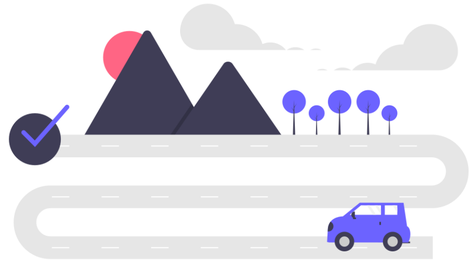
CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

"Everybody should learn to program a computer because it teaches you how to think." (Steve Jobs)

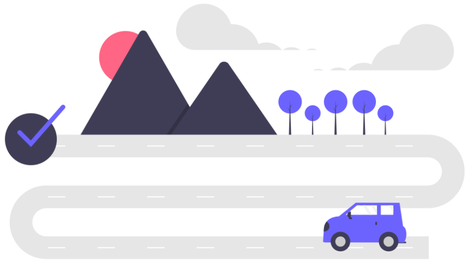
Stanford University

Recap



- How to compile CUDA codes: PTX and binary; CUDA compute capability
- CUDA profilers and cuda-memcheck
- L1 and L2 cache, coalesced memory access, cache lines
- Offset and strided memory access
- Shared memory, bank conflicts
- Matrix transpose example

NVIDIA guest lecture



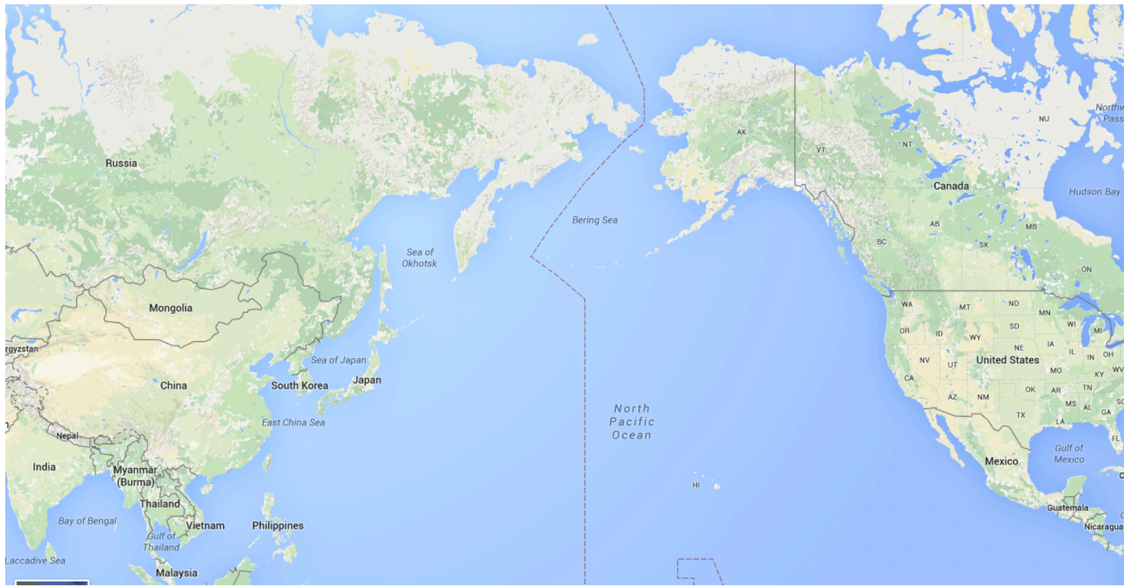
- Concurrency, occupancy, latency; Little's Law
- Branching and control flow
- Tensor cores
- Memory system; L1 and L2 cache; shared memory

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?

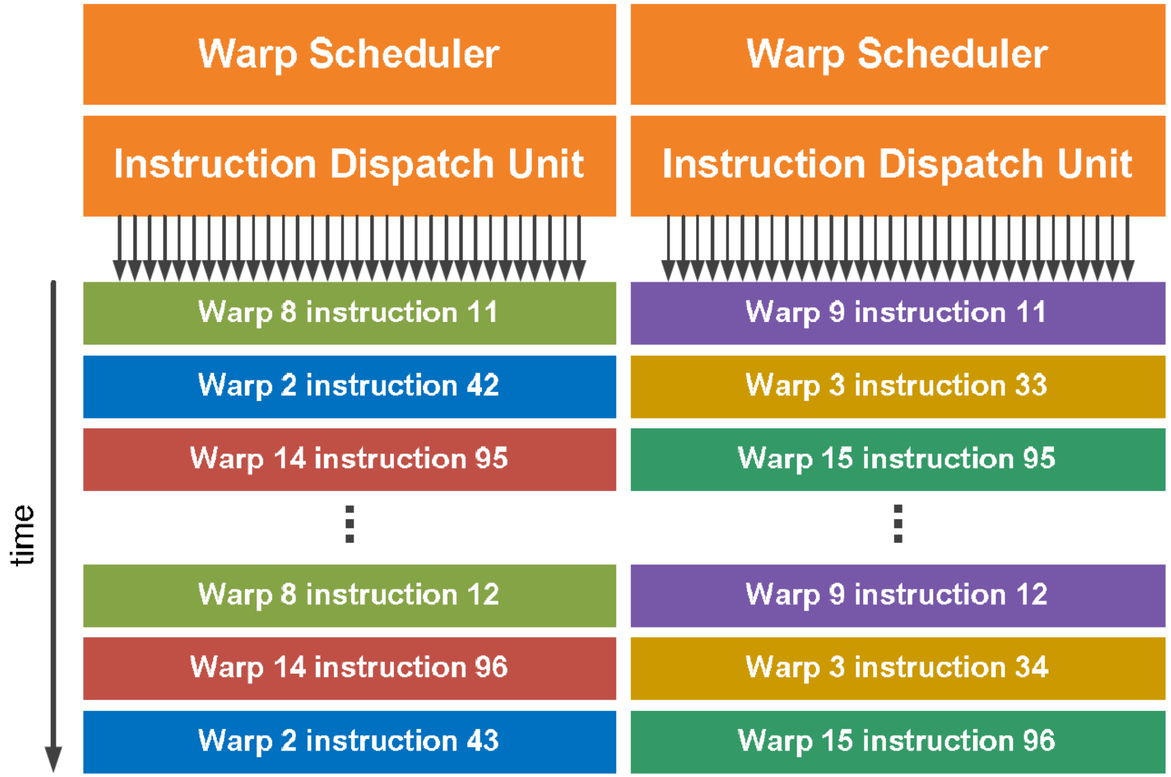


Hiding latency

- Concurrency is used to hide long latencies:
  - memory access
  - floating point units
  - any long sequence of operations

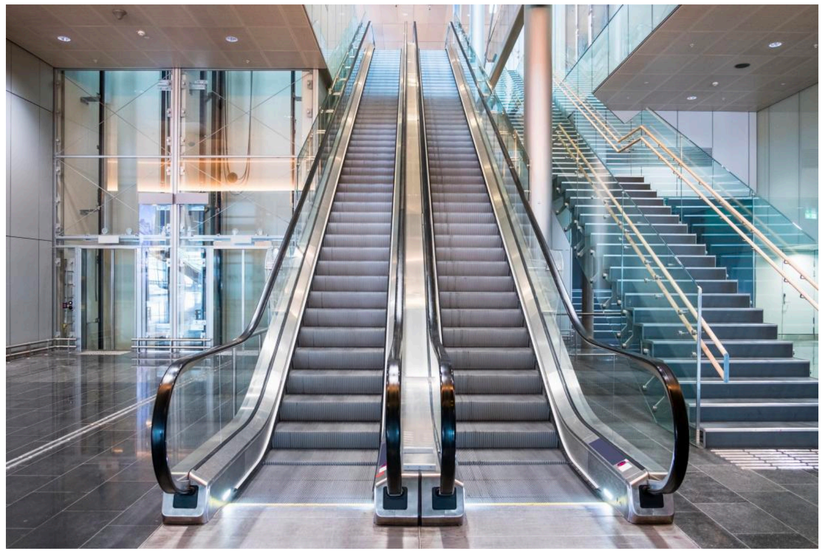
Hide latency through concurrency

Processors are optimized in the same way:



Little's Law

- Imagine an escalator:
  - 1 sec per step
  - 100 steps
- If the escalator is full, 1 person reaches the top every second.
- If there is only 1 person on the escalator, it takes 100 seconds to reach the top.



Little's Law

- This applies both to instruction throughput and memory requests.
- To achieve peak performance we need:

number of items in systems	≫	latency	x	throughput
item = instruction or memory request		time to process a single item		item/sec at maximum throughput
Escalator example:		100 sec	x	1 item/sec

The CME 213 Law of Latency

- The situation is worse on GPU systems where we have a very large number of concurrent computing units.
- In that case, the Law reads

number of items in systems $\gg$ no. of pipelines x latency x throughput

- This can be a very large number!

What concurrency means for a GPU

- On a GPU this means that we need:
  - if **compute bound:** a lot of *concurrent* arithmetic instructions (flops)
  - if **data traffic bound:** a lot of *concurrent* memory requests

How to maximize concurrency

Instruction-level parallelism:

- This is a more advanced topic.
- Requires a thread to issue **independent** instructions that can be pipelined.
- Will not work if there is a sequential dependency between instructions.

Have as many **live threads** as possible:

- We will look at this in more details.
- Make sure that as many threads as possible are able to fit on an SM.
- This depends on the **hardware limits** for an SM.

Example of instruction-level parallelism

The basic five-stage RISC architecture.

- Instruction fetch (IF)
- Instruction decode (ID)
- Instruction execute (Ex)
- Memory access (Mem)
- Register write-back (WB)

Instr. No.	Pipeline stage								
1	IF	ID	EX	MEM	WB				
2		IF	ID	EX	MEM	WB			
3			IF	ID	EX	MEM	WB		
4				IF	ID	EX	MEM	WB	
5					IF	ID	EX	MEM	WB
Clock cycle	1	2	3	4	5	6	7	8	9

Occupancy

- The relevant metric is occupancy.
- Occupancy:
  - Number of **concurrent thread blocks per multiprocessor.**
  - Divide by max threads per multiprocessor gives the occupancy as a **percentage.**
- 100%: you have maxed out the number of threads that can possibly run on an SM.
- Understanding occupancy requires knowing the **hardware resource limits.**

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

How can we (easily) make sense of this?

