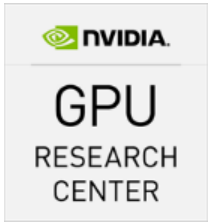
Parallel Computing with GPUs: Shared

Dr Paul Ric

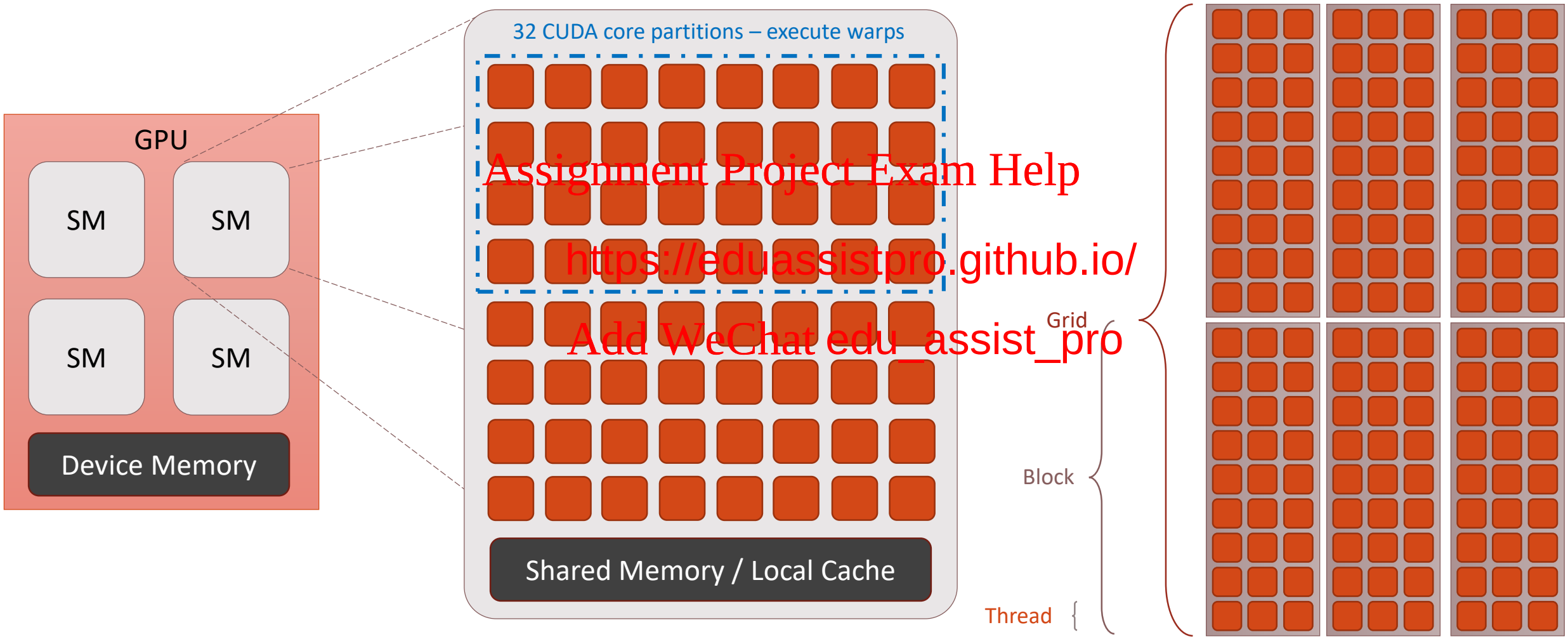
http://paulrichmond.shef.ac

COM4521/



Average Mark: 71%

Grids, Blocks, Warps & Threads



Grids, Blocks, Warps & Threads

- Blocks map to SMs
  - SMs may have more than one block
  - Blocks are split into warps by hardware (always pick block size multiple of 32)
  - Blocks do not migrate between SMs
  - No guarantee of order
  - No communication or blocks
- Threads map to CUDA cores
  - Executed in partitions of 32, called warps
  - Lots of warps means lots of opportunity to hide memory movement

