Lecture 3
Control flow and synchronisation

Prof Wes Armour
wes.armour@eng.ox.ac.uk

Oxford e-Research Centre
Department of Engineering Science

Learning outcomes

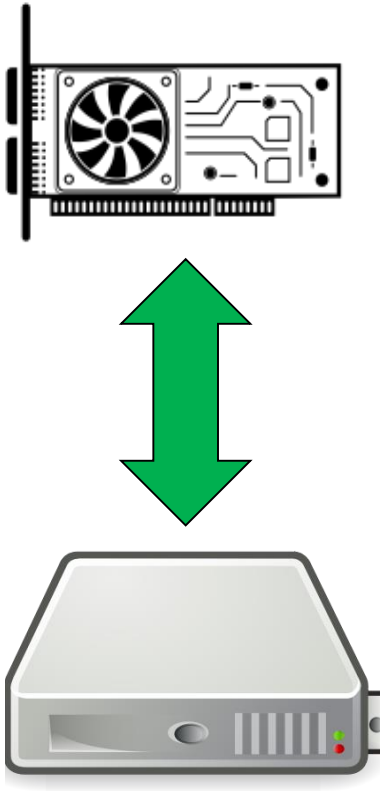
In this third lecture we will take a look at control flow and synchronisation.

You will learn about:

- Warps and control flow.
- Warp divergence.
- Synchronisation on the host and device.
- Atomic operations on the GPU.

A quick recap – initiate host

1. **Initiate host**
   - declare variables,
   - allocate memory on the host.



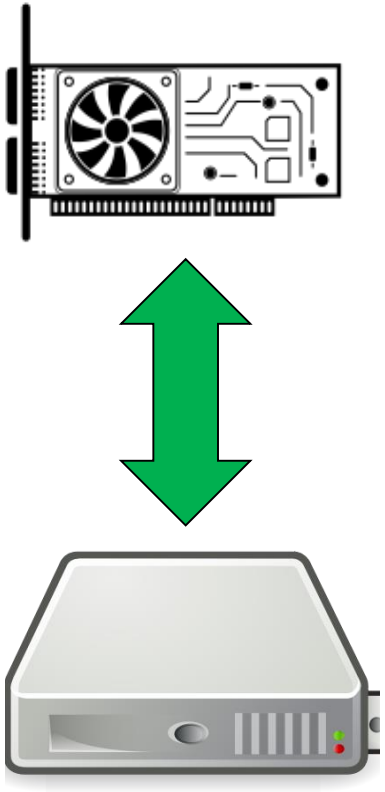
By RRZEicons [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], from Wikimedia Commons

```
// Allocate pointers for host and device memory
float *h_input, *h_output;

// malloc() host memory (this is in your RAM)
h_input  = (float*) malloc(mem_size);
h_output = (float*) malloc(mem_size);
```

A quick recap – initiate device

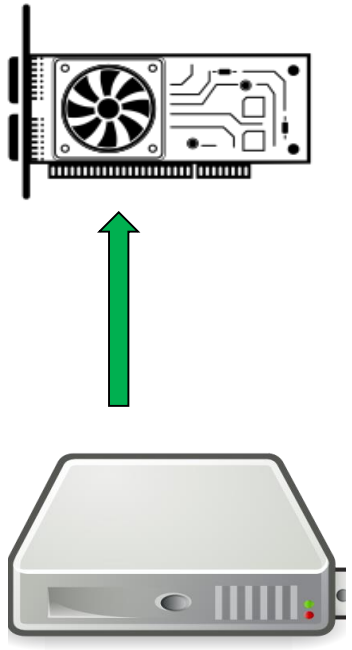
2. **Initiate device**
   - declare device variables/pointers,
   - allocate memory on the device.

```
// Allocate pointers for device memory
float *d_input, *d_output;

// Allocate device memory input and output
arrays
cudaMalloc((void**)&d_input,  mem_size);
cudaMalloc((void**)&d_output, mem_size);
```



A quick recap – copy data to device

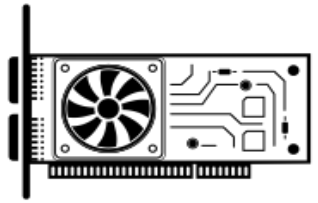
3. **Copy data from the host to device**



```
// Copy host memory to device input array
cudaMemcpy(device, host, mem_size, cudaMemcpyHostToDevice);
```

A quick recap – execute kernel

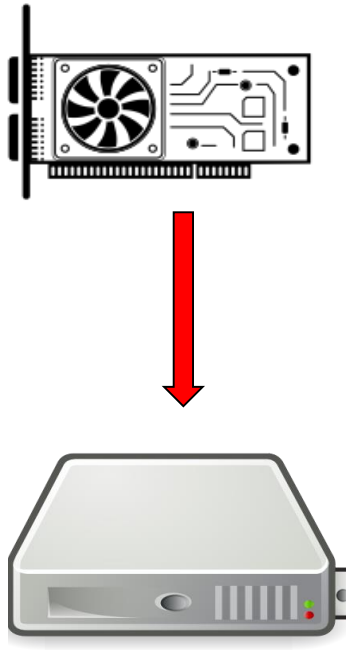
```
// Execute Kernel
my_kernel<<<nblocks,nthreads>>>(d_input, d_output);
```