CUDA API

API can assist programmers in choosing a thread block size based on register and shared memory requirements.

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`

- Can provide an **occupancy prediction** based on the block size and shared memory usage of a kernel.
- This function reports occupancy in terms of the number of concurrent thread blocks per multiprocessor.

`cudaOccupancyMaxPotentialBlockSize`

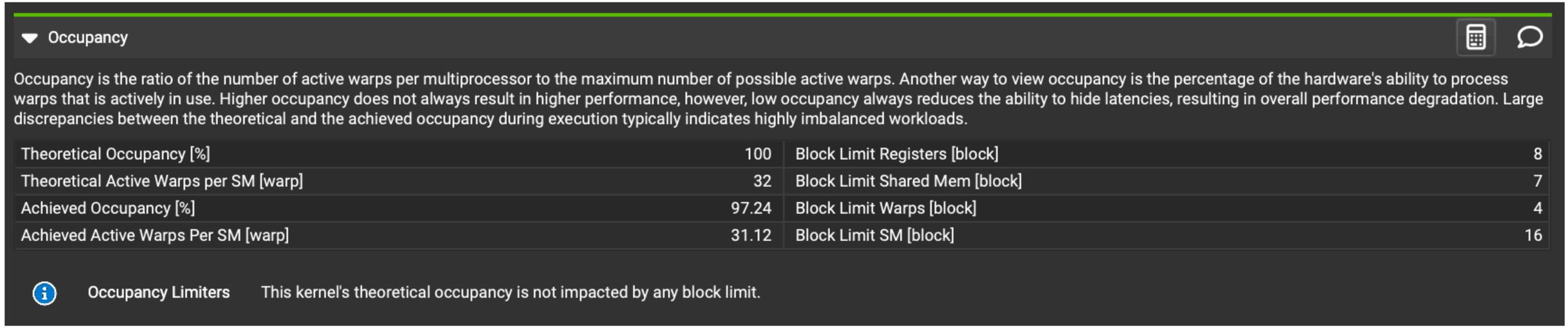
`cudaOccupancyMaxPotentialBlockSizeVariableSMem`

Heuristically calculate an execution configuration that achieves the maximum multiprocessor-level occupancy.

Occupancy calculator using Nsight Compute

- Occupancy calculator is available in Nsight Compute.
- See Nsight Compute Occupancy Calculator documentation.

Occupancy information



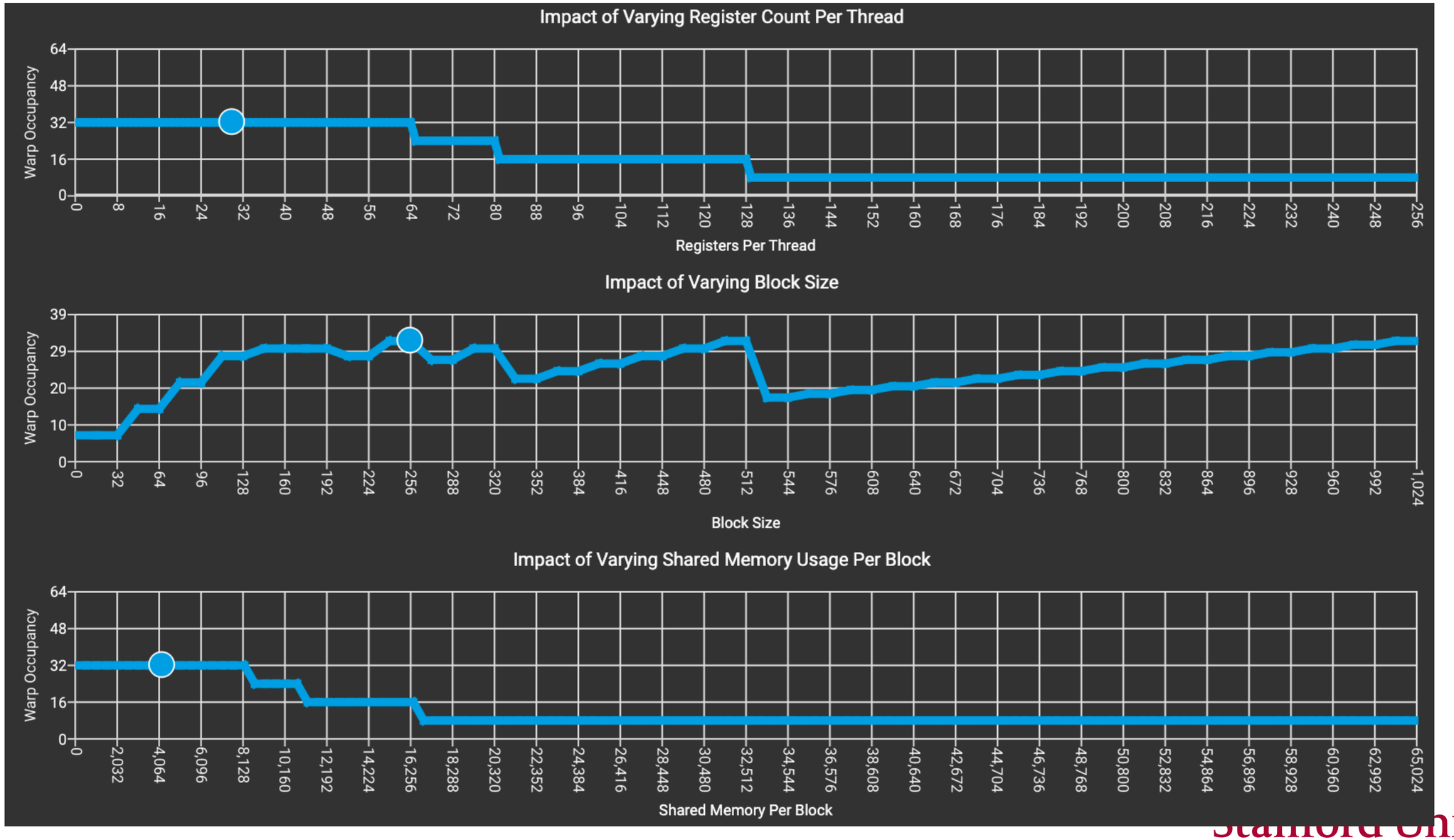
Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100	Block Limit Registers [block]	8
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	7
Achieved Occupancy [%]	97.24	Block Limit Warps [block]	4
Achieved Active Warps Per SM [warp]	31.12	Block Limit SM [block]	16

Occupancy Limiters This kernel's theoretical occupancy is not impacted by any block limit.

Occupancy graphs



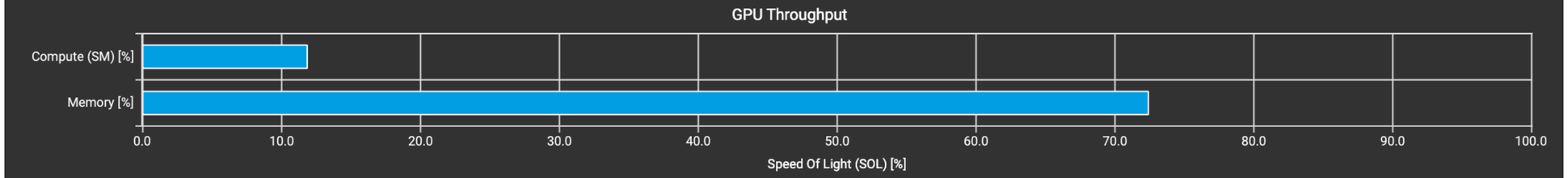
Example of Speed of Light statistics

GPU Speed Of Light Throughput — All

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor.

Compute (SM) Throughput [%]	11.84	Duration [msecond]	24.38
Memory Throughput [%]	72.40	Elapsed Cycles [cycle]	35465081
L1/TEX Cache Throughput [%]	13.15	SM Active Cycles [cycle]	35423232.56
L2 Cache Throughput [%]	16.63	SM Frequency [cycle/nsecond]	1.45
DRAM Throughput [%]	72.40	DRAM Frequency [cycle/nsecond]	6.49

High Memory Throughput Memory is more heavily utilized than Compute: Look at the Memory Workload Analysis section to identify the DRAM bottleneck. Check memory replay (coalescing) metrics to make sure you're efficiently utilizing the bytes transferred. Also consider whether it is possible to do more work per memory access (kernel fusion) or whether there are values you can (re)compute.



