

# CME 213

SPRING 2019

Eric Darve

# CUDA summary

Warp, block, grid

Memory access:

- Global memory: coalesced access = warp requests and uses a full 32-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

# Branch divergence

- Should be avoided.
- This leads to idle threads.

# Homework 4

## PDE solver using CUDA

- We want to solve the following PDE:

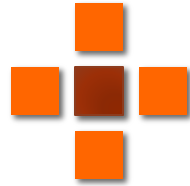
$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

- 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:

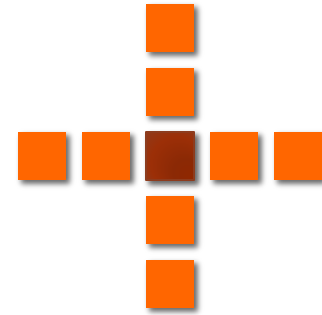
$$T^{n+1} = AT^n$$

- Different stencils can be used depending on the order.

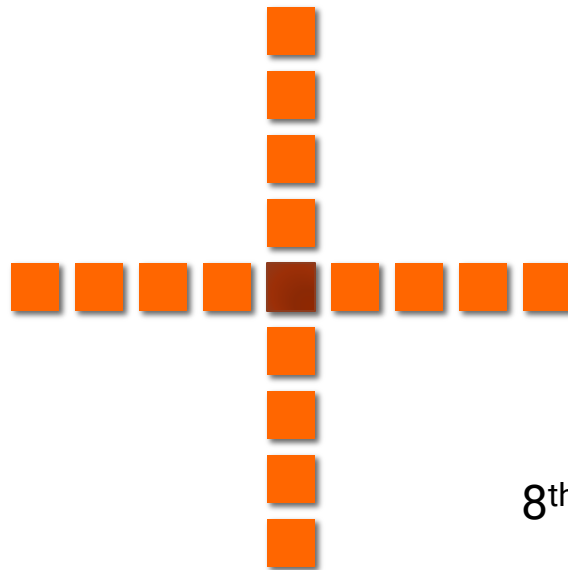
# Stencils



2<sup>nd</sup> order stencil

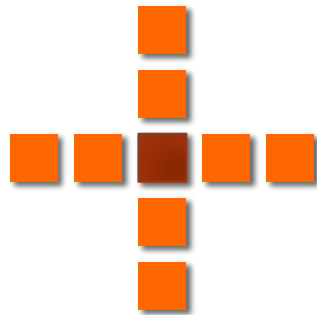


4<sup>th</sup> order stencil



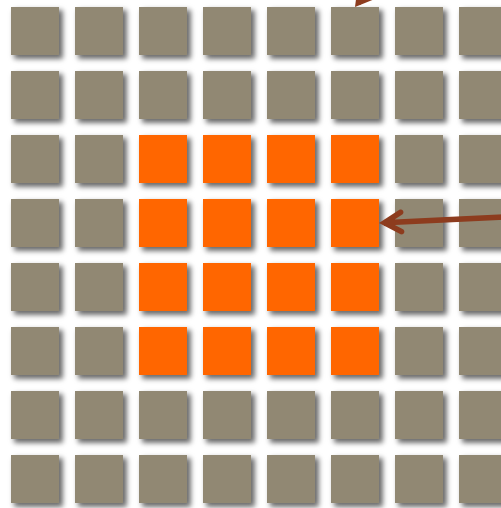
8<sup>th</sup> order stencil

# The grid



4<sup>th</sup> 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 a basic finite-difference centered stencil.



