CME 213, ME 339—Spring 2021

Eric Darve, ICME, Stanford

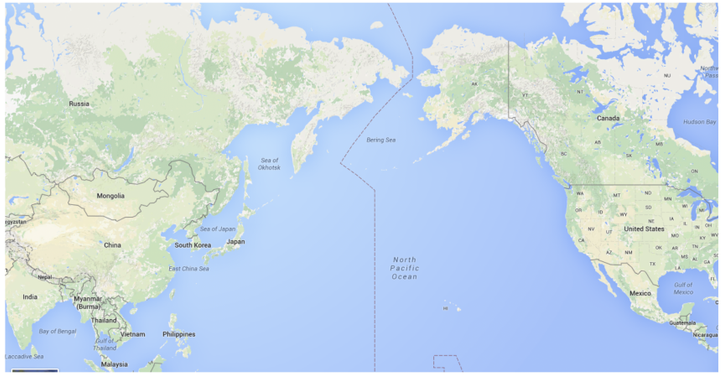
"Software is like entropy: It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases." (Norman Augustine)

Concurrency and latency

Imagine you are a pencil manufacturer.

You outsource your manufacturing plants to China but your market is in the US.

How do you organize the logistics of the transport?

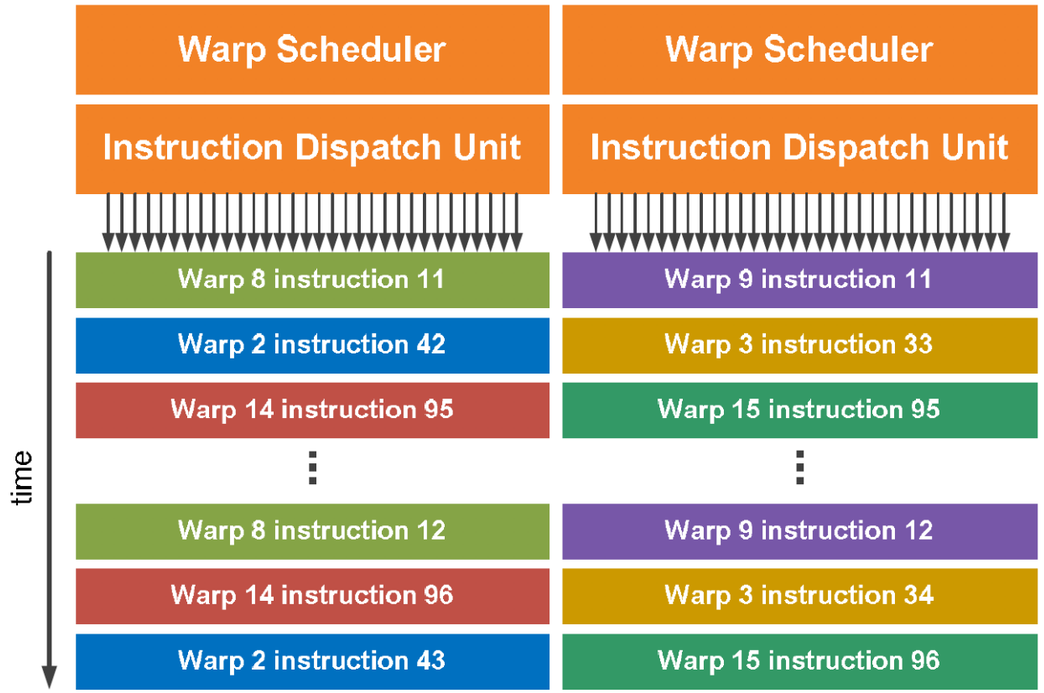


Concurrency is used to hide long latencies:

- memory access
- floating point units
- any long sequence of operations

Processors are optimized in the same way

Hide latency through concurrency



How to maximize concurrency?

- Have as many live threads as possible
- Instruction-level parallelism

Hardware limits

- Max dim. of grid: y/z 65,535 (x is large, $2^{31} - 1$)
- Max dim. of block: x/y 1,024; z 64
- Max # of threads per block: 1,024
- Max blocks per SM: 16
- Max resident warps: 32
- Max threads per SM: 1,024
- # of 4-byte registers per SM: 64 K
- Max shared mem per SM: 64 KB

Spreadsheet

How can we make sense of this?

CUDA API

Achieve best potential occupancy; recommended parameter selections

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`

Reports an occupancy prediction based on the block size and shared memory usage of a kernel

`cudaOccupancyMaxPotentialBlockSize`
`cudaOccupancyMaxPotentialBlockSizeVariableSMem`

Returns the minimum grid size and recommended block size to achieve maximum occupancy

Occupancy spreadsheet!

CUDA Occupancy Calculator

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):	7.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	30	
User 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	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	7.5
Threads per Warp	32
Max Warps per Multiprocessor	32
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1024
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	0

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	8	32	4
Registers	(Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)		4352	65536	15

Note: SM is an abbreviation for (Streaming) Multiprocessor

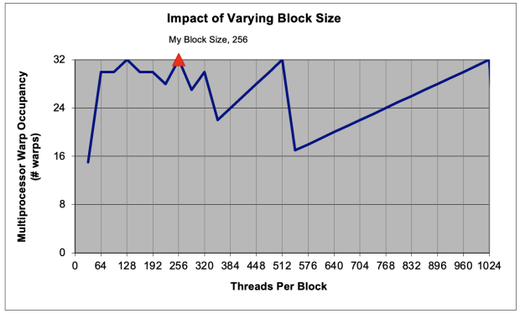
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	4	8	32
Limited by Registers per Multiprocessor	8		
Limited by Shared Memory per Multiprocessor	15		

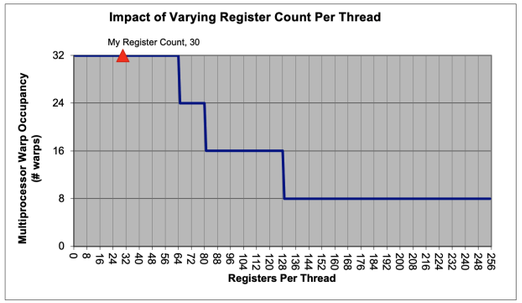
Note: Occupancy limiter is shown in orange

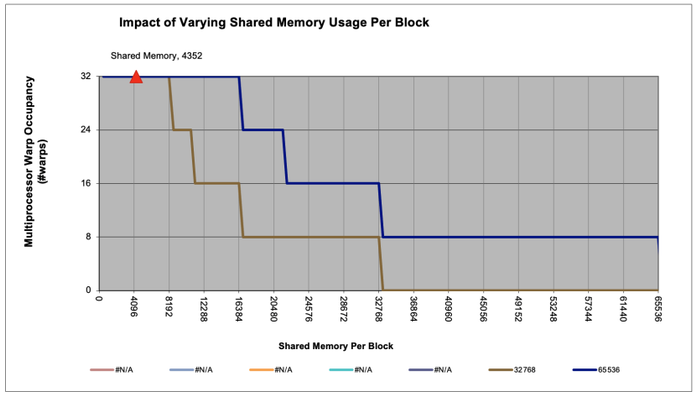
Physical Max Warps/SM = 32
Occupancy = 32 / 32 = 100%

CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

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.