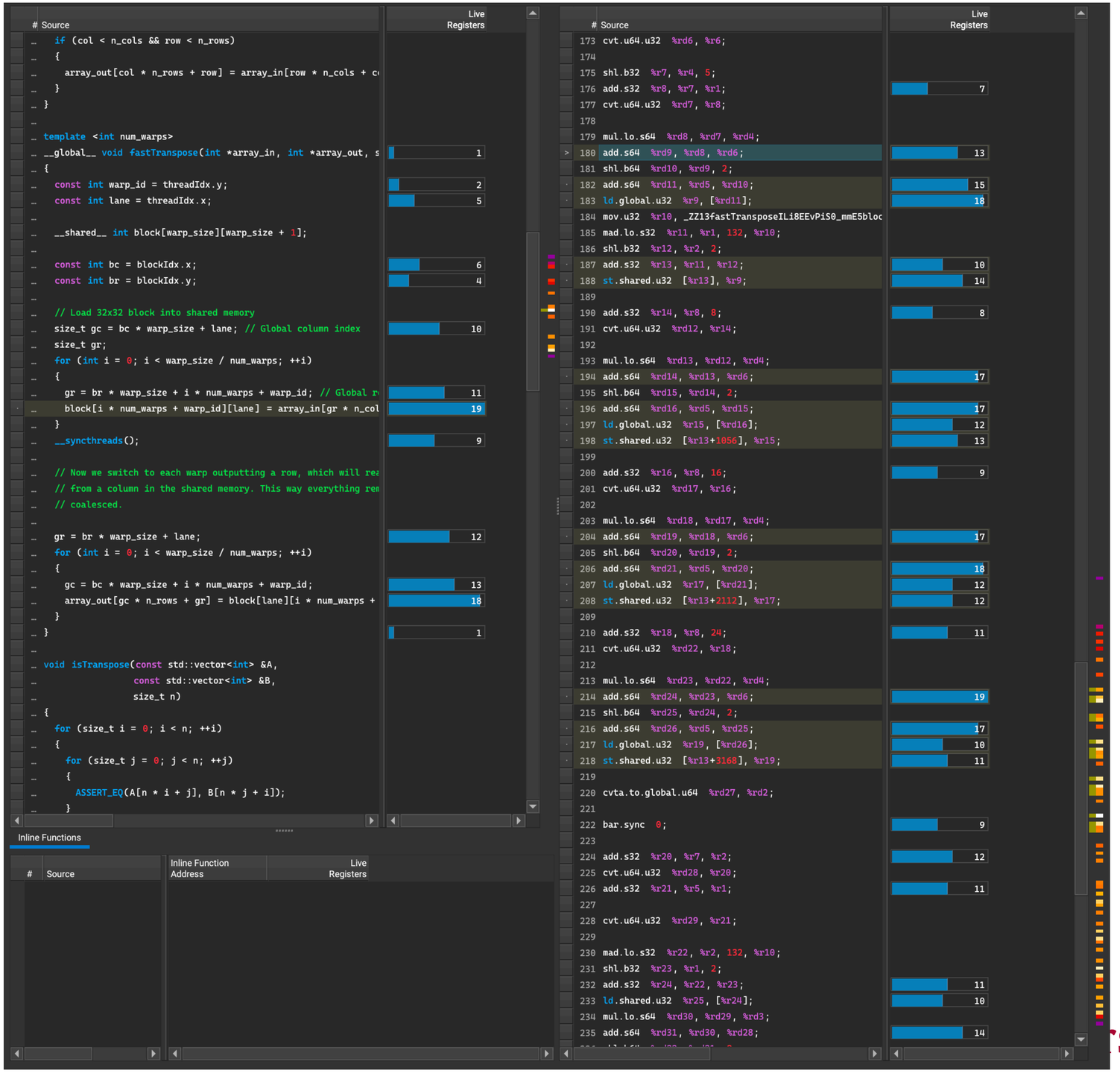
Compute Throughput Breakdown

SM: Inst Executed Pipe Lsu [%]	11.84
SM: Issue Active [%]	5.67
SM: Inst Executed [%]	5.67
SM: Mio Inst Issued [%]	4.60
SM: Pipe Alu Cycles Active [%]	4.11
SM: Pipe Fma Cycles Active [%]	3.62
SM: Mio Pq Write Cycles Active [%]	2.63
SM: Mio Pq Read Cycles Active [%]	2.63
SM: Inst Executed Pipe Adu [%]	1.97
SM: Mio2rf Writeback Active [%]	1.97
SM: Inst Executed Pipe Cbu Pred On Any [%]	0.33
IDC: Request Cycles Active [%]	0
SM: Inst Executed Pipe Xu [%]	0
SM: Inst Executed Pipe Uniform [%]	0
SM: Inst Executed Pipe Tex [%]	0
SM: Inst Executed Pipe Ipa [%]	0
SM: Inst Executed Pipe Fp16 [%]	0
SM: Pipe Fp64 Cycles Active [%]	0
SM: Pipe Shared Cycles Active [%]	0
SM: Pipe Tensor Cycles Active [%]	0

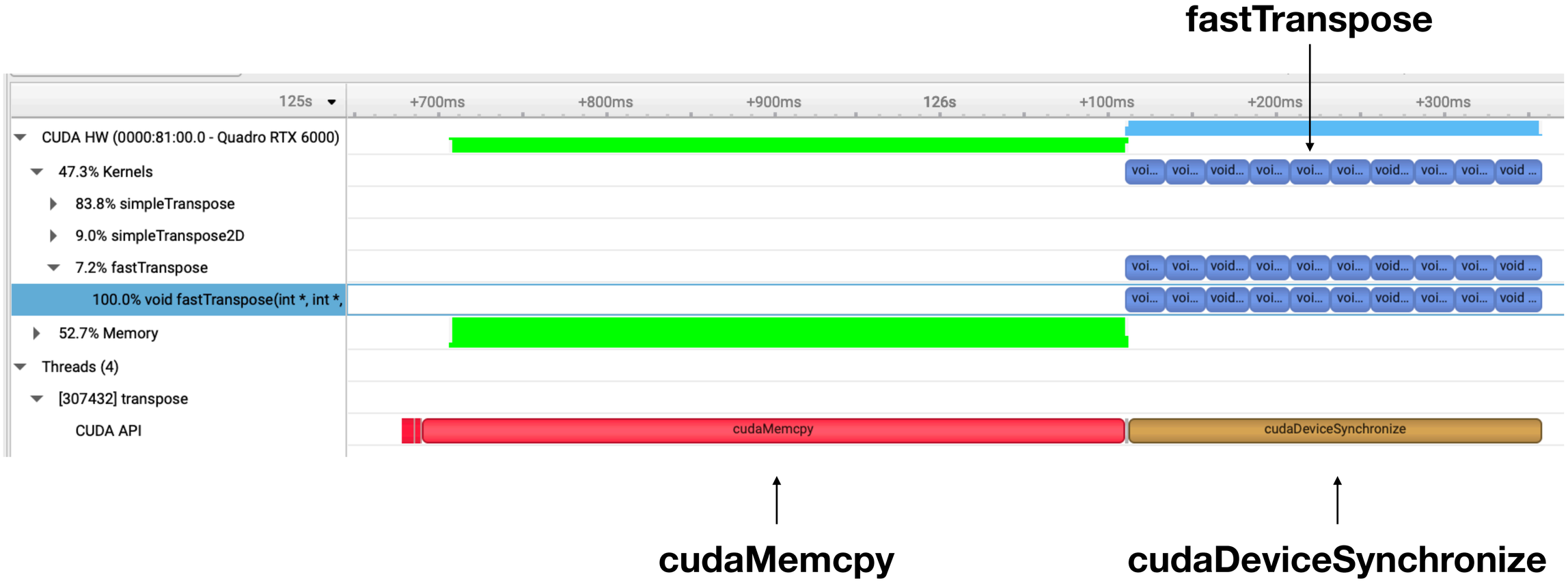
Memory Throughput Breakdown

DRAM: Cycles Active [%]	72.40
DRAM: Dram Sectors [%]	42.59
L2: T Sectors [%]	16.63
L1: Lsuin Requests [%]	11.84
L2: Xbar2lts Cycles Active [%]	10.33
L2: Lts2xbar Cycles Active [%]	8.31
L2: D Sectors Fill Device [%]	8.31
L2: D Sectors [%]	8.31
L1: M L1tex2xbar Req Cycles Active [%]	6.58
L1: Data Pipe Lsu Wavefronts [%]	6.12
L1: M Xbar2l1tex Read Sectors [%]	5.26
L2: T Tag Requests [%]	4.16
L1: Lsu Writeback Active [%]	3.29
L1: Data Bank Writes [%]	1.97
L1: Data Bank Reads [%]	1.97
L1: Texin Sm2tex Req Cycles Active [%]	0.04
L1: F Wavefronts [%]	0.00
L2: D Sectors Fill Sysmem [%]	0
L1: Data Pipe Tex Wavefronts [%]	0
L1: Tex Writeback Active [%]	0
L2: D Atomic Input Cycles Active [%]	0

Source and PTX code analysis



Example of Nsight Systems overview



Occupancy calculator (deprecated)

Occupancy calculator spreadsheet (deprecated)

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	32	
User Shared Memory Per Block (bytes)	2048	

(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)	2048	65536	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

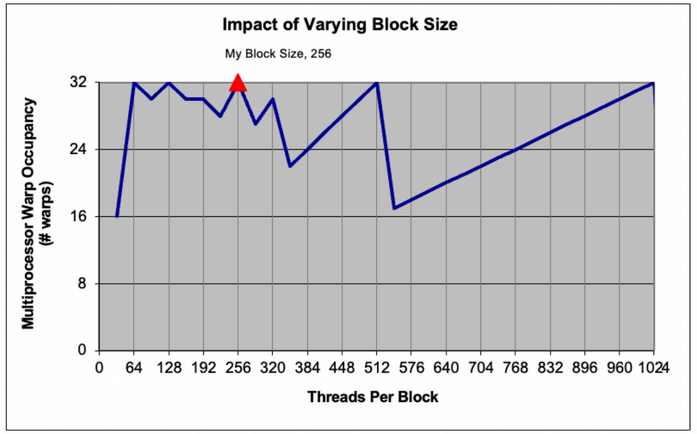
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	**4**	**8**	**32**
Limited by Registers per Multiprocessor	8		
Limited by Shared Memory per Multiprocessor	32		

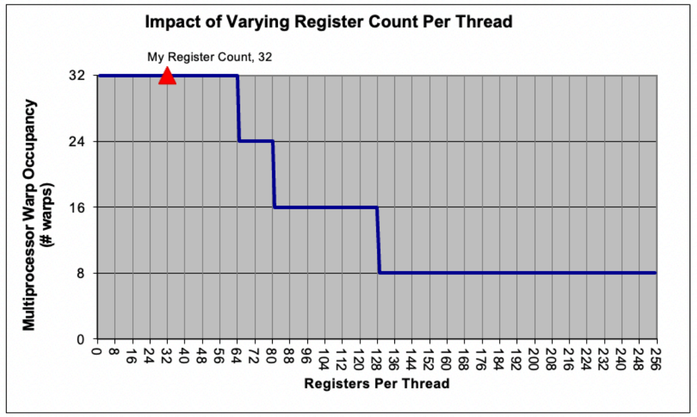
Note: Occupancy limiter is shown in orange

