Parallel Computing with GPUs

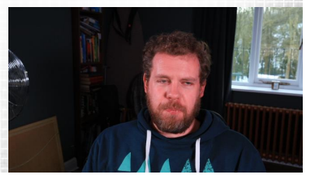
Shared Memory
Part 1 – Introduction to Shared Memory

The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

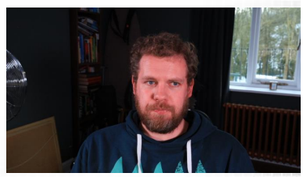
This Lecture (learning objectives)

- Shared Memory
  - Repeat important concepts of grids, blocks and warps
  - Identify a use case for shared memory
  - Demonstrate the use of shared memory on a simple problem
  - Recognise potential issues caused by use of shared memory (boundary conditions)

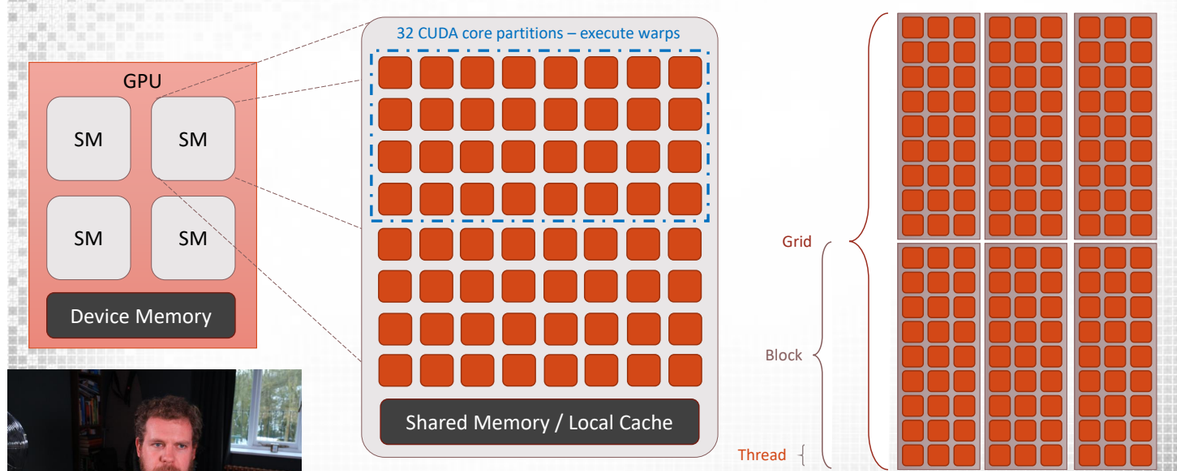


Review of last week

- 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.g.
  - Load data from memory to registers
  - Perform thread-local computations
  - Store results back to global memory
- We will now consider another important type of memory
  - Shared memory



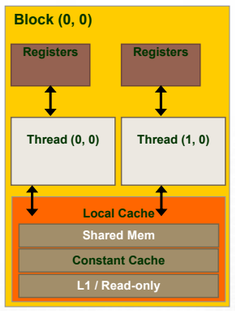
Grids, Blocks, Warps & Threads



Shared Memory

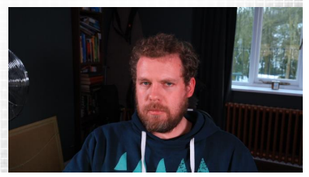
- Architecture Details
  - In Maxwell 64KB of Shared Memory is dedicated
- Its just another Cache, right?
  - User configurable
  - Requires manually loading and synchronising data