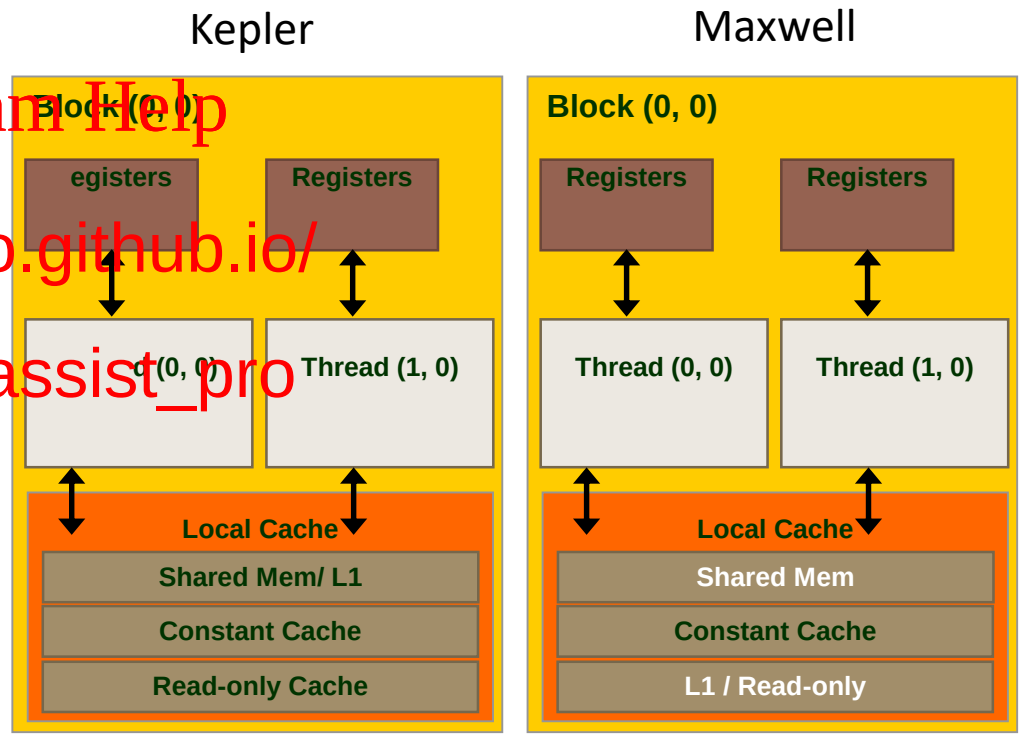
Review of last week

- We have seen the importance of different types of memory
  - And observed the performance improvement from read-only and constant cache usage
- So far we have seen how CUDA can be used for performing thread local computations; e.
  - Load data from memo
  - Perform thread-local computations
  - Store results back to global memory
- We will now consider another important type of memory
  - Shared memory

- Shared Memory
- Shared Memory Bank Conflicts
- 2D Shared Memory Bank Conflicts
- Boundary Conditions f ading
- Host-side Configurati y

Shared Memory

- Architecture Details
  - In Kepler (64KB) of Shared Memory is split between Shared Memory and L1 cache
    - The ratio to SM and L1 can be configured
  - In Maxwell 64KB of Shared Memory is dedicated
- Its just another Cache, right?
  - User configurable
  - Requires manually loading and synchronising data