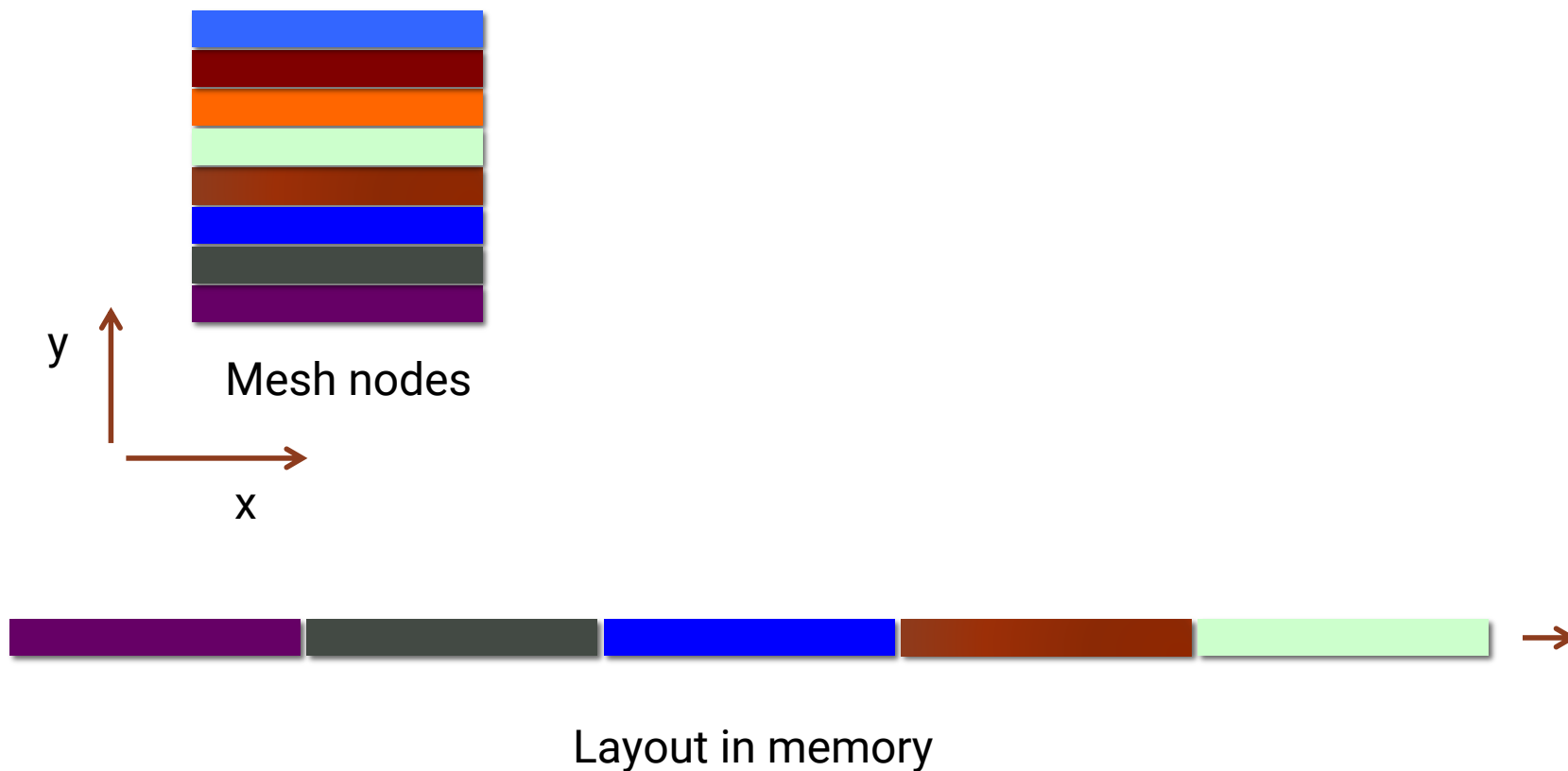
# Mesh grid

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  
};
```

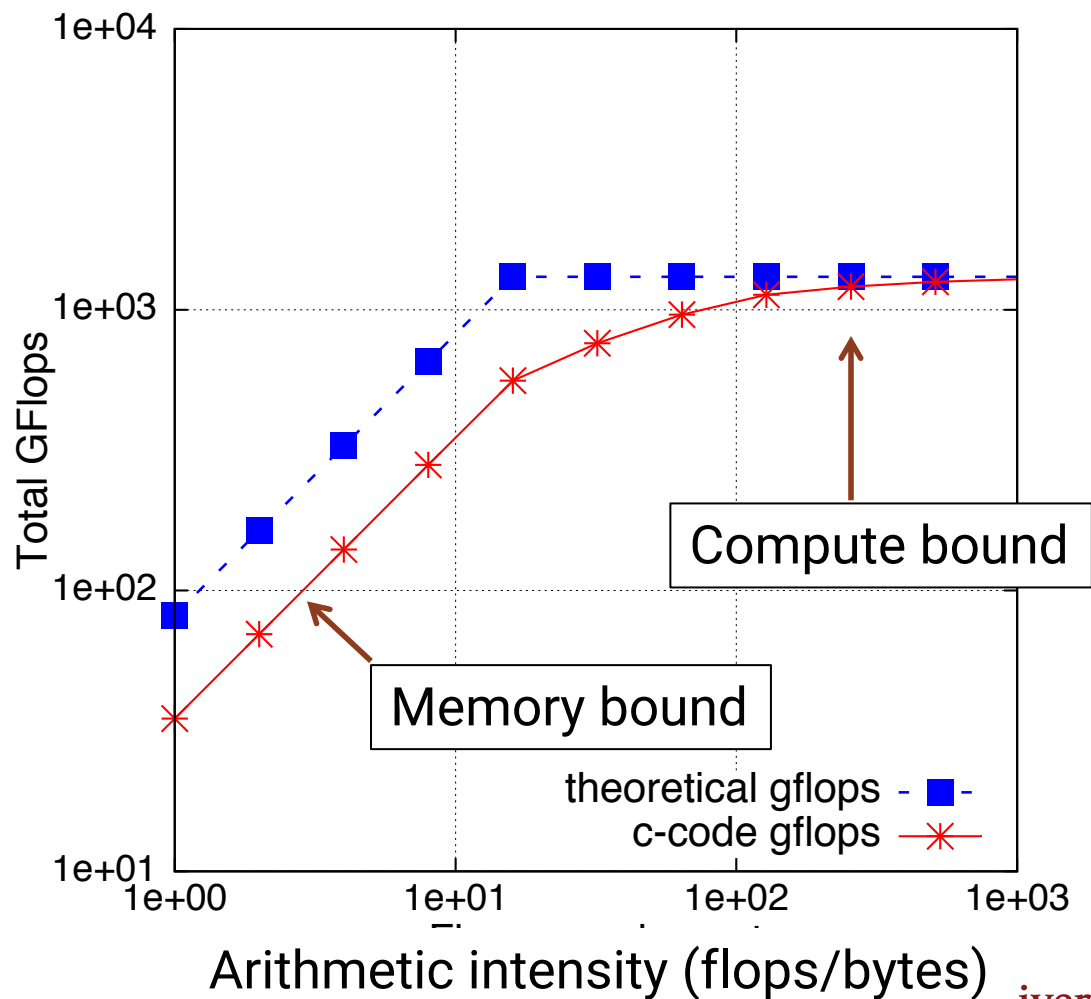
# Memory layout

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



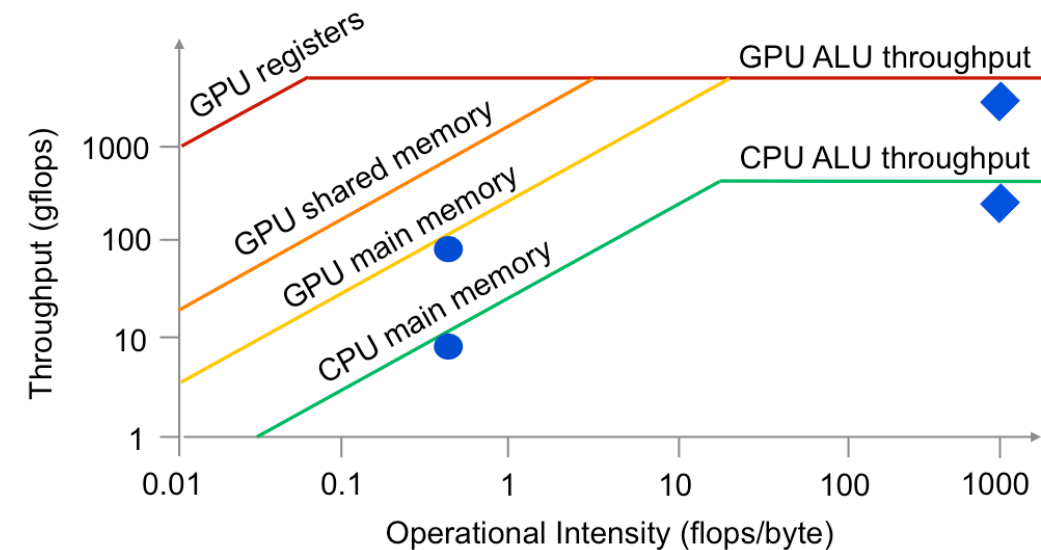
# Roofline model

Maximum GFlops Measurements, Titan



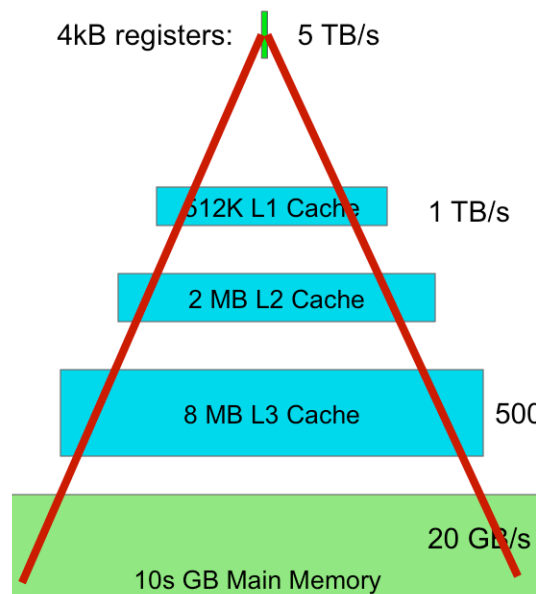
NVIDIA  