Command line

```
$ nvcc --ptxas-options=-v -o transpose transpose.cu
```

`--ptxas-options=-v`

Output

```
ptxas info    : Compiling entry function '...fastTranspose...' for 'sm_75'
ptxas info    : Function properties for ...fastTranspose...
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 30 registers, 4224 bytes smem, 384 bytes cmem[0]
```

Occupancy calculator demo

CUDA Occupancy Calculator

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

1.) Select Compute Capability (click):	7.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	30	
User Shared Memory Per Block (bytes)	4224	

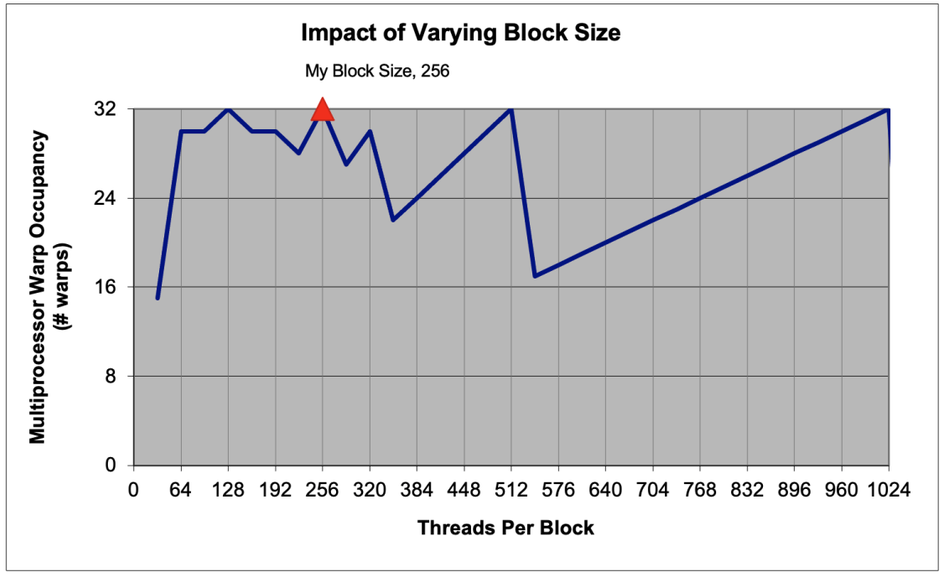
(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	**1024**	(Help)
Active Warps per Multiprocessor	**32**	
Active Thread Blocks per Multiprocessor	**4**	
Occupancy of each Multiprocessor	**100%**	
Physical Limits for GPU Compute Capability:	**7.5**	
Threads per Warp	32	
Max Warps per Multiprocessor	32	

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

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

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.



Branching and divergent execution path

32 threads = 1 warp

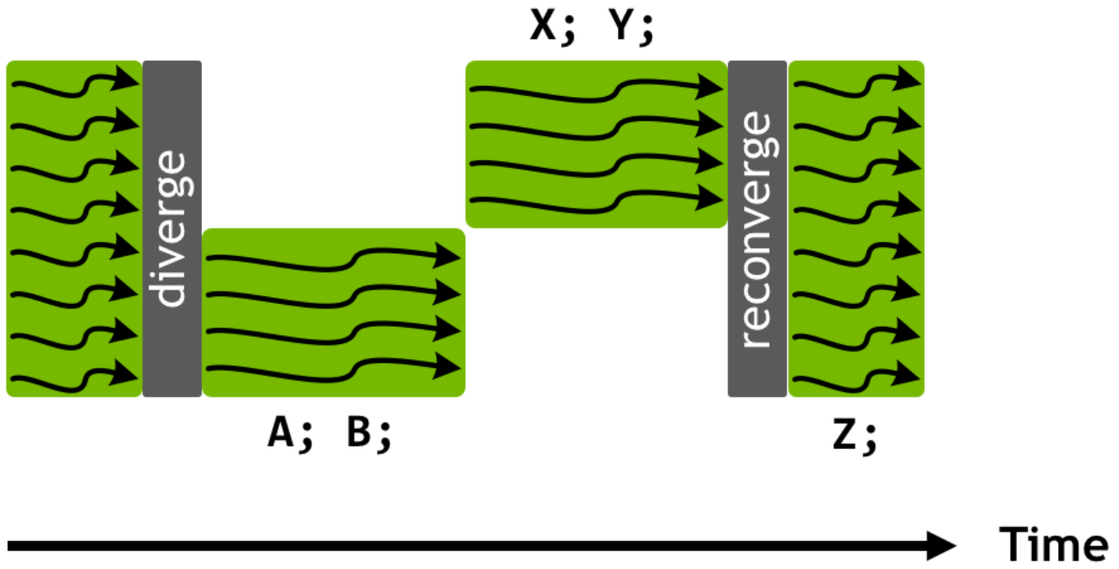
SIMT: Single Instruction, Multiple Thread

Up to Pascal

Single program counter shared amongst all 32 threads

What happens with branches?

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

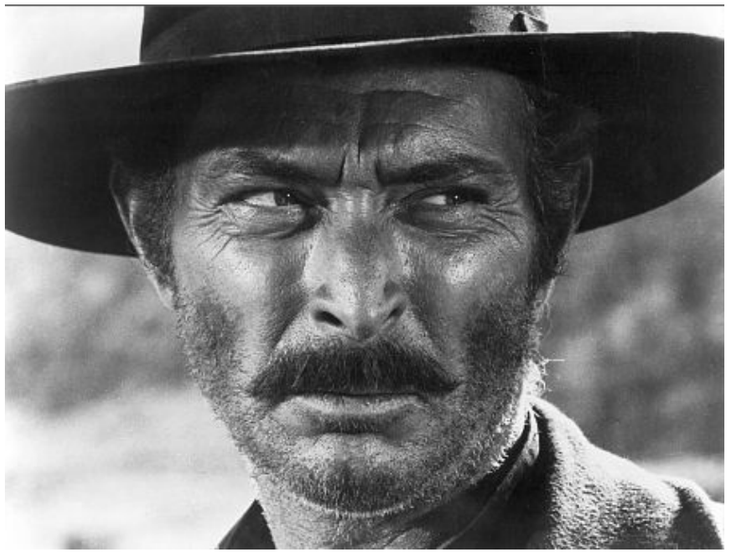


Active mask

Specifies which threads of the warp are active at any given time

Execution becomes serialized

```
__global__ void branch_thread(float* out){
  int tid = threadIdx.x;
  if (tid%2 == 0) {
     ...;
  } else {
     ...;
  }
}
```