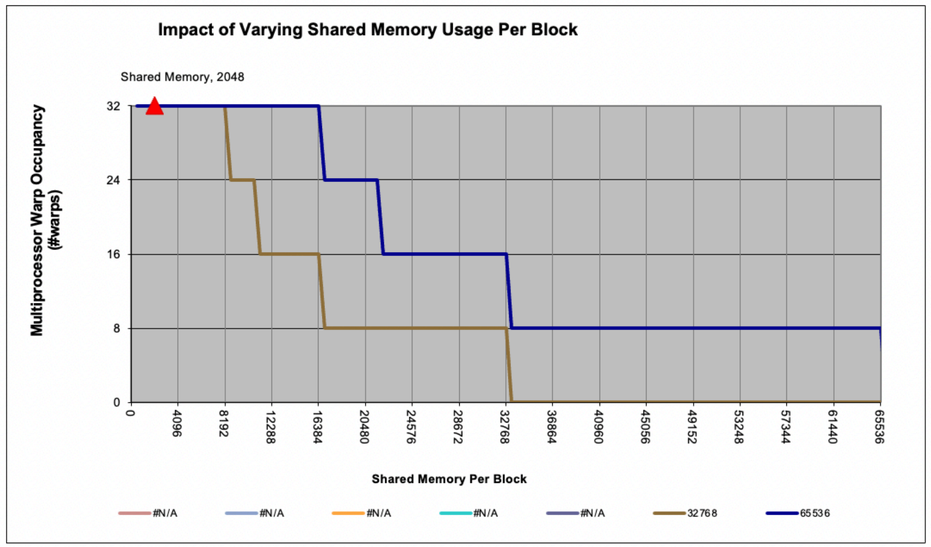
Physical Max Warps/SM = 32

Occupancy = 32 / 32 = 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 figure out the number of registers used by a kernel

Register usage is reported by the `--ptxas-options=-v` compiler option.

Example:

- Command line

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

Output

Output on Google Colab, Tesla K80

```
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z13fastTransposeILi8EEvPiS0_mm' for 'sm_52'
ptxas info    : Function properties for _Z13fastTransposeILi8EEvPiS0_mm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 24 registers, 4224 bytes smem, 352 bytes cmem[0]
ptxas info    : Compiling entry function '_Z17simpleTranspose2DPiS_mm' for 'sm_52'
ptxas info    : Function properties for _Z17simpleTranspose2DPiS_mm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 14 registers, 352 bytes cmem[0]
ptxas info    : Compiling entry function '_Z15simpleTransposePiS_mm' for 'sm_52'
ptxas info    : Function properties for _Z15simpleTransposePiS_mm
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 20 registers, 352 bytes cmem[0]
```

Occupancy calculator demo

1.) Select Compute Capability (click):	**5.2**	(Help)
1.b) Select Shared Memory Size Config (bytes)	**98304**	
1.d) Select Global Load Caching Mode	**L1+L2 (ca)**	
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	24	
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	**2048**	(Help)
Active Warps per Multiprocessor	**64**	
Active Thread Blocks per Multiprocessor	**8**	
Occupancy of each Multiprocessor	**100%**	

Physical Limits for GPU Compute Capability:	**5.2**
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	32768
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	98304
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	2
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	64	8
Registers (Warp limit per SM due to per-warp reg count)	8	42	10
Shared Memory (Bytes)	4352	49152	22

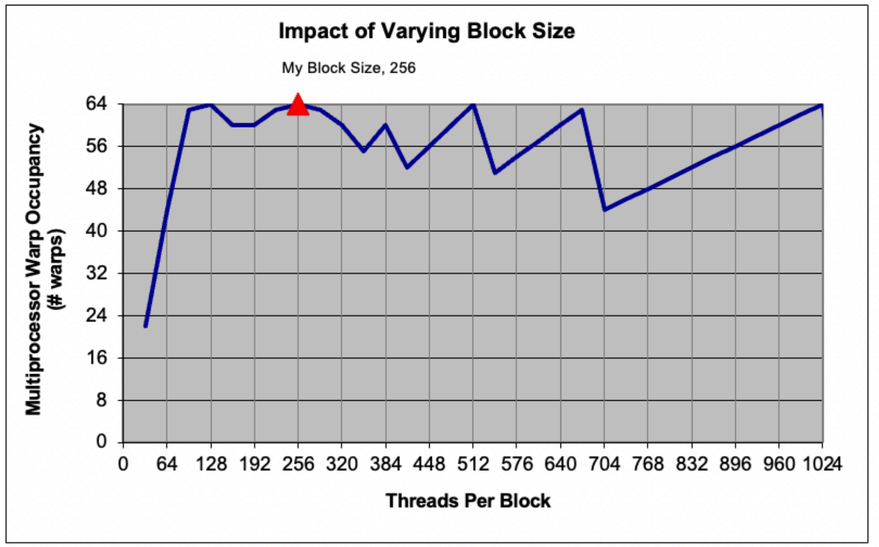
Note: SM is an abbreviation for (Streaming) Multiprocessor

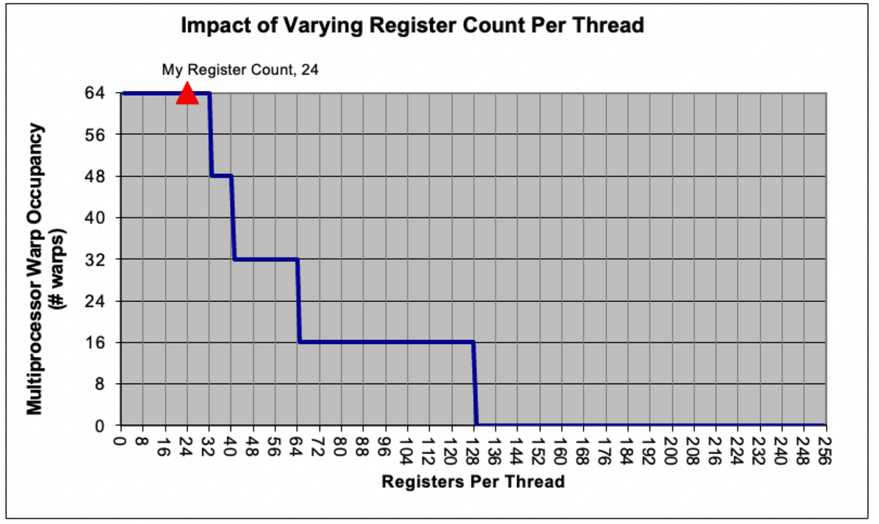
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	**8**	**8**	**64**
Limited by Registers per Multiprocessor	10		
Limited by Shared Memory per Multiprocessor	22		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

represent the range of possible block sizes, register counts, and shared memory allocation.





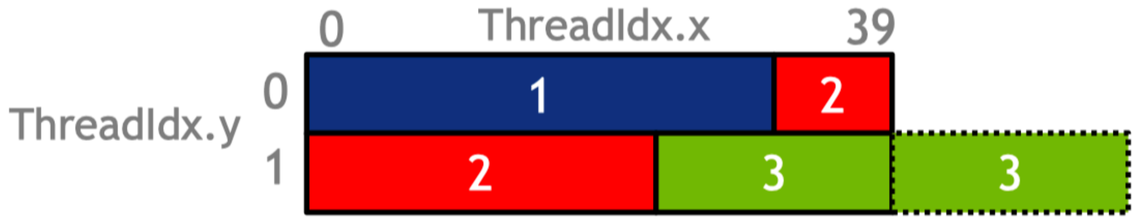
Optimization: avoid branching

- All threads in a warp are expected to issue the same instruction every cycle.
- What happens if there is a branch in your code:
  - `if/while` ?
  - **`for` loop with varying number of iterations.**

Implementation of branching in CUDA

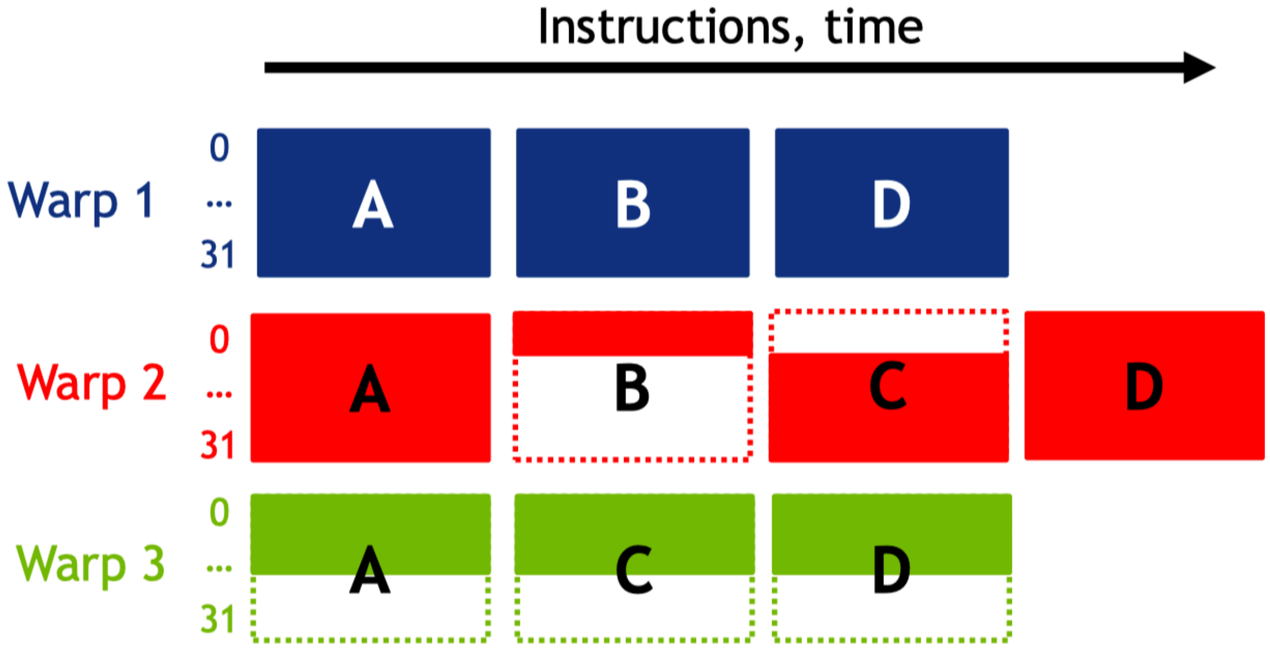
- Up to Pascal
- **Single program counter** shared amongst all 32 threads
- How it works:
  - Active mask
  - Specifies which threads of the warp are active at any given time
- Execution becomes **serialized**