K20x GPU

- Dense matrix multiply ◆
- Sparse matrix multiply ●

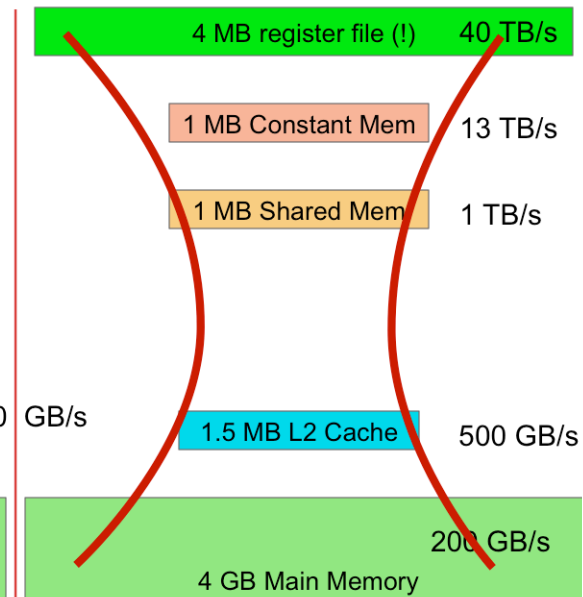


## Rooflines and memory hierarchies

Intel® 8 core Sandy Bridge CPU



NVIDIA® GK110 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:

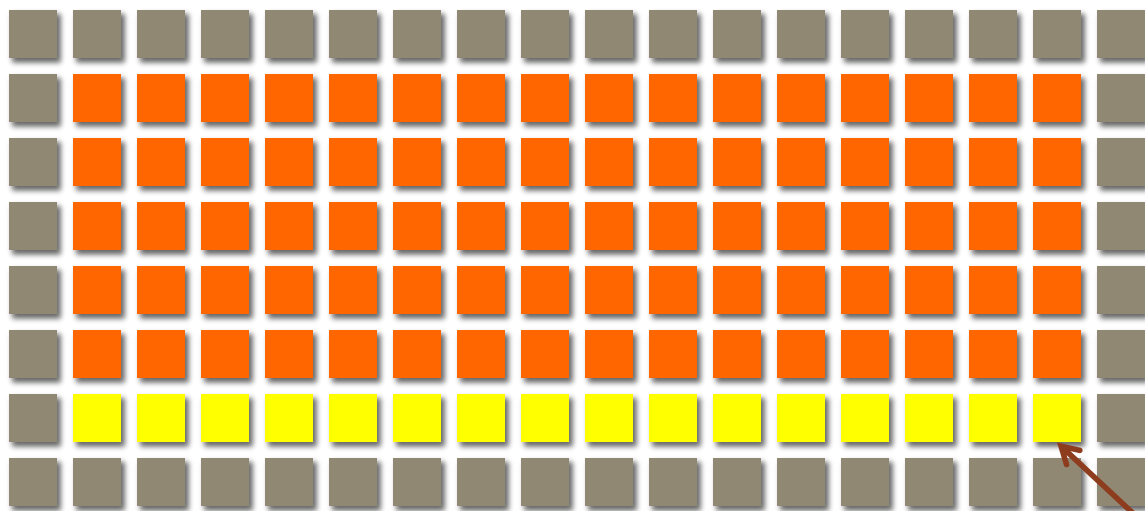
1. Use cache or shared memory:  
Once a data is read from memory and is in cache/shared memory, use