4. **Execute kernel on GPU**

A quick recap – copy results from device

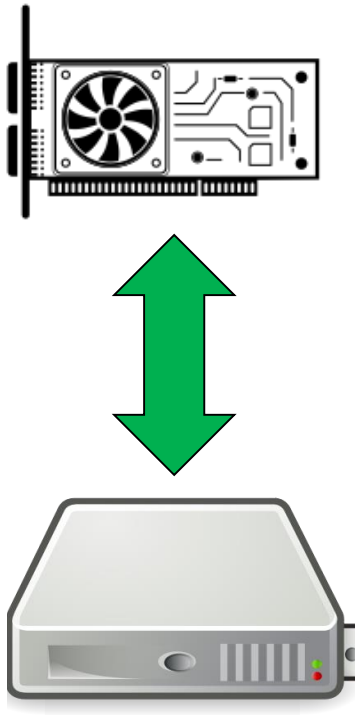
```
// Copy result from device to host
cudaMemcpy(host, device, mem_size, cudaMemcpyDeviceToHost);
```



5. **Copy results from the device to host**

A quick recap – free memory

```
// Cleanup memory on the device
cudaFree(d_input);
cudaFree(d_output);
```



6. **Free memory**
   - Free device memory: `cudaFree();`
   - Free host memory: `free();`

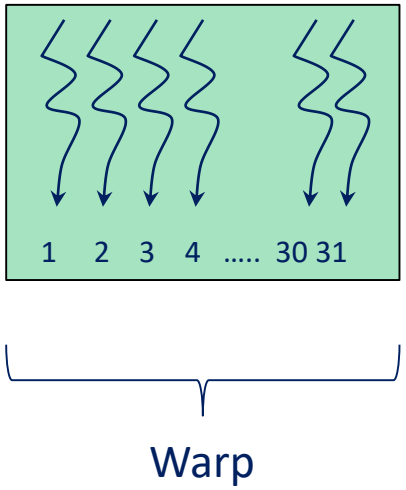
```
// Cleanup memory on the host
free(h_input);
free(h_output);
```

Warps

As introduced in Lecture one, code executes on a GPU in groups of threads, the threads are naturally grouped into lots of 32, a warp.

Threads execute in “lock-step” – each thread within a warp executes the same instruction at the same time (just on different data).

But what happens when different threads within the same warp need to do different things?



Warps and control flow

Different warps can execute different code, i.e. they can be at different points in their instruction streams (for example warp one calculates a "+", warp three calculates a "*").

This has no impact on performance.

How about if we have an `if` statement within a warp? (i.e. we have different possible code paths)??

```
if(radius <= 10.0) {
    area = PI * radius * radius;
} else {
    area = 0.0;
}
```

Warps and control flow

In this case if all threads *in a warp* have a radius less than or equal to 10 then the execution path for all threads follows the same direction. This is also true if all threads in the warp have a radius greater than 10.

What if some threads in the warp have a radius less than 10 and some more than 10?

```
if(radius <= 10.0) {
    area = PI * radius * radius;
} else {
    area = 0.0;
}
```

Warp divergence

GPUs don’t have the ability to execute the `if` and `else` at the same time.

The hardware serialises the separate execution paths in the `if` statement.

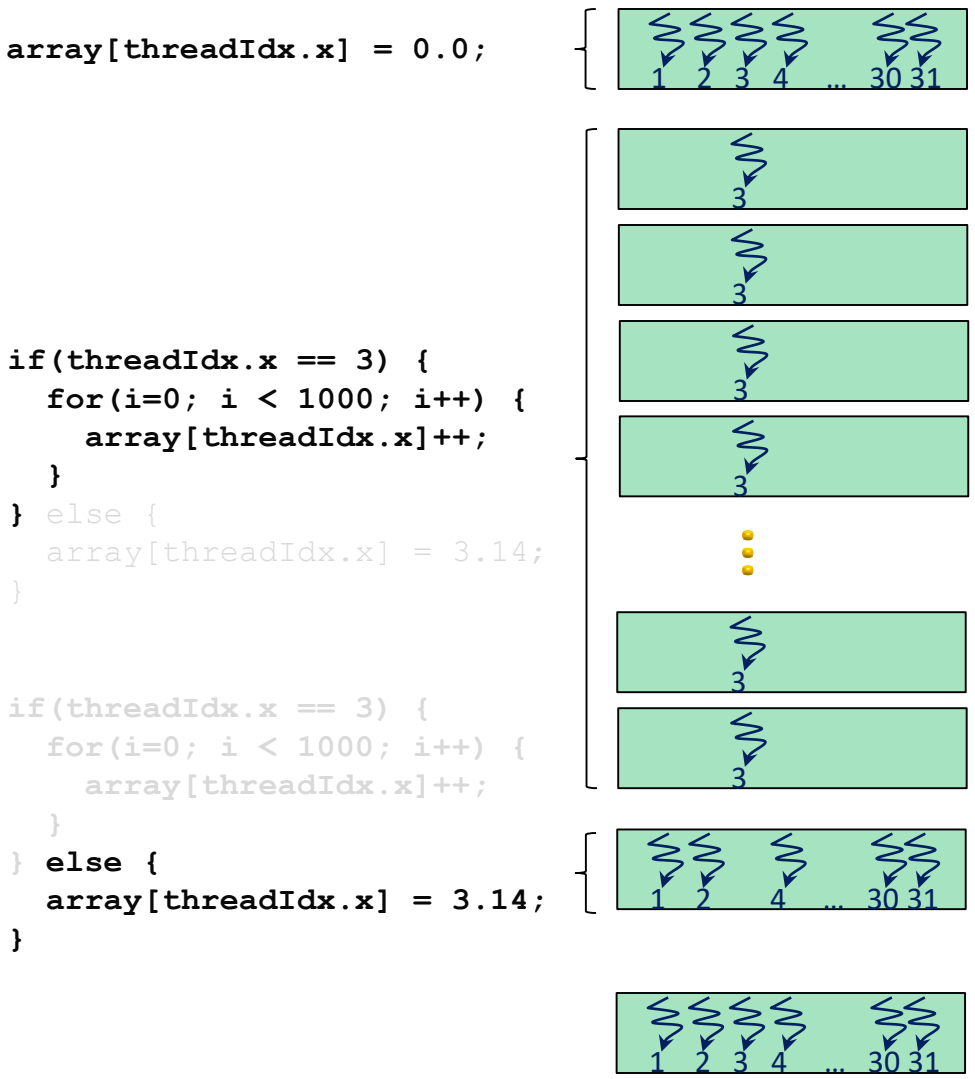
In this case both execution paths for the warp are executed but threads that don’t participate in a particular execution path are “masked”.

```
if(radius <= 10.0) {
    area = PI * radius * radius;
} else {
    area = 0.0;
}
```

