CME 213

Spring 2017

Eric Darve

CUDA summary

Warp, block, grid

Memory access:

- Global memory: coalesced access = warp requests and uses a full 128-byte memory segment
- Potential issues: misaligned access, strided access
- Shared memory:
  - Bandwidth: 1 4-byte word every 2 cycles
  - Bank conflict if threads access different memory locations in the same bank
  - Bank conflict: access becomes serialized
  - Successive 4-byte words are assigned to successive banks

CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	2.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	14	
Shared Memory Per Block (bytes)	4224	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1536	(Help)
Active Warps per Multiprocessor	48	
Active Thread Blocks per Multiprocessor	6	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	2.0
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	8
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	32768
Max Registers per Thread Block	32768
Max Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Max Shared Memory per Block	49152
Register allocation unit size	64
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	2

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	72	9
Shared Memory (Bytes)	4224	49152	11

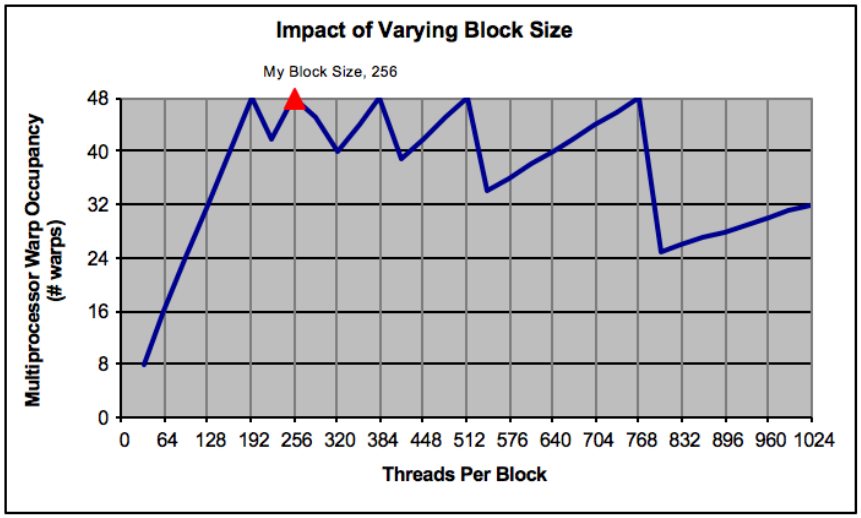
Note: SM is an abbreviation for (Streaming) Multiprocessor

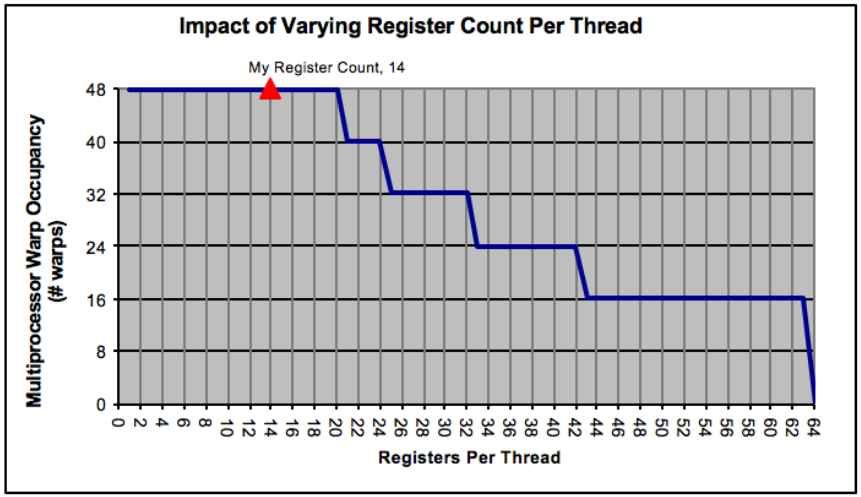
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	8	48
Limited by Registers per Multiprocessor	9		
Limited by Shared Memory per Multiprocessor	11		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 48
Occupancy = 48 / 48 = 100%

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.





How to get kernel information

```
[darve@certainty-a code]$ nvcc --ptxas-options=-v -O3 -arch=sm_20 transpose.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z17simpleTranspose2DPiS_ii' for 'sm_20'
ptxas info    : Function properties for _Z17simpleTranspose2DPiS_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 8 registers, 56 bytes cmem[0]
ptxas info    : Compiling entry function '_Z15simpleTransposePiS_ii' for 'sm_20'
ptxas info    : Function properties for _Z15simpleTransposePiS_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 8 registers, 56 bytes cmem[0]
ptxas info    : Compiling entry function '_Z13fastTransposeILi8EEvPiS0_ii' for 'sm_20'
ptxas info    : Function properties for _Z13fastTransposeILi8EEvPiS0_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 14 registers, 4224 bytes smem, 56 bytes cmem[0]
```

```
$ nvcc --ptxas-options=-v
```

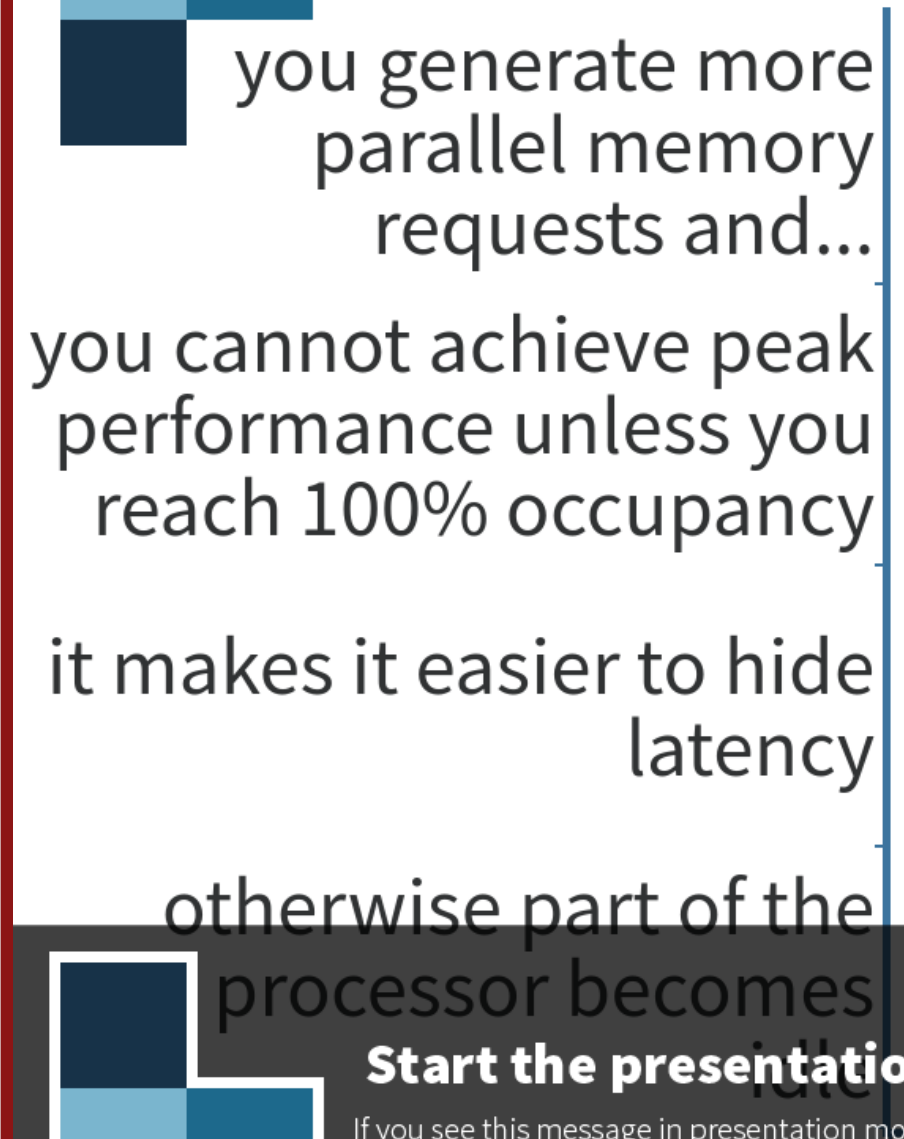
How to choose the number of warps per block?