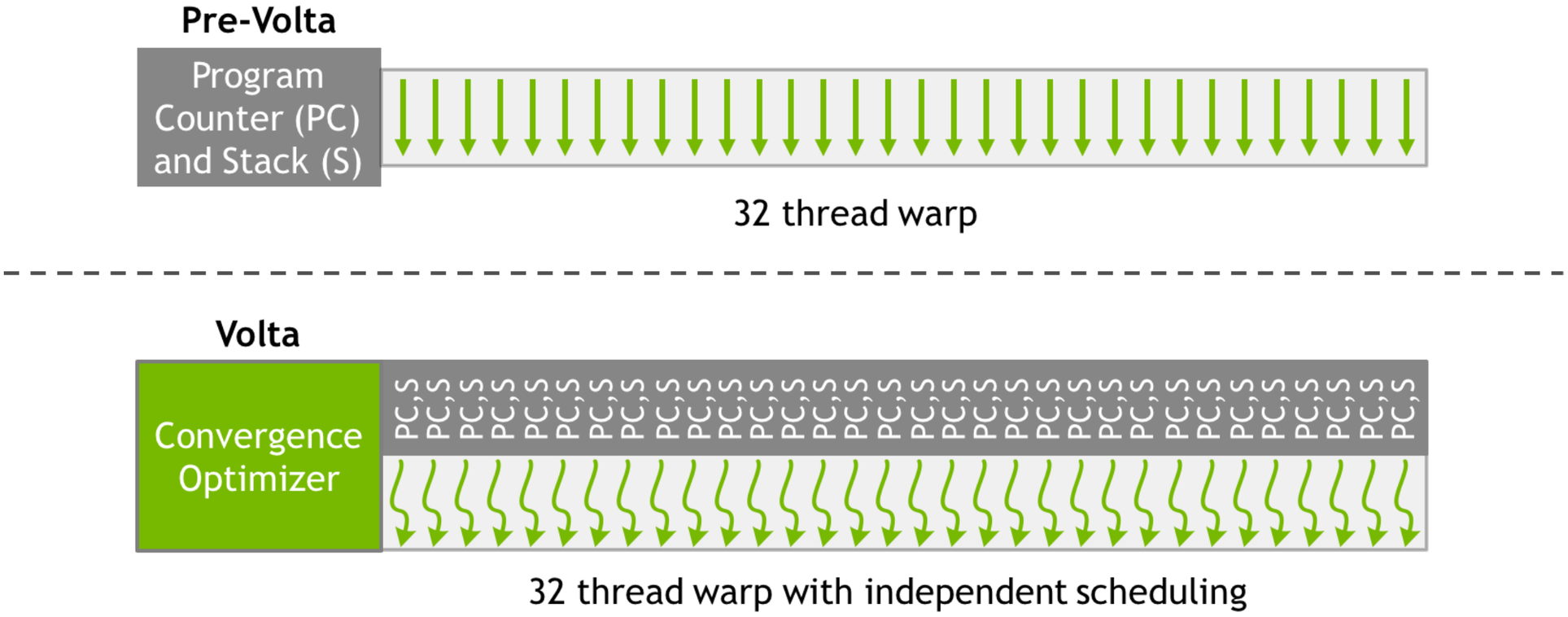


```
__global__ void branch_warp(float* out){
  int wid = threadIdx.x/32;
  if (wid%2 == 0) {
      ...;
  } else {
      ...;
  }
}
```

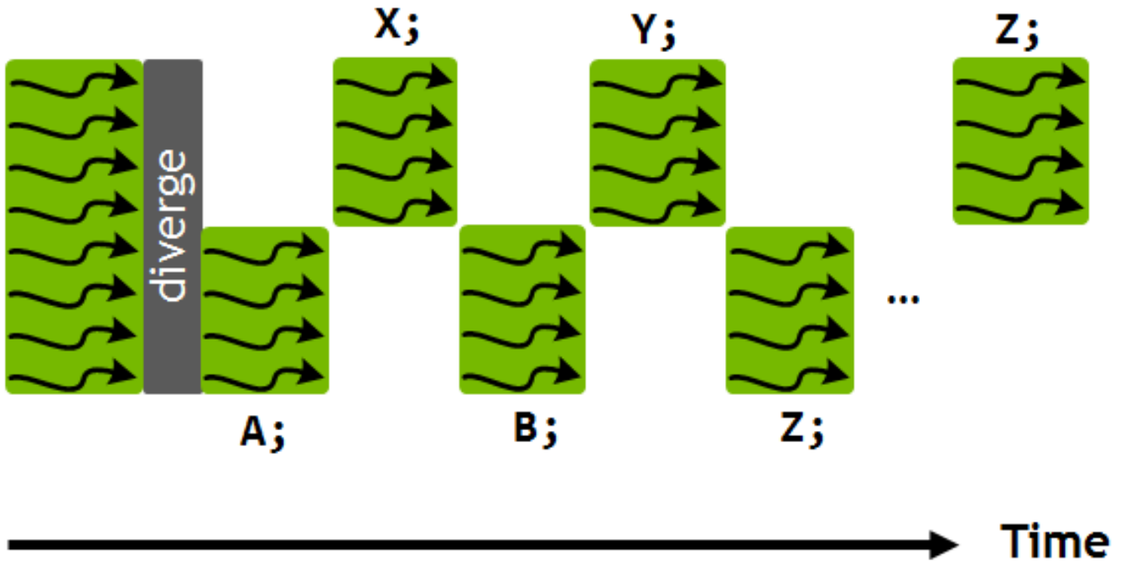
Volta!

Mitigates this problem a bit

But fundamentally performance hit is still significant


Pre-Volta
Program Counter (PC) and Stack (S)
32 thread warp
Volta
Convergence Optimizer
PC,S
32 thread warp with independent scheduling

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Gives more flexibility in programming and performance.

Avoids certain deadlock bugs due to divergence.

Volta allows the correct execution of many concurrent algorithms.