Warp divergence

This is called ***warp divergence***

Warp divergence can cause significant slow down in code execution. In the worst case this can cause a loss of 32x in performance if only one thread needs to follow a computationally expensive branch and the rest of the threads do nothing.



Warp divergence in more detail

NVIDIA GPUs have *predicated* instructions which are carried out only if a logical flag is true.

```
p: a = b + c; // computed only if p is true
```

In the example on the right, all threads compute the logical predicate and the two predicated instructions when some threads within the warp have `x` values `< 0.0`, and others have `x ≥ 0`.

```
if (x<0.0) {
        z = x-2.0;
} else {
        z = sqrt(x);
}
```

The resulting code looks as follows:

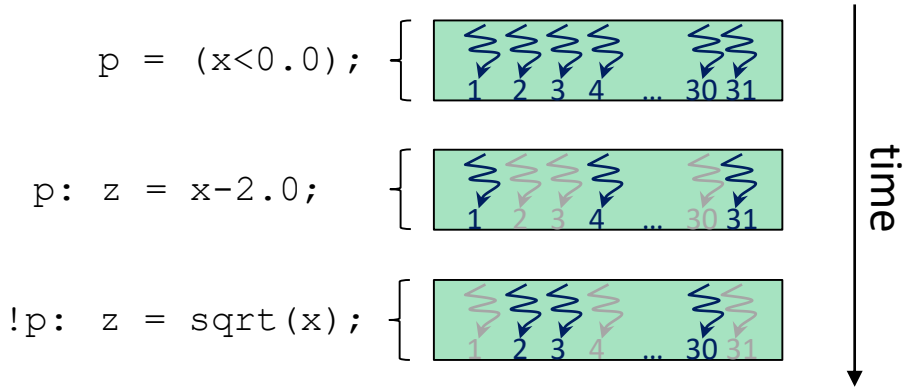
```
    p = (x<0.0);
 p: z = x-2.0;        // single instruction
!p: z = sqrt(x);
```

Warp divergence

Note that:

`sqrt(x)` would usually produce a `NaN` when `x<0`, but it's not really executed when `x<0` so there's no problem.

All threads execute both conditional branches, so execution cost is sum of both branches
⇒ potentially large loss of performance



Warp divergence

Another example on the right:

- `x[n]` is only read here if `n>=0`
- don't have to worry about illegal memory accesses when `n` is negative.

```
if (n>=0) {
        z = x[n];
} else {
        z = 0;
}
```

Warp voting

As mentioned previously, if all threads *within a warp* follow the execution path then there is no performance penalty.

If branches are big the nvcc compiler inserts code to check if all threads in the warp take the same branch, called ***warp voting***, and then branches accordingly.

Note:

- It doesn't matter what is happening with other warps – each warp is treated separately.
- If each warp only goes one way that's very efficient.
- Warp voting costs a few instructions, so for very simple branches the compiler just uses predication without voting.

```
          p = ...
          if (any(p)) {
p:                  ...
p:                  ...
          }

          if (any(!p)) {
!p:                 ...
!p:                 ...
          }
```

Warp divergence

In some cases it is possible to determine at compile time that all threads in the warp must branch in the same direction.

Look at the example on the right.

If `case` is a run-time argument, `case` will be the same for all threads during program execution and hence there's no need to vote.

```
if (case==1) {
        z = x*x;
} else {
        z = x+2.3;
}
```

Warp divergence

As previously illustrated warp divergence can lead to a big loss of parallel efficiency, something that's very damaging to GPU performance. **It's one of the first things to look out for in a new application**.

