CME 213

SPRING 2012-2013

Eric Darve

Homework 3

PDE SOLVER USING CUDA

- We want to solve the following PDE:

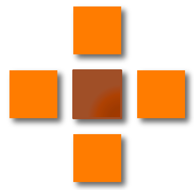
$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

- Use a finite-difference scheme for the spatial operator and the Euler scheme for the time integration.
- We get an update equation of the form:

$$T^{n+1} = AT^n$$

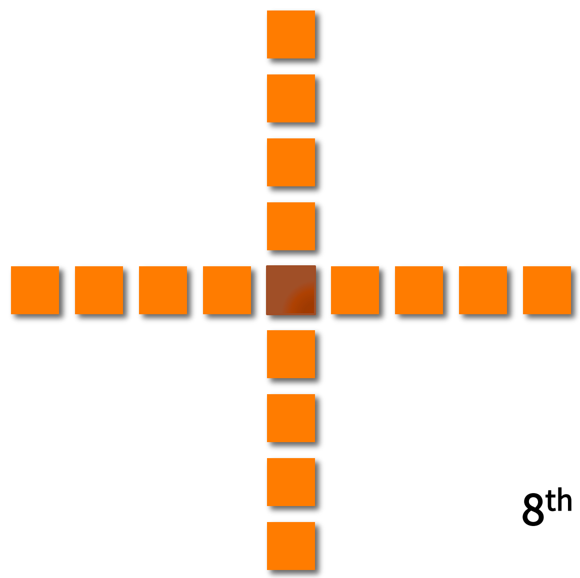
- Different stencils can be used depending on the order.

STENCILS



2nd order stencil

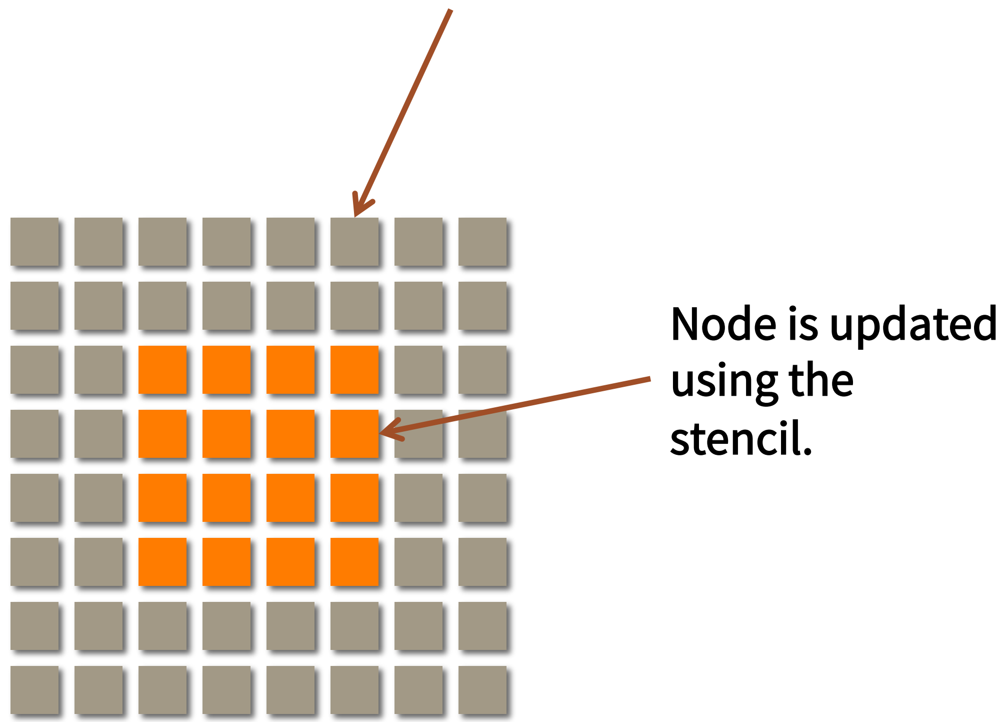
4th order stencil



8th order stencil

THE GRID

4th order stencil



Boundary condition

- The stencil can only be applied on the inside.
- Near the boundary a special stencil needs to be used (one-sided).
- To simplify the homework, we considered a case where the analytical solution is known.
- Nodes on the boundary are simply updated using the exact solution.
- You don't have to worry about that.
- Goal of the homework: implement a CUDA routine to update nodes inside the domain using the basic finite-difference centered stencil.

