

# Lecture 3: control flow and synchronisation

Prof. Mike Giles

`mike.giles@maths.ox.ac.uk`

Oxford University Mathematical Institute

# Warp divergence

Threads are executed in warps of 32, with all threads in the warp executing the same instruction at the same time.

What happens if different threads in a warp need to do different things?

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

This is called *warp divergence* – CUDA will generate correct code to handle this, but to understand the performance you need to understand what CUDA does with it

# Warp divergence

This is not a new problem.

Old CRAY vector supercomputers had a logical merge vector instruction

$$z = p ? x : y;$$

which stored the relevant element of the input vectors  $x, y$  depending on the logical vector  $p$ , equivalent to

```
for (i=0; i<I; i++) {  
    if (p[i]) z[i] = x[i];  
    else      z[i] = y[i];  
}
```

# Warp divergence

Similarly, 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 previous example, all threads compute the logical predicate and two predicated instructions

```
    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:

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

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

# Warp divergence

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

```
p = ...
```

```
if (any(p)) {
```

```
p:    ...
```

```
p:    ...
```

```
}
```

```
if (any(!p)) {
```

```
!p:   ...
```

```
!p:   ...
```

```
}
```

# Warp divergence

Note:

- 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



# Warp divergence

In some cases, can determine at compile time that all threads in the warp must go the same way

e.g. if `case` is a run-time argument

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

In this case, there's no need to vote

# Warp divergence

Warp divergence can lead to a big loss of parallel efficiency – one of the first things I look out for in a new application.

In worst case, effectively lose factor  $32\times$  in performance if one thread needs expensive branch, while rest do nothing

Typical example: PDE application with boundary conditions

- if boundary conditions are cheap, loop over all nodes and branch as needed for boundary conditions
- if boundary conditions are expensive, use two kernels: first for interior points, second for boundary points

# Warp divergence

Another example: processing a long list of elements where, depending on run-time values, a few require very expensive processing

GPU implementation:

- first process list to build two sub-lists of “simple” and “expensive” elements
- then process two sub-lists separately

Note: none of this is new – this is what we did 35 years ago on CRAY and Thinking Machines systems.

What’s important is to understand hardware behaviour and design your algorithms / implementation accordingly

# Synchronisation

Already introduced `__syncthreads()` ; which forms a barrier – all threads wait until every one has reached this point.

When writing conditional code, must be careful to make sure that all threads do reach the `__syncthreads()` ;

Otherwise, can end up in *deadlock*

# Typical application

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

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

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

# 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've not used these, and don't currently see a need for them

# Warp voting

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  $n^{th}$  predicate

Again, I've never used these

# Atomic operations

Occasionally, an application needs threads to update a counter in shared memory.

