6,8) tidx = 2 tidy = 3	(7,8) tidx = 3 tidy = 3	
(3,7) tidx = 0 tidy = 3	(4,7) tidx = 0 tidy = 3	(5,7) tidx = 1 tidy = 3	(6,7) tidx = 2 tidy = 3	(7,7) tidx = 3 tidy = 3	(8,7) tidx = 3 tidy = 3
(3,6) tidx = 0 tidy = 2	(4,6) tidx = 0 tidy = 2	(5,6) tidx = 1 tidy = 2	(6,6) tidx = 2 tidy = 2	(7,6) tidx = 3 tidy = 2	(8,6) tidx = 3 tidy = 2
(3,5) tidx = 0 tidy = 1	(4,5) tidx = 0 tidy = 1	(5,5) tidx = 1 tidy = 1	(6,5) tidx = 2 tidy = 1	(7,5) tidx = 3 tidy = 1	(8,5) tidx = 3 tidy = 1
(3,4) tidx = 0 tidy = 0	(4,4) tidx = 0 tidy = 0	(5,4) tidx = 1 tidy = 0	(6,4) tidx = 2 tidy = 0	(7,4) tidx = 3 tidy = 0	(8,4) tidx = 3 tidy = 0
	(4,3) tidx = 0 tidy = 0	(5,3) tidx = 1 tidy = 0	(6,3) tidx = 2 tidy = 0	(7,3) tidx = 3 tidy = 0	

Stencil

Have threads in the same warp load halo to shared memory



	(4,8) tidx = 0 tidy = 3	(5,8) tidx = 1 tidy = 3	(6,8) tidx = 2 tidy = 3	(7,8) tidx = 3 tidy = 3	
🟥 (3,7) tidx = 0 tidy = 3	(4,7) tidx = 0 tidy = 3	(5,7) tidx = 1 tidy = 3	(6,7) tidx = 2 tidy = 3	(7,7) tidx = 3 tidy = 3	🟦 (8,7) tidx = 3 tidy = 3
🟥 (3,6) tidx = 0 tidy = 2	🟦 (4,6) tidx = 0 tidy = 2	🟦 (5,6) tidx = 1 tidy = 2	🟦 (6,6) tidx = 2 tidy = 2	🟦 (7,6) tidx = 3 tidy = 2	🟦 (8,6) tidx = 3 tidy = 2
🟥 (3,5) tidx = 0 tidy = 1	🟥 (4,5) tidx = 0 tidy = 1	🟥 (5,5) tidx = 1 tidy = 1	🟥 (6,5) tidx = 2 tidy = 1	🟥 (7,5) tidx = 3 tidy = 1	🟦 (8,5) tidx = 3 tidy = 1
🟥 (3,4) tidx = 0 tidy = 0	(4,4) tidx = 0 tidy = 0	(5,4) tidx = 1 tidy = 0	(6,4) tidx = 2 tidy = 0	(7,4) tidx = 3 tidy = 0	🟦 (8,4) tidx = 3 tidy = 0
	(4,3) tidx = 0 tidy = 0	(5,3) tidx = 1 tidy = 0	(6,3) tidx = 2 tidy = 0	(7,3) tidx = 3 tidy = 0	

Stencil

Version	N	Performance (TP/s)	Speedup
Naive	512	2.00	1.0
Shared memory	512	2.79	1.4
Non-divergent threads	512	2.85	1.43