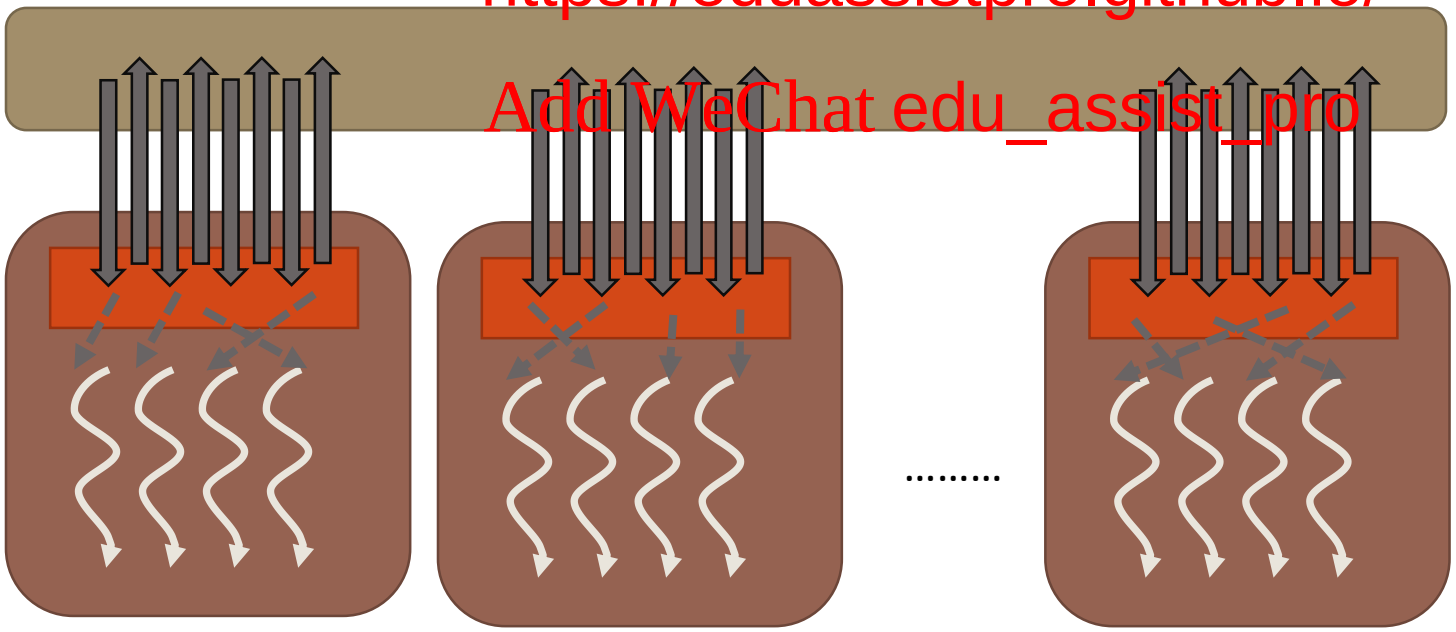


Shared Memory

- Performance
  - Shared memory is very fast
  - Bandwidth > 1 TB/s
- Block level computation
  - Challenges the thread
  - Allows data to be shar e same block
  - User configurable cache at the thread
  - Still no broader synchronisation beyond the level of thread blocks

Block Local Computation

- Partition data into groups that fit into shared memory
- Load subset of data into shared memory
- Perform computation on the subset
- Copy subset back to g



Move, execute, move

- From Host view
  - Move: Data to GPU memory
  - Execute: Kernel
  - Move: Data back to host
- From Device view
  - Move: Data from device registers
  - Execute: instructions
  - Move: Data back to device memory

- From Host view
  - Move: Data to GPU memory
  - Execute: Kernel
  - Move: Data back to host
- From Device view
  - Move: Data from device memory to cache
  - ute: subset of kernel (reusing values)
  - Data back to device memory

From Block View

- Move: Data from local cache
- Execute: instructions
- Move: Data back to local cache (or device memory)

Thread level parallelism

Block level parallelism

A Case for Shared Memory

```
__global__ void sum3_kernel(int *c, int *a)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int left, right;

    //load value at i-1
    left = 0;
    if (i > 0)
      left = a[i - 1];

    //load value at i+1
    right = 0;
    if (i < (N - 1))
      right = a[i + 1];

    c[i] = left + a[i] + right; //sum three values
}
```


Do we have a candidate for block level parallelism using shared memory?

A Case for Shared Memory

```
__global__ void sum3_kernel(int *c, int *a)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int left, right;

    //load value at i-1
    left = 0;
    if (i > 0)
      left = a[i - 1];

    //load value at i+1
    right = 0;
    if (i < (N - 1))
      right = a[i + 1];

    c[i] = left + a[i] + right; //sum three values
}
```


- Currently: Thread-local computation
- Bandwidth limited
  - Requires three loads per thread (at index `i-1`, `i`, and `i+1`)
- Block level solution: load each value only once!

CUDA Shared memory

- Shared memory between threads in the same block can be defined using `__shared__`
- Shared variables are only accessible from within device functions
  - Not addressable in host code
- Must be careful to av
  - Multiple threads writi mory variable
    - Results in undefined behaviour
  - Typically write to shared memory using `threadIdx`
  - Thread level synchronisation is available through `__syncthreads()`
    - Synchronises threads in the block


```
__shared__ int s_data[BLOCK_SIZE];
```

Example

```
__global__ void sum3_kernel(int *c, int *a)
{
  __shared__ int s_data[BLOCK_SIZE];

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int left, right;

  s_data[threadIdx.x] = a[i];
  __syncthreads();

  //load value at i-1
  left = 0;
  if (i > 0){
    left = s_data[threadIdx.x - 1];
  }

  //load value at i+1
  right = 0;
  if (i < (N - 1)){
    right = s_data[threadIdx.x + 1];
  }

  c[i] = left + s_data[threadIdx.x] + right; //sum
}
```

- Allocate a shared array
  - One integer element per thread
- Each thread loads a single item to shared memory
- `__syncthreads` to ensure memory data is populated by s elements through shared memory

What is wrong with this code?

Example

```
__global__ void sum3_kernel(int *c, int *a)
{
  __shared__ int s_data[BLOCK_SIZE];

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int left, right;

  s_data[threadIdx.x] = a[i];
  __syncthreads();

  //load value at i-1
  left = 0;
  if (i > 0){
    if (threadIdx.x > 0)
      left = s_data[threadIdx.x - 1];
    else
      left = a[i - 1];
  }

  //load value at i+1
  right = 0;
  if (i < (N - 1)){
    if (threadIdx.x <(BLOCK_SIZE-1))
      right = s_data[threadIdx.x + 1];
    else
      right = a[i + 1];
  }

  c[i] = left + s_data[threadIdx.x] + right; //sum
}
```

- Additional step required!
- **Check boundary conditions for the edge of the block**