```
__shared__ int count;
```

```
...
```

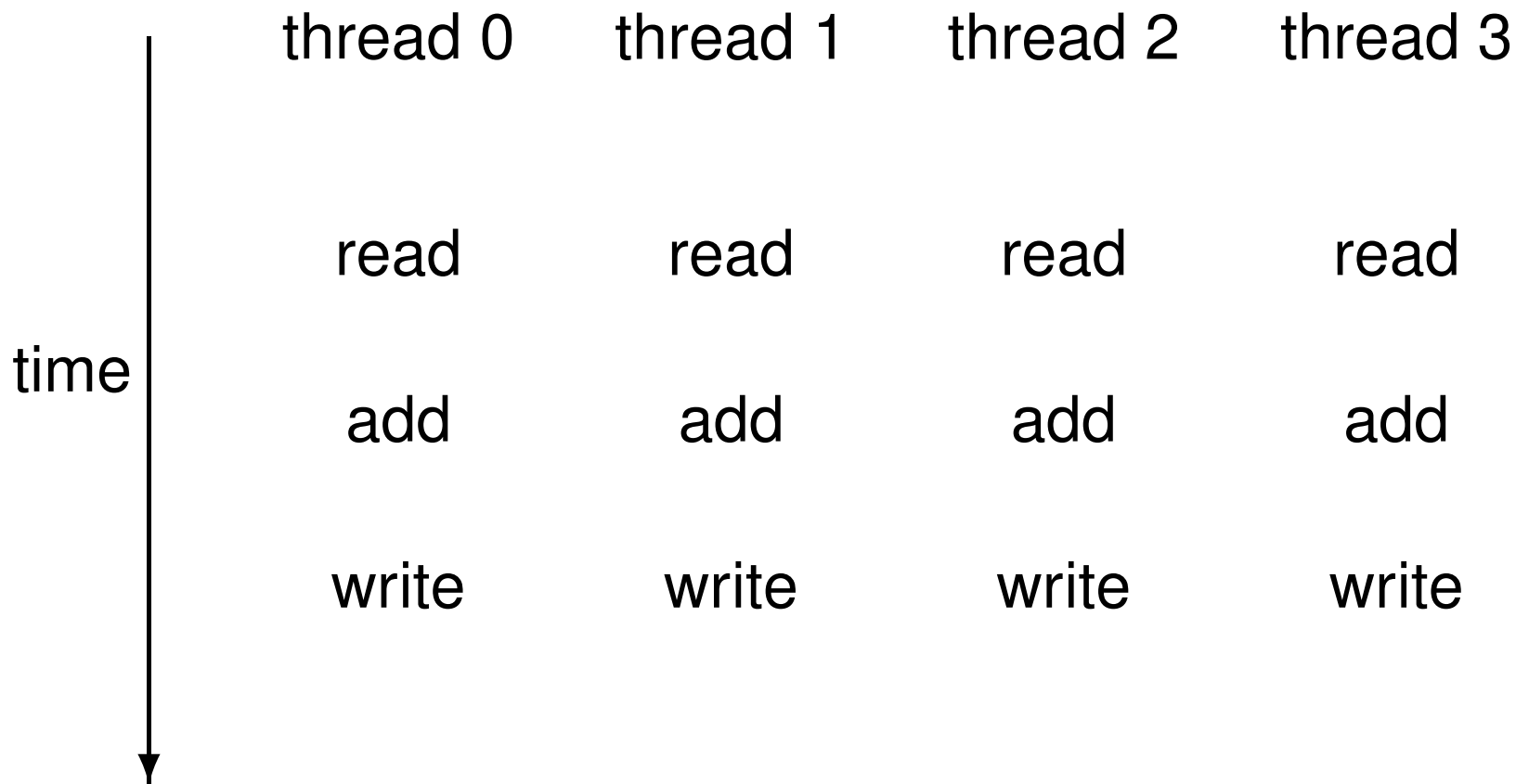
```
if ( ... ) count++;
```

In this case, there is a problem if two (or more) threads try to do it at the same time



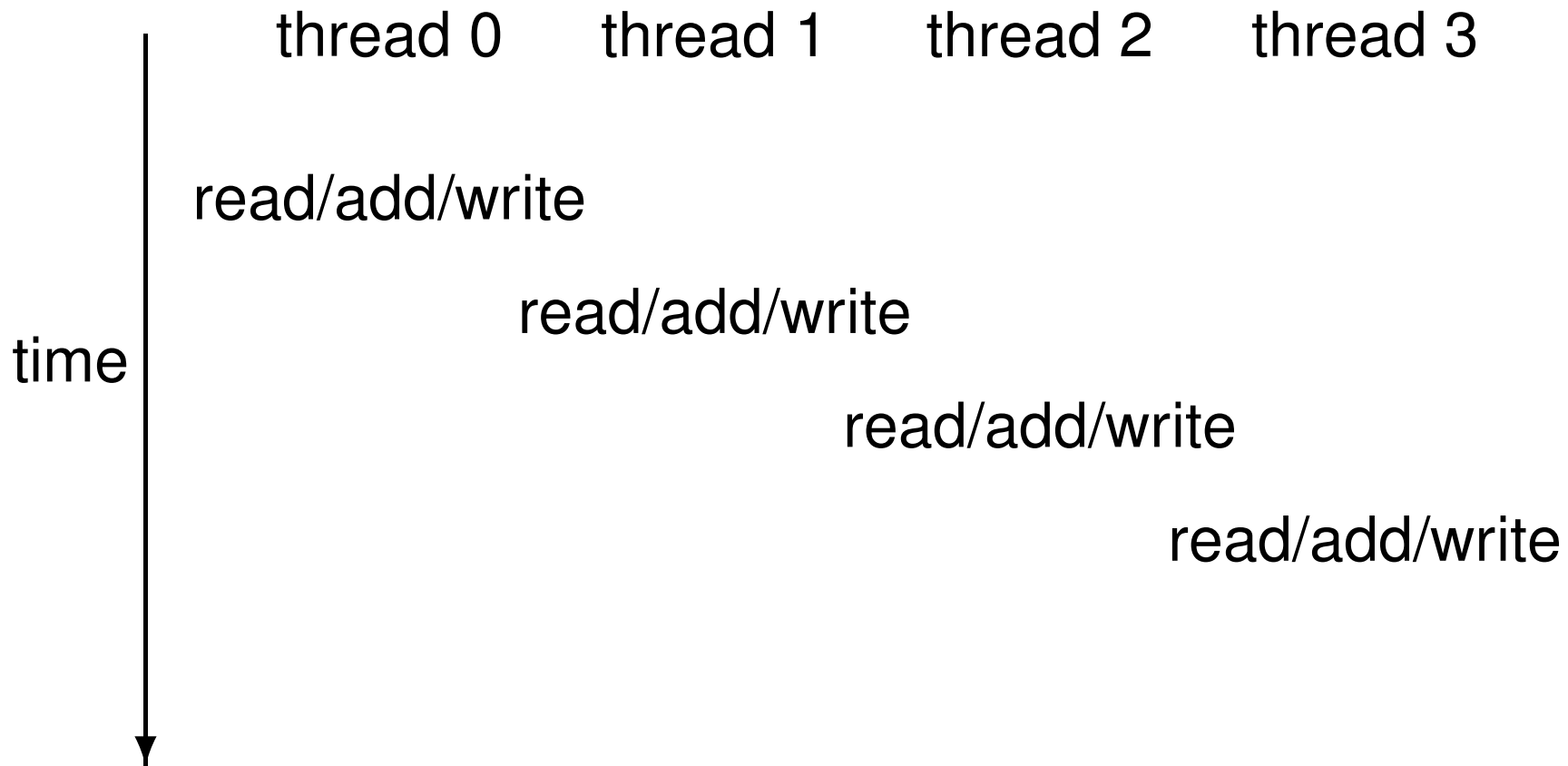
# Atomic operations

Using standard instructions, multiple threads in the same warp will only update it once.



# Atomic operations

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



# Atomic operations

Several different atomic operations are supported:

- **addition / subtraction**  
`atomicAdd, atomicSub`
- **minimum / maximum**  
`atomicMin, atomicMax`
- **increment / decrement**  
`atomicInc, atomicDec`
- **exchange / compare-and-swap**  
`atomicExch, atomicCAS`
- **bitwise AND / OR / XOR**  
`atomicAnd, atomicOr, atomicXor`

Fast for variables in shared memory, only slightly slower for variables in device global memory (operations performed in L2 cache)

# Atomic operations

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

# Global atomic lock

