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)

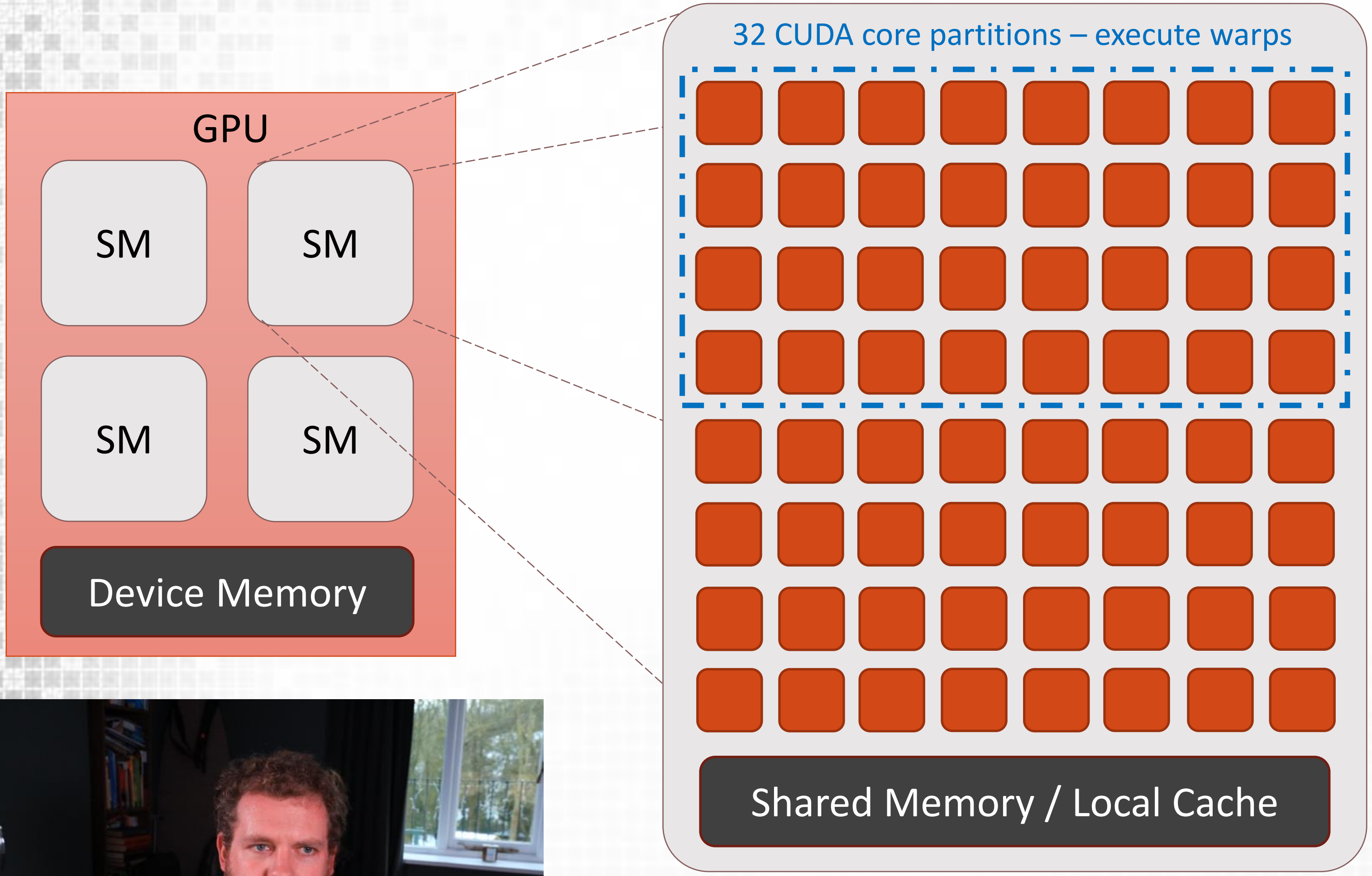
- Shared Memory
  - Repeat important concepts of grids, blocks and warps
  - Identify a use case for shared memory
  - Demonstrate the use of shared memory on a simple problem
  - Recognise potential issues caused by use of shared memory (boundary conditions)