Definition: **starvation-free algorithm**

Algorithms that are guaranteed to execute correctly so long as the system ensures that all threads have adequate access to a contended resource.

```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```

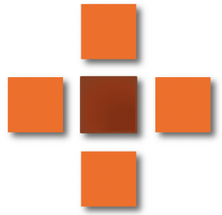
Volta ensures that the thread holding the lock is able to make progress.

Homework 4

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

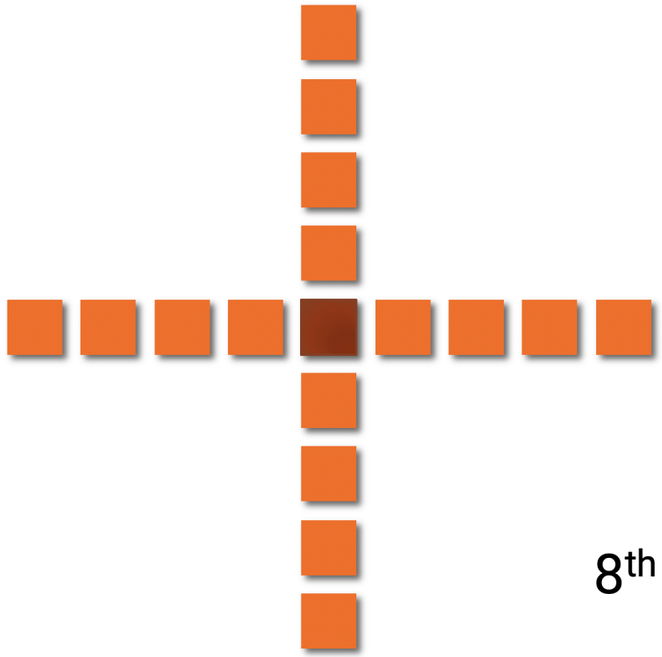
Finite-difference

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



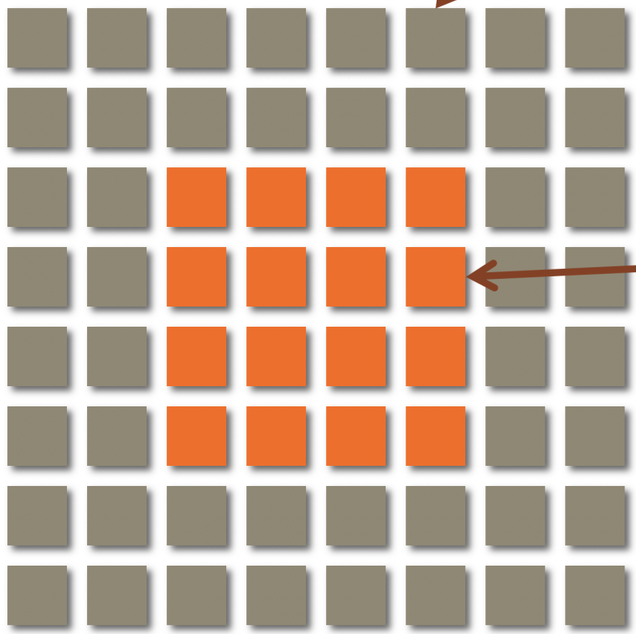
2nd order stencil