Example of branching



- **Warp 2 has a branch**
- **Warp 1 and 3 do not branch**

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```

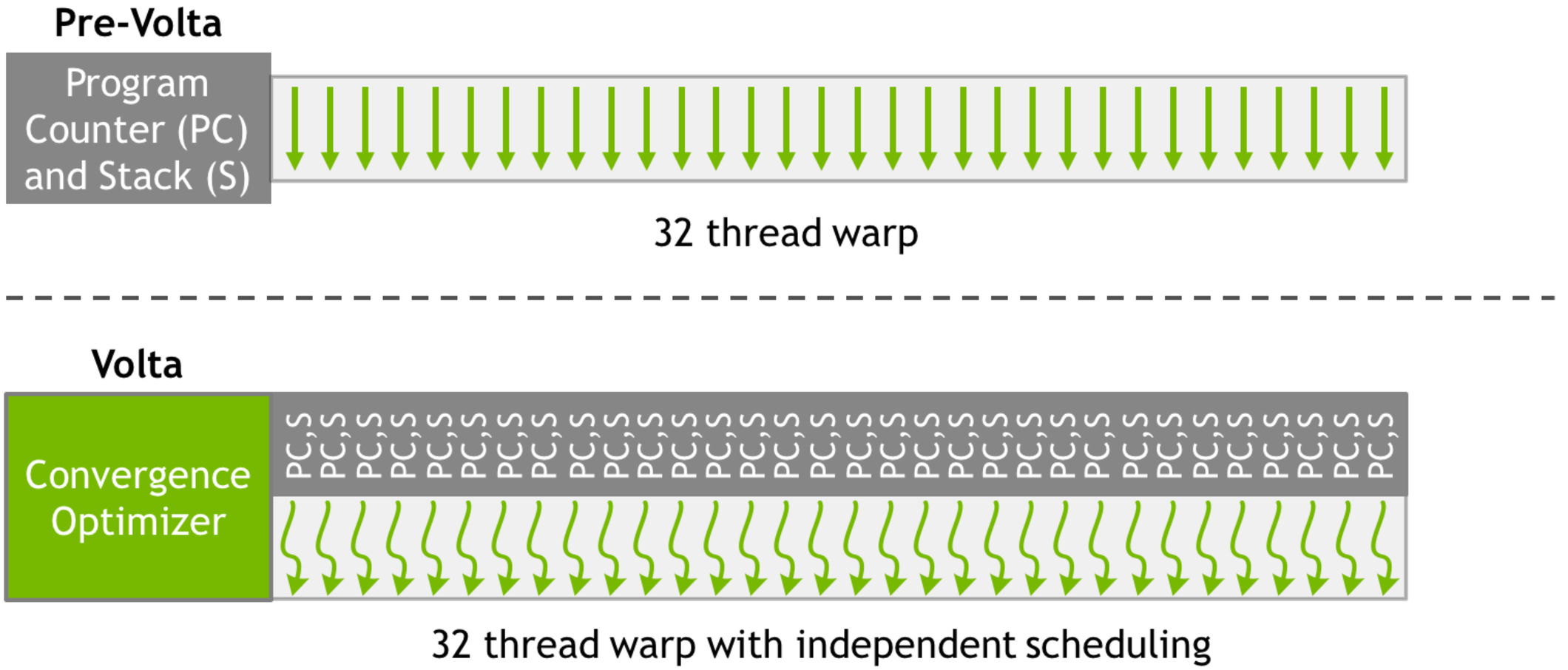


- **Not all threads in warp 3 have work to do**
- **Example: n_threads= 96 but matrix size = 80.**

Volta and later!

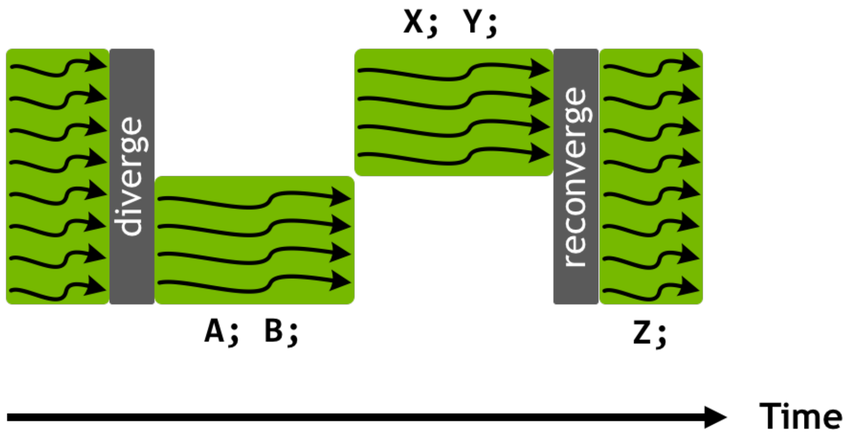
- Mitigates this problem a bit.
- But fundamentally performance hit is mostly the same.
- **Independent Thread Scheduling** allows full concurrency between threads, regardless of warp.
- The GPU maintains **execution state per thread,** including a program counter and call stack, and can **yield execution at a per-thread granularity,** either to make better use of execution resources or to allow one thread to wait for data to be produced by another.
- **A schedule optimizer** determines how to group active threads from the same warp together into **SIMT units.**
- This retains the high throughput of SIMT execution as in prior NVIDIA GPUs, but with much more flexibility: **threads can now diverge and reconverge at sub-warp granularity.**

Pre and post Volta



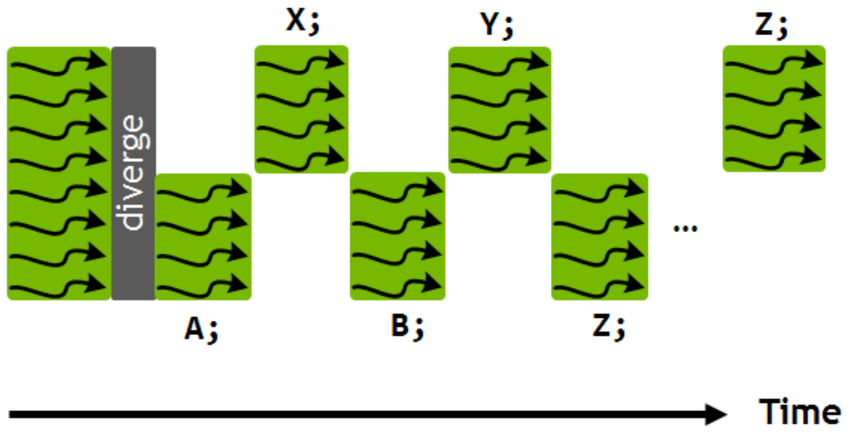
Example of branch execution

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



Pre-Volta

Threads diverge and reconverge at sub-warp granularity.



Volta

Extra level of flexibility

- Gives more flexibility in programming and performance.
- Avoids certain deadlock bugs due to divergence.

Correct execution of advanced parallel algorithms

- 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.

Example: CUDA programming with `mutex`

```
__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);
}
```

- CUDA with `mutex`.
- **Correct execution requires independent thread scheduling.**
- Volta ensures that the thread holding the lock is able to make progress.
- Starvation-free implementation

Homework 4

Heat diffusion

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

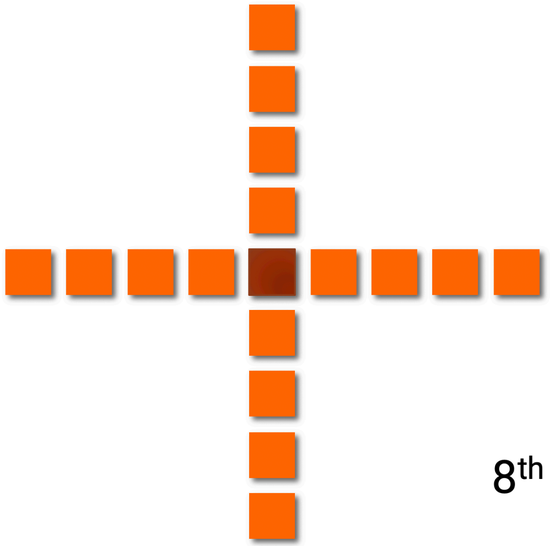
After finite-difference discretization:

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

Stencils

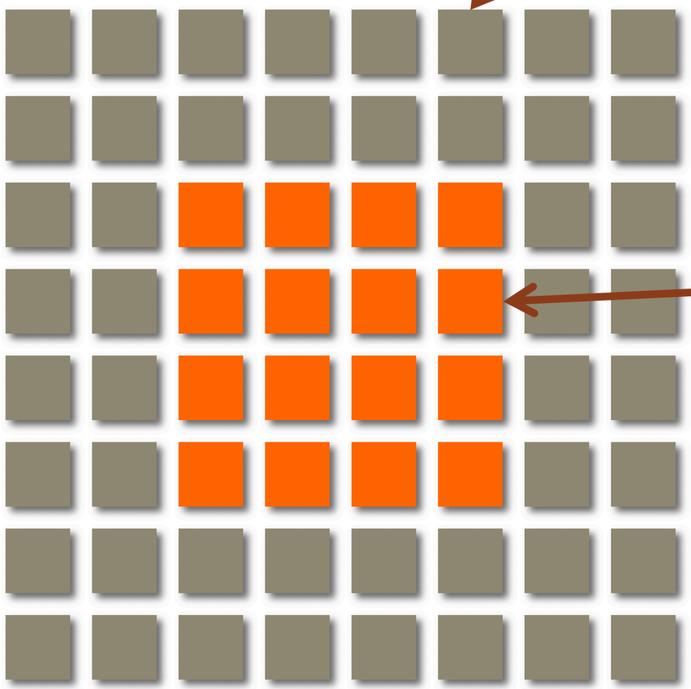
2nd order stencil

4th order stencil



8th order stencil

4^{th} order stencil



Updated in a separate routine; boundary conditions are used

Node is updated using the stencil.