```
// 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

Problem: 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

```
__global__ void kernel(...) {  
    ...  
  
    if (threadIdx.x==0) {  
        do {} while(atomicCAS(&lock, 0, 1));  
        ...  
        __threadfence(); // wait for writes to finish  
  
        // free lock  
        lock = 0;  
    }  
}
```

# \_\_threadfence

- `__threadfence_block();`

wait until all global and shared memory writes are visible to

- all threads in block

- `__threadfence();`

wait until all global and shared memory writes are visible to

- all threads in block
- all threads, for global data

# Summary

- lots of esoteric capabilities – don't worry about most of them
- essential to understand warp divergence – can have a very big impact on performance
- `__syncthreads()` is vital – will see another use of it in next lecture
- the rest can be ignored until you have a critical need – then read the documentation carefully and look for relevant NVIDIA sample codes



# Key reading

## CUDA C++ Programming Guide:

- Section 5.4.2: control flow and predicates
- Section 5.4.3: synchronization
- Section 7.5: `__threadfence()` and variants
- Section 7.6: `__syncthreads()` and variants
- Section 7.14: atomic functions
- Section 7.19: warp voting

# 2D Laplace 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

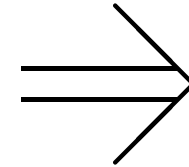
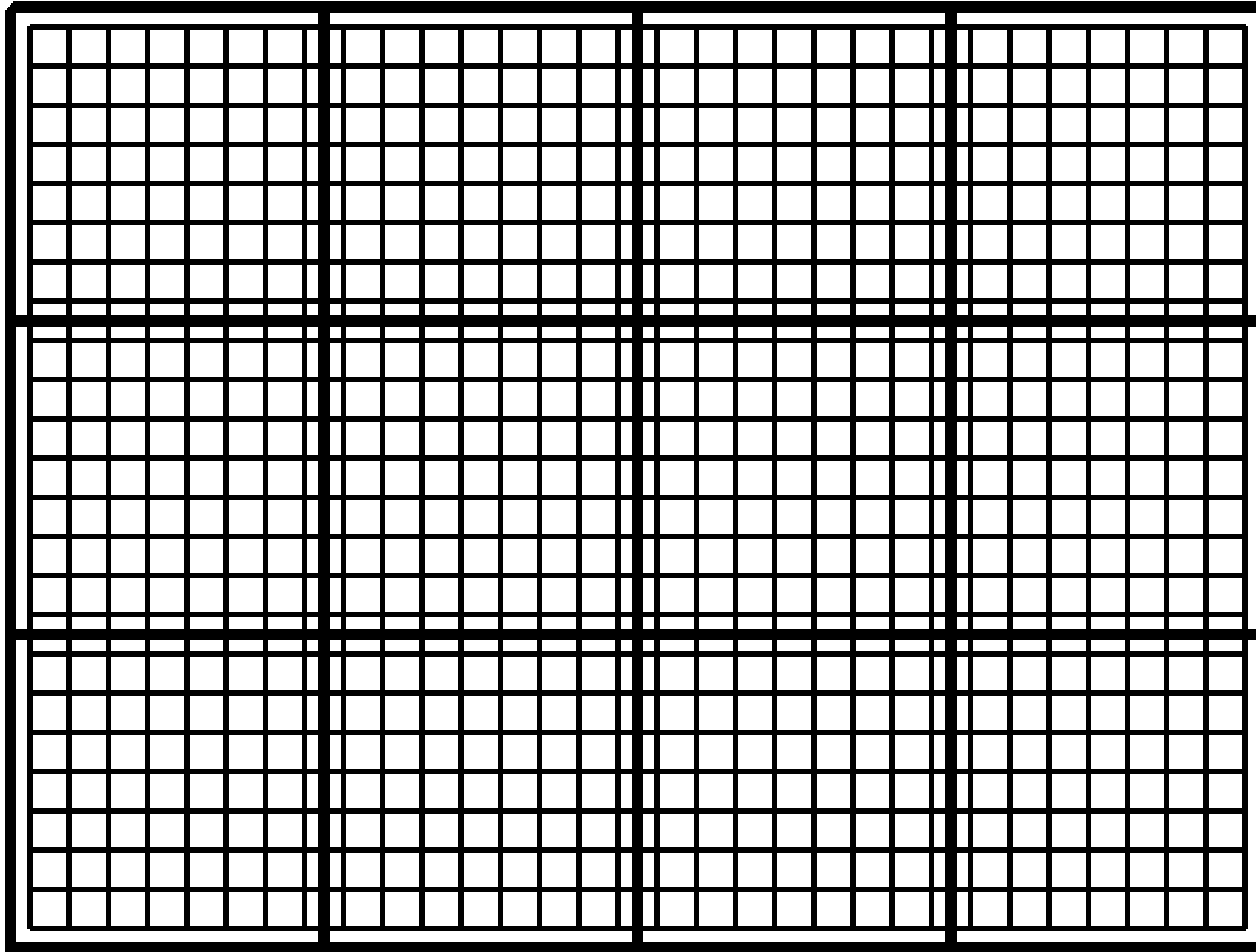
        if (i==0 || i==I-1 || j==0 || j==J-1)
            u2[id] = u1[id];
        else
            u2[id] = 0.25*( u1[id-1] + u1[id+1]
                           + u1[id-I] + u1[id+I] );
    }
}
```

# 2D Laplace solver

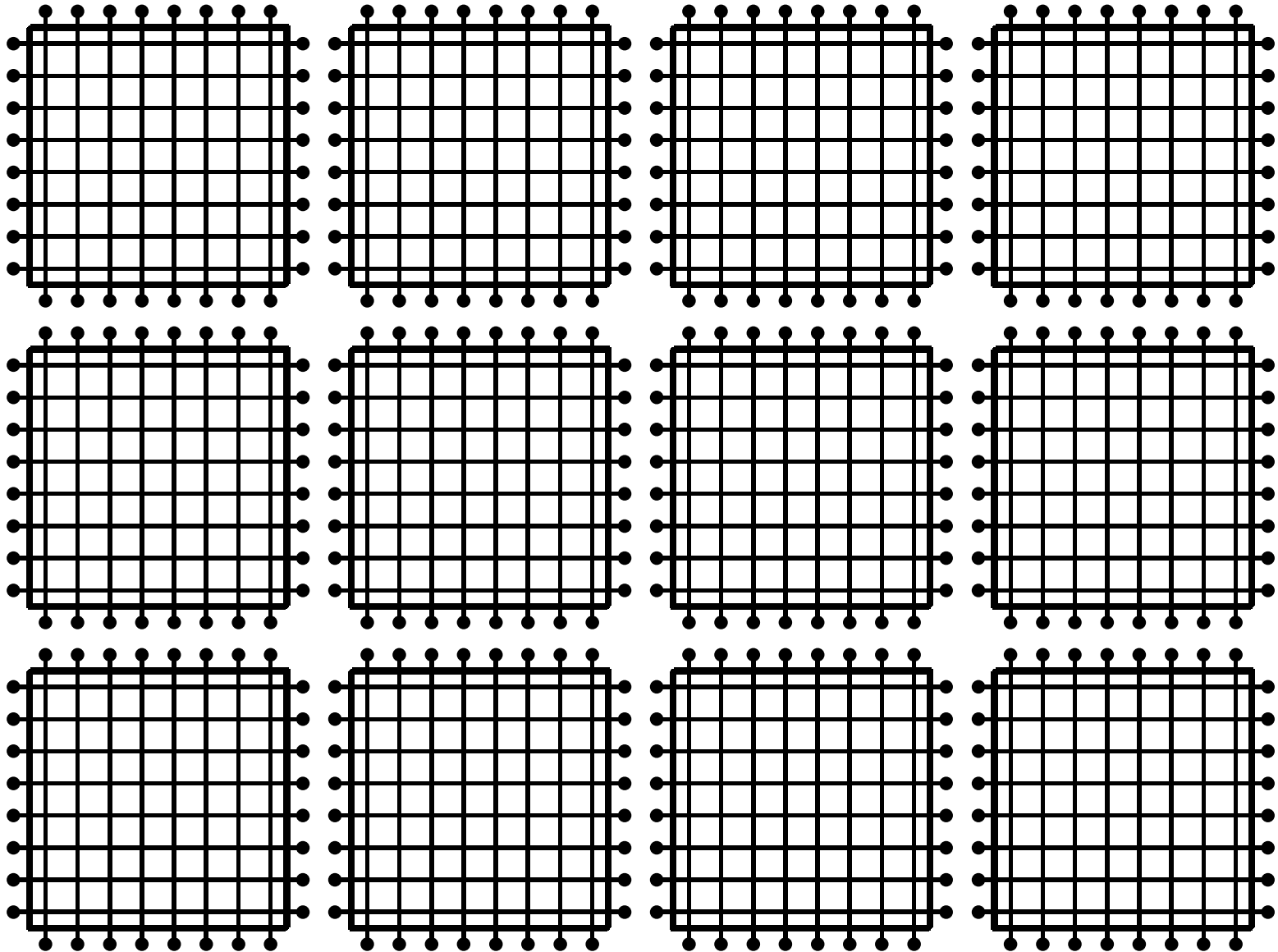
How do we tackle this with CUDA?

- 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

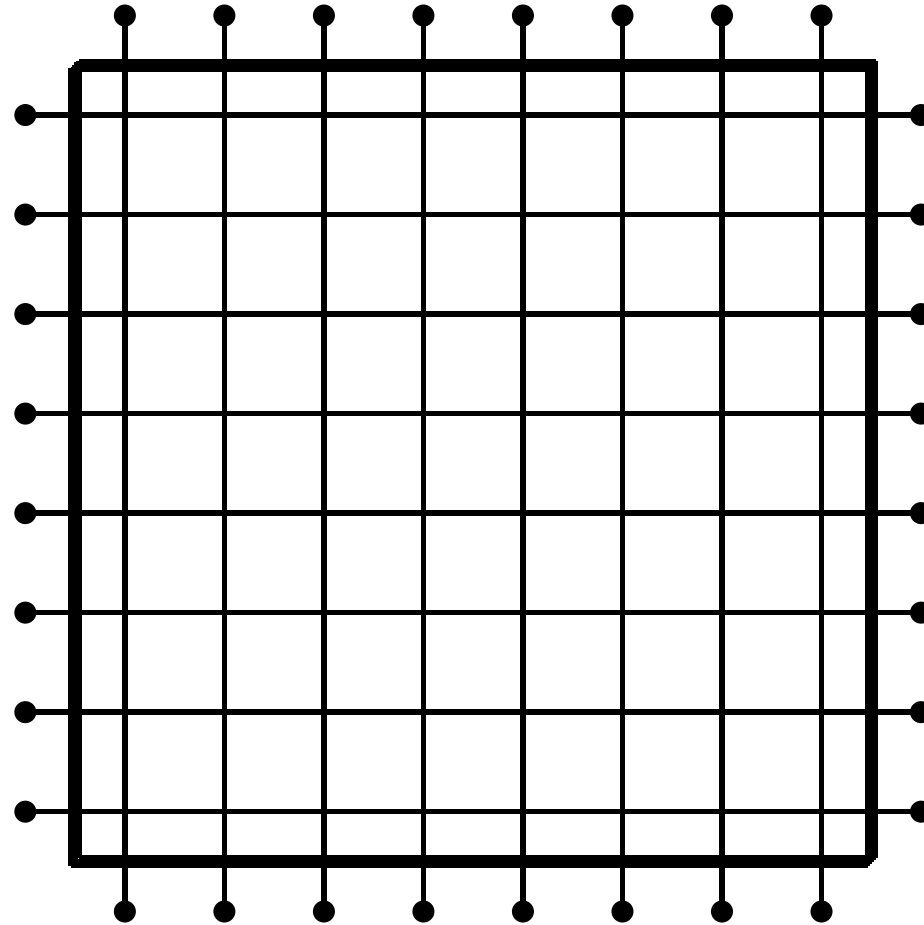
# 2D Laplace solver



# 2D Laplace solver



# 2D Laplace solver



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

# 2D Laplace 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] );
    }
}
```