MESH GRID

- We have an array the contains the values of T at mesh nodes.

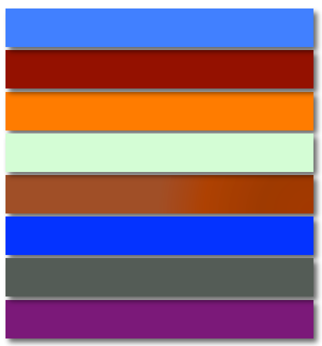
```
class Grid {
  public:
    std::vector<float> hGrid_;
    float *             dGrid_;
    ...
};
```

hGrid_ grid on the host

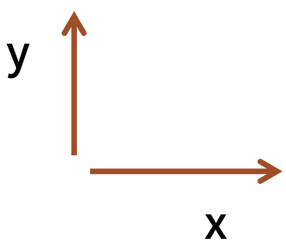
dGrid_ grid on the GPU

Memory layout

We use a simple 1D array to store grid information:



Mesh nodes



Layout in memory

PERFORMANCE GUIDELINES

1. How many flops do we need to perform?
2. How many words do we need to read from / write to memory?

- We need to understand which one is the limiting factor.
- Then we can design a reasonable algorithm.

- Take the order 2 stencil:

1. How many flops?
2. How many words?

Answers

1. How many flops?
   10 add/mul
2. How many words?
   - Read: 5
   - Write: 1
   - Total: 6

- Operations are very fast on the hardware.
- Threads are mostly going to wait on memory for this problem.
- How can we address this?

Optimizing memory access

There are two main ideas:

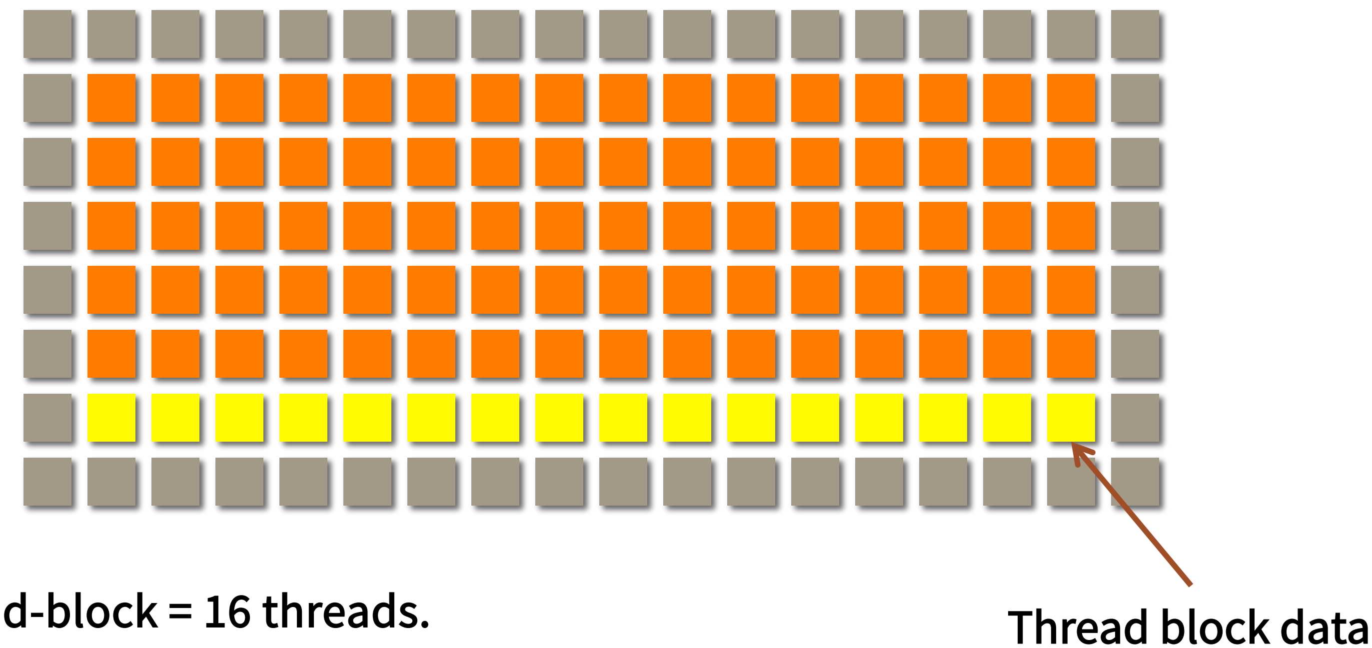
1. Use cache or shared memory:
   Once a data is read from memory and is in cache/shared memory, use it as much as possible, that is use it for several different stencils that need that point
2. Memory accesses should be coalesced: threads in a warp need to read from contiguous memory locations.