Problems with Shared memory

- In the example we saw the introduction of boundary conditions
  - Global loads still present at boundaries
  - We have introduced divergence in the code (remember the SIMD model)
  - This is even more prevalent in 2D examples where we *tile* data into shared memory


```
//boun
left
if (i > 0){
  if (threadIdx.x > 0
    left = s_data[thre              ;
  else
    left = a[i - 1];
}
```

- Shared Memory
- Shared Memory Bank Conflicts
- 2D Shared Memory Bank Conflicts
- Boundary Conditions f ading
- Host-side Configurati y

Shared Memory Bank Conflicts

- Shared memory is arranged into 4byte (32bit banks)
  - A load or store of N addresses spanning N distinct banks can be serviced simultaneously
    - Overall bandwidth of $\times N$ a single module
    - Kepler+ can also serve broadcast accesses simultaneously
- A bank conflict occurs uest addresses from the same bank
  - Results in serialisation of the access
- Bank conflicts only occur between threads in a warp
  - There are 32 banks and 32 threads per warp
  - If two threads in a warp access the same bank this is said to be a 2-way bank conflict

Think about you block sized array of floats

```
bank = (index * stride) % 32
```

Stride=1 Stride=2 Stride=3

Access Strides

- Stride refers to the size (in increments of the bank size) between each threads memory access pattern
  - If threads access consecutive 4 byte values (e.g. `int` then the stride is 1.
    - No conflicts
  - If a thread accesses consecutive 8 bytes values (e.g. `double`) then the stride is 2.
    - 2 way conflicts
  - In general odd strides result in no conflicts

Stride (4 byte)	1			
TPB	128			
threadIdx.x			**index**	**bank**
0			0	1
1			1	2
2			2	3
3			3	4
4			4	
5			5	
6			6	
7			7	8
8			8	9
9			9	10
10			10	11
31			31	12
			Banks Used	32
			Max Conflicts	1

```
bank = (index*stride) % 32
```

More on SM banks

- Irregular access is fine as long as no bank conflicts
- Multiple threads can access the same bank conflict free if they access addresses broadcast
- Broadcast can be to any number of threads in a warp


```
__shared__ float s_data[??];
//read from shared memory using broadcast

some_thread_value = s_data[0] ;
```

Strided access example

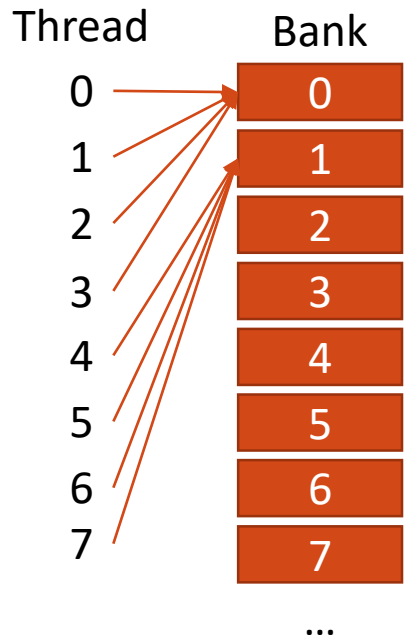
```
__shared__ char s_data[BLOCK_SIZE];

//load or calculate some_thread_value

s_data[threadIdx.x] = some_thread_value;
__syncthreads();
```


Thread	Bank
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
...	...
31	31

- What is the stride?
- What is the level of conflict?
- How can this be improved?

Strided access example

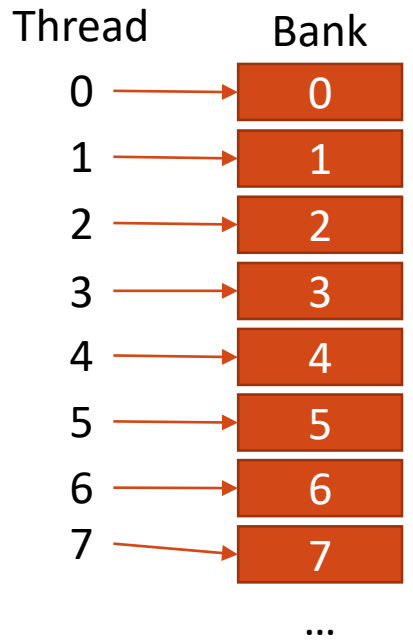
```
__shared__ char s_data[BLOCK_SIZE];

//load or calculate some_thread_value

s_data[threadIdx.x] = some_thread_value;
__syncthreads();
```



- What is the stride? Less than 1 (0.25)
- What is the level of conflict? 4 way
- How can this be improved? Increase the stride

Increase the stride (OK solution)

```
__shared__ char s_data[BLOCK_SIZE*4];

//load or calculate some_thread_value

s_data[threadIdx.x*4] = some_thread_value;
__syncthreads();
```