Stencil

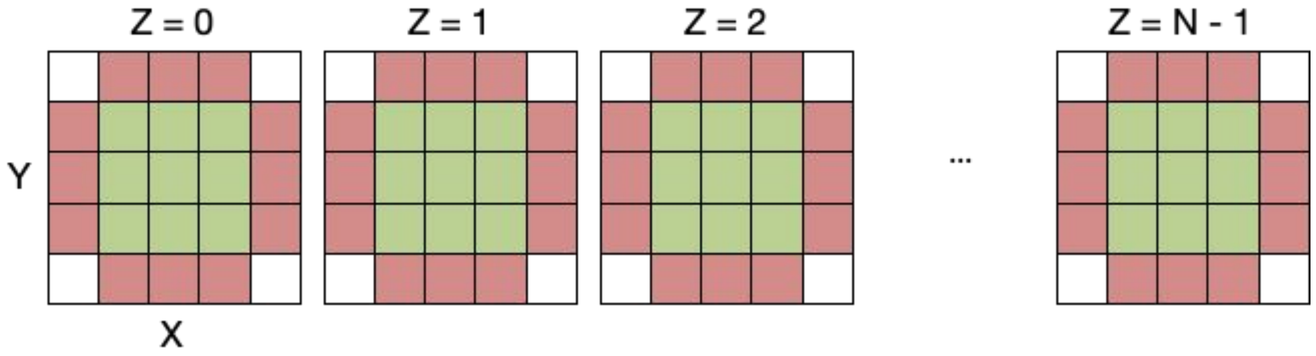
Small speedup - why?

- This only eliminates the *serialization of the memory requests*
- The data is still loaded from different cache lines (uncoalesced access)
- This is a bandwidth-bound kernel where loading data from the memory is the bottleneck - **how do you maximize data reduction further?**

Stencil

Blocking (2.5D blocking)

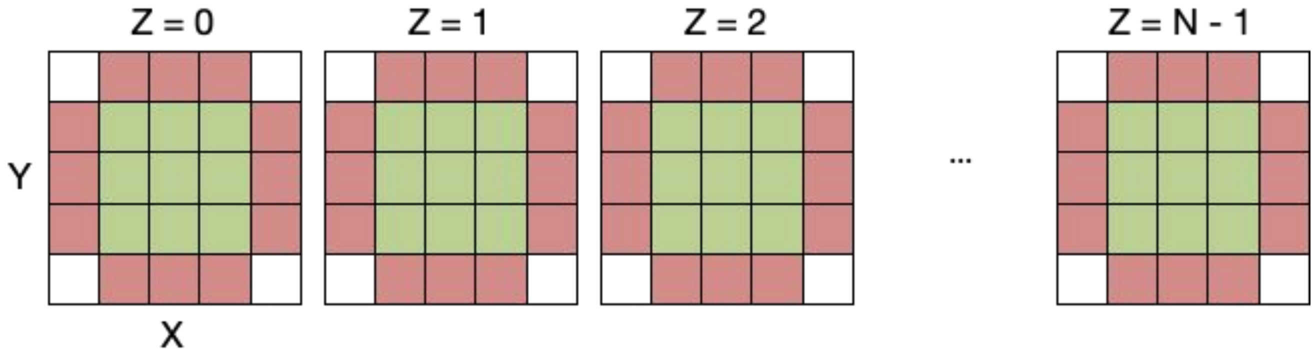
Block data in the X-Y plane and “stream” along the Z dimension



Stencil

Blocking (2.5D blocking)

Block data in the X-Y plane and "stream" along the Z dimension



We now have T' x T' thread blocks instead of T'xT'xT'

Each thread now computes N points along the Z plane (instead of just 1)

Stencil

Benefits?

- We only have to "pay" for halo along the X and Y dimensions
- Allows for larger X-Y block which reduces the surface to volume ratio
- Fewer thread blocks (lower scheduling cost)
- Better resource utilization (possibly)

Stencil

Version	N	Performance (TP/s)	Speedup
Naive	512	2.00	1.0
Shared memory	512	2.79	1.4
Non-divergent threads	512	2.85	1.43
2.5D	512	4.06	2.03

Stencil

Can we further optimize this?

- Store 1 slice (as opposed to keeping 1 old and 1 new slice) and keep temporary running sums (in register)
    - Lower shared memory usage -> possibly more threads resident in each SMX and better latency hiding
    - No shifting of data required (swapping new and old slices)

Stencil

Version	N	Performance (TP/s)	Speedup
Naive	512	2.00	1.0
Shared memory	512	2.79	1.4
Non-divergent threads	512	2.85	1.43
2.5D	512	4.06	2.03
2.5D with single slice	512	4.37	2.19

Questions?