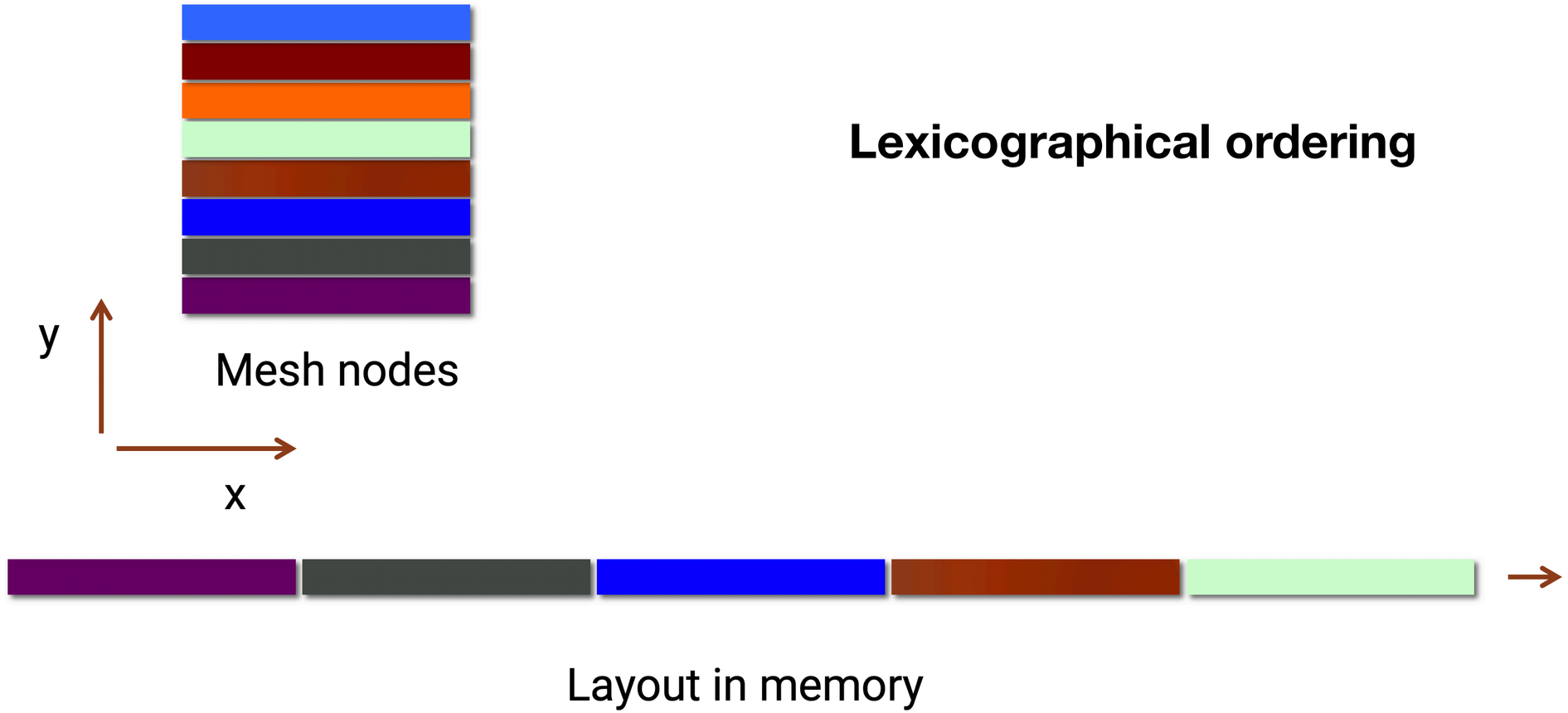
Goals of homework

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

Benchmarks:

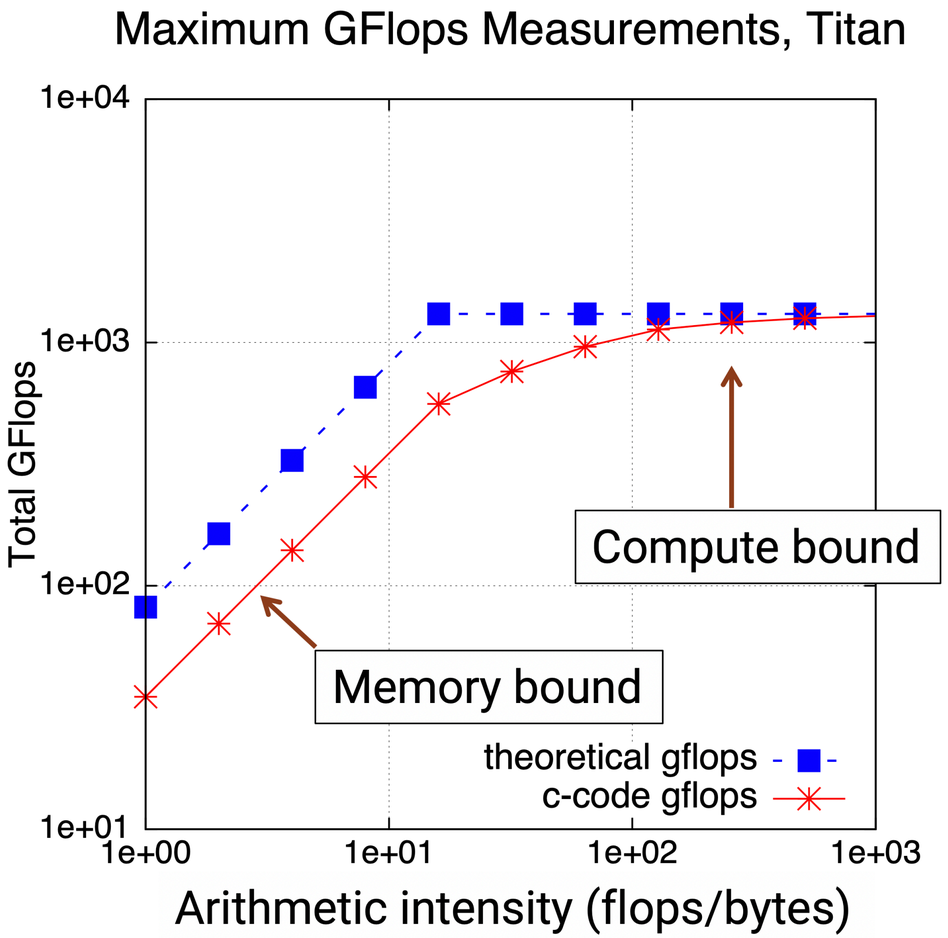
- Different stencil orders
- Algorithms with different memory access patterns
- Shared memory algorithm