4th order stencil



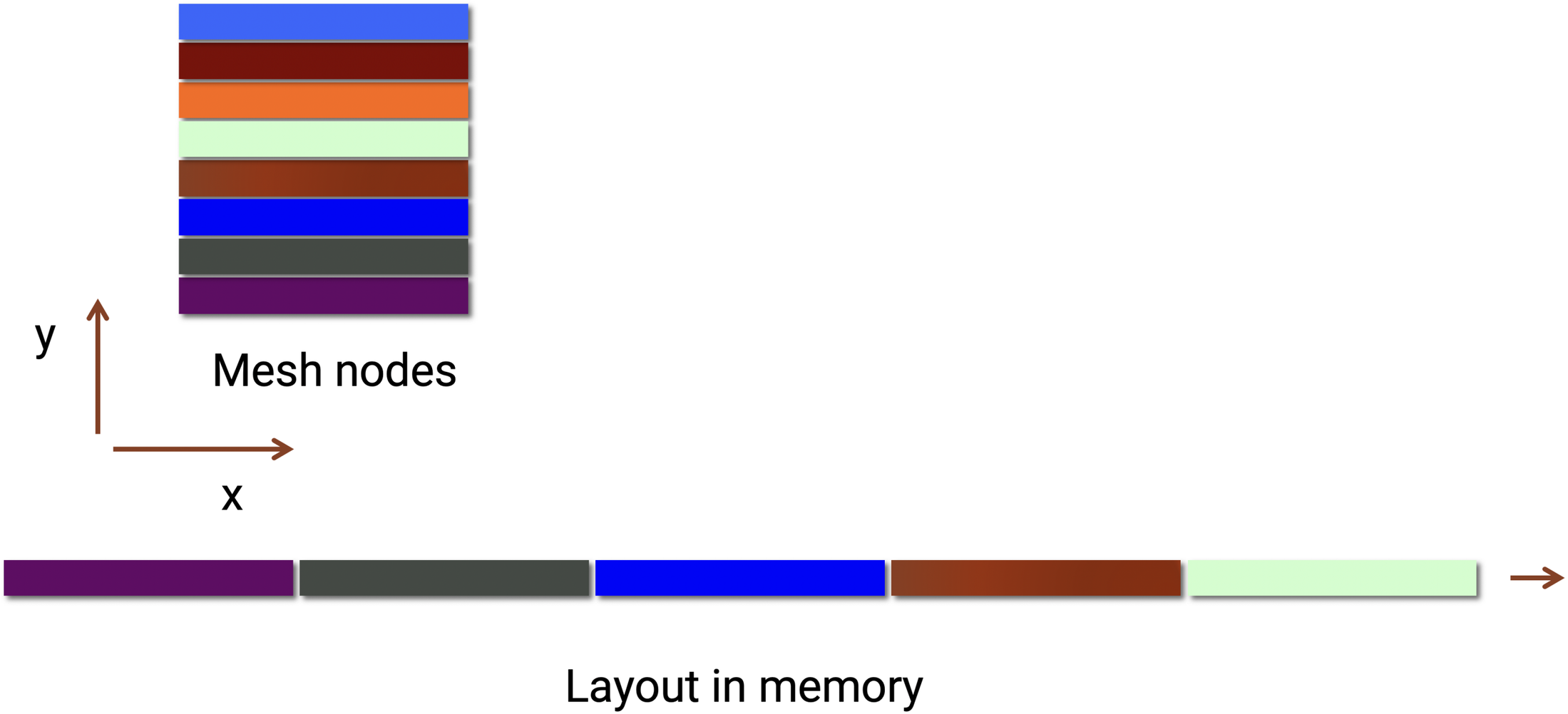
8th order stencil

4th order stencil

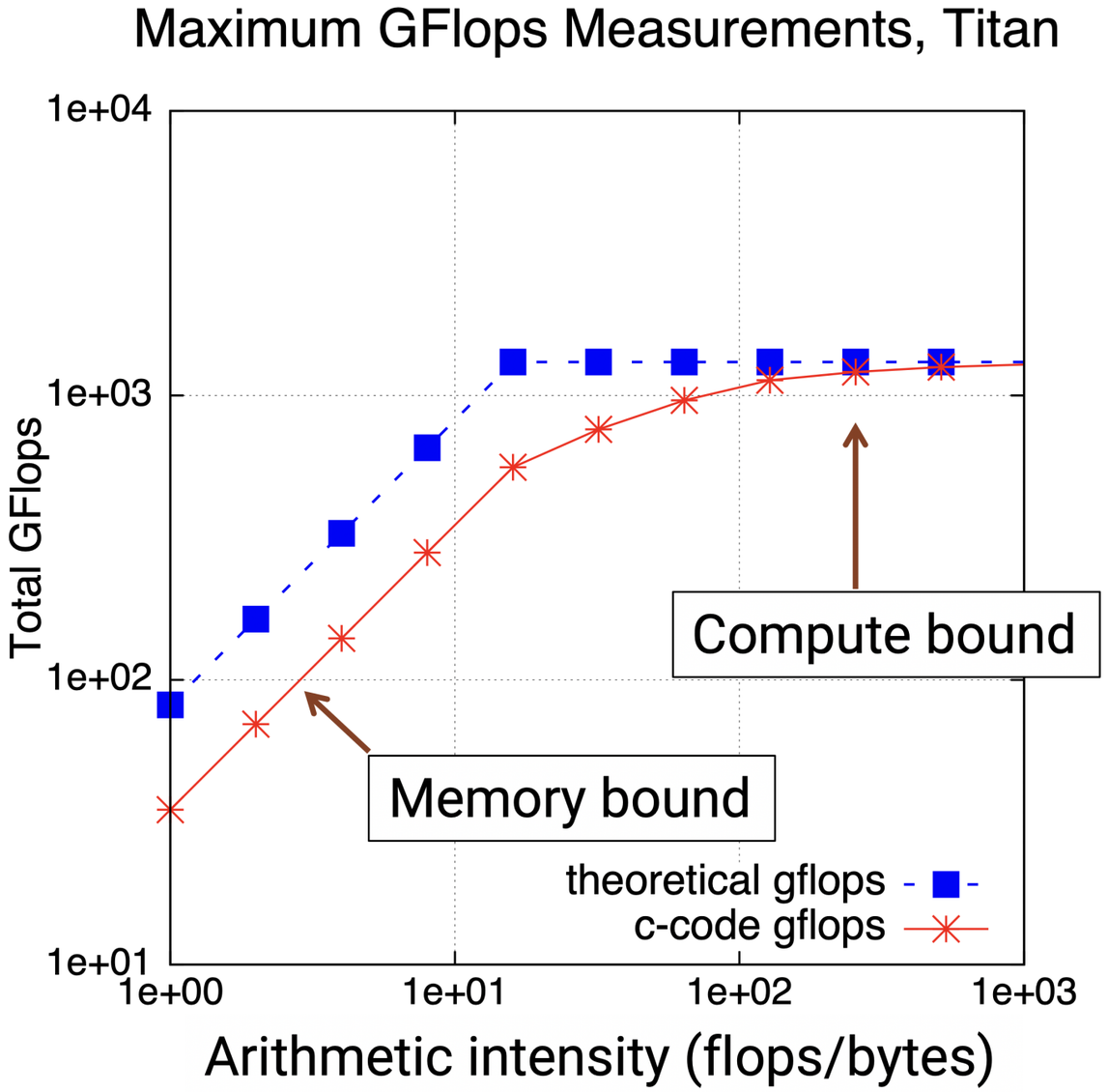


Goal of the homework

Implement a CUDA routine to update nodes inside the domain using a finite-difference centered stencil

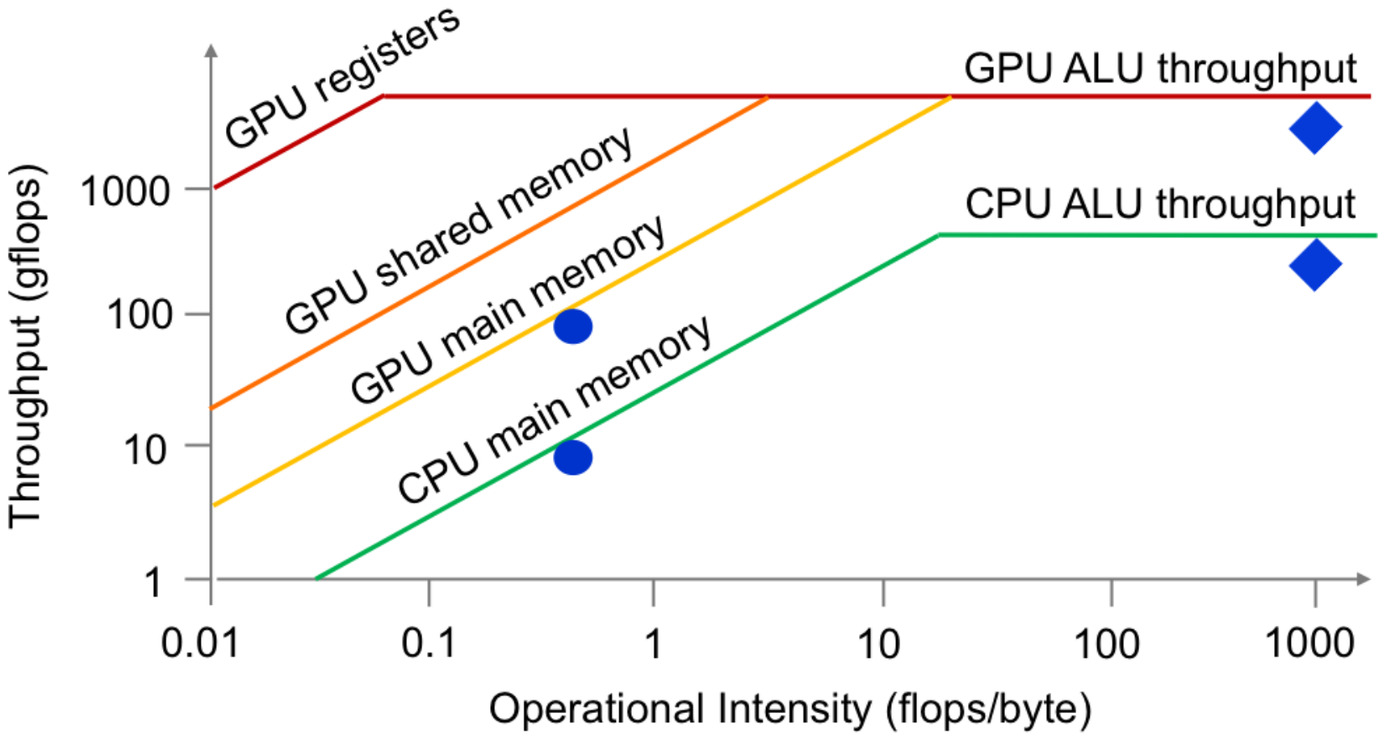

y
x
Mesh nodes
Layout in memory

NVIDIA
K20x GPU

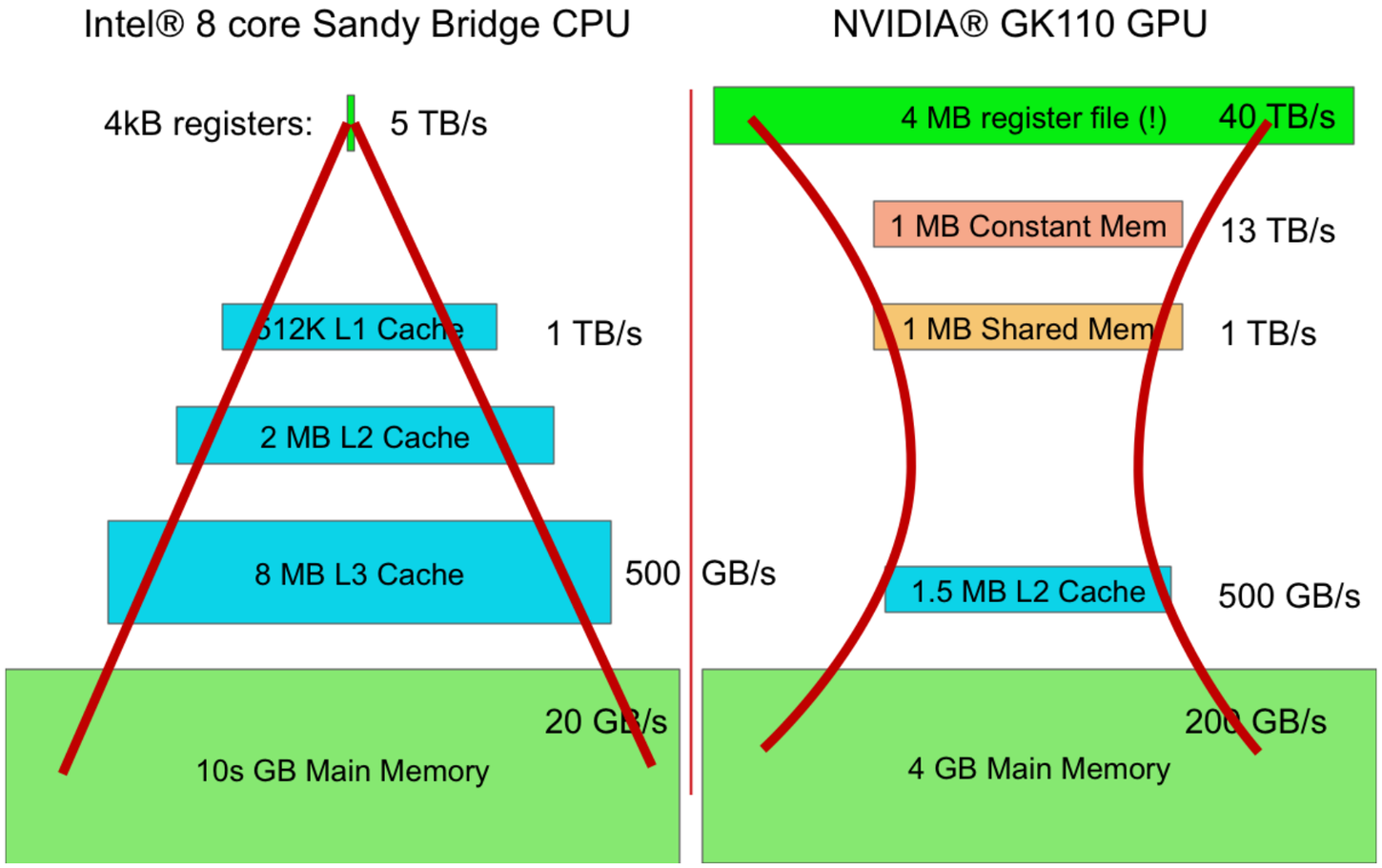


Roofline Design – Matrix kernels

- Dense matrix multiply
- Sparse matrix multiply



Where is my Memory?



Where are we in the roofline plot?

Order 2 stencil

How many flops?

How many words?