As we have shown, in the worst case, it's possible to lose a factor of 32× in performance if one thread needs to take an expensive branch, while rest do nothing.

A typical example of this is a PDE application with boundary conditions:

- If the boundary conditions are cheap, loop over all nodes and branch as needed for the boundary conditions.

- If boundary conditions are expensive, use two kernels:
  - First for interior points,
  - Second for boundary points

The most important thing is to understand hardware behaviour and design your algorithms / implementation accordingly.

Synchronisation

As we've mentioned before all threads within a warp execute in lock-step. They are explicitly synchronised.

We have also introduced `__syncthreads();` that provides a barrier function to synchronise threads within a thread block.

When writing conditional code its very important to ensure that all threads are able to reach the `__syncthreads();` **Otherwise the code can end up in *deadlock*.**

See the example on the right.

```
if(radius <= 10.0) {
          area = PI * radius * radius;
          __syncthreads();
} else {
          area = 0.0;
}
```

A typical application

Previously seen in lecture one, a typical CUDA program might look something like:

```
// load in data to shared memory
...
...
...

// synchronisation to ensure this has finished
__syncthreads();

// now do computation using shared data
...
...
...
```

Advanced synchronisation

There are other synchronisation instructions which are similar but have extra capabilities:

- `int __syncthreads_count(predicate)` → counts how many predicates are true.
- `int __syncthreads_and(predicate)` → returns non-zero (true) if all predicates are true.
- `int __syncthreads_or(predicate)` → returns non-zero (true) if any predicate is true.

I don’t know anybody who has used these and can’t think of a use for them.

Host and device synchronisation

What if we want to synchronise **everything** – *in our main() code we want to ensure that we reach a point where everything on the host and device is synchronised?*

```
cudaDeviceSynchronize();
```

Ensures that all asynchronous (sometimes called non-blocking) tasks are finished before code execution proceeds further.

This is a function used in host code.

Warp voting instructions

There are similar *warp voting* instructions which operate at the level of a warp:

- `int __all(predicate)` → returns non-zero (true) if all predicates in warp are true.
- `int __any(predicate)` → returns non-zero (true) if any predicate is true.
- `unsigned int __ballot(predicate)` → sets n^{th} bit based on the n^{th} predicate.

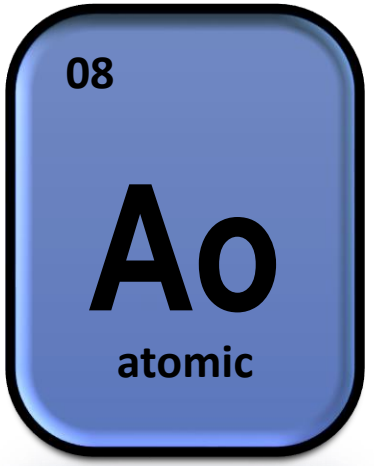
I haven't found a use for these in my codes.

Atomic operations

There will be instances where you need two or more threads to update a single variable.

Obviously due to the undefined scheduling of threads, several threads could try to update the variable at the same time leading to an undefined result.

To get around this issue, CUDA provides several atomic operations to do this for you, these guarantee that only one thread at a time can update the variable.



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Atomic operations

Lets look at an example (right). Here we declare an integer variable in shared memory.

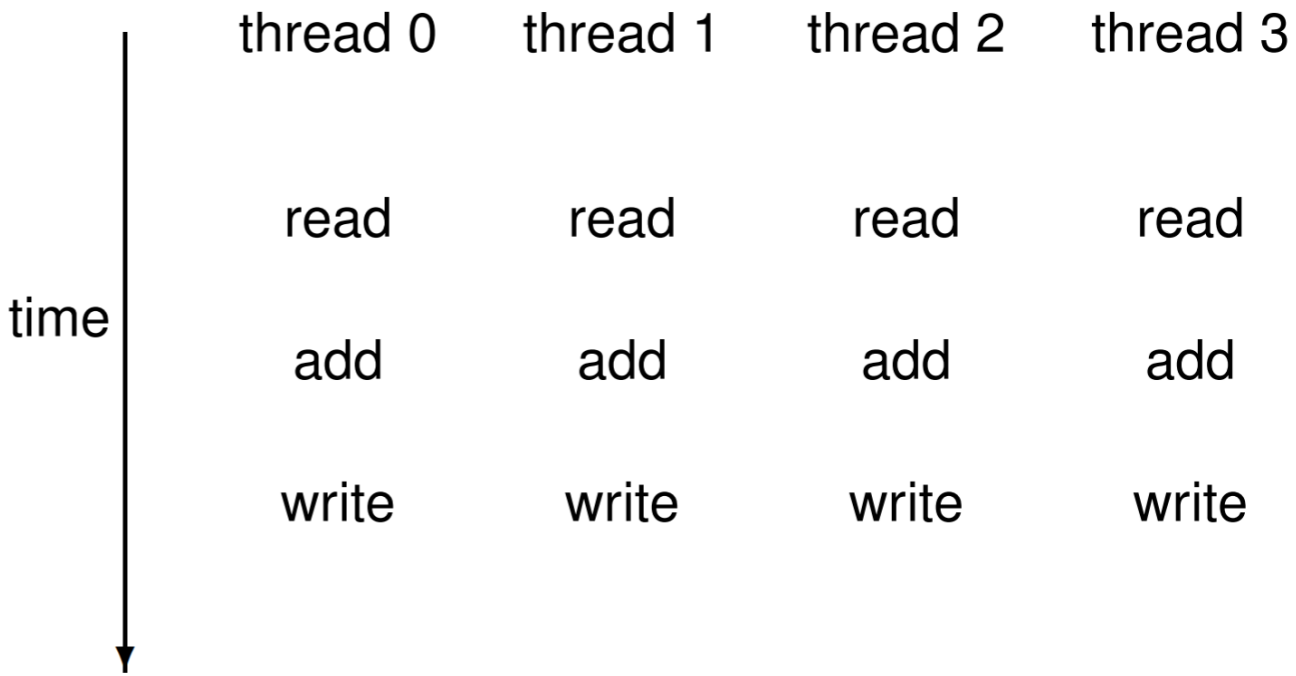
We then have some kernel code that has a conditional, for those threads whose conditional statement evaluates to true, they all try to add 1 to count.