We are going to see how this works for this problem.

THREAD-BLOCK

The first concern is: what is a thread block going to do?

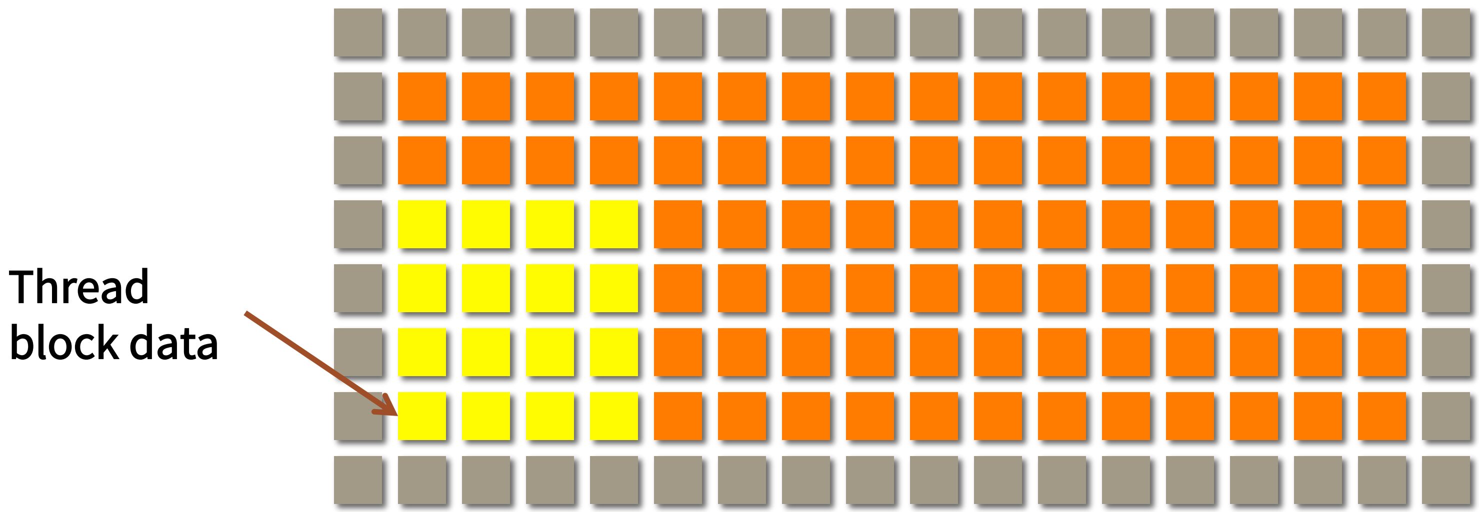
Idea 1: work on a line of the mesh



- Thread-block = 16 threads.
- Reads: 16*3+2. Writes: 16. Total = 66
- Flops: 10*16 = 160
- Ratio: flops/words = 2.4