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

How many flops?

10 additions / multiplications

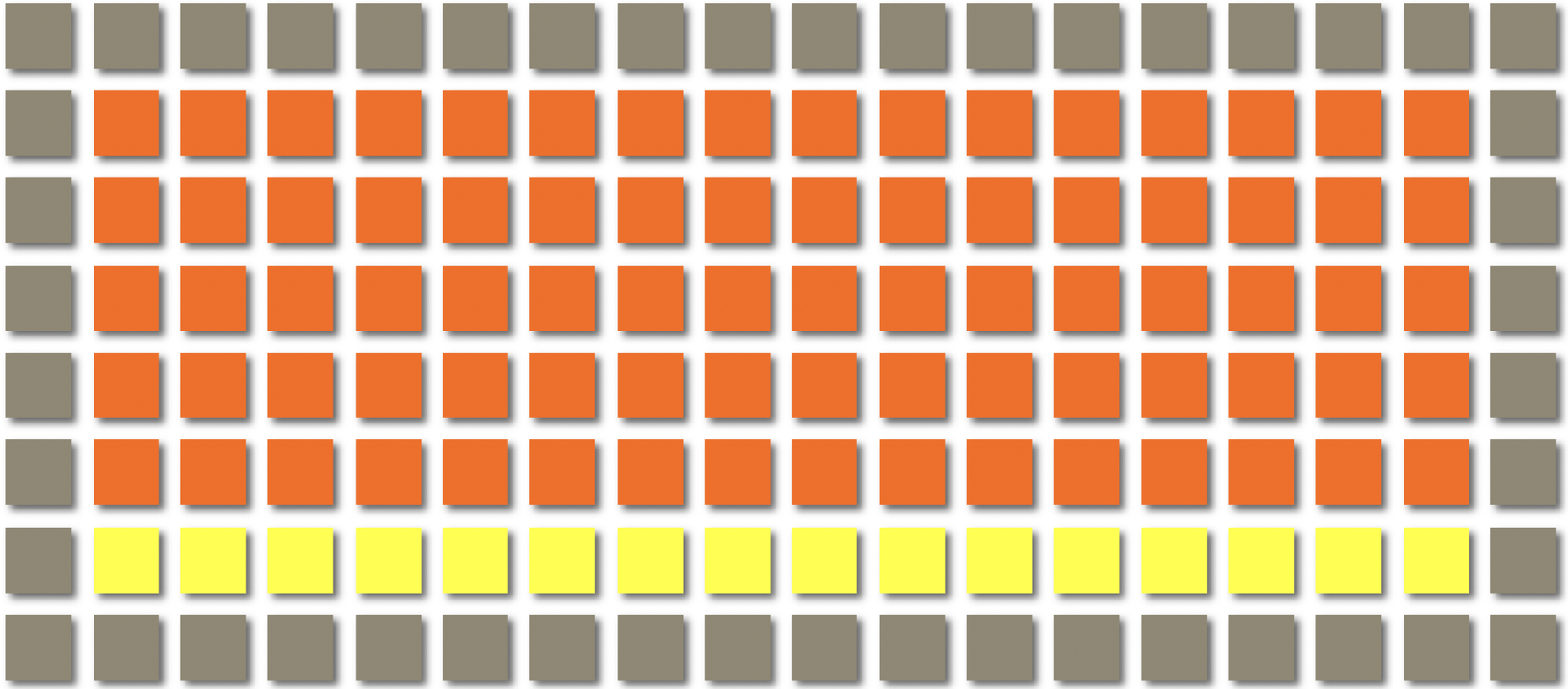
How many words?

- Read: 5
- Write: 1
- Total: 6

Two main ideas

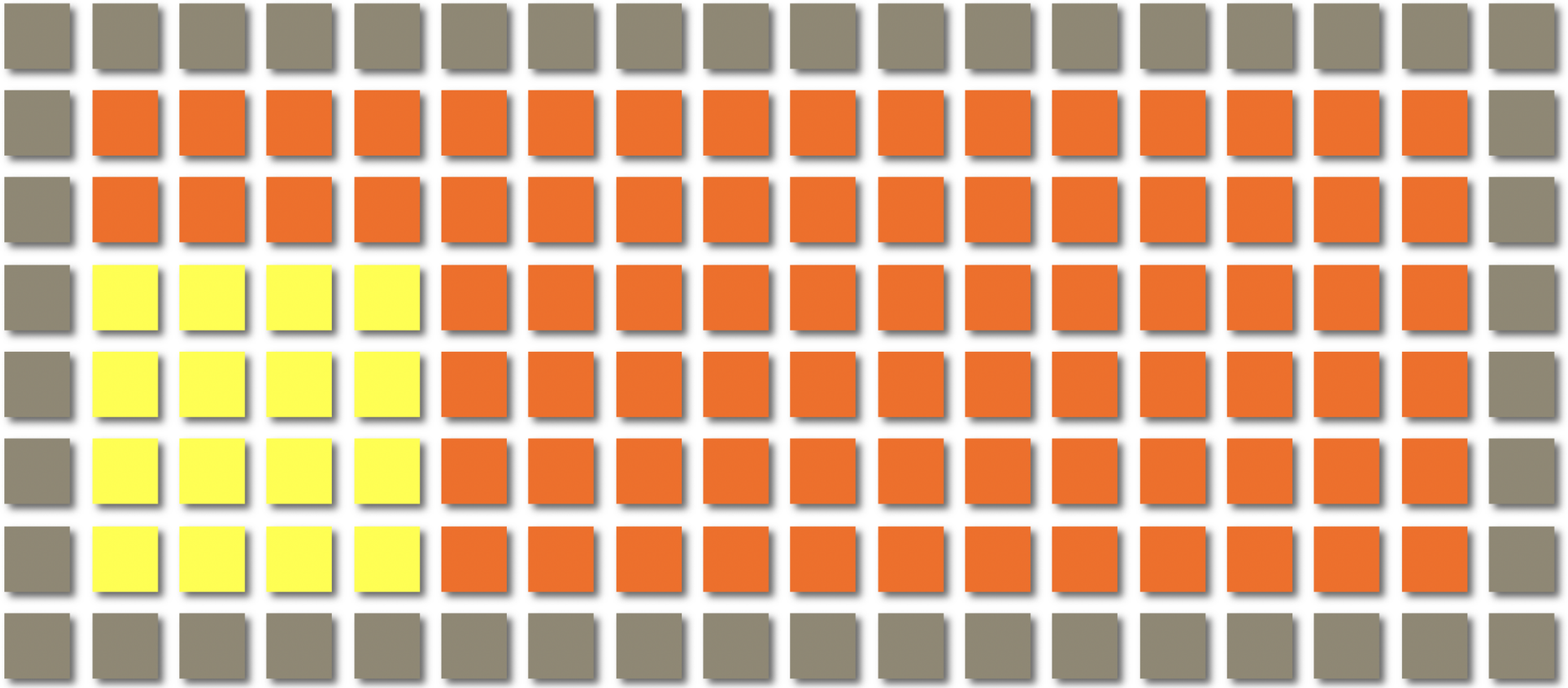
1. Use cache or shared memory
2. Memory accesses should be coalesced

Idea 1



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

Idea 2



Improvement

Ratio: flops/word

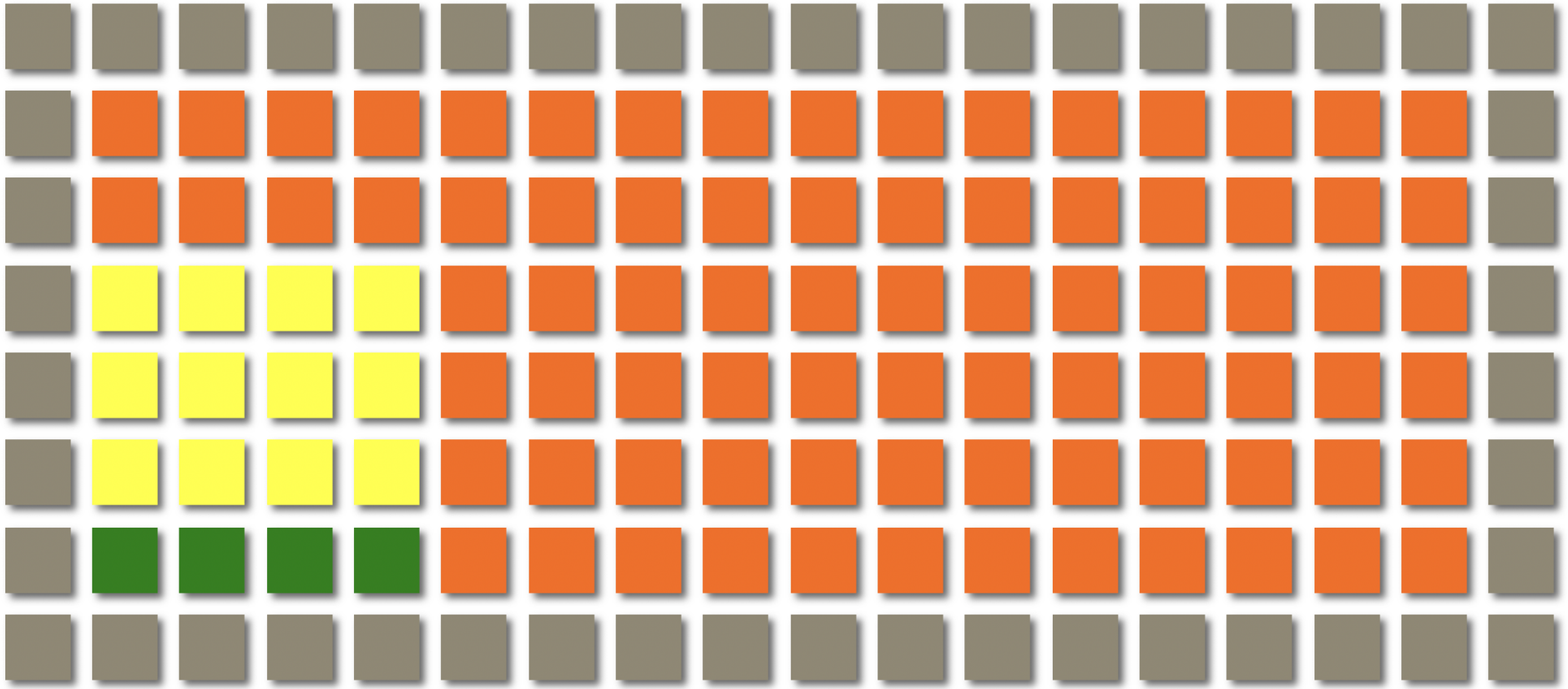
2.4 → **3.3**

Peak

For an $n \times n$ block:

- Memory traffic: $2n^2 + 4n$
- Flops: $10n^2$

Maximum intensity: 5 flops/words