- With our implementation, the number of warps must divide the L1 cache line size of 32.
- Generally, you pick the smallest number of warps that gives you the highest occupancy.
- Here that number is 8 warps = 256 threads / block.

```
for(int i = 0; i < warp_size / num_warps; ++i) {
  int gc = bc * warp_size + i * num_warps + warp_id;
  array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
}
```

4 iterations required to complete read

It's best to maximize occupancy because



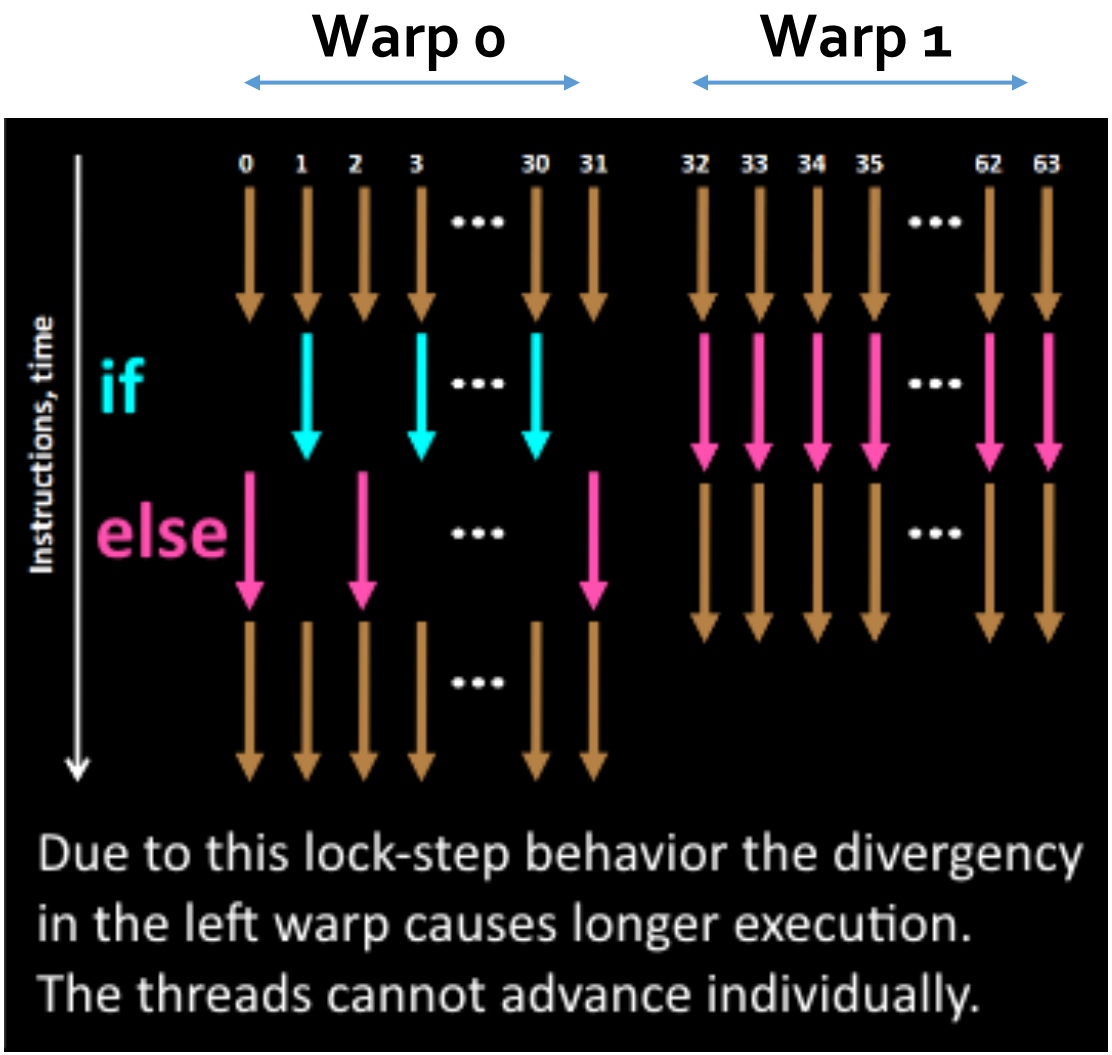
Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

Branch divergence

- Should be avoided at all cost.
- This leads to idle threads.


Warp 0
Warp 1
0 1 2 3 ... 30 31
32 33 34 35 ... 62 63
Instructions, time
if
else
Due to this lock-step behavior the divergency
in the left warp causes longer execution.
The threads cannot advance individually.