BETTER SHAPE

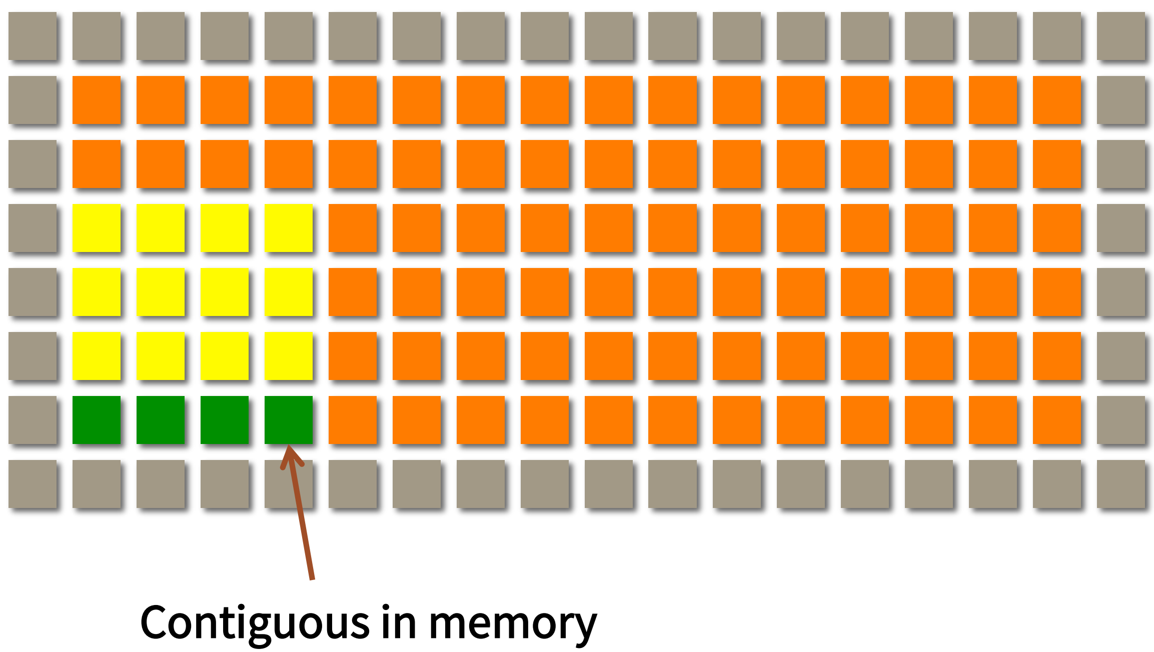
Idea 2: you can convince yourself that the optimal shape is a square.



- Thread-block = 16 threads.
- Reads: 16+16. Writes: 16. Total = 48
- Flops: 10*16 = 160
- Ratio: flops/words = 3.3

COALESCED MEMORY READS

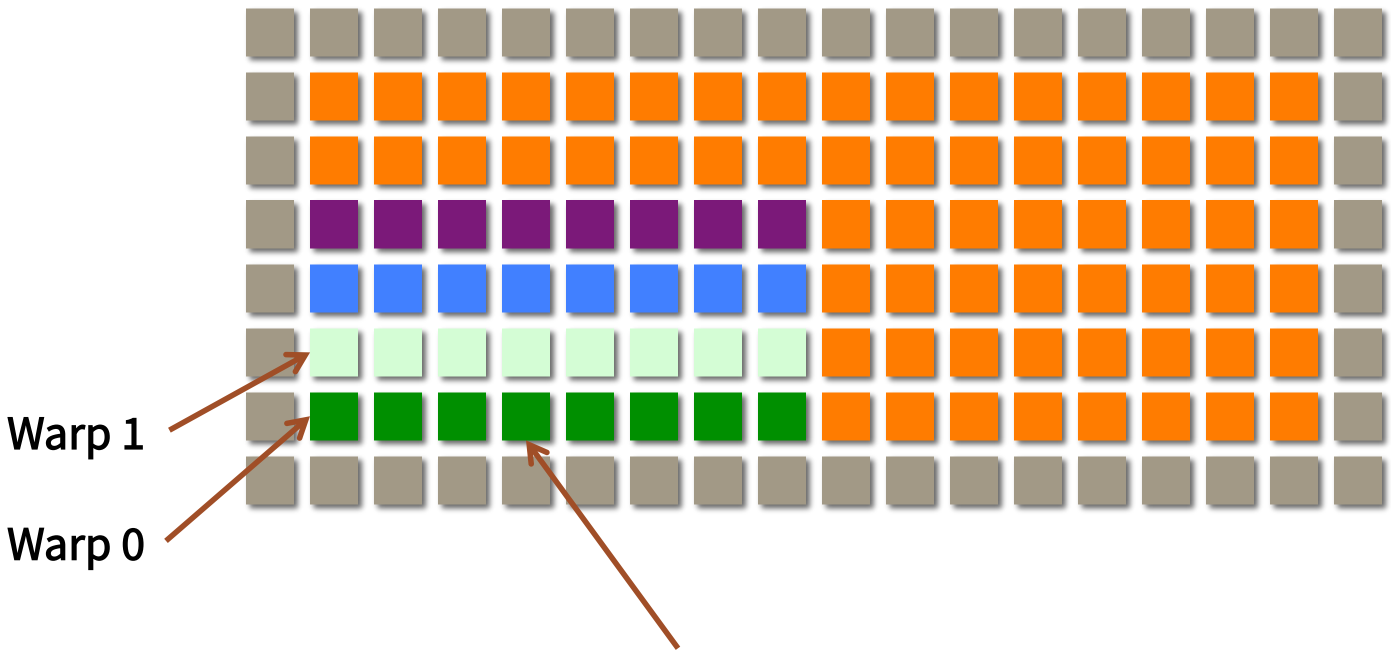
- We saw that the hardware does best at reading 32 floats (= 128 bytes) contiguous in memory for each warp.