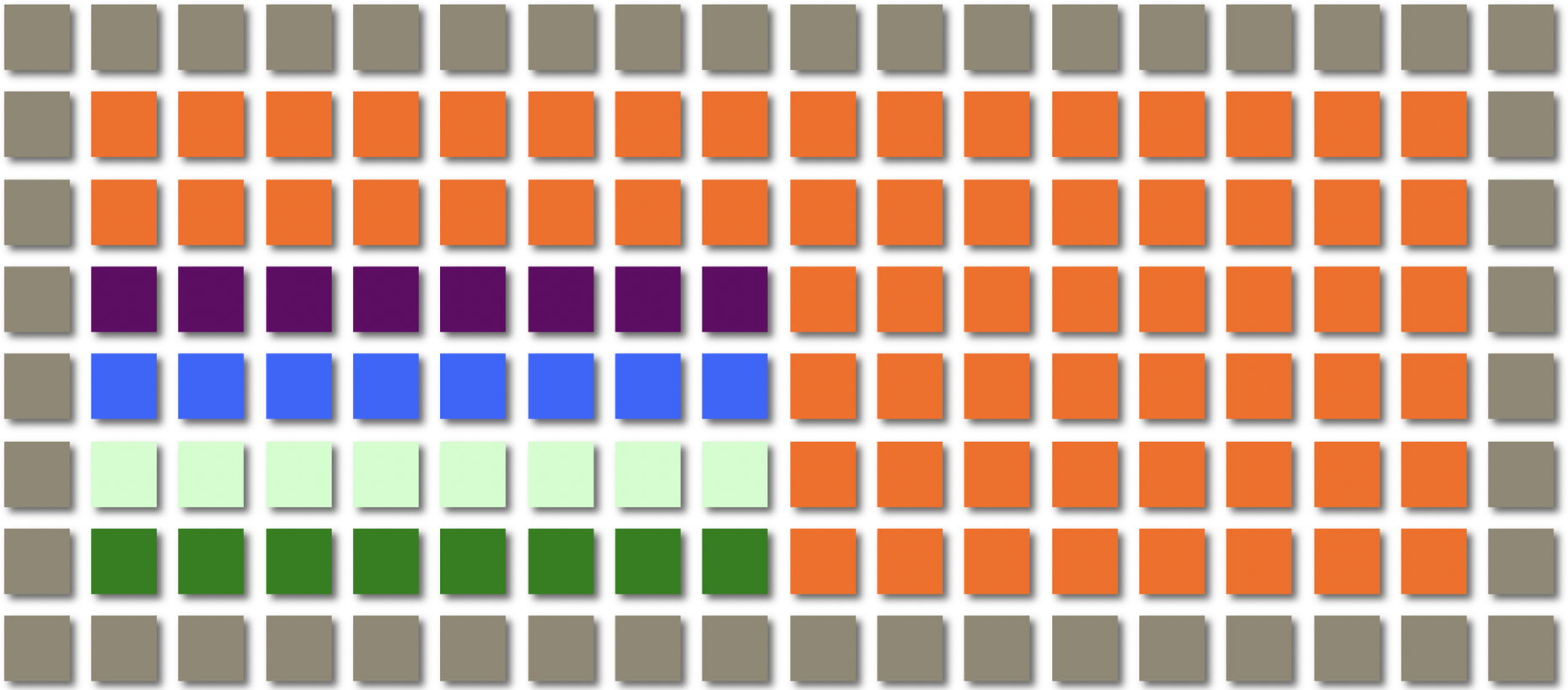


Only mesh nodes along x are contiguous.

This size must be a multiple of 32.

A warp must work on a chunk aligned along x.

Warp memory access



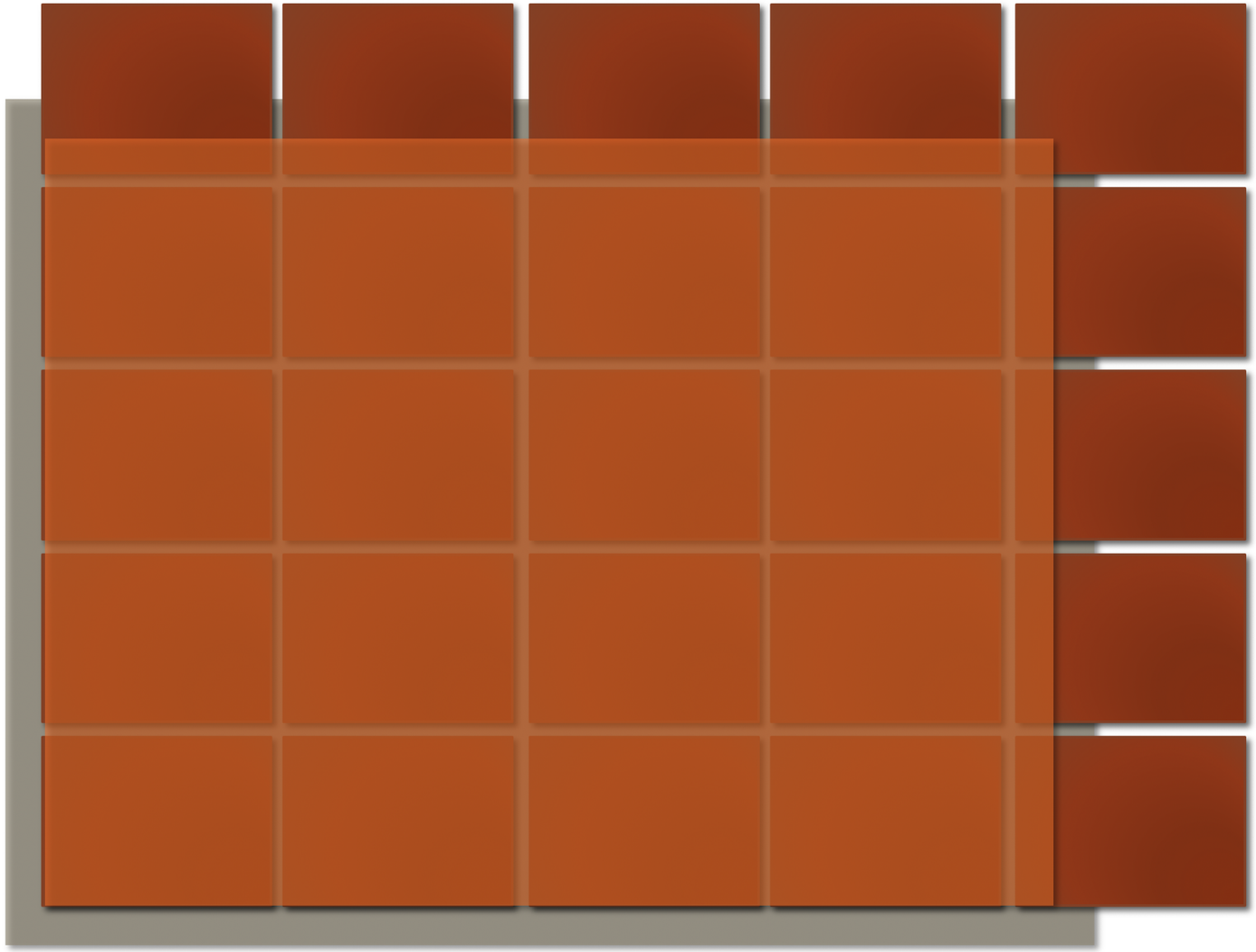
Thread block should be rectangular

Along x: dimension 64

Along y: determines number of threads in block

Example: 512 → dimension y = 8

Check your bounds



Shared memory algorithm

Step 1: all threads load data in shared memory

Load in shared memory

Step 2: threads inside the domain apply the stencil and write to the output array

Update inside nodes

Sample output