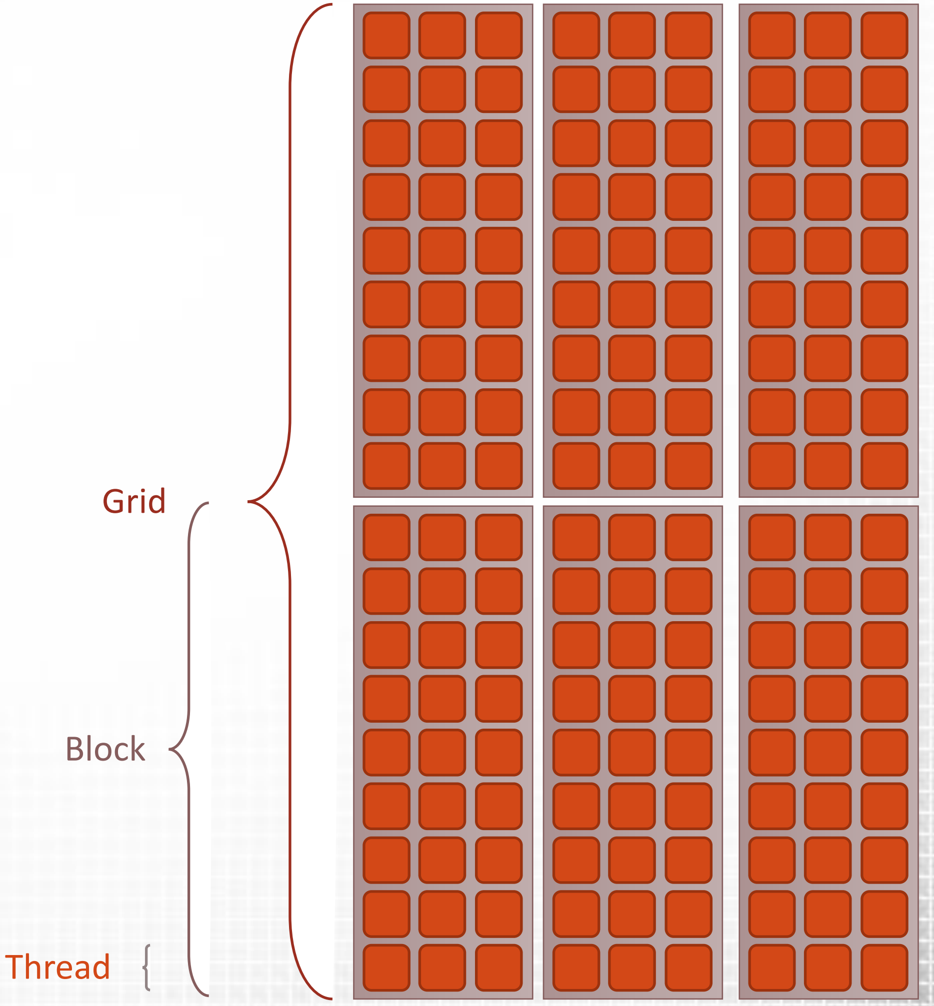
Review of last week

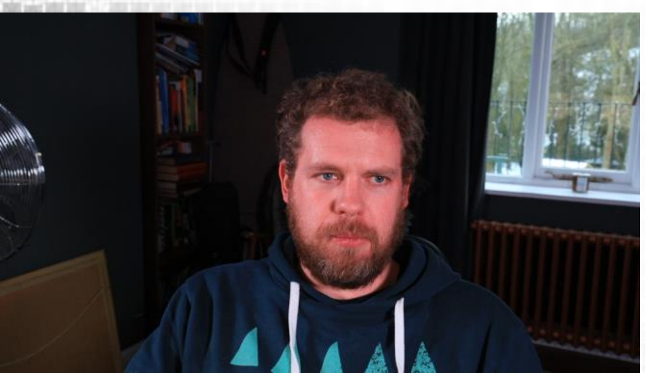
- We have seen the importance of different types of memory
  - And observed the performance improvement from read-only and constant cache usage
- So far we have seen how CUDA can be used for performing thread local computations; e.g.
  - Load data from memory to registers
  - Perform thread-local computations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