- Only mesh nodes along x are contiguous.
- This size must therefore be a multiple of 32.
- A warp must work on a chunk aligned along x.

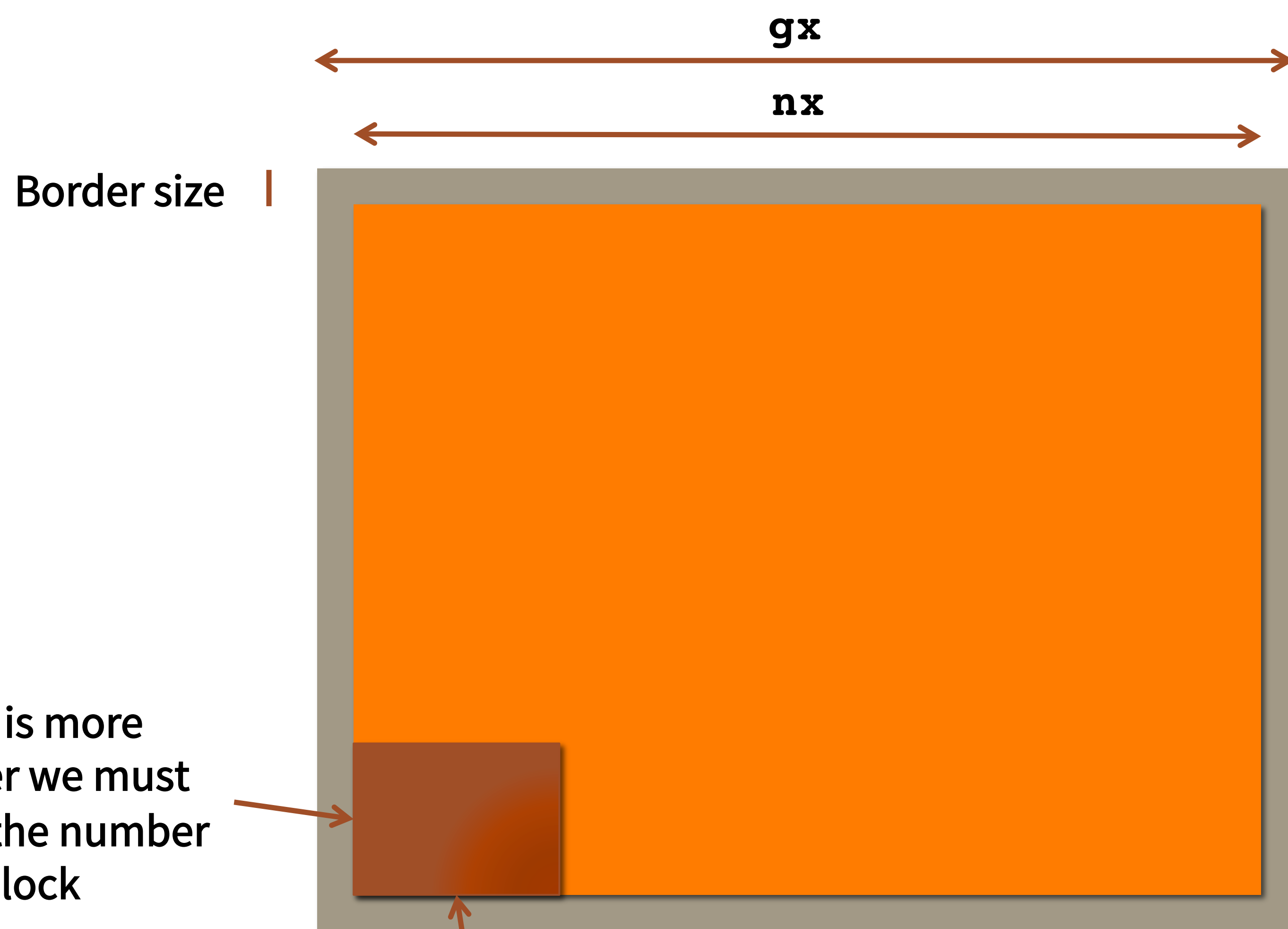
Warp assignment

Warp 0 = tid 0 to 31
Warp 1 = tid 32 to 63
Warp 2 = tid 64 to 95
Warp 3 = tid 96 to 128



Assume a warp size of 8: this is a perfect read and write to memory

CHOOSE YOUR BLOCK SIZE



That dimension is more flexible; however we must be careful with the number of threads in a block