Maxwell/Pascal



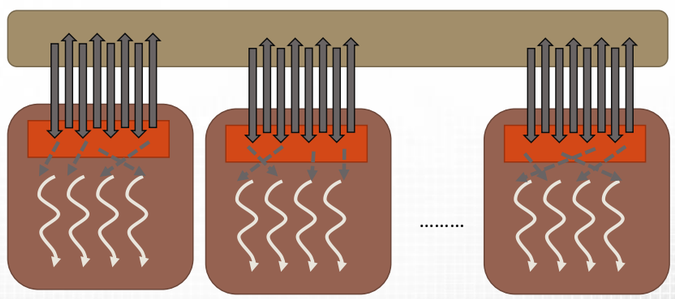
Shared Memory

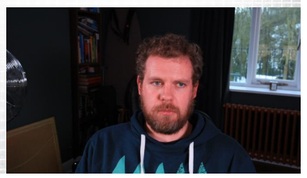
- Performance
  - Shared memory is very fast
  - Bandwidth > 1 TB/s
- Block level computation
  - Challenges the thread level view…
  - Allows data to be shared between threads in the same block
  - User configurable cache at the thread block level
  - Still no broader synchronisation beyond the level 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 global memory

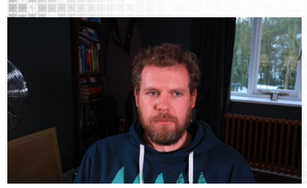




Move, execute, move

Thread level parallelism

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



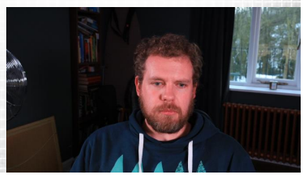
Block level parallelism

- From Host view
  - Move: Data to GPU memory
  - Execute: Kernel
  - Move: Data back to host
- From Device view
  - Move: Data from device memory to local cache
  - Execute: subset of kernel (reusing cached values)
  - Move: Data back to device memory
- From Block View
  - Move: Data from local cache
  - Execute: instructions
  - Move: Data back to local cache (or device 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
}
```

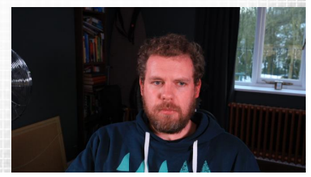
Do we have a candidate for block level parallelism using shared memory?



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
}
```

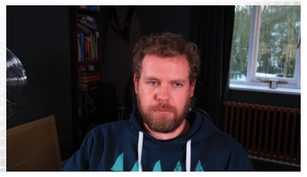
- Currently: Thread-local computation
- Bandwidth limited
  - Requires three loads per thread (at index `i-1`, `i`, and `i+1`)
- Block level solution: load each value only once!



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 avoid race conditions
  - Multiple threads writing to the same shared memory 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];
```

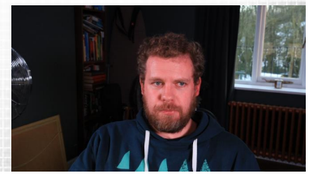


Example

```
__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
}
```

What is wrong with this code?

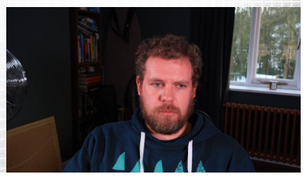
- Allocate a shared array
  - One integer element per thread
- Each thread loads a single item to shared memory
- Call `__syncthreads` to ensure shared memory data is populated by all threads
- Load all elements through shared memory



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

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

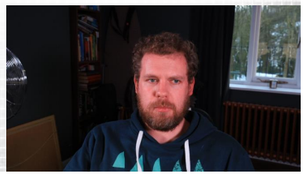
Dynamically Assigned Shared Memory

- 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 = sizeof(float)*DIM*DIM;
image_kernel<<<blocksPerGrid, threadsPerBlock, sm_size >>>(d_image);

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

}
```

Is equivalent to

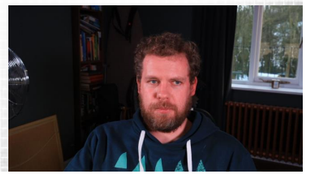
```
image_kernel<<<blocksPerGrid, threadsPerBlock>>>(d_image);

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

}
```



Summary

- Shared Memory
  - Repeat important concepts of grids, blocks and warps
  - Identify a use case for shared memory
  - Demonstrate the use of shared memory on a simple problem
  - Recognise potential issues caused by use of shared memory (boundary conditions)