Layout of data in memory



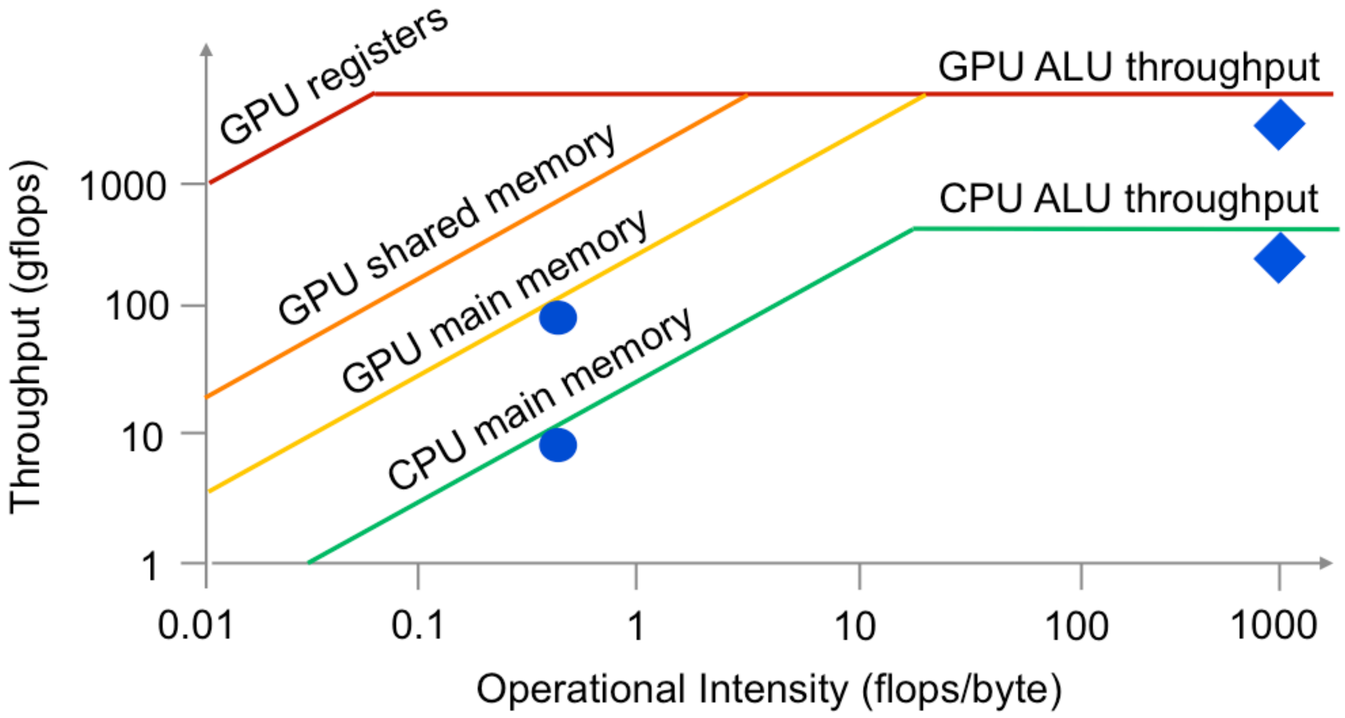
Memory vs compute bound kernels

NVIDIA
K20x GPU

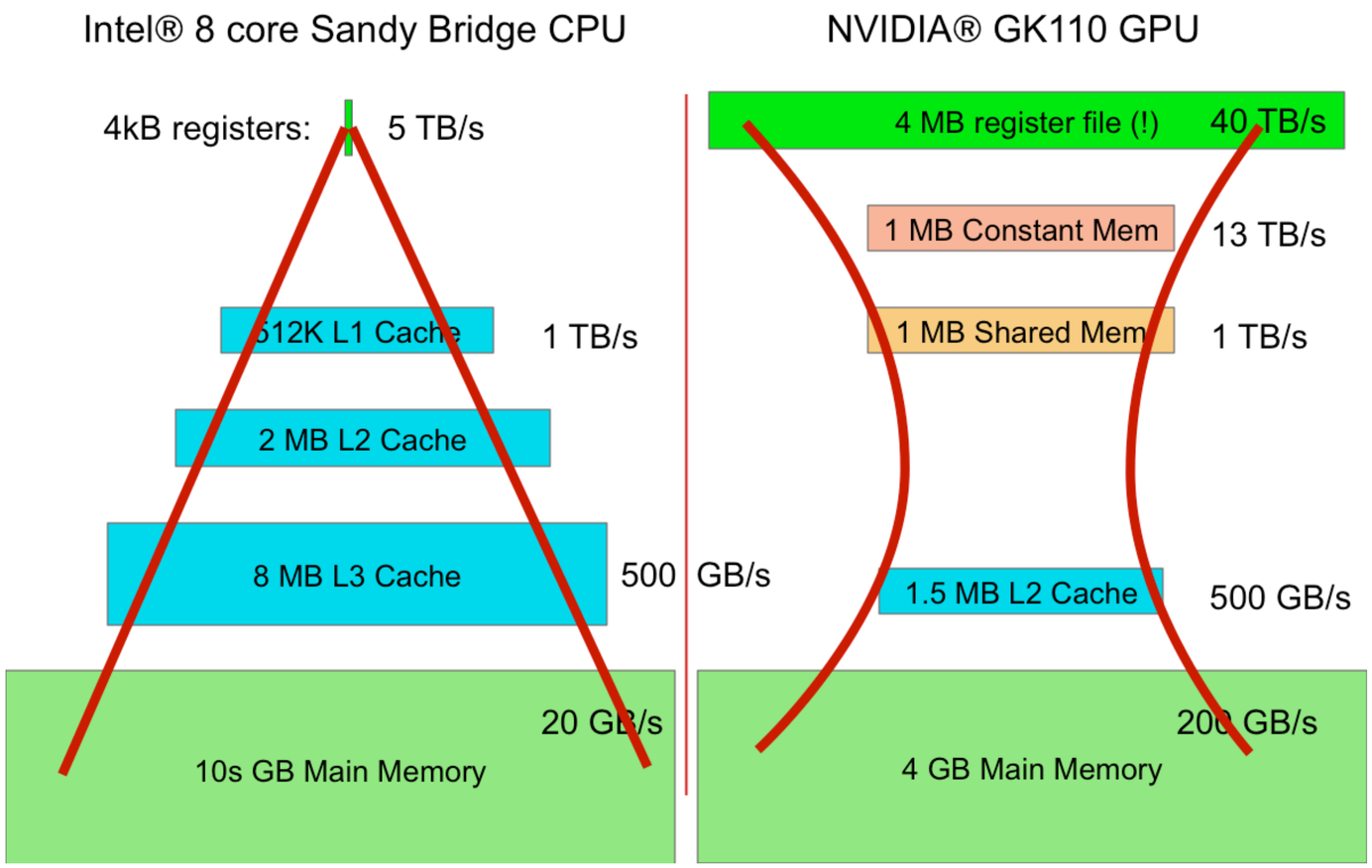


Example: performance analysis of matrix kernels
Roofline design

- Dense matrix multiply
- Sparse matrix multiply



Bottleneck analysis depends on the memory hierarchy



Algorithm analysis

- Where are we in the roofline design plot?
- How can we count the number of flops and LD/ST (load/stores)?

Order 2 stencil

- How many flops?
- How many words?

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

Count

How many flops?

- 10 additions / multiplications

How many words? 1 word = 4 bytes = 1 float or 1 int

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

Our tools

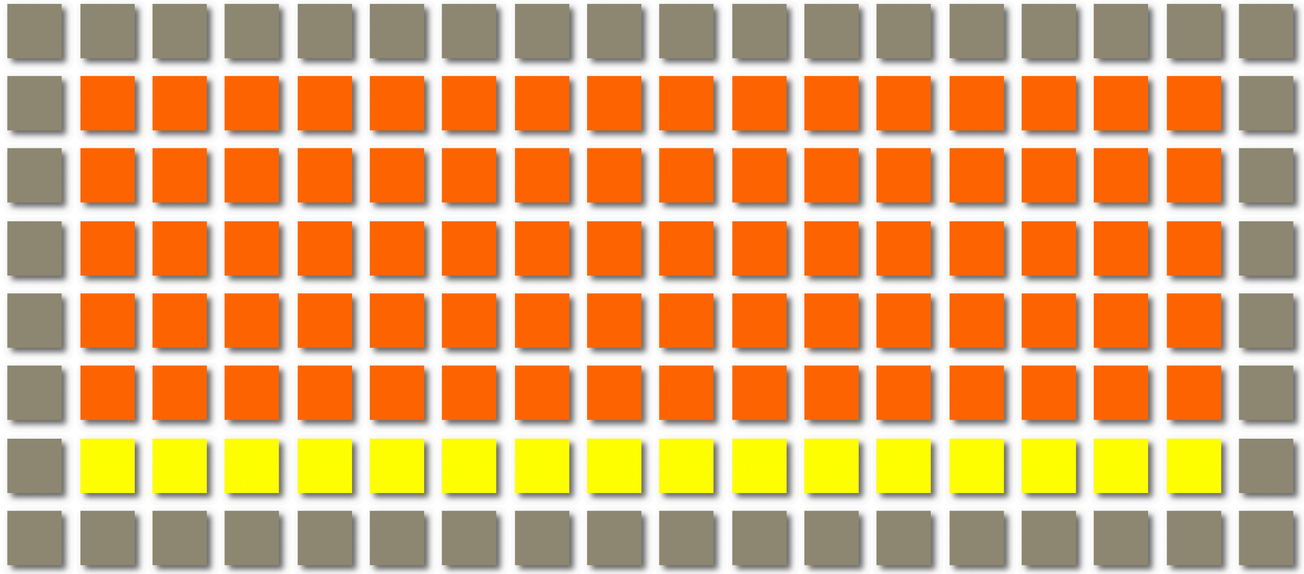
Reduce memory traffic:

- Use cache (L2/L1) or shared memory

Maximize bandwidth:

- Coalesced memory access

Idea 1



Assign a thread-block to a row of the mesh

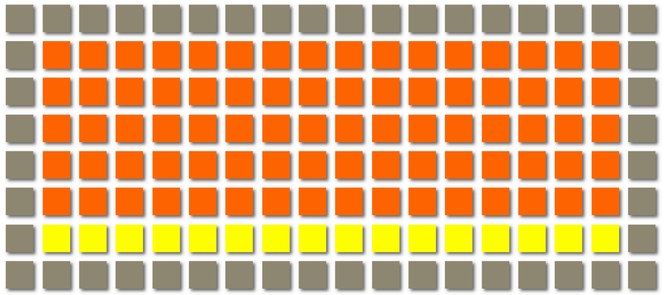
Pro: simple scheme

Con: relatively low performance

Performance analysis

Assume that thread-block = 16 threads.

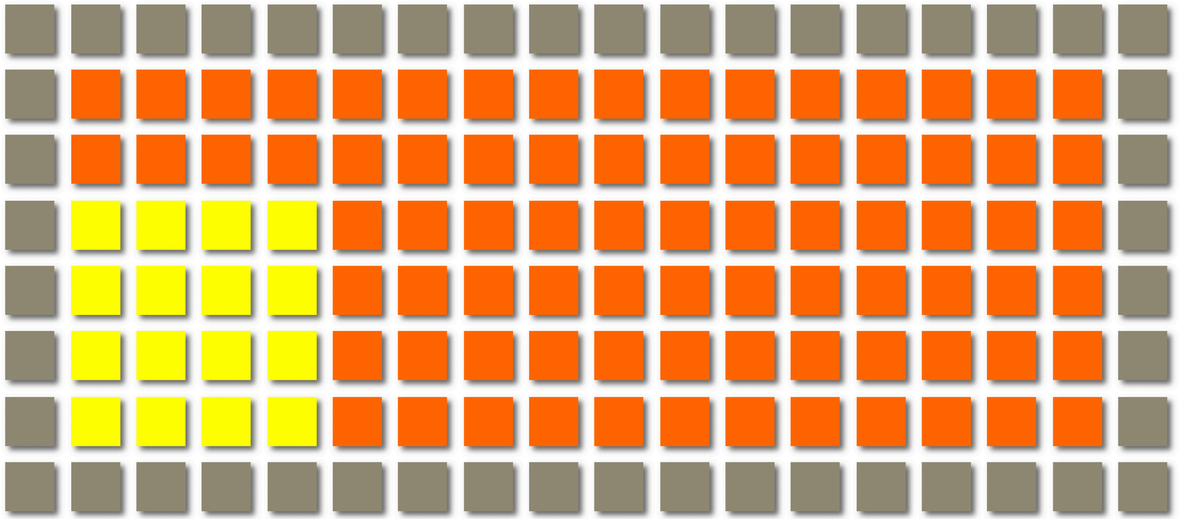
- Reads: 16*3+2. Writes: 16. Total = 66
- Flops: 10*16 = 160
- Ratio: flops/word = **2.4**



Reduce memory traffic

- A point is accessed 5 times in this scheme.
- Goal is to maximally reuse the data before evicting it from the cache.
- Nodes on the “boundary” do not get reused much.
- Nodes in the center are reused by all their neighbors.
- **Design goal: minimize surface given a volume.**
- Solution: sphere; in practice: **square.**

Idea 2



- A thread block should work on a mesh block.
- This ensures that few nodes fall on the boundary.
- **Most nodes are in the inside region where the data is reused.**