Branch occurrences

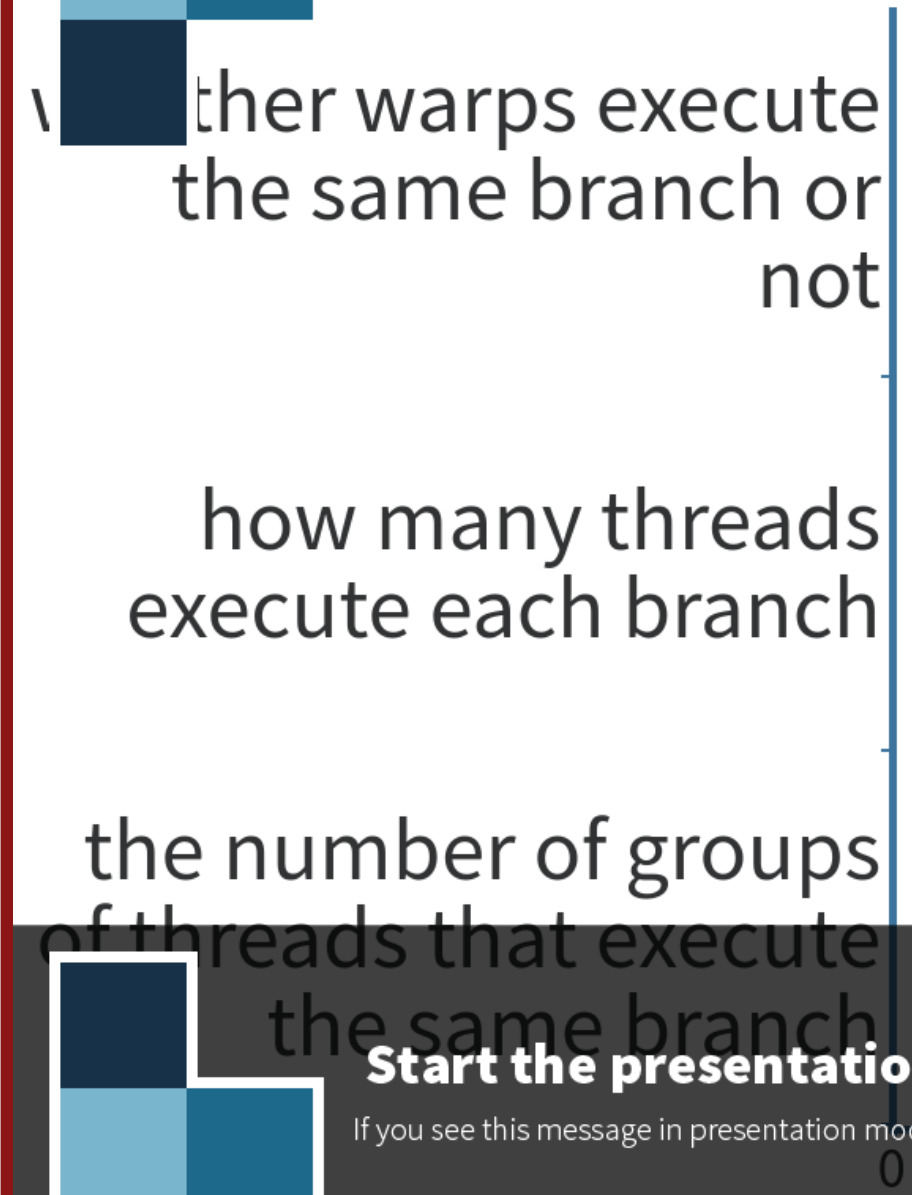
- `If/while` statements
- `For loop` of varying lengths, e.g., processing an unstructured grid
- Branch divergence is an issue only for threads belonging to the same warp.
- If warps do different things, there is no performance impact.

```
__global__ void branch_thread(float* out) {
    int tid = threadIdx.x;

    if(tid%2 == 0) {
        ...;
    } else {
        ...;
    }
}

__global__ void branch_warp(float* out) {
    int wid = threadIdx.x/32;

    if(wid%2 == 0) {
        ...;
    } else {
        ...;
    }
}
```

Divergence!

No divergence!

erformance when executing a branch depen



Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

Homework 3

PDE solver using CUDA

- We want to solve the following PDE:

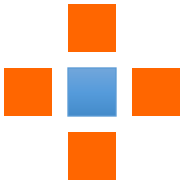
$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

- 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:

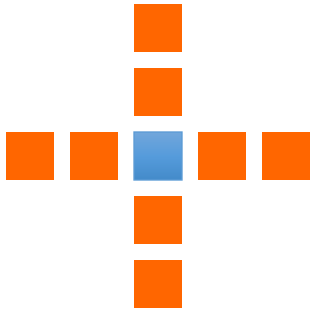
$$T^{n+1} = AT^n$$

- Different stencils can be used depending on the order.

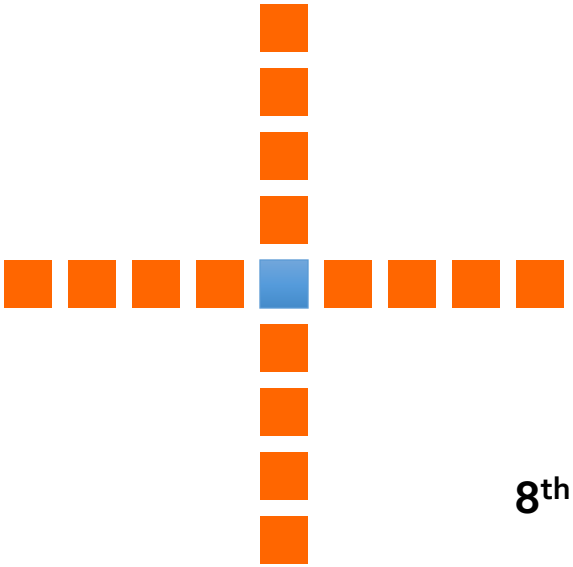
Stencils



2nd order stencil

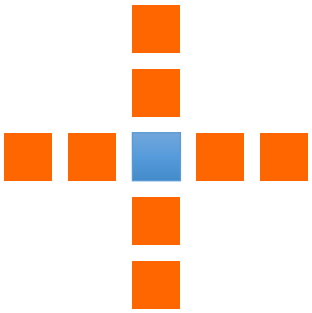


4th order stencil

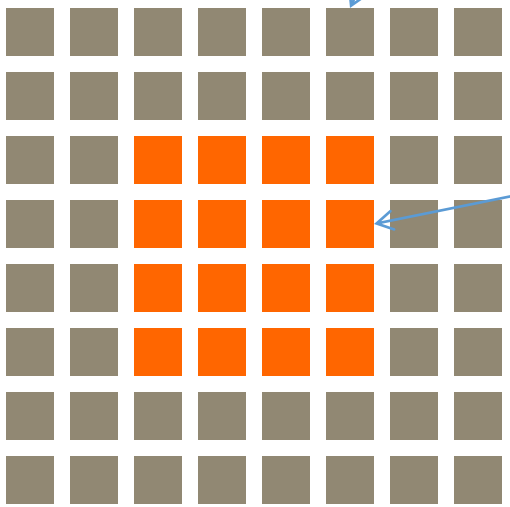


8th order stencil

The grid



4th order stencil



Updated in a separate routine; boundary conditions are used

Node is updated using the 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.
- So 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.