- Next Lecture: Shared Memory Bank Conflicts

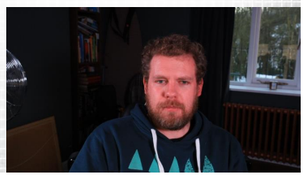


Parallel Computing with GPUs

Shared Memory
Part 2 – Bank Conflicts

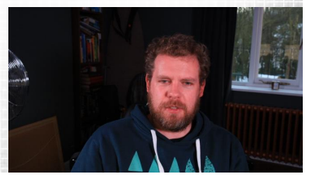
The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/



This Lecture (learning objectives)

- Shared Memory Bank Conflicts
  - Explain the concepts of memory banks and how conflicts can occur
  - Give examples of access strides and the impact on bank conflicts
  - Present solutions for avoiding bank conflicts



Shared Memory Bank Conflicts

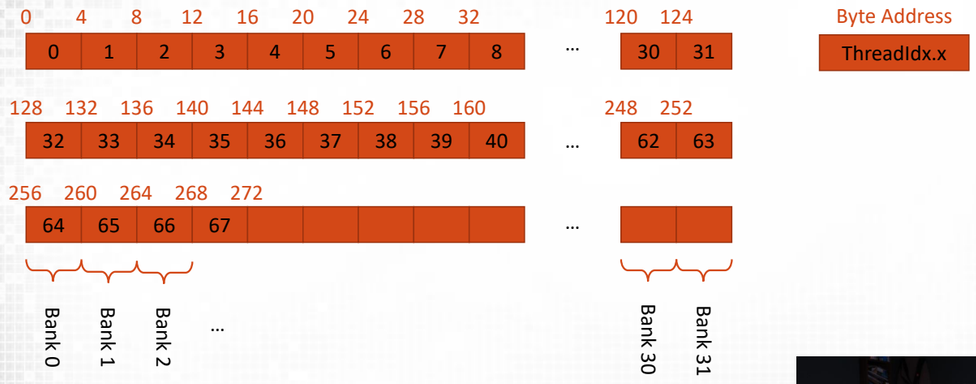
- Shared memory is arranged into 4byte (32bit banks)
  - A load or store of N addresses spanning N distinct banks can be serviced simultaneously
    - Overall bandwidth of $\times N$ a single module
    - Kepler+ can also serve broadcast accesses simultaneously
- A bank conflict occurs when two threads request addresses from the same bank (without reading in broadcast pattern)
  - 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

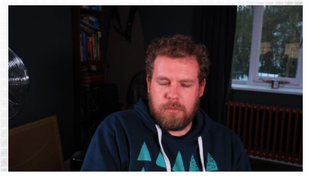
Think about you block sized array of floats

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

The University Of Sheffield.

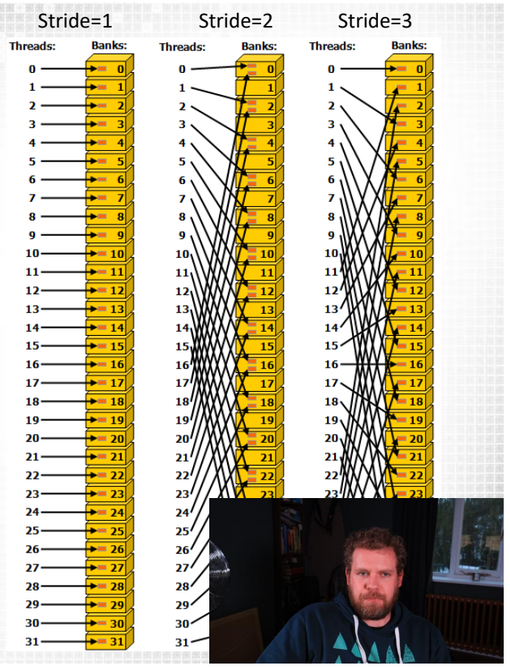
Logical View of Shared Memory Banks





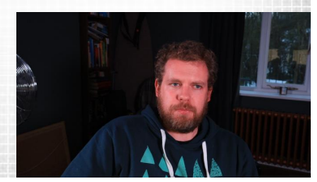
Access Strides

- 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` or `float`) then the stride is 1.
    - No conflicts
  - If a threads 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		5	6
6		6	7
7		7	8
8		8	9
9		9	10
10		10	11
31		31	12
		Banks Used	32
		Max Conflicts	1