Improvement

Ratio: flops/word

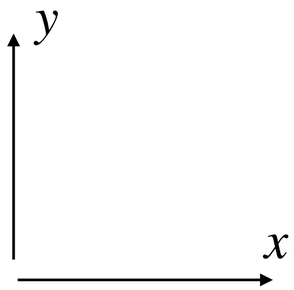
2.4 → 3.3

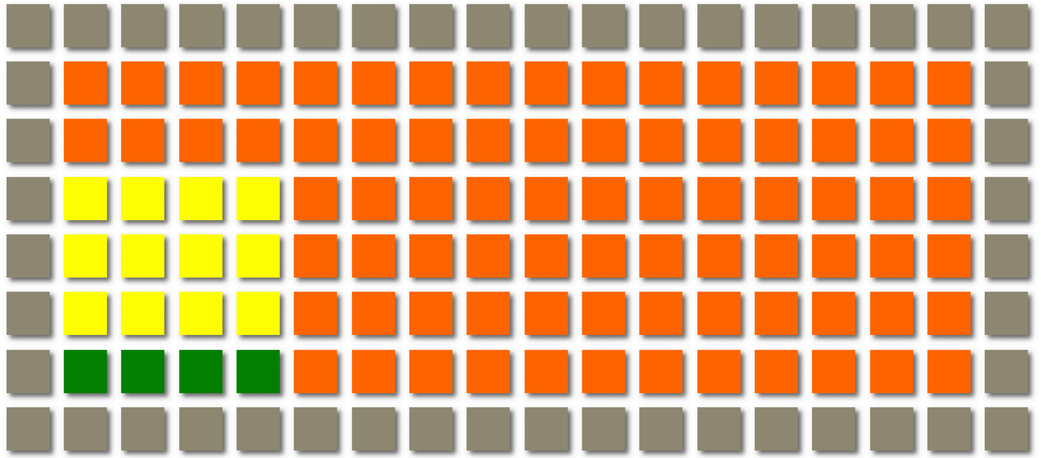
assuming a thread block with 16 threads.

Theoretical peak

- Not achievable.
- But let's assume that nodes on the surfaces are few.
- For an $n \times n$ block:
  - Memory traffic: $2n^2 + 4n$
  - Flops: $10n^2$
- Maximum intensity: 5 flops/words.
- Still memory bound but performance improves as we get closer to that number.

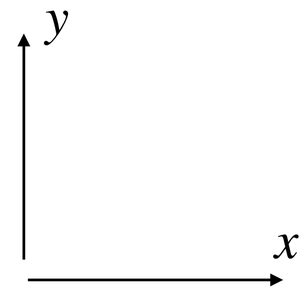
Workload of a warp

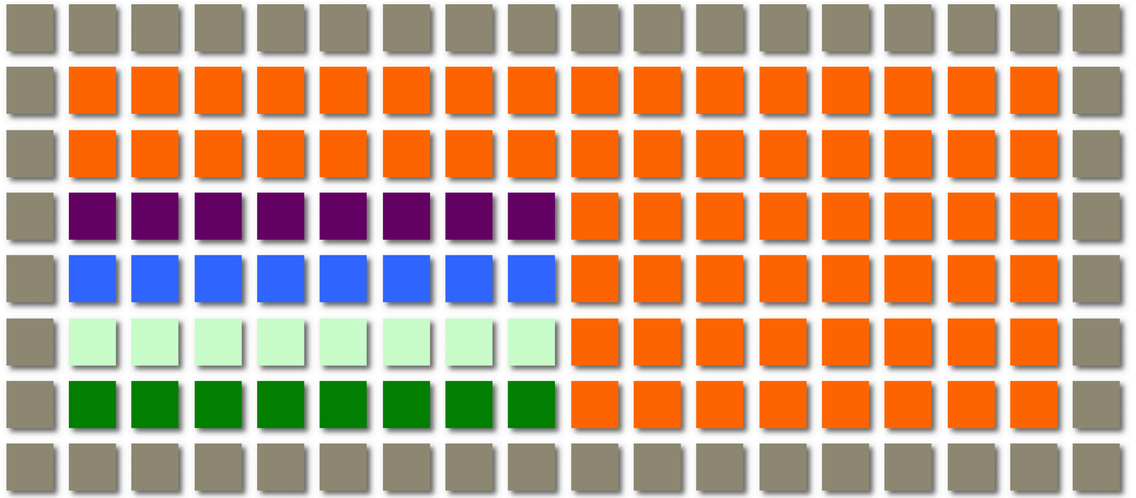




- Only mesh nodes along x are contiguous.
- This size must be a multiple of 32.
- A warp must work on a chunk aligned along x.

Warp memory accesses



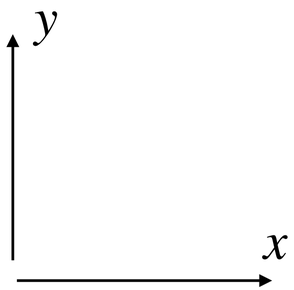


- Each warp is assigned to a row.
- Each warp **loops in the y direction** to process all the entries.
- Data access is coalesced + large data reuse in the cache.

Example of thread block size

- Thread block should be rectangular
- **Along x : recommended, dimension 64**
- Along y : determines number of threads in block
- Example: 512 → dimension y = 8

Check your bounds



- Along x: if `threadIdx.x` too large return immediately.
- Along y: check if iteration goes beyond the domain size. Check the bounds of the `for` loop along y.

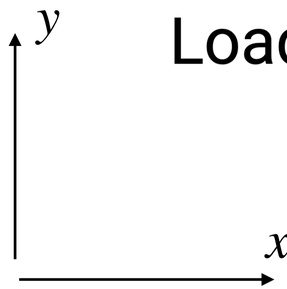
Shared memory algorithm

Step 1: all threads load data in shared memory

All threads are involved in loading data from memory.

As before:

- Along x : if threadIdx.x too large, skip.
- Along y: check if `for` loop iteration goes beyond the domain size.
- **End with `__syncthreads();`**



Load in shared memory

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

- More `if` statements are required to check whether a thread has work to do and find the bounds of the `for` loop.
- Some threads will not have any work during that phase.
- This is fine because the performance limiting step is the read from memory.

Update inside nodes

Simple things to watch for

- Most of the instructions are within inner `for` loops; focus your attention on these.
- Remove all **if/branches** inside the `for` loop.
- Remove all operations that are **independent** of the iteration index.
- Reduce the **instruction count** as much as possible inside the inner loop.
- Loop unrolling: **if the iteration count is known at compile time,** the compiler will **unroll** the loop for optimization; this is beneficial in some cases. Template parameters can be useful.

Sample output

- Compiled with `--fmad=false`
- No fused multiply add (FMA) operation used.
- The arithmetic matches the CPU. Roundoff errors are identical.

```
Order: 8, 4096x4096, 100 iterations
                     time (ms)     GBytes/sec
            CPU        2797.83        43.1748
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from DiffusionTest
[ RUN      ] DiffusionTest.GlobalTest
Order: 8, 4096x4096, 100 iterations
                     time (ms)     GBytes/sec
         Global        49.1457        2457.92

          L2Ref           LInf           L2Err
       0.447065              0               0

[       OK ] DiffusionTest.GlobalTest (287 ms)
[ RUN      ] DiffusionTest.BlockTest
Order: 8, 4096x4096, 100 iterations
                     time (ms)     GBytes/sec
          Block        37.5009        3221.15

          L2Ref           LInf           L2Err
       0.447065              0               0

[       OK ] DiffusionTest.BlockTest (284 ms)
[ RUN      ] DiffusionTest.SharedTest
Order: 8, 4096x4096, 100 iterations
                     time (ms)     GBytes/sec
         Shared        34.3608        3515.52

          L2Ref           LInf           L2Err
       0.447065              0               0

[       OK ] DiffusionTest.SharedTest (279 ms)
```

With fused multiply add (FMA)

Compiled *without* --fmad=false

- With fused multiply add (FMA) instructions, the roundoff errors are different from the CPU.
- This is because arithmetic operations in finite-precision are **not associative.**
- Note that performance is the same because we are memory bound.

```
darve@icme-gpu1:~/hw4/code/solution$ srun --gres=gpu:1 -p CME ./main -gsb
Order: 8, 4096x4096, 100 iterations
                    time (ms)     GBytes/sec
            CPU        2799.16        43.1544
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from DiffusionTest
[ RUN      ] DiffusionTest.GlobalTest
Order: 8, 4096x4096, 100 iterations
                    time (ms)     GBytes/sec
         Global        46.6404        2589.94
There were 5856 total locations where there was a difference between the cpu and gpu

          L2Ref           LInf          L2Err
       0.447065    3.57628e-07     4.3774e-08

main.cu:115: Failure
Failed
There was an error in the computation, quitting...

[  FAILED  ] DiffusionTest.GlobalTest (325 ms)
[ RUN      ] DiffusionTest.BlockTest
Order: 8, 4096x4096, 100 iterations
                    time (ms)     GBytes/sec
          Block        37.5009        3221.15
There were 5856 total locations where there was a difference between the cpu and gpu

          L2Ref           LInf          L2Err
       0.447065    3.57628e-07     4.3774e-08

main.cu:140: Failure
Failed
There was an error in the computation, quitting...

[  FAILED  ] DiffusionTest.BlockTest (311 ms)
[ RUN      ] DiffusionTest.SharedTest
Order: 8, 4096x4096, 100 iterations
                    time (ms)     GBytes/sec
         Shared        34.3511        3516.51
There were 5856 total locations where there was a difference between the cpu and gpu

          L2Ref           LInf          L2Err
       0.447065    3.57628e-07     4.3774e-08

main.cu:175: Failure
Failed
There was an error in the computation, quitting...

[  FAILED  ] DiffusionTest.SharedTest (337 ms)
[----------] 3 tests from DiffusionTest (974 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (975 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 3 tests, listed below:
[  FAILED  ] DiffusionTest.GlobalTest
[  FAILED  ] DiffusionTest.BlockTest
[  FAILED  ] DiffusionTest.SharedTest
```

Summary of best strategy

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

- Then loop along y reusing loaded data; go as far as possible while generating sufficient concurrent work, e.g., have enough active thread blocks.