This causes a problem. All threads will try to update the variable at the same time leading to an undefined (sometimes seemingly random) result.

```
__shared__ int count;

...

if ( ... ) count++;
```

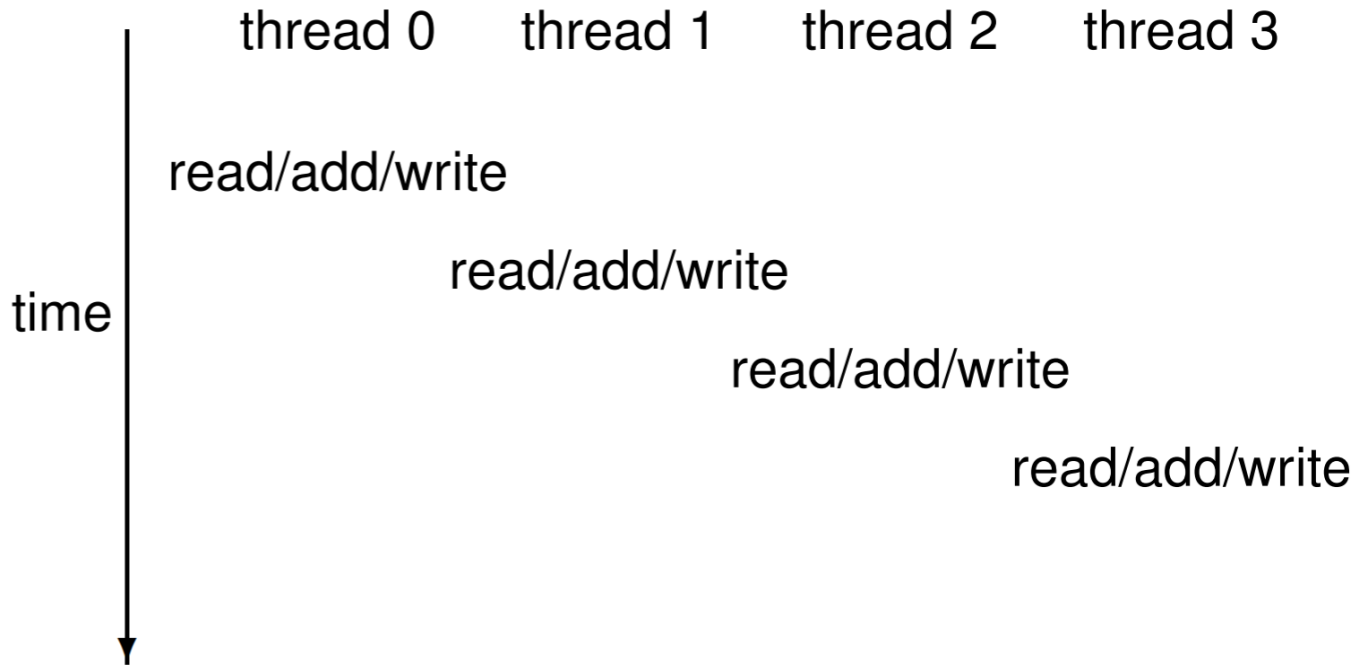
Atomic operations

Using standard instructions, multiple threads in the same warp will only update it once (one thread will manage to update the counter, others will not be able to).



Atomic operations

With atomic instructions, the read/add/write becomes a single operation, and they happen one after the other (they serialise).



Atomic operations

There are several different atomic operations supported, for both arithmetic functions and also bitwise functions.

See: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions for a full list.

Atomics only support integer variable types, apart from `atomicAdd()` and `atomicExch()`, which also support half/float/double and float types respectively. Although this is dependent on hardware generation.

Atomic add

An example of `atomicAdd();`

```
type atomicAdd(type* address, type val);
```

`type` can be:

- `int, unsigned int, unsigned long long int`
- `__half, __half2, float, double`

The function reads the value located at `address` (in global or shared memory), lets call this “old”.

It then computes (“old” + “val”).

The result is placed back into memory at `address`.

The function returns the value “old”.

Atomic operations

Some other atomics that might be useful are listed on the right.

There is a subtraction version (like addition), but only for integers. But I think you can just do:

```
atomicAdd(type* address, type -val);
```

There is a maximum and minimum, these compare the value at address with val and store the maximum or minimum.