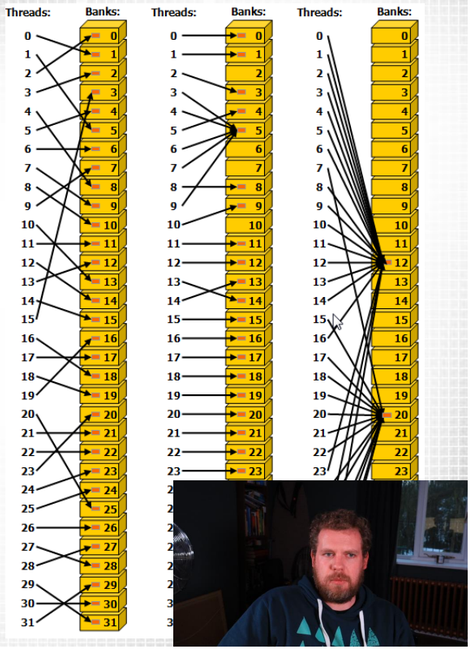
`bank = (index*stride) % 32`



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 the same addresses (e.g. a broadcast access pattern)
- 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] ;
```

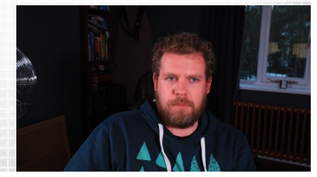


Strided access example

```
__shared__ float s_data[BLOCK_SIZE];

//load or calculate some_thread_value

s_data[threadIdx.x*2] = some_thread_value;
__syncthreads();
```

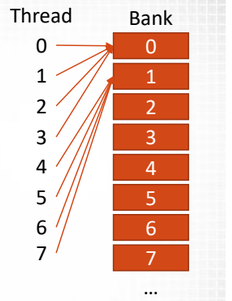
Thread	Bank
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
	…
31	31

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



Strided access example

```
__shared__ float s_data[BLOCK_SIZE*2];

//load or calculate some_thread_value

s_data[threadIdx.x*2] = some_thread_value;
__syncthreads();
```



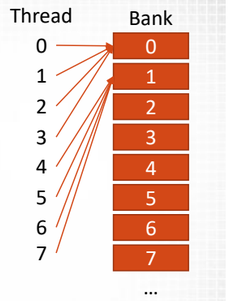
- What is the stride? 2
- What is the level of conflict? 2 way
- How can this be improved? Remove the stride (use a stride of 1)

Edge case for broadcast access

```
__shared__ char s_data[BLOCK_SIZE];

//load or calculate some_thread_value

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


Edge case for broadcast access

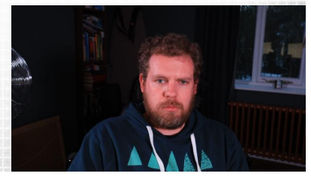
```
__shared__ char s_data[BLOCK_SIZE];

//load or calculate some_thread_value

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



- Stride: 0.25 BUT no conflict (for compute capability > 2.0)
  - Why?
  - "A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 32-bit word or within two 32-bit words whose indices *i* and *j* are in the same 64-word aligned segment (i.e., a segment whose first index is a multiple of 64) and such that *j=i+32* (even though the addresses of the two sub-words fall in the same bank): In that case, for read accesses, the 32-bit words are broadcast to the requesting threads and for write accesses, each sub-word is written by only one of the threads (which thread performs the write is undefined)". - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory-3-0
  - E.g. Adjacent threads accessing the same bank falls under the broadcast pattern

Large Data Types and Phasing

- For larger data types Volta+ Hardware allows scheduling of address accesses in phases
  - Bank conflicts only occur between threads in the same phase
- Double and 8B data types
  - 2 Phases
    - Process addresses in first 16 threads of warp
    - Process addresses in the last 16 of the warp
- 16B data types (E.g. float4)
  - 4 Phases
    - Each phase processes addresses in a quarter of the warp



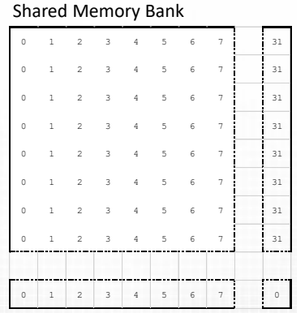
Bank conflicts with 2D tiles