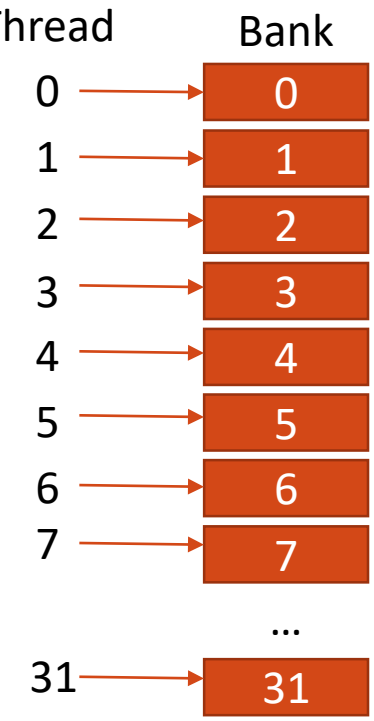
- What is the stride? 1
- What is the level of conflict? 1 way (no conflict)
- How can this be improved? Use less memory!

Increase the stride (good solution)

```
__shared__ char s_data[BLOCK_SIZE+1];

//load or calculate some_thread_value

s_data[CONFLICT_FREE(threadIdx.x)] = some_thread_value;
__syncthreads();
```

where

```
#define CHAR_MULTIPLIER 4
#define CONFLICT_FREE(x) (x*CHAR_MULTIPL          K_SIZE+1))
```

- What is the stride? 1
- What is the level of conflict? 1 way (no conflict)
- How much shared memory is required? BLOCK_SIZE+1

Increase the stride (good solution)

stride	0.25			
multipier	4			
block_size	128			
threadIdx.x			adjusted index	bank
0			0	0
1			4	1
2			8	2
3			12	3
4				
5				
6			24	6
7			28	7
8			32	8
9			36	9
10			40	10
…			…	…
127			**121**	30
		Banks Used		32
		Max Conflicts		1

`=(tid*multiplier) % (block_size+1)`

`=(index*stride) % 32`

`SIZE+1` unique

- Much better than `BLOCK_SIZE*4` unique indices

- ❑Shared Memory
- ❑Shared Memory Bank Conflicts
- ❑2D Shared Memory Bank Conflicts
- ❑Boundary Conditions f ading
- ❑Host-side Configurati y

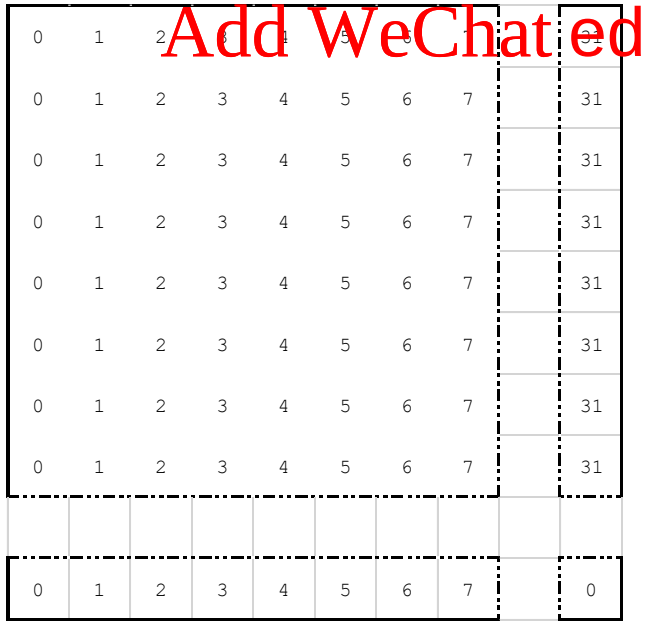
Bank conflicts with 2D tiles

```
__global__ void image_kernel(float *image)
{
  __shared__ float s_data[BLOCK_DIM][BLOCK_DIM];

  for (int i = 0; i < BLOCK_DIM; i++){
    some_thread_value += f(s_data[threadIdx.x][i]);
}
```

- Example where each thread (of 2D block) operates on a row
  - Loads values by column


```
bank = threadIdx.x * stride
```