Thread-block: x dimension should be a multiple of 32

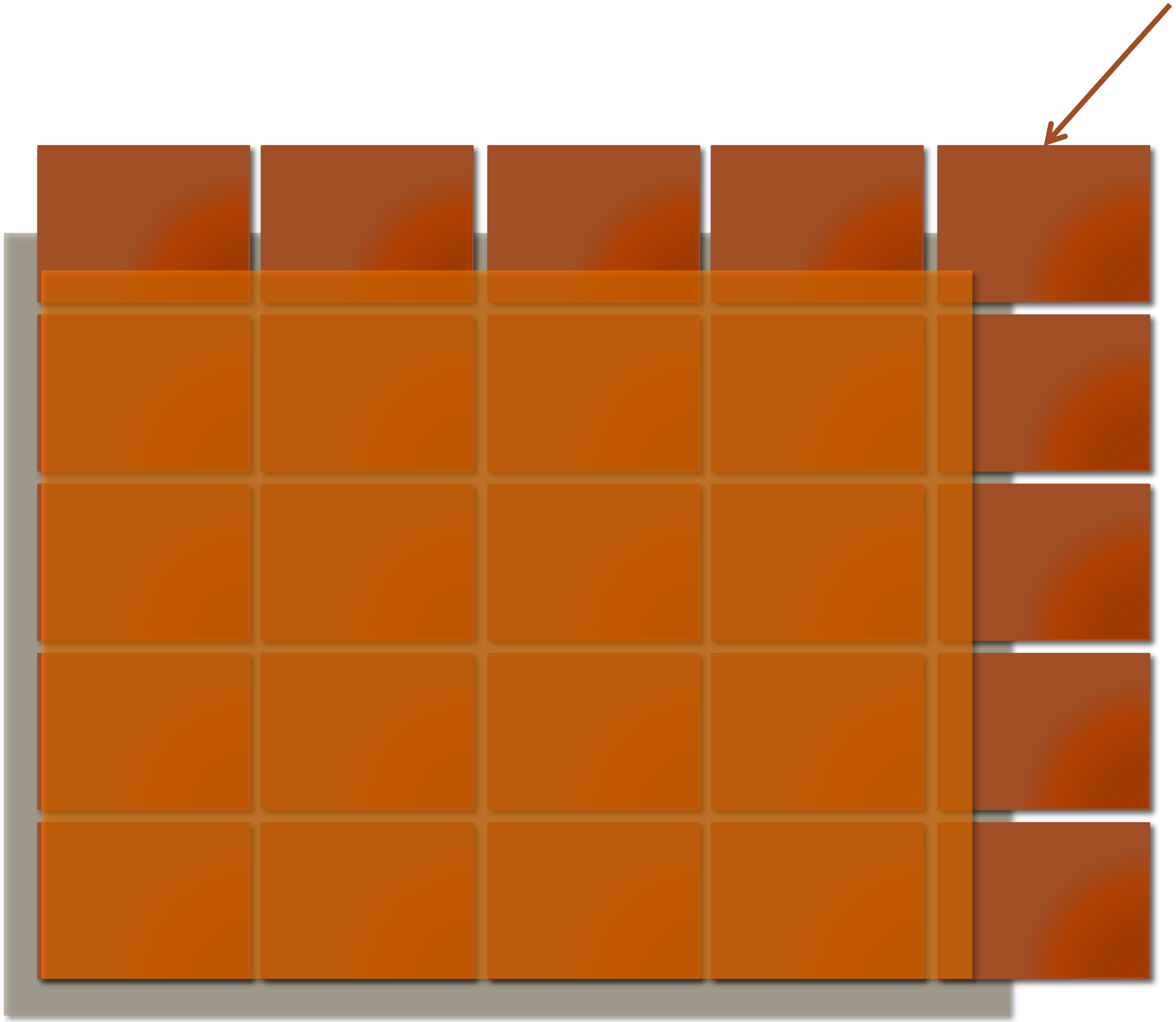
GOOD DIMENSIONS

Guidelines:

- `blockDim.x` multiple of 32
- Total number of threads should be approximately 192.
- Find `blockDim.y`

CHECK YOUR BOUNDS

- Use an if statement to check whether the thread is inside or not
- If not; return.