```
__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 % 32
```

Shared Memory Bank



Bank conflicts with 2D tiles

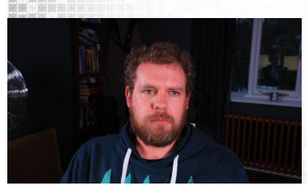
```
__global__ void image_kernel(float *image)
{
  __shared__ float s_data[BLOCK_DIM][BLOCK_DIM];

  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

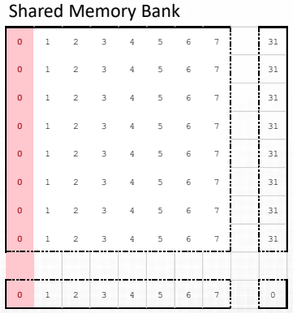
```
bank = threadIdx.x * stride % 32
```

BLOCK_DIM=32

i=0



Shared Memory Bank



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

Bank conflicts with 2D tiles

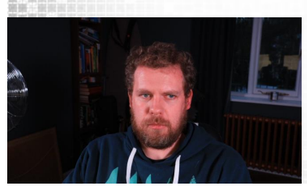
```
__global__ void image_kernel(float *image)
{
  __shared__ float s_data[BLOCK_DIM][BLOCK_DIM];

  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

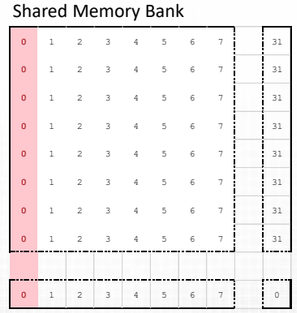
```
bank = threadIdx.x * stride % 32
```

BLOCK_DIM=32

i=0



Shared Memory Bank



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

Bank conflicts with 2D tiles

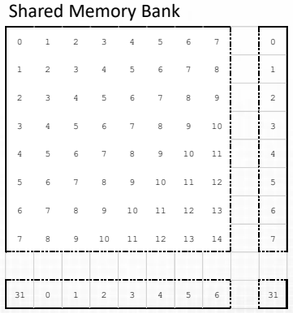
```
__global__ void image_kernel(float *image)
{
  __shared__ float s_data[BLOCK_DIM][BLOCK_DIM+1];

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

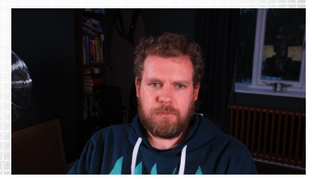
- **Memory Padding Solution**

```
bank = threadIdx.x * stride %32
```

BLOCK_DIM+1=33

Shared Memory Bank





Bank conflicts with 2D tiles

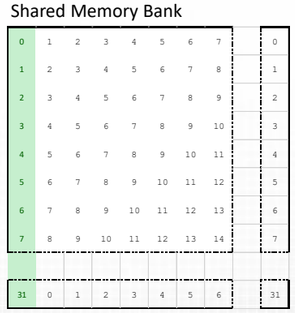
```
__global__ void image_kernel(float *image)
{
  __shared__ float s_data[BLOCK_DIM][BLOCK_DIM+1];

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