Shared Memory Bank



Bank conflicts with 2D tiles

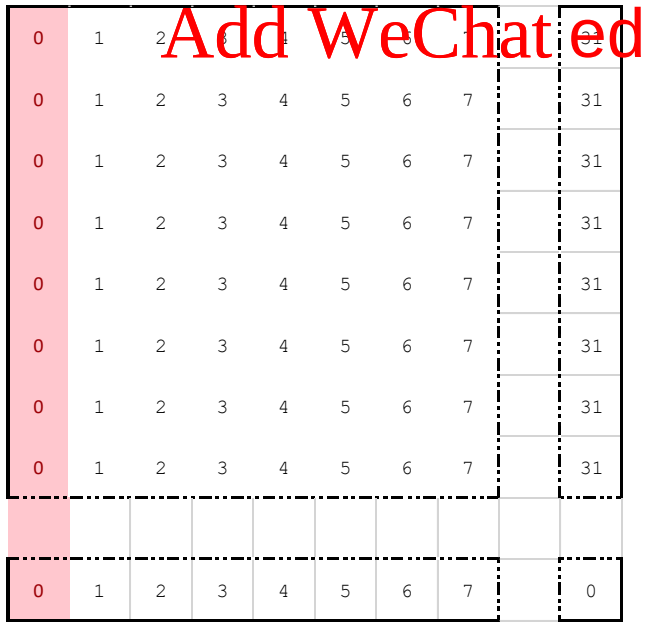
```
__global__ void image_kernel(float *image)
{
  __shared__ float s_data[BLOCK_DIM][BLOCK_DIM];

  for (int i = 0; i < BLOCK_DIM; i++){
    some_thread_value += f(s_data[threadIdx.x][i]);
}
```

- Example where each thread (of 2D block) operates on a row
  - Loads values by column

bank = threadIdx.x * stride

BLOCK_DIM=32

Shared Memory Bank



- 32 way bank conflicts!
  - Very bad
- Stride = 32

i=0

Bank conflicts with 2D tiles

```
__global__ void image_kernel(float *image)
{
  __shared__ float s_data[BLOCK_DIM][BLOCK_DIM];

  for (int i = 0; i < BLOCK_DIM; i++){
    some_thread_value += f(s_data[threadIdx.x][i]);
}
```



```
bank = threadIdx.x * stride
```

BLOCK_DIM=32

Shared Memory Bank

0	1	2	3	4	5	6	7		31
0	1	2	3	4	5	6	7		31
0	1	2	3	4	5	6	7		31
0	1	2	3	4	5	6	7		31
0	1	2	3	4	5	6	7		31
0	1	2	3	4	5	6	7		31
0	1	2	3	4	5	6	7		31
0	1	2	3	4	5	6	7		31
0	1	2	3	4	5	6	7		0

i=0

- ❑ Example where each thread (of 2D block) operates on a row
  - ❑ Loads values by column

How to fix

- ❑ **Memory padding**
- ❑ **Transpose the matrix**
  - ❑ **Or operate on columns (loading by row) if possible**

Bank conflicts with 2D tiles

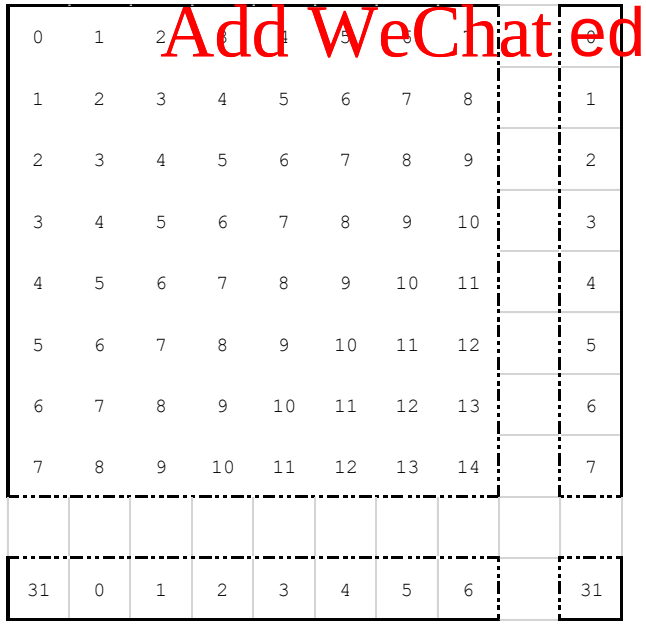
```
__global__ void image_kernel(float *image)
{
  __shared__ float s_data[BLOCK_DIM][BLOCK_DIM+1];

  for (int i = 0; i < BLOCK_DIM; i++){
    some_thread_value += f(d_data[threadIdx.x][i]);
}
```