There is also a function for exchange, so the val is stored in place of the value at address.

```
type atomicSub(type* address, type val);

type atomicExch(type* address, type val);

type atomicMin(type* address, type val);

type atomicMax(type* address, type val);
```

Atomic compare-and-swap

One other useful atomic is compare-and-swap:

```
int atomicCAS(int * address,int compare,int val);
```

- If compare equals old value stored at address then val is stored instead.
- In either case, routine returns the value of old.
- Seems a bizarre routine at first sight, but can be very useful for atomic locks (more to come).

Atomic operations

A final few thoughts on atomics.

These are generally fast for variables held in shared memory, and only slightly slower for data in device memory (because operations are performed in L2).

However on **older hardware** atomics can be verrryyyyy slooooooowwww!!!

Even on newer hardware where atomics are quicker, I'd recommend using them only when you really need to.

https://www.flickr.com/photos/fatboyke/2668411239

Global atomic lock

Lets imagine a scenario where we need to do more work on a shared variable that the supplied atomics will allow us to do.

We can achieve this using atomics to “lock” a variable.

The code on the right is an attempt to do this.

```
// global variable: 0 unlocked, 1 locked
__device__ int lock=0;

__global__ void kernel(...) {
        ...

        if (threadIdx.x==0) {
                // set lock
                do {} while(atomicCAS(&lock,0,1));
                ...

                // free lock
                lock = 0;
        }
}
```

Global atomic lock

The previous code has a problem. This is that when a thread writes data to device memory the
order of completion is not guaranteed, so global writes may not have completed by the time the lock is unlocked.

This can be fixed using:

```
__threadfence();
```

```
// global variable: 0 unlocked, 1 locked
__device__ int lock=0;

__global__ void kernel(...) {
        ...

        if (threadIdx.x==0) {
                // set lock
                do {} while(atomicCAS(&lock,0,1));
                ...
                 __threadfence();

                // free lock
                lock = 0;
        }
}
```

Global atomic lock

```
__threadfence_block();
```

Wait until all global and shared memory writes are visible to:

- All threads in a block.

```
__threadfence();
```

Wait until all global and shared memory writes are visible to:

- All threads in a block.
- All threads, for global data.

Mikes notes on Practical 3 – Laplace finite difference solver

Jacobi iteration to solve discrete Laplace equation on a uniform grid:

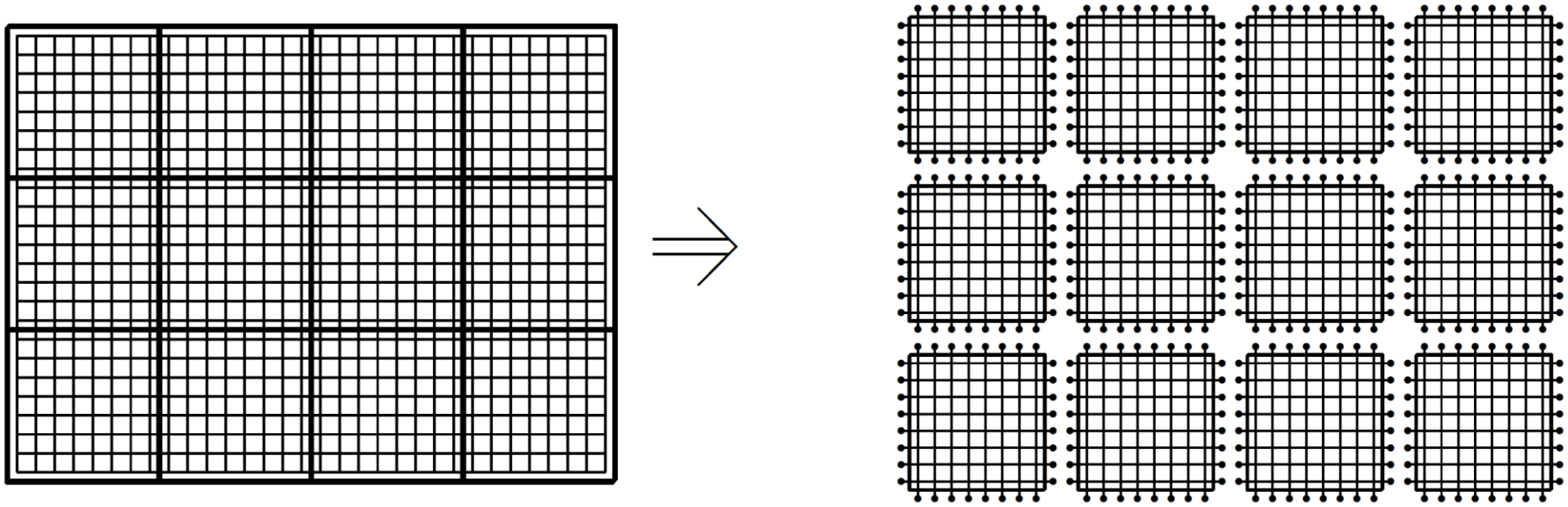
```
for (int j=0; j<J; j++) {
  for (int i=0; i<I; i++) {

    id = i + j*I;     // 1D memory location

    if (i==0 || i==I-1 || j==0 || j==J-1)
      u2[id] = u1[id];
    else
      u2[id] = 0.25*( u1[id-1] + u1[id+1]
                    + u1[id-I] + u1[id+I] );
  }
}
```