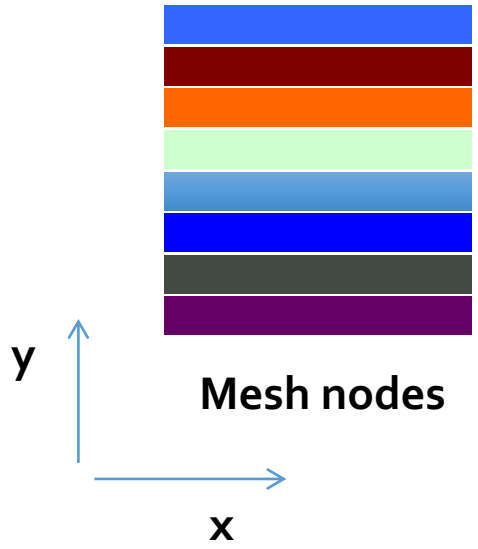
Mesh grid

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

```
class Grid {
  public:
    Grid(int gx, int gy);
    Grid(const Grid&);
    ~Grid();



    std::vector<float> hGrid_;
    float*             dGrid_;

  private:
    int gx_, gy_;              //total grid extents
};
```

Memory layout

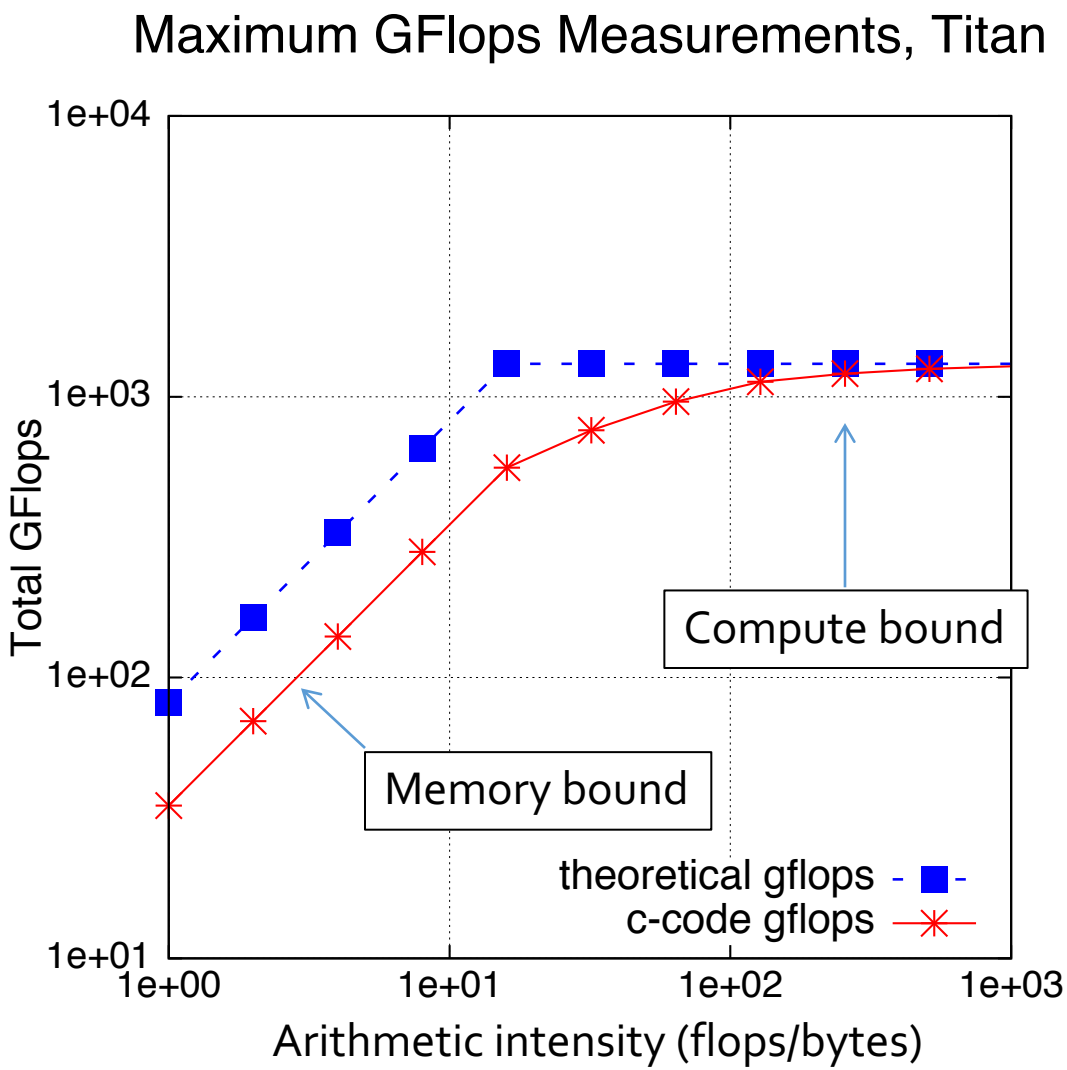
We use a simple 1D array to store grid information:



Mesh nodes

Layout in memory

Roofline model



NVIDIA K20x GPU

Where are we in the roofline plot?

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?

```
return curr[0] + xcfl * (curr[-1] + curr[1] - 2.f * curr[0]) +
       ycfl * (curr[width] + curr[-width] - 2.f * curr[0]);
```

Answers

1. How many flops?
   10 additions / multiplications
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?

```
return curr[0] + xcfl * (curr[-1] + curr[1] - 2.f * curr[0]) +
       ycfl * (curr[width] + curr[-width] - 2.f * curr[0]);
```

Optimizing memory access

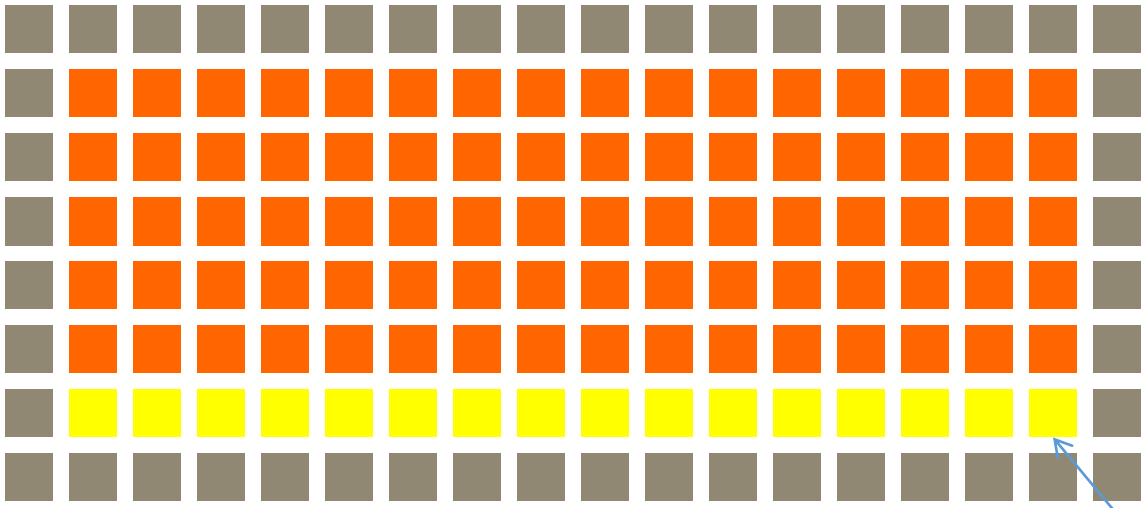
There are two main ideas:

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

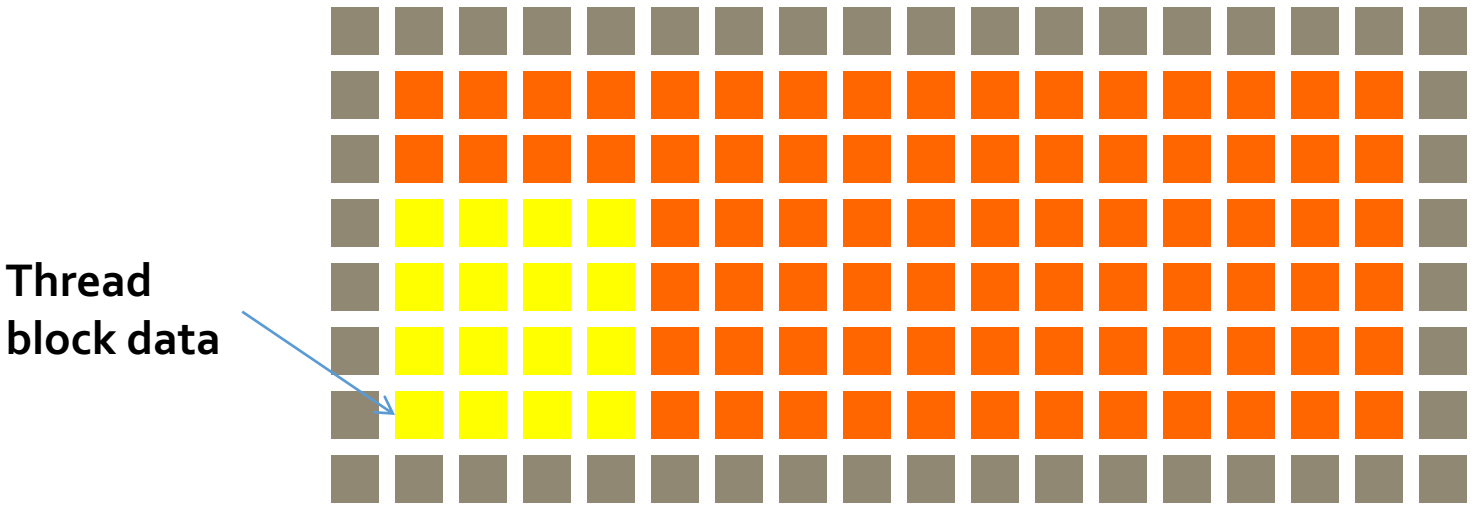
- The first concern is: what is a thread block going to do?
- Idea 1: work on a line of the mesh



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

Better shape

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



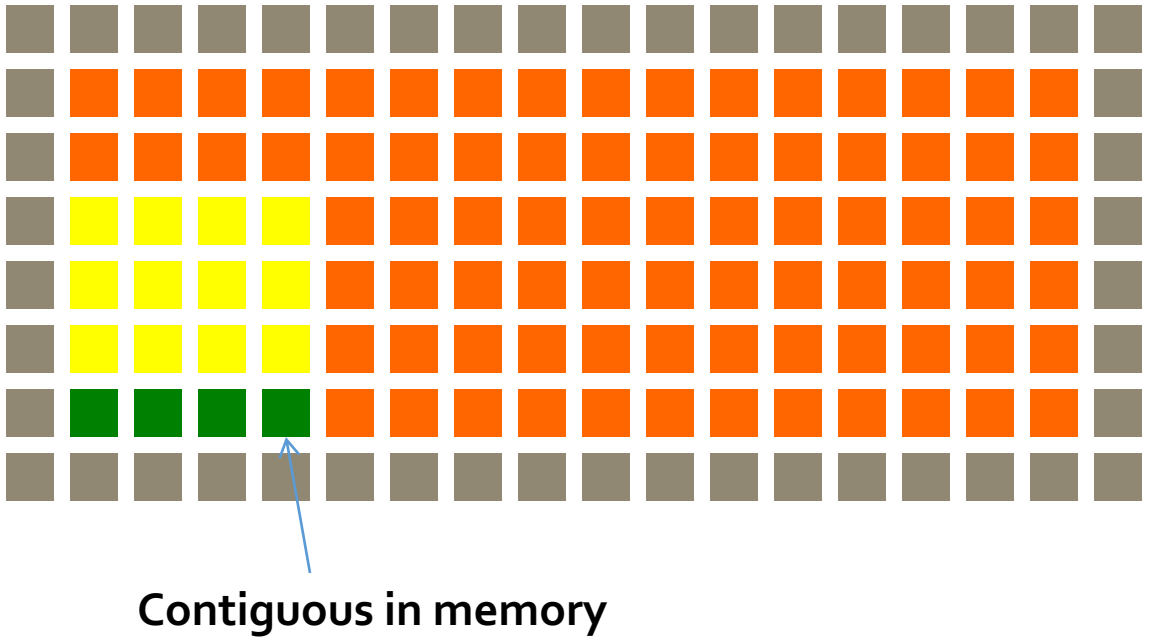
- Thread-block = 16 threads.
- Reads: 16+16. Writes: 16. Total = 48
- Flops: 10*16 = 160
- Ratio: flops/word = 3.3

Asymptotic intensity

- For an n x n block:
  - Memory traffic: $2n^2 + 4n$
  - Flops: $10n^2$
- Maximum intensity: 5 flops/words
- Kernel with higher-order compact stencils will have better performance.
- We see that for this problem, we cannot make the kernel compute bound. The peak bandwidth is going to be the limiting factor.

Coalesced memory reads

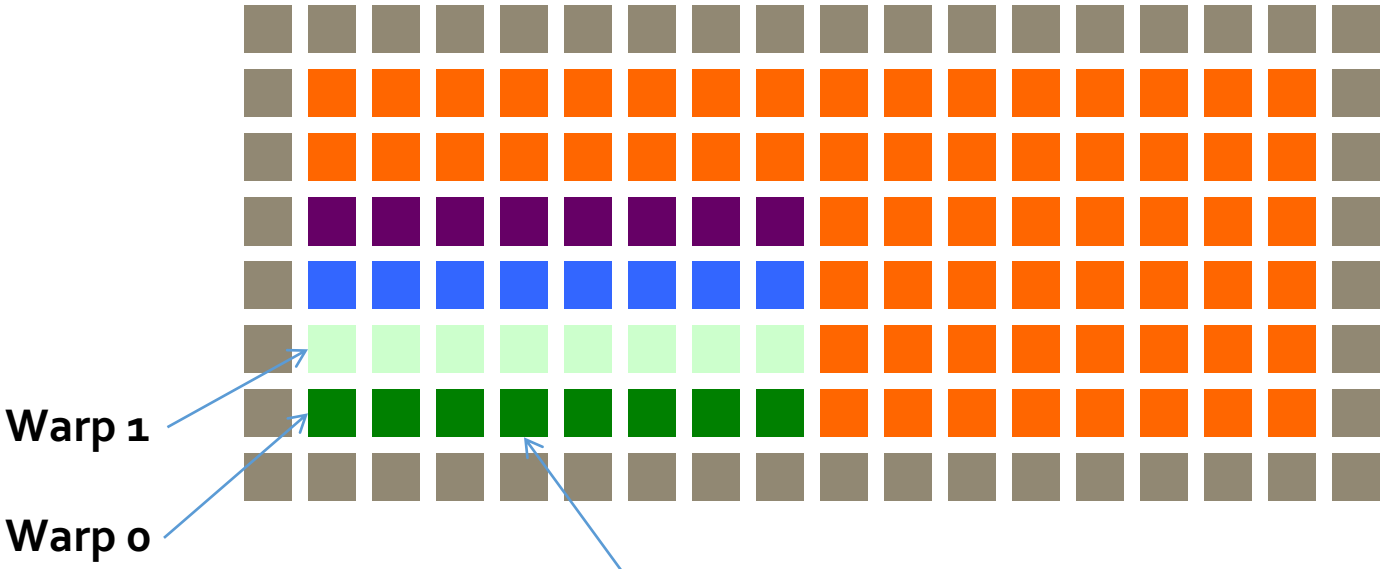
- We saw that the hardware does best at reading 32 floats (= 128 bytes) contiguous in memory for each warp.



- 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.

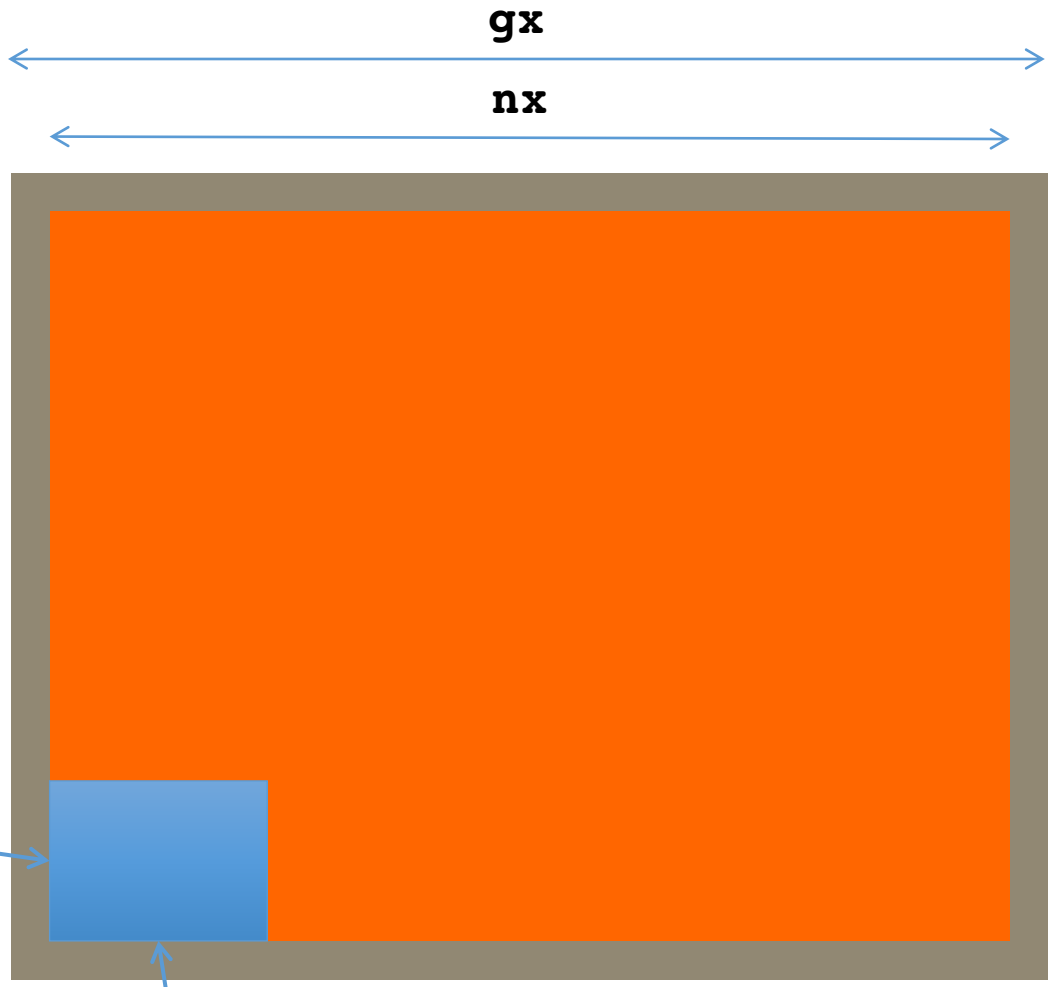
Warp assignment

- 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

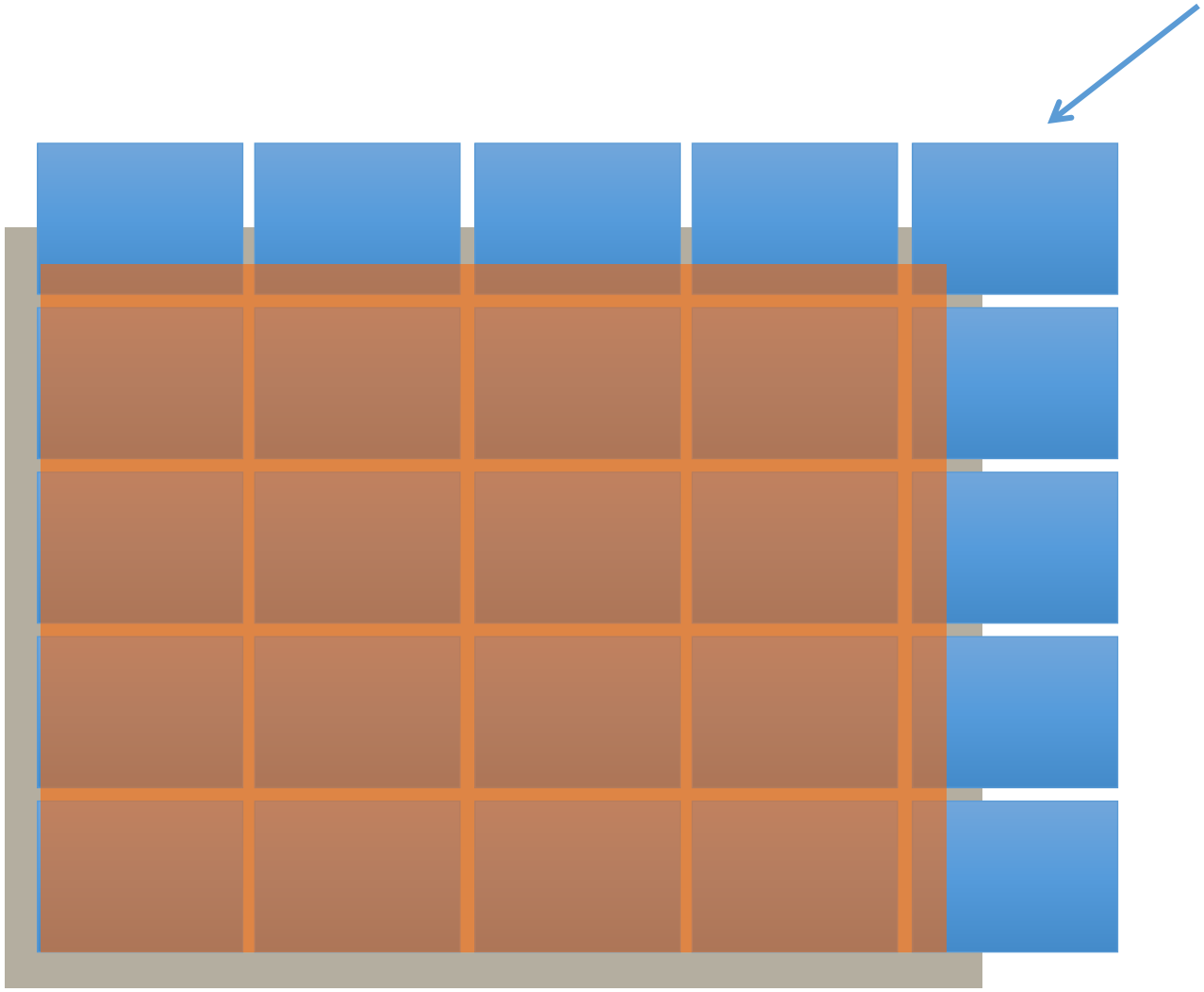
Good dimensions

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.

```
template<int order>
__global__
void gpuStencil(float* next, const float* curr, int gx, int nx, int ny,
                float xcfl, float ycfl) {
    // TODO
}
```

Domain size

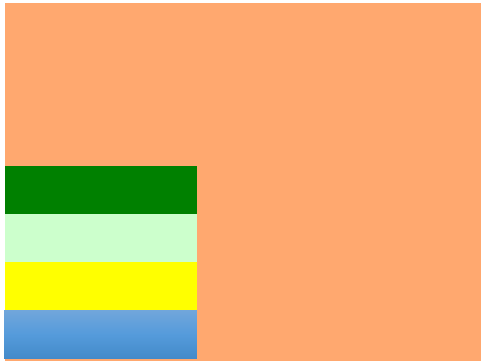
- You will see in the first implementation that the domain that a thread block is working on is shaped like a rectangle:

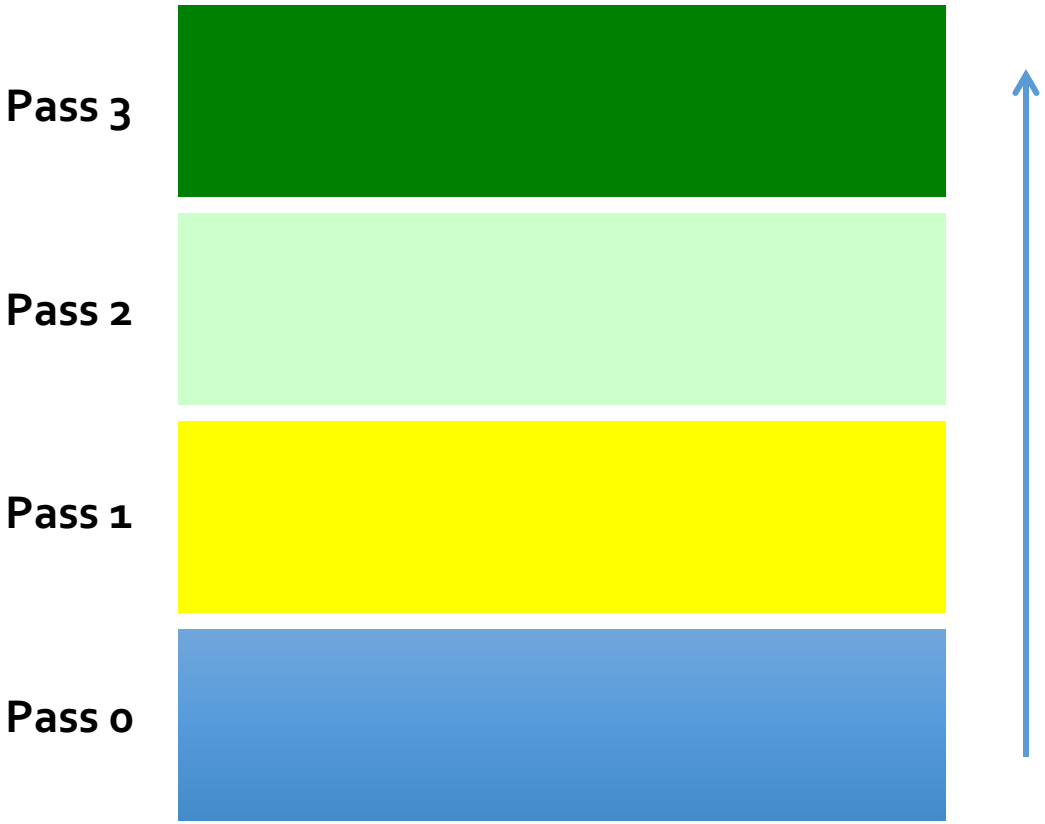
- It's better to have a square as we saw.
- But we are limited by the number of threads we can have in a block.
- ??????

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.

```
template<int order, int numYPerStep>
__global__
void gpuStencilLoop(float* next, const float* curr, int gx, int nx, int ny,
                    float xcfl, float ycfl) {
    // TODO
}
```

Shared memory

- Instead of 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.

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.**

You have to carefully track all the indices.

Example of output

Here is an example case to give you an idea of what to expect.

Mesh size: **`nx = 4096, ny = 4096.`**

Number of time steps: 10

Sample output from code:

```
size =  4096 order of stencil =  8
Order: 8
                    time (ms)     GBytes/sec
         CPU          826.692         14.612
      Global          48.7157         247.961
       Block          40.2375         300.208
      Shared          27.1916          444.24

                       L2Ref           LInf           L2Err
      Global    1.25248e+06    6.0329e-07    5.14062e-08
       Block    1.25248e+06    6.0329e-07    5.14062e-08
      Shared    1.25248e+06    6.0329e-07    5.14062e-08
```

Notes

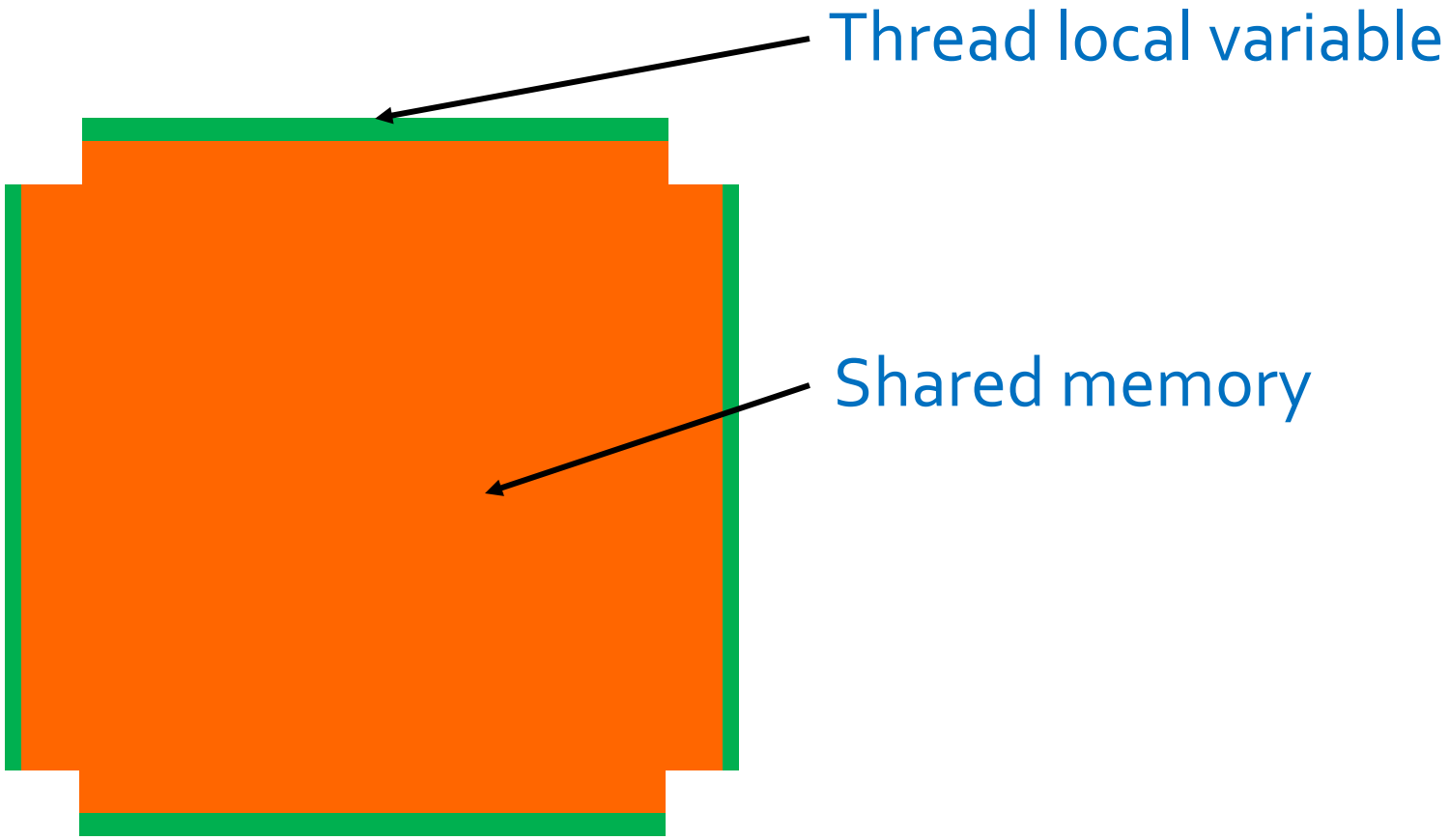
- To run the code you can use options `-g -b or -s`. This determines which GPU algorithm can run. g=global, b=block (algo. 2), s=shared
- See sample code for details.
- Read the CPU code for reference.
- You may get slightly different numbers from the previous slide but it should be close.
- There is 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.

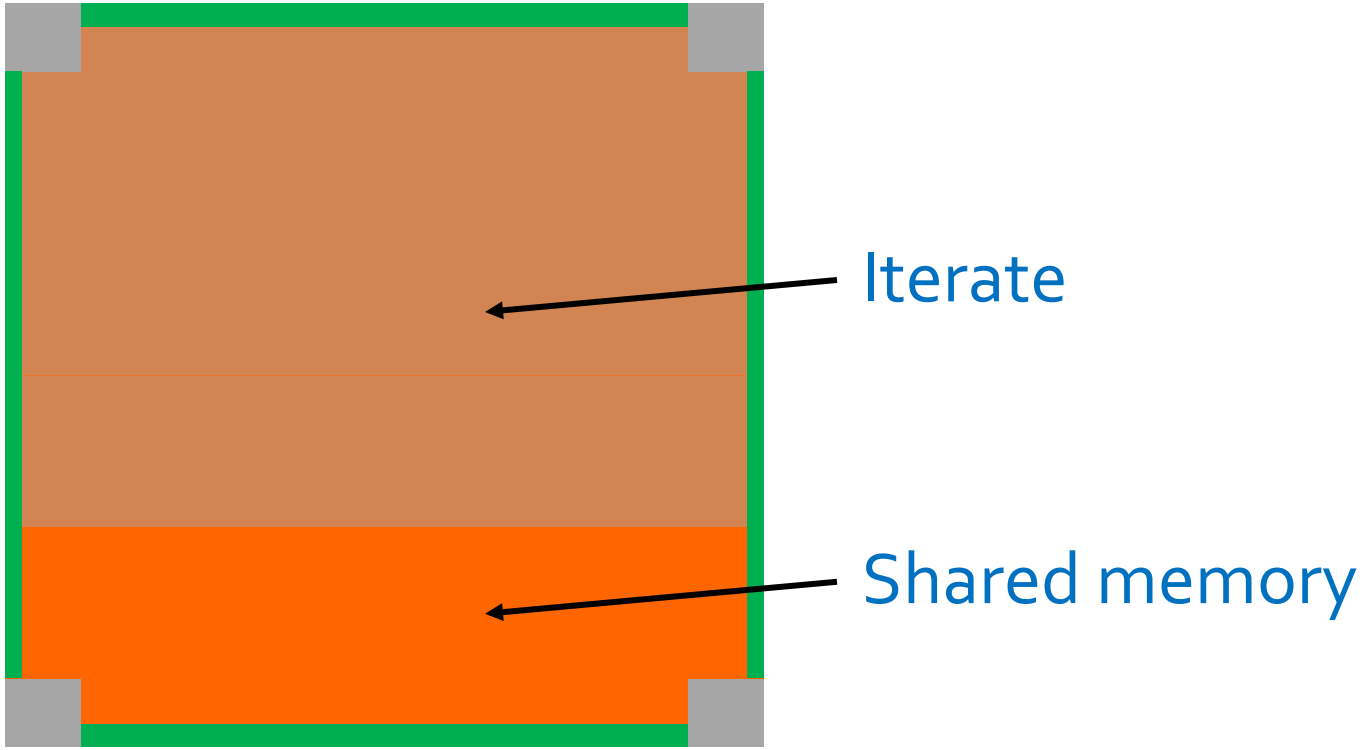
Sample output with -G compilation

CUDFLAGS=-G -O3 -arch=sm_20 …

```
size =  4096 order of stencil =  8
Order: 8
                    time (ms)     GBytes/sec
          CPU         826.605        14.6135
       Global         137.992        87.5387
        Block         129.139        93.5394
       Shared         191.499        63.0793

                       L2Ref           LInf           L2Err
       Global    1.25248e+06              0              0
        Block    1.25248e+06              0              0
       Shared    1.25248e+06              0              0
```

Further optimizations





- This allows using a rectangular shape for the shared memory.
- This leads to larger squares and less memory traffic (more data reuse).
- Size of squares can be optimized (trade-off) to minimize memory traffic (large squares) and maximize concurrency (small squares).