❑Memory Padding Solution

bank = threadIdx.x * stride

BLOCK_DIM+1=33

Shared Memory Bank



Bank conflicts with 2D tiles

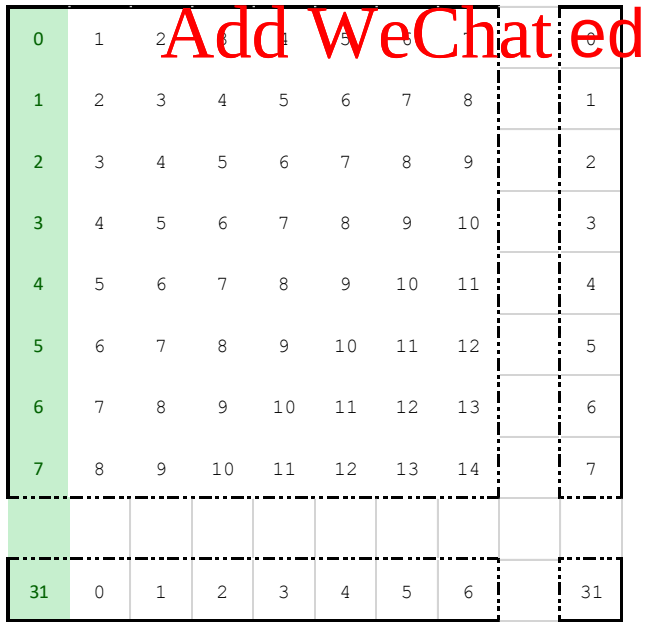
```
__global__ void image_kernel(float *image)
{
  __shared__ float s_data[BLOCK_DIM][BLOCK_DIM+1];

  for (int i = 0; i < BLOCK_DIM; i++){
    some_thread_value += f(d_data[threadIdx.x][i]);
}
```

❑**Memory Padding Solution**

bank = threadIdx.x * stride

BLOCK_DIM+1=33

Shared Memory Bank



i=0

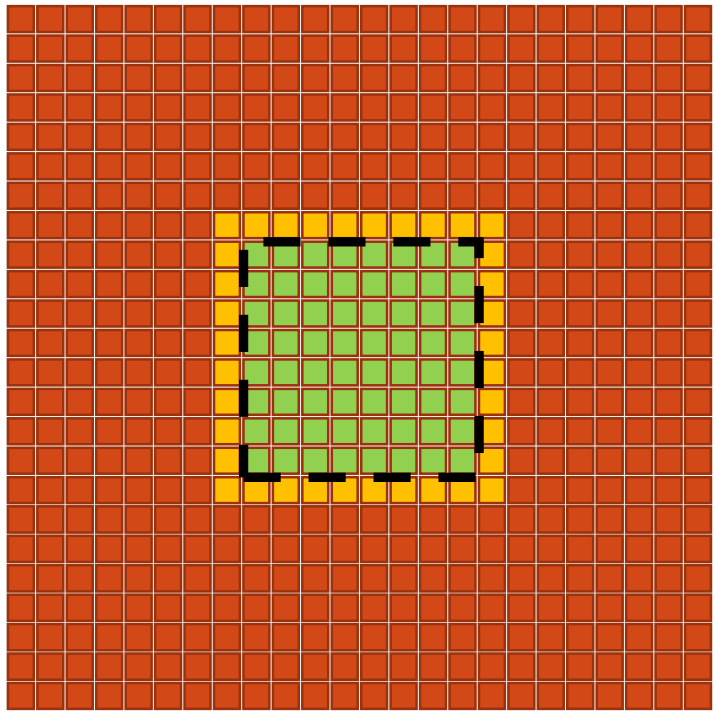
very thread in warp reads from different bank

❑*Alternative: Transpose solution left to you!*

- ❑Shared Memory
- ❑Shared Memory Bank Conflicts
- ❑2D Shared Memory Bank Conflicts
- ❑Boundary Conditions f
- ❑Host-side Configurati

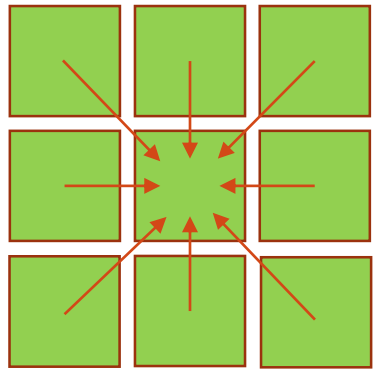
Boundary Conditions & Shared Memory Tiling

- Consider a 2D problem where data is gathered from neighbouring cells
  - Each cell reads 8 values (gather pattern)
  - Sounds like a good candidate for shared memory
    - We can tile data into