```
bank = threadIdx.x * stride % 32
```

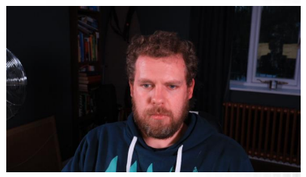
BLOCK_DIM+1=33

i=0

Shared Memory Bank



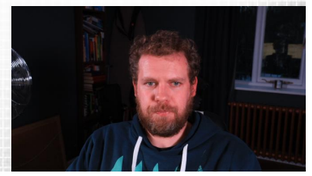
- Memory Padding Solution



- Every thread in warp reads from different bank
- *Alternative: Transpose solution left to you!*

Summary

- Shared Memory Bank Conflicts
  - Explain the concepts of memory banks and how conflicts can occur
  - Give examples of access strides and the impact on bank conflicts
  - Present solutions for avoiding bank conflicts
- Next Lecture: Boundary Conditions



Kepler Shared Memory Bank Size

- In Kepler it is possible to specify the size of shared memory banks
  - This changes the bank conflicts depending on your access pattern
- `cudaDeviceSharedMemConfig`
  - Options are;
    - `cudaSharedMemBankSizeDefault`: default size
    - `cudaSharedMemBankSizeFourByte`: 32 bit size banks
    - `cudaSharedMemBankSizeEightByte`: 64 bit size banks
  - Can improve performance of access for even strides.
- Not available in Maxwell +.

Parallel Computing with GPUs

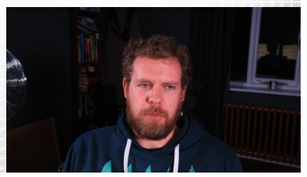
Shared Memory
Part 3 – Boundary Conditions

The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

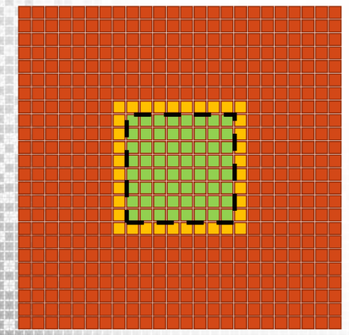
This Lecture (learning objectives)

- Boundary Conditions
  - Demonstrate the impact of boundary conditions for 2D gather problems
  - Compare and contrast different solutions to solving boundary problems



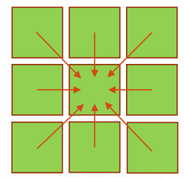
Boundary Conditions & Shared Memory Tiling

- 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 memory



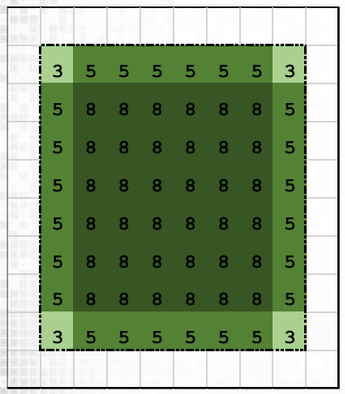
Thread Block size is 8x8

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



Gather pattern

Problem with our tiling approach



- Diagram shows number of cached reads
- Memory access pattern is good for threads at centre of the block
  - 6x6x8=288 cached reads
- Memory access for threads at the boundary of the block is poor
  - 132 cached reads
  - **92 un-cached reads**

Boundary Conditions Solutions (Easy)

- 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 DIM x DIM compute values
    - Causes under utilisation
- Use more shared memory per thread
  - 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%

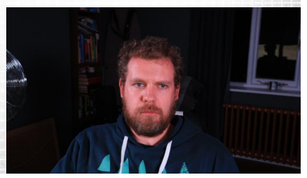
Boundary Conditions Solution (Harder)

- Use more shared memory per thread
  - Launch same `DIM × DIM` threads
  - Allocate `DIM+2 × DIM+2` elements of space in SM
  - Distribute the loading of SM evenly between threads
    - Thread position in the block must be translated to a position in SM for each load
    - Only last warp will have imbalance of at worse one load
- 100 loads
- 512/512 cached reads

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	0	1	2	3	4	5
6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35

Summary

- Boundary Conditions
  - Demonstrate the impact of boundary conditions for 2D gather problems
  - Compare and contrast different solutions to solving boundary problems



Acknowledgements and Further Reading

- Overview of Shared Memory Bank Conflicts
  - http://cuda-programming.blogspot.co.uk/2013/02/bank-conflicts-in-shared-memory-in-cuda.html
- Architecture Specific Guidance
  - http://acceleware.com/blog/maximizing-shared-memory-bandwidth-nvidia-kepler-gpus
  - https://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf