CME 213

SPRING 2019

Eric Darve

Previously in CME213

Warp = 32 threads

Block: 1D, 2D, 3D group of threads; max = 1,024 threads

Grid: 1D, 2D, 3D group of blocks; required for performance and large problems

Memory access:

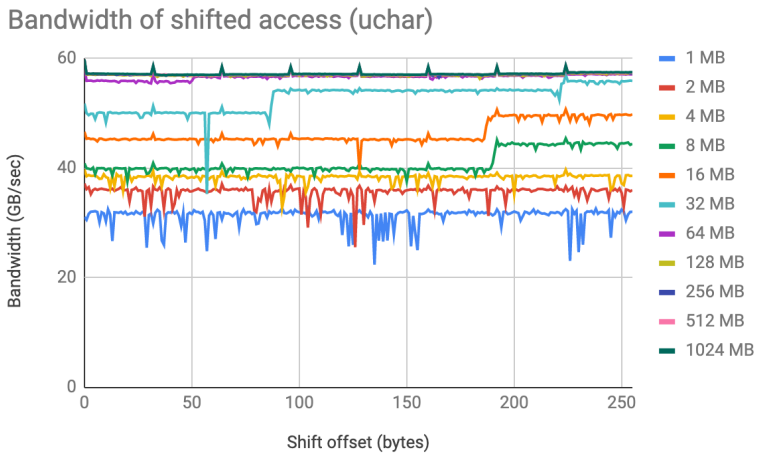
- Analysis based on warp: memory requests made by all threads in warp
- Peak performance: full segment/cache line used by warp = coalesced access
- Loss in performance reflected by fraction of cache line not used by warp.