it as much as possible, that is use it for several different stencils that need that point
2. Memory accesses should be coalesced: threads in a warp need to read from contiguous memory locations.

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



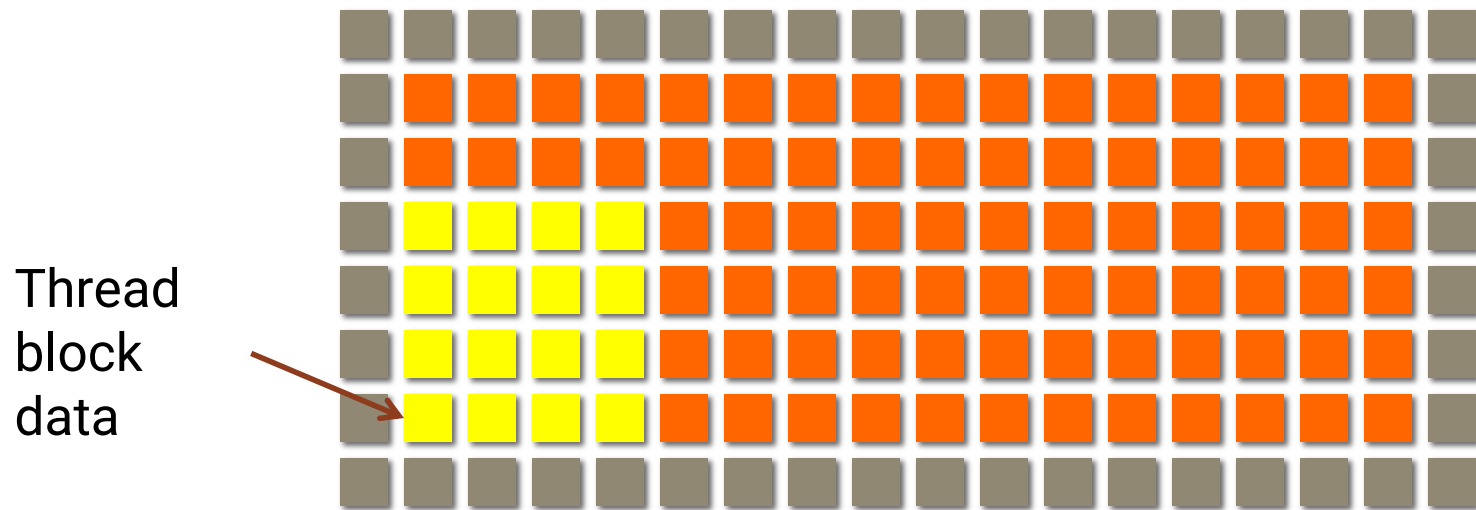
- Thread-block = 16 threads.
- Reads:  $16 \times 3 + 2$ . Writes: 16. Total = 66