- We will now consider another important type of memory
  - Shared memory

Grids, Blocks, Warps & Threads



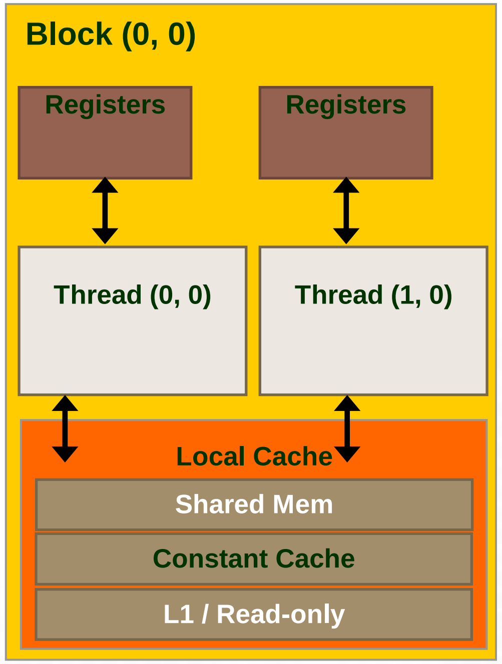




Shared Memory

- ❑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

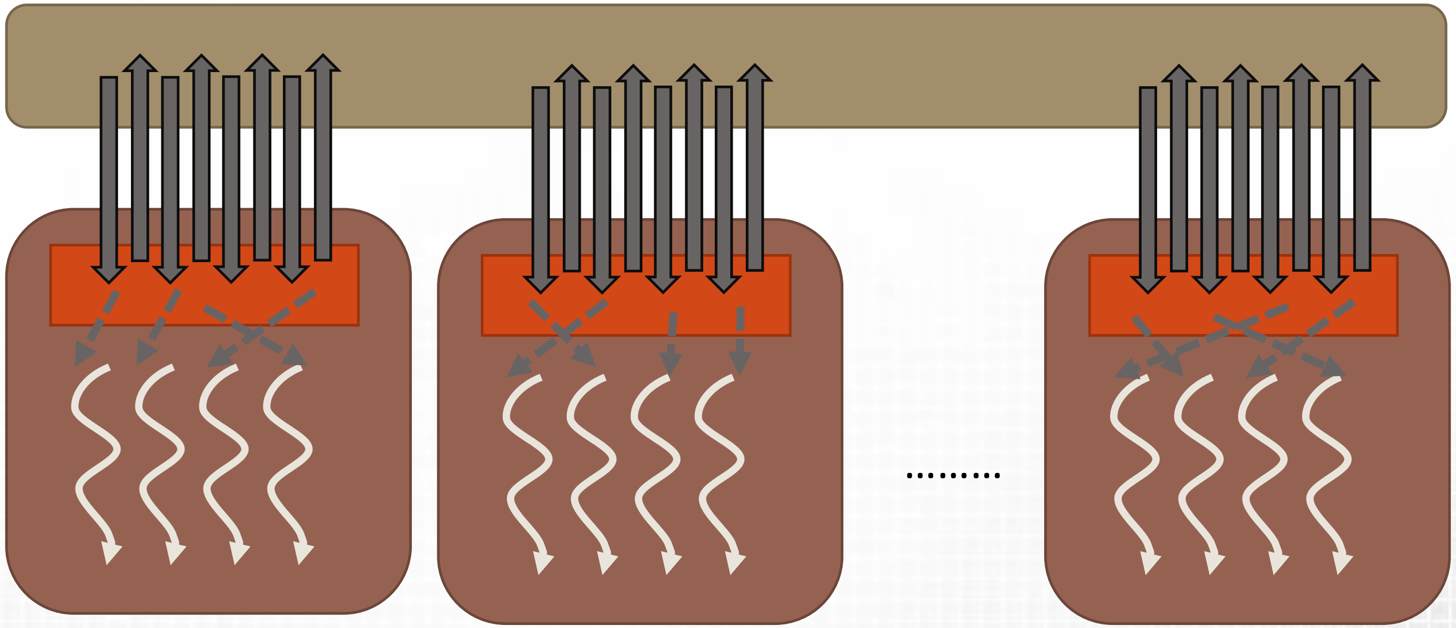


Shared Memory

- 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

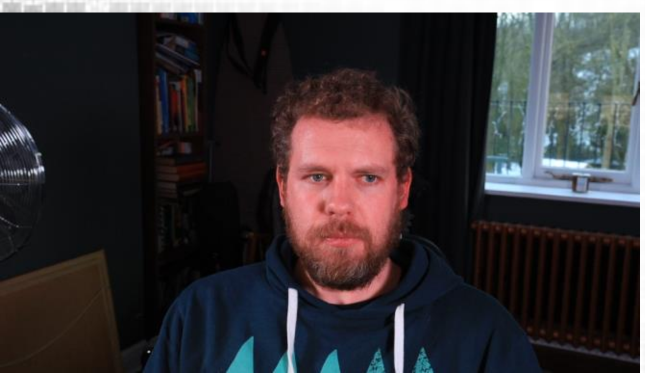
- ❑Partition data into groups that fit into shared memory
- ❑Load subset of data into shared memory
- ❑Perform computation on the subset
- ❑Copy subset back to global memory



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)

The University Of Sheffield.

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