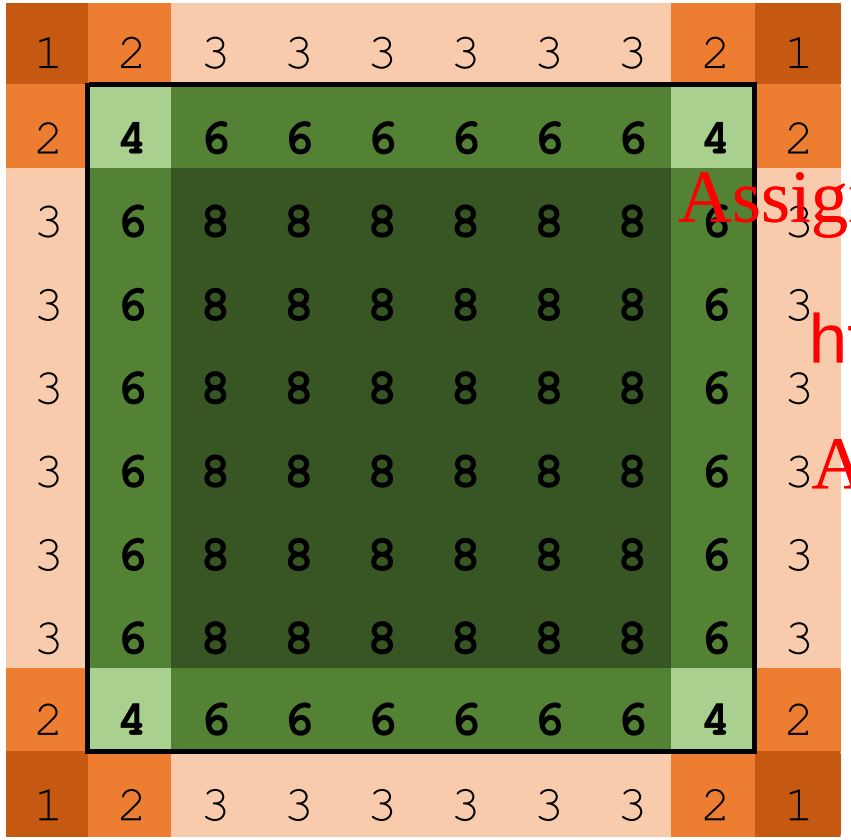
Thread Block size is 8x8

- Data tiled into shared memory
- Data not tiled into shared memory



Gather pattern

Problem with our tiling approach



- Memory access pattern is good for values inside the boundary
- Memor f boundary is loaded multiple
  - 92 un-cached reads
  - 92 loads

Boundary Condition Improvements

- Launch more threads
  - Launch thread block of `DIM+2 × DIM+2`
  - Allocate one element of space per thread in SM
  - Every thread loads one value
  - Only threads in inner s
    - Causes under utilisatio
- Use more shared memory per thre
  - Launch same `DIM × DIM` threads
  - Allocate `DIM+2 × DIM+2` elements of space in SM
  - Threads on boundary load multiple elements
    - Causes unbalanced loads
  - All threads perform compute values

$$Utilisation = \frac{DIM^2}{(DIM+2)^2}$$

DIM	Utilisation
8	64%
12	73%
16	79%
20	83%
24	85%
28	87%
32	89%
36	90%
40	91%
44	91%
48	92%

- Shared Memory
- Shared Memory Bank Conflicts
- 2D Shared Memory Bank Conflicts
- Boundary Conditions f ading
- Host-side Configurati

Dynamically Assigned Shared Memory

- It is possibly to dynamically assign shared memory at runtime.
- Requires both a host and device modification to code
  - Device: Must declare shared memory as extern
  - Host: Must declare shared memory size in kernel launch parameters


```
unsigned int sm_size =
image_kernel<<<blocksPe                    e >>>(d_image);

__global__ void image_kernel(float *image
{
  extern __shared__ float s_data[];

}
```

Is equivalent to

```
image_kernel<<<blocksPerGrid, threadsPerBlock>>>(d_image);

__global__ void image_kernel(float *image)
{
  __shared__ float *s_data[DIM][DIM];

}
```

Summary

- Shared Memory introduces the idea of block level computation rather than just thread level computation
- Shared Memory is a limited resource but can be very useful for reducing global memory bandwidth
  - Where data is reused
- Shared Memory requires synchronization unlike other general purpose caches (i.e. L1, texture)
- For optimal performance memory banks must be considered and boundary conditions must be handled
- There are hardware specific options for configuring how Shared Memory is used

Acknowledgements and Further Reading

- http://cuda-programming.blogspot.co.uk/2013/02/bank-conflicts-in-shared-memory-in-cuda.html
- http://acceleware.com/blog/maximizing-shared-memory-bandwidth-nvidia-kepler-gpus

Shared Memory Preferences

- In Compute 2.0+ (Fermi) and Compute 3.0+ devices (Kepler) it is possible to configure the ratio of SM and L1 with host function
  - `cudaDeviceSetCacheConfig(enum cudaFuncCache)`
    - does this for all kernels
  - `cudaFuncSetCach` `uncCache)`
    - for a single kernel
  - Possible values are;
    - `cudaFuncCachePreferNone`: defaul `uration`
    - `cudaFuncCachePreferShared`: 48KB SM and 16 KB L1
    - `cudaFuncCachePreferL1`: 16KB SM and 64 KB L1
    - `cudaFuncCachePreferEqual`: 32KB SM and 32KB L1 (only available on Kepler)
  - Not required in Maxwell