- Flops:  $10 \times 16 = 160$
- Ratio: flops/word = 2.4

Thread block data



## Better shape

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



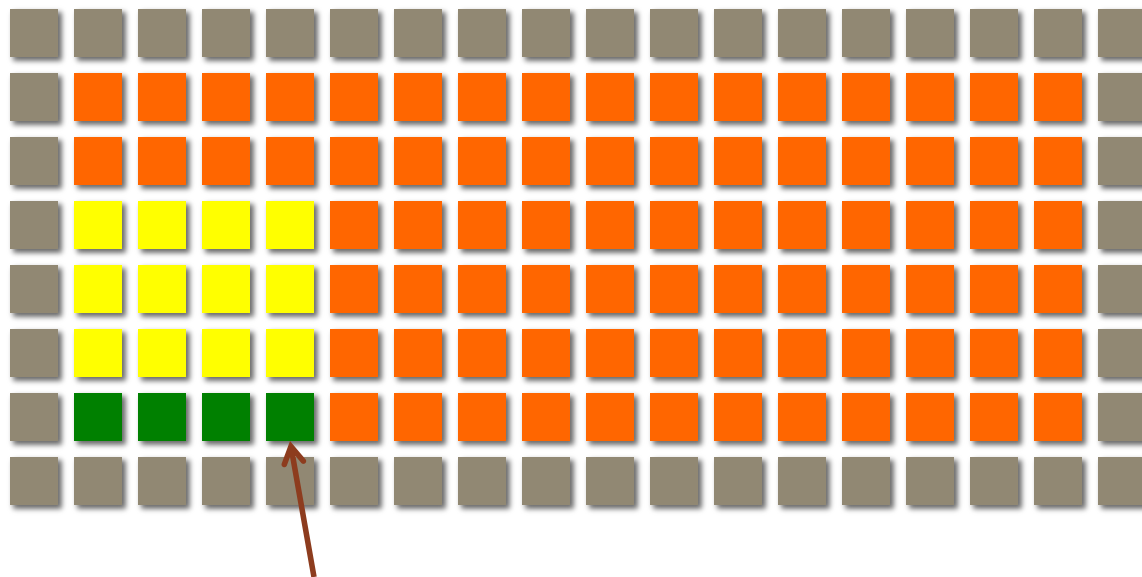
- Thread-block = 16 threads.
- Reads:  $16+16$ . Writes: 16. Total = 48
- Flops:  $10 \times 16 = 160$
- Ratio: flops/word = 3.3

# Asymptotic intensity

- For an  $n \times n$  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 contiguous in memory for each warp.



Contiguous in memory

- 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.

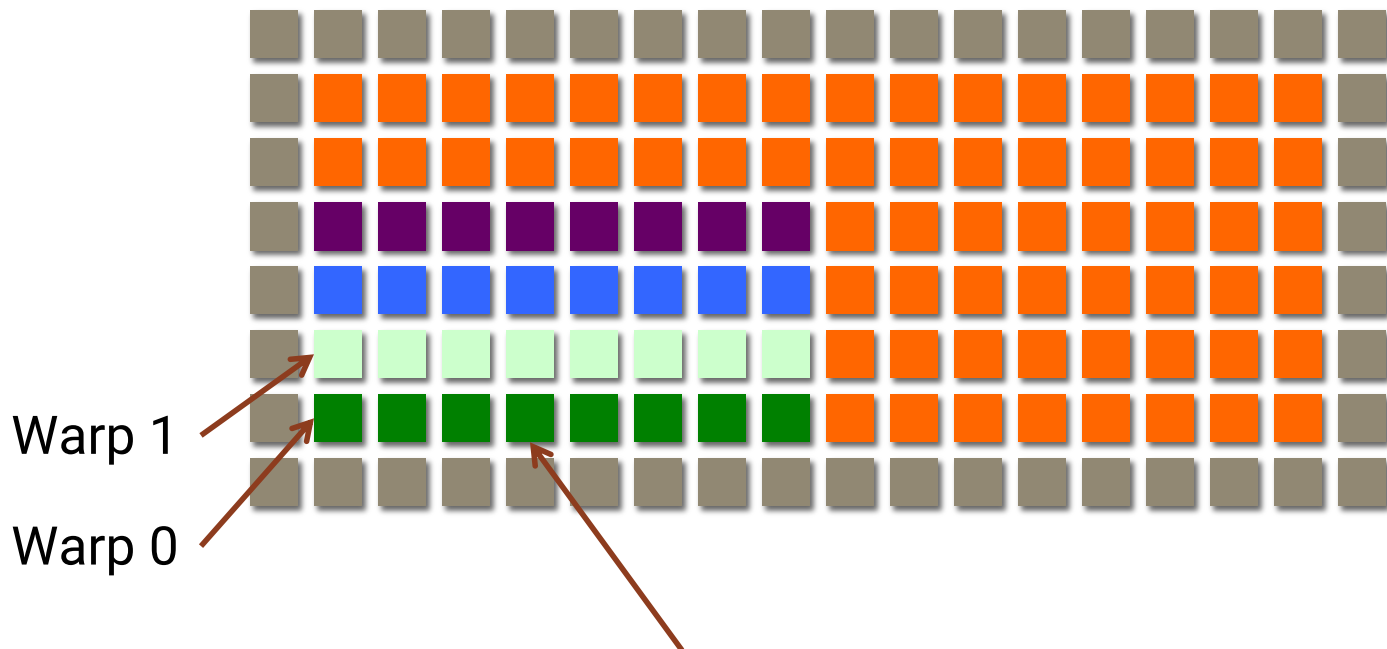
## Warp assignment

Warp 0 = tid 0 to 31

Warp 1 = tid 32 to 63

Warp 2 = tid 64 to 95

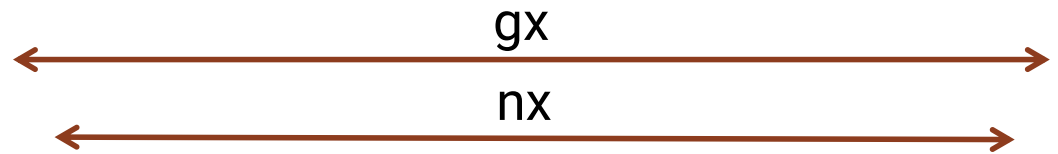
Warp 3 = tid 96 to 128



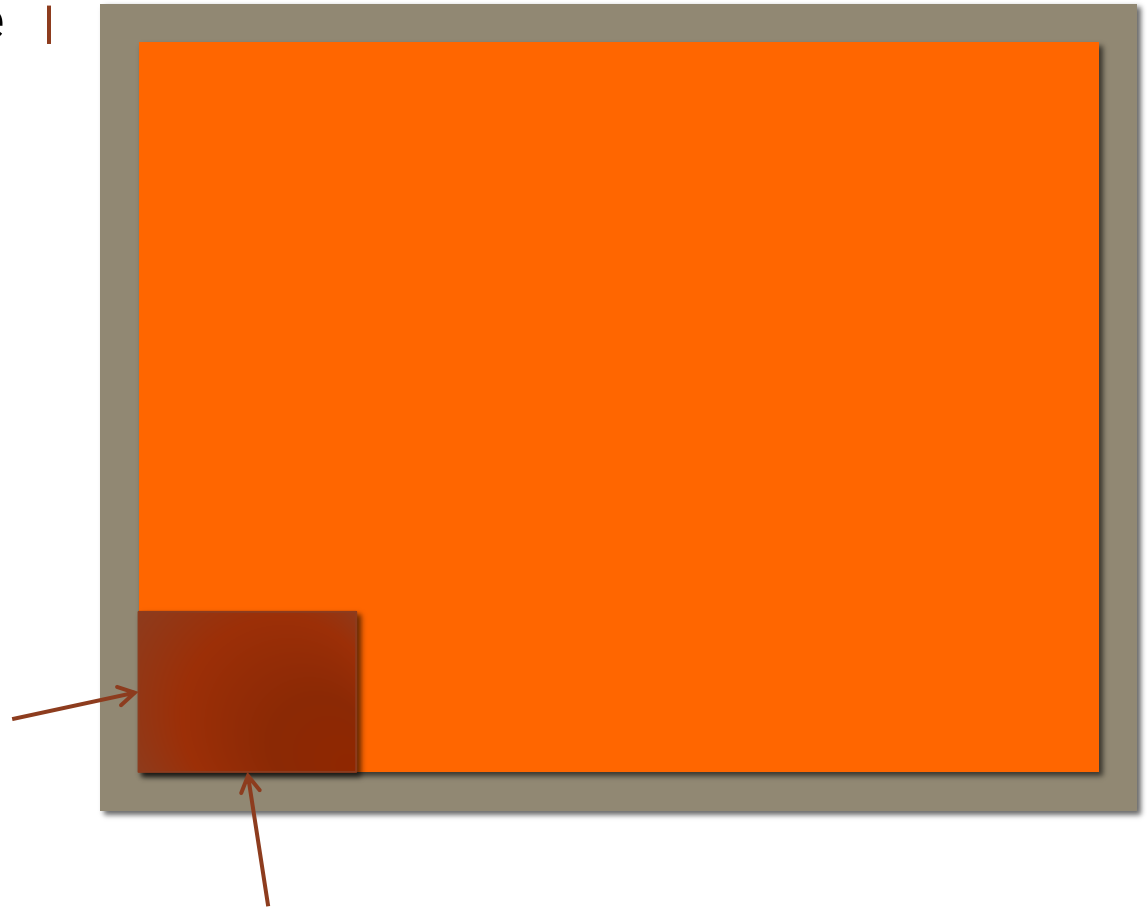
Assume a warp size of 8: this is a perfect read and write to memory

Choose your block size

Border 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

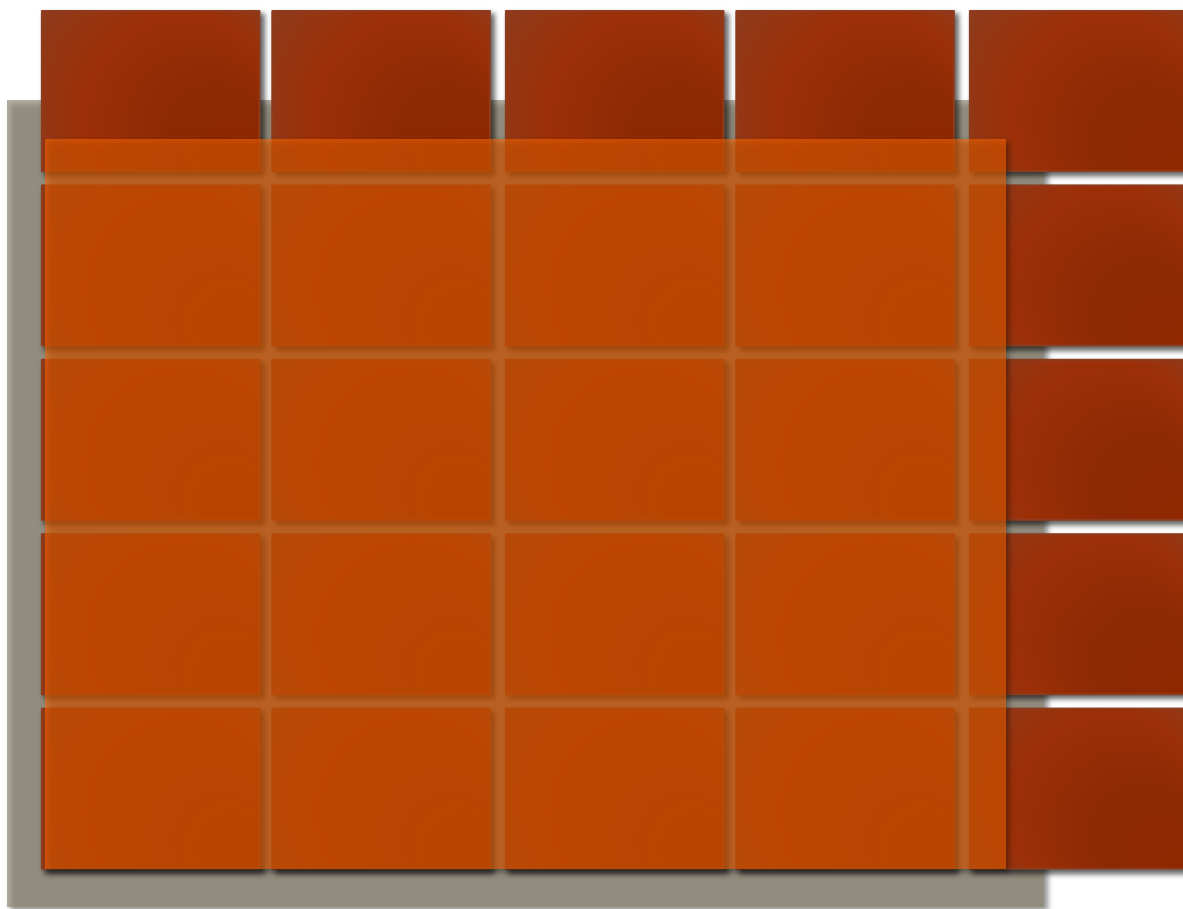
Guidelines:

- `blockDim.x` multiple of 32
- Total number of threads should be approximately 256/512.

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) {
```



## Domain size

- In the first implementation, the domain that a thread block is working on is shaped like a strip:



- It's better to have a square as we saw.
- Let's reshape our block and give more work to each thread.

# Solution

We can ask threads to process multiple elements:

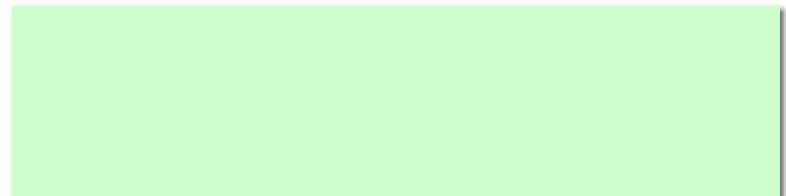
- Each thread loops multiple times until the whole block has been processed.
- Number of passes = `numYPerStep`



Pass 3



Pass 2



Pass 1



Pass 0



## 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) {
```

# Shared memory

Instead of relying on the cache, we can use shared memory.

Two-step process:

- Load in shared memory
- 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.

You can 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:

Order: 8, 4096x4096, 100 iterations

		time (ms)	GBytes/sec	
	CPU	4259.53	28.359	
	Global	294.945	409.554	
	Block	194.064	622.453	
	Shared	147.28	820.179	
		L2Ref	LInf	L2Err
	Global	0.447065	0.000895977	3.31018e-05
	Block	0.447065	0.000895977	3.31018e-05
	Shared	0.447065	0.000895977	3.31018e-05

Order: 2, 4096x4096, 100 iterations

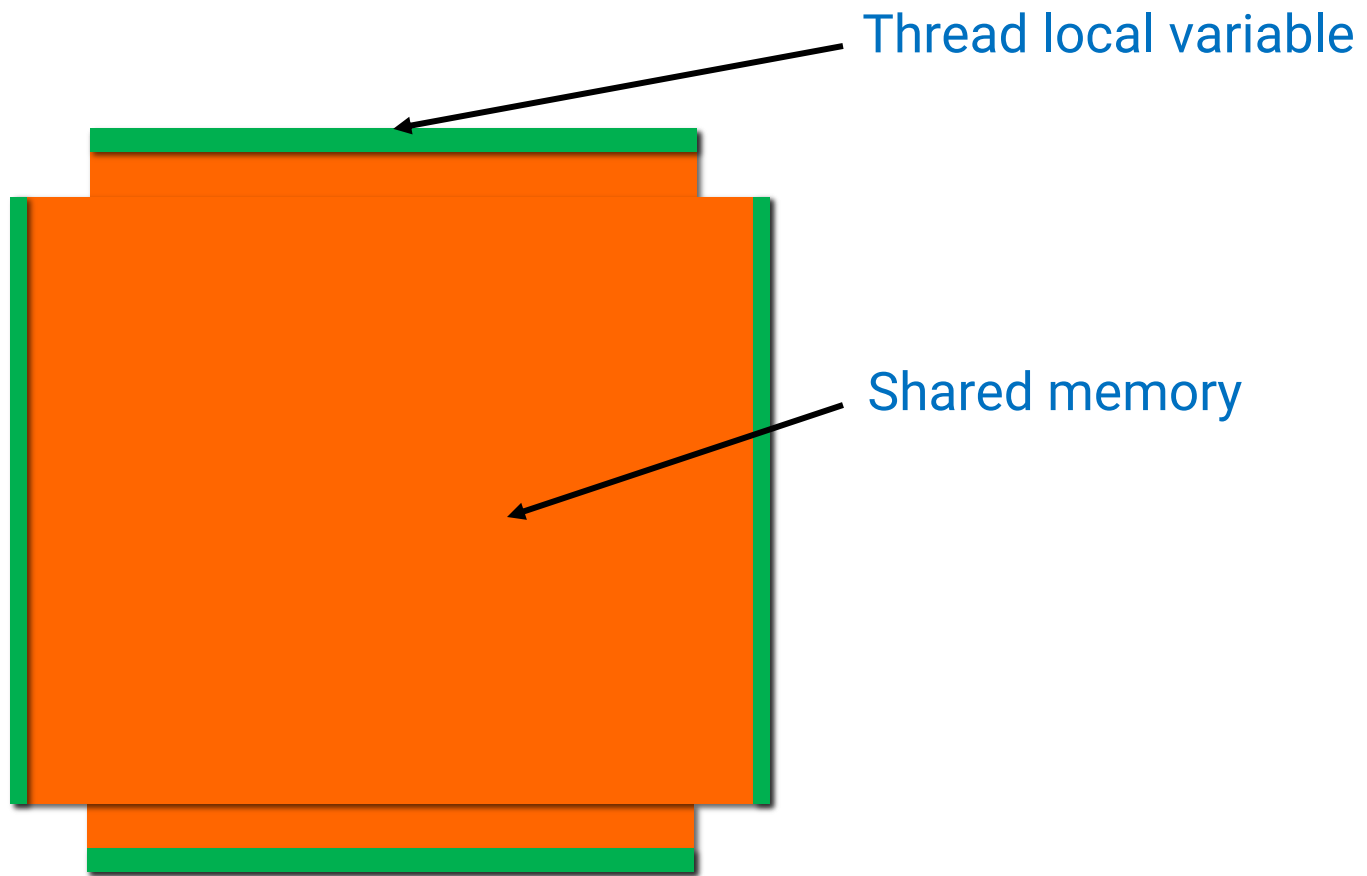
		time (ms)	GBytes/sec	
	CPU	2144.5	18.7761	
	Global	231.485	173.943	
	Block	103.555	388.83	
	Shared	108.694	370.447	
		L2Ref	LInf	L2Err
	Global	0.418194	0.00103641	3.20589e-05
	Block	0.418194	0.00103641	3.20589e-05
	Shared	0.418194	0.00103641	3.20589e-05

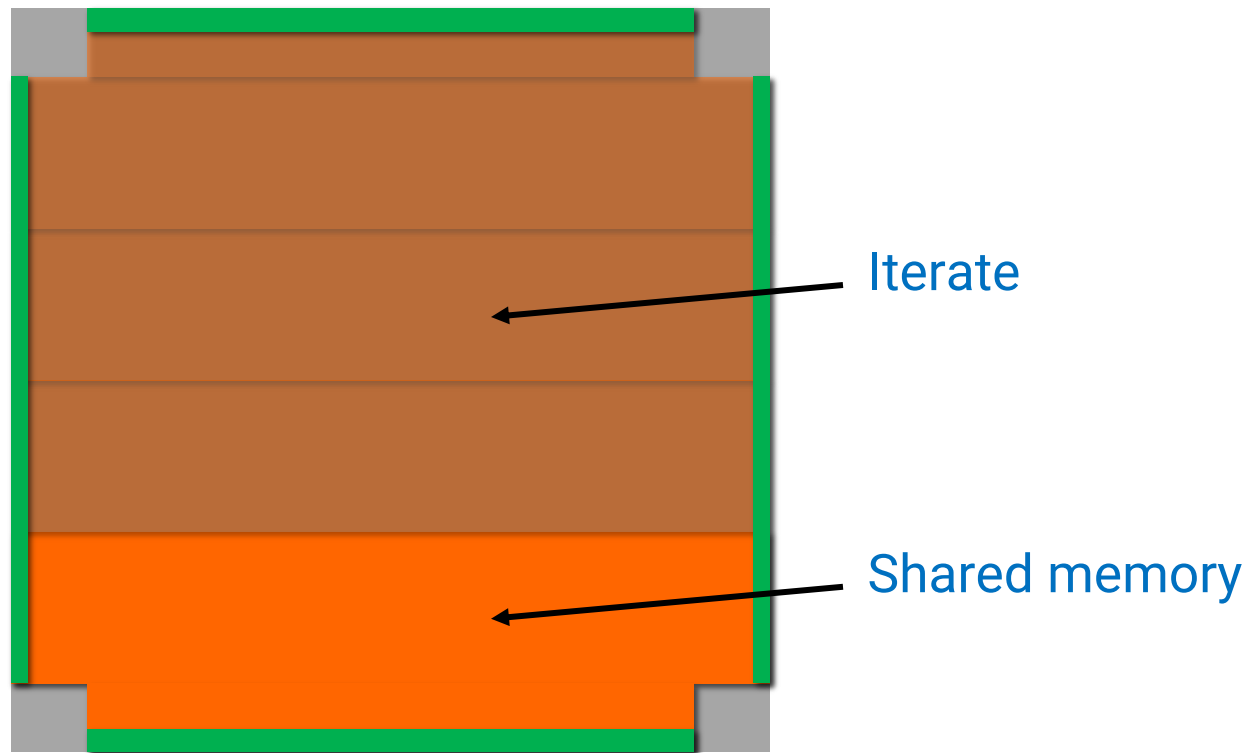
# Notes

- 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.



## 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).