Practical 3 – Laplace finite difference solver

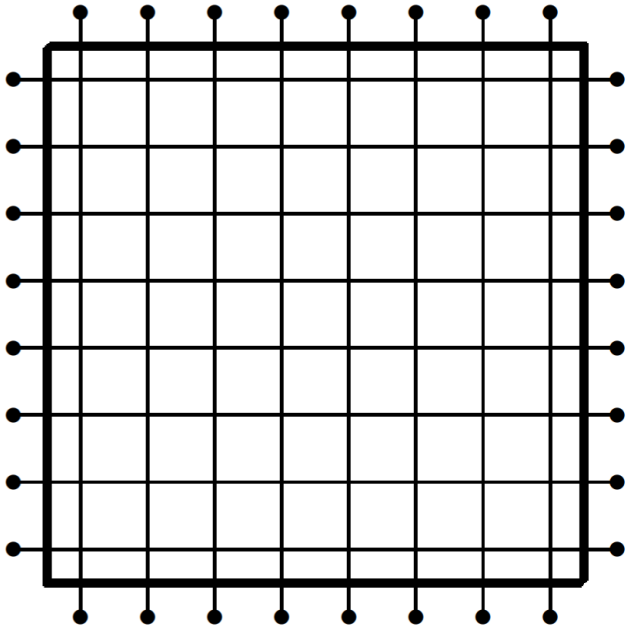
How do we tackle this with CUDA?

- each thread responsible for one grid point
- each block of threads responsible for a block of the grid
- conceptually very similar to data partitioning in MPI distributed-memory implementations, but much simpler
- (also similar to blocking techniques to squeeze the best cache performance out of CPUs)
- great example of usefulness of 2D blocks and 2D “grid”s

Practical 3 – Laplace finite difference solver



Practical 3 – Laplace finite difference solver



Each block of threads processes one of these grid blocks, reading in old values and computing new values.

Practical 3 – Laplace finite difference solver

```
__global__ void lap(int I, int J,
          const float* __restrict__ u1,
                float* __restrict__ u2) {

  int i  = threadIdx.x + blockIdx.x*blockDim.x;
  int j  = threadIdx.y + blockIdx.y*blockDim.y;
  int id = i + j*I;

  if (i==0 || i==I-1 || j==0 || j==J-1) {
    u2[id] = u1[id];   // Dirichlet b.c.'s
  }
  else {
    u2[id] = 0.25 * ( u1[id-1] + u1[id+1]
                    + u1[id-I] + u1[id+I] );
  }
}
```

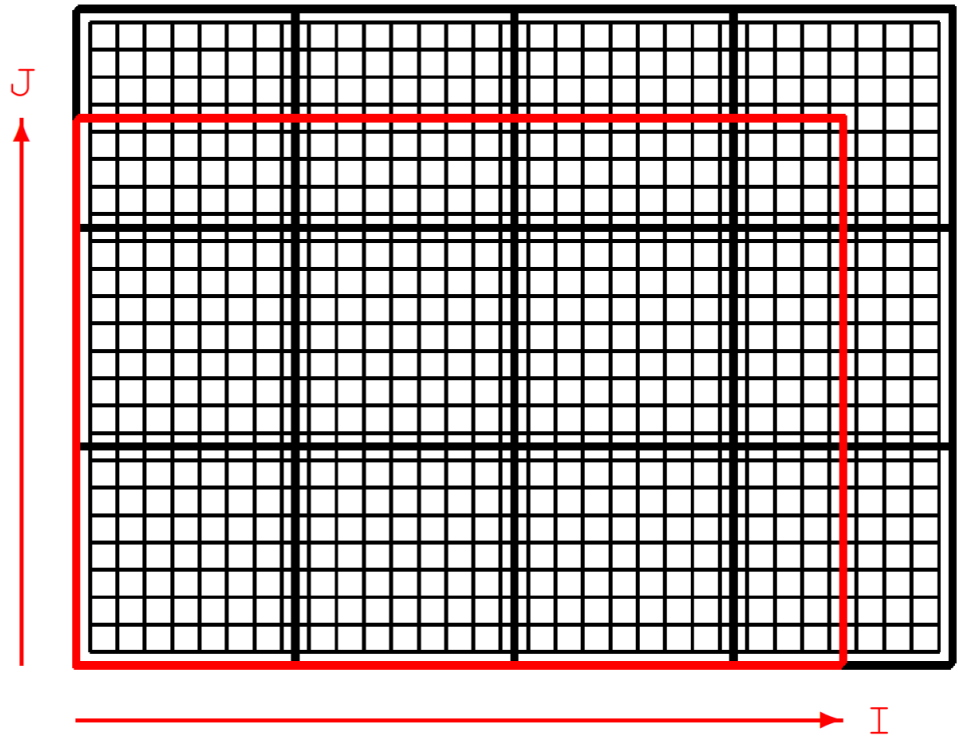
Practical 3 – Laplace finite difference solver

Assumptions made in the previous kernel are:

- `I` is a multiple of `blockDim.x`
- `J` is a multiple of `blockDim.y`
- hence grid breaks up perfectly into blocks