# 2D Laplace solver

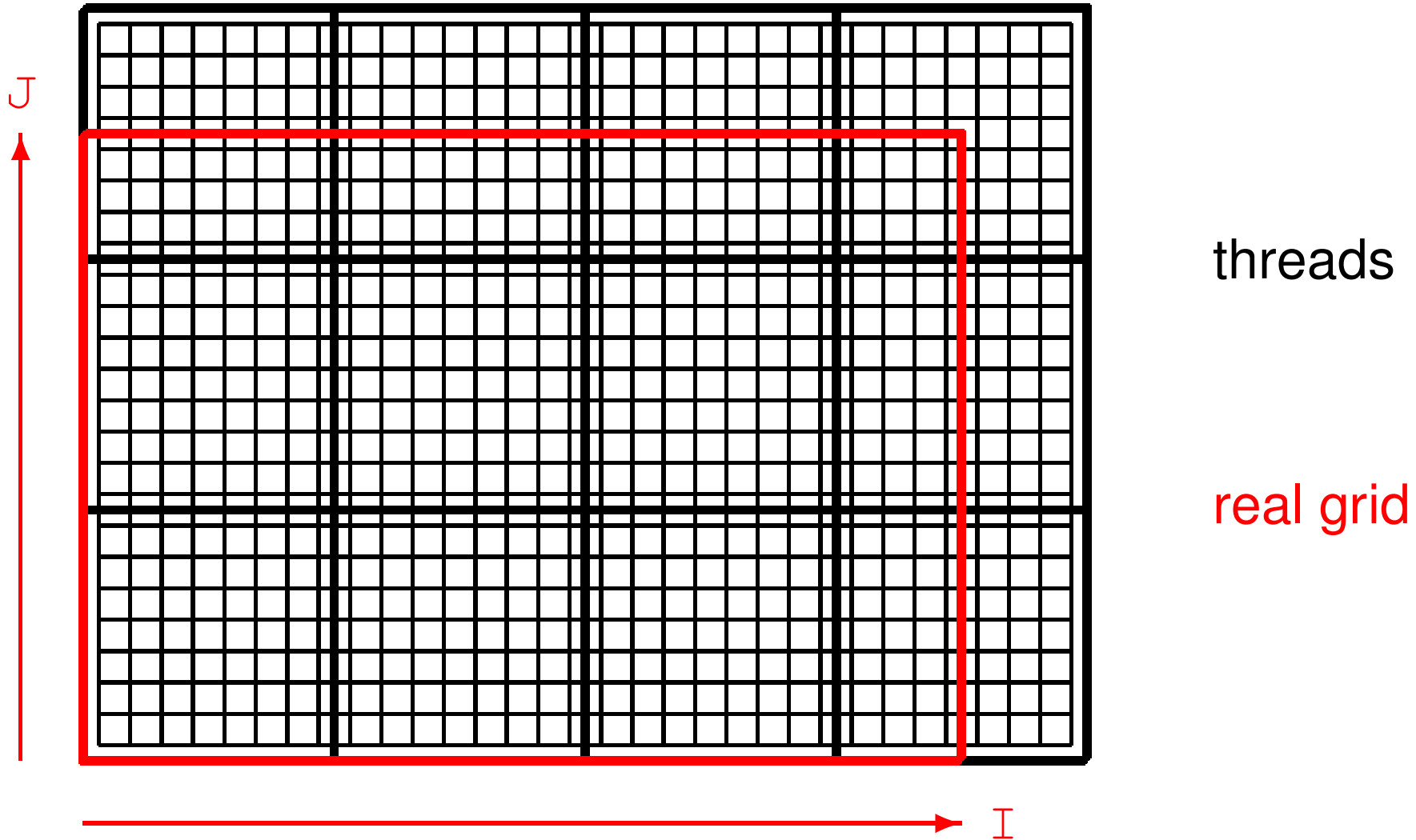
Assumptions:

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

Can remove these assumptions by testing whether  $i, j$  are within grid



# 2D Laplace solver



# 2D Laplace solver

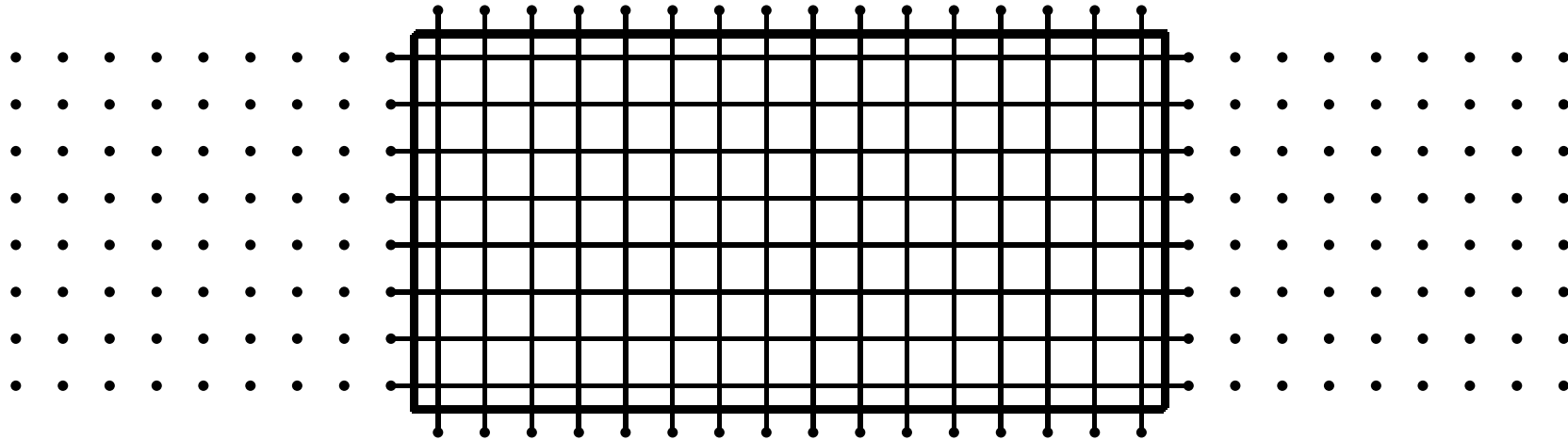
```
__global__ void lap(int I, int J,
                   const float* __restrict__ u1,
                   float* __restrict__ u2) {

    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] );
    }
}
```

# 2D Laplace 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 8:1 if cache line is 32 bytes == 8 floats

# 3D Laplace solver

- practical 3
- 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, perhaps due to many more active threads – in either case the application is probably bandwidth-limited