```
$ ./main -gsb
Order: 8, 4096x4096, 100 iterations
                 time (ms)     GBytes/sec
        CPU        2681.71        45.0444
     Global        48.7767        2476.51
      Block        37.4602        3224.65
     Shared        33.9051        3562.77

                    L2Ref           LInf          L2Err
     Global      0.447065    3.57628e-07    4.38207e-08
      Block      0.447065    3.57628e-07    4.38207e-08
     Shared      0.447065    3.57628e-07    4.38207e-08
```

If you compile with the −G option:

```
$ ./main -gsb
Order: 8, 4096x4096, 100 iterations
                 time (ms)     GBytes/sec
        CPU        2683.12        45.0206
     Global         208.855        578.373
      Block         111.014        1088.12
     Shared         136.626        884.138

                   L2Ref           LInf           L2Err
     Global       0.447065              0              0
      Block       0.447065              0              0
     Shared       0.447065              0              0
```

Order 2

```
$ ./main -gsb
Order: 2, 4096x4096, 100 iterations
                  time (ms)     GBytes/sec
        CPU          908.71        44.3104
     Global         37.6773        1068.69
      Block         35.3023        1140.59
     Shared         36.0975        1115.46

                    L2Ref           LInf          L2Err
     Global       0.418194    2.98023e-07    4.87715e-08
      Block       0.418194    2.98023e-07    4.87715e-08
     Shared       0.418194    2.98023e-07    4.87715e-08
```

What is the best blocking strategy?

Go as wide as you can along x while satisfying memory and concurrency constraints

Then loop along y reusing loaded data