We can remove these assumptions by testing whether `i`, `j` are within grid

Practical 3 – Laplace finite difference solver



threads

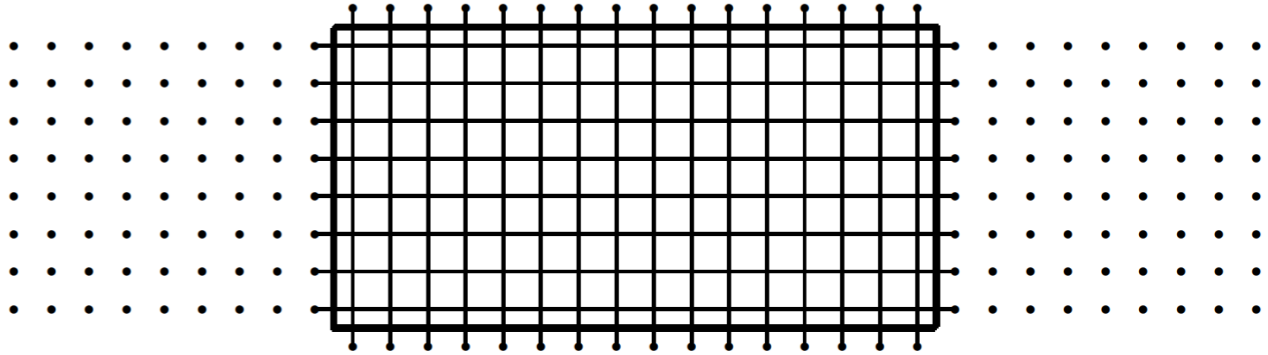
real grid

Practical 3 – Laplace finite difference solver

```
__global__ void lap(int I, int J,
          const float* __restrict__ u1,
                float* __restrict__ u2) {

  int i  = threadIdx.x + blockIdx.x*blockDim.x;
  int j  = threadIdx.y + blockIdx.y*blockDim.y;
  int id = i + j*I;

  if (i==0 || i==I-1 || j==0 || j==J-1) {
    u2[id] = u1[id];   // Dirichlet b.c.'s
  }
  else if (i<I && j<J) {
    u2[id] = 0.25f * ( u1[id-1] + u1[id+1]
                     + u1[id-I] + u1[id+I] );
  }
}
```

Practical 3 – Laplace finite difference solver

How does cache function in this application?



- if block size is a multiple of 32 in x-direction, then interior corresponds to set of complete cache lines
- “halo” points above and below are full cache lines too
- “halo” points on side are the problem – each one requires the loading of an entire cache line
- optimal block shape has aspect ratio of roughly 32:1 (or 8:1 if cache line is 32 bytes)

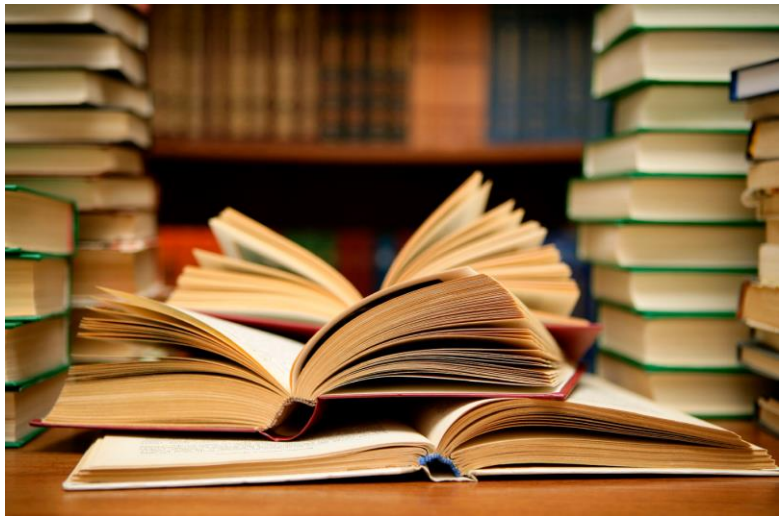
Practical 3 – 3D Laplace solver

- each thread does an entire line in z-direction
- x, y dimensions cut up into blocks in the same way as 2D application
- `laplace3d.cu` and `laplace3d_kernel.cu` follow same approach described above

- this used to give the fastest implementation, but a new version uses 3D thread blocks, with each thread responsible for just 1 grid point
- the new version has lots more integer operations, but is still faster (due to many more active threads?)

Key reading

CUDA Programming Guide, version 9.0:
Section 5.4.2: control flow and predicates
Section 5.4.3: synchronization
Appendix B.5: __threadfence() and variants
Appendix B.6: __syncthreads() and variants
Appendix B.12: atomic functions
Appendix B.13: warp voting
Appendix C: Cooperative Groups – this is new in CUDA 9.0 and may lead to changes/updates in some of the material in this lecture



https://www.flickr.com/photos/abee5/8314929977

What have we learnt?

In this lecture we have learnt about Warps and control flow.

We have looked at how warp divergence and cause a significant loss in performance.

We have looked at synchronisation on the device, between threads in a thread block and why this is important. We also looked at synchronisation on the host.

Finally we covered atomic operations on the GPU.