Algorithm 1: global memory

- This is the first algorithm
- Choose a size along x and y.
- Each thread updates 1 mesh point.
- Test to make sure the thread is inside the domain.

DOMAIN SIZE

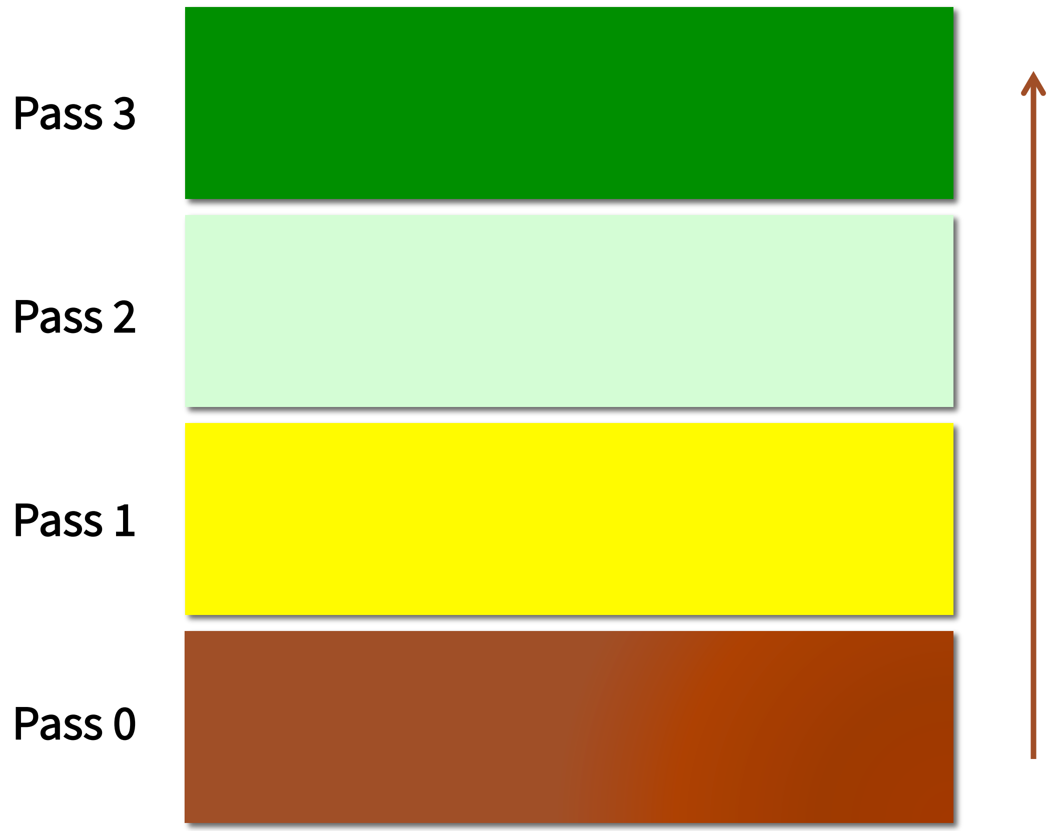
- You will see in the first implementation that the domain that a thread block is working on is shaped like a rectangle:

- It’s better to have a square as we saw.
- But we are limited by the number of threads we can have in a block.
- ??????

SOLUTION

- We can ask threads to process multiple elements:

- Each thread loops multiple times until the whole block has been processed.
- Number of passes = **`numYPerStep`**



ALGORITHM 2

- Choose a size along x and y.
- Each thread updates multiple mesh points, looping along the y direction.
- Test to make sure the thread is inside the domain. This is a bit more difficult this time because of the loop.

SHARED MEMORY

- Instead on relying on the cache, we can use shared memory.
- Two step process:
  1. Load in shared memory
  2. Apply stencil to nodes inside

Load in shared memory

Update inside nodes

ALGORITHM 3

- This one is optional because of the extra difficulty.
- If you were able to easily do the first two algorithms, try out this one for extra bonus points (total score is still capped at 100 though).
- Don't feel obligated to implement this one.

- Use a loop along y as in algorithm 2.
- Step 1: all threads load data in shared memory
- Step 2: threads inside the domain apply the stencil and write to the output array.

- Not hard in principle but you have to carefully track all the indices.

EXAMPLE OF OUTPUT

- I ran a test case to give you an idea of what to expect.
- Mesh size: nx = 2048, ny = 2048.
- Number of time steps: 10
- Sample output from code:

```
[darve@node020 PA3]$ ./heat -g -b -s
Order: 8
                     time (ms)     GBytes/sec
            CPU        941.884        3.20623
         Global        12.4968        241.653
The L2 norm of the reference solution is: 3.1439e+05
The LInf norm of the relative error is:    1.04e-06
The relative L2 norm error is:              1.4e-07

          Block        12.1455        248.643
The L2 norm of the reference solution is: 3.1439e+05
The LInf norm of the relative error is:    1.04e-06
The relative L2 norm error is:              1.4e-07

         Shared        7.60986         396.84
The L2 norm of the reference solution is: 3.1439e+05
The LInf norm of the relative error is:    1.04e-06
The relative L2 norm error is:              1.4e-07
```

Notes

- To run the code you can use options –g –b or –s. This determines which GPU algorithm can run. g=global, b=block (algo. 2), s=shared
- See sample code for details.
- Read the CPU code for reference.
- You may get slightly different numbers from me but it should be "close."
- There is a discrepancy between CPU and GPU because of roundoff errors.
- However all GPU kernels should produce exactly the same output. Check how the errors in the previous slide are all exactly equal.
- If you compile with the –G option, the GPU code will match the CPU code exactly. However there is a performance hit. Use it when debugging only.

SAMPLE OUTPUT WITH –G COMPILATION

```
[darve@node020 PA3]$ ./heat -g -b -s
Order: 8
                    time (ms)     GBytes/sec
            CPU       929.689        3.24829
         Global       31.9415        94.5447
The L2 norm of the reference solution is: 3.1439e+05
The LInf norm of the relative error is:           0
The relative L2 norm error is:                    0

          Block       30.0462        100.509
The L2 norm of the reference solution is: 3.1439e+05
The LInf norm of the relative error is:           0
The relative L2 norm error is:                    0

         Shared       40.9707        73.7088
The L2 norm of the reference solution is: 3.1439e+05
The LInf norm of the relative error is:           0
The relative L2 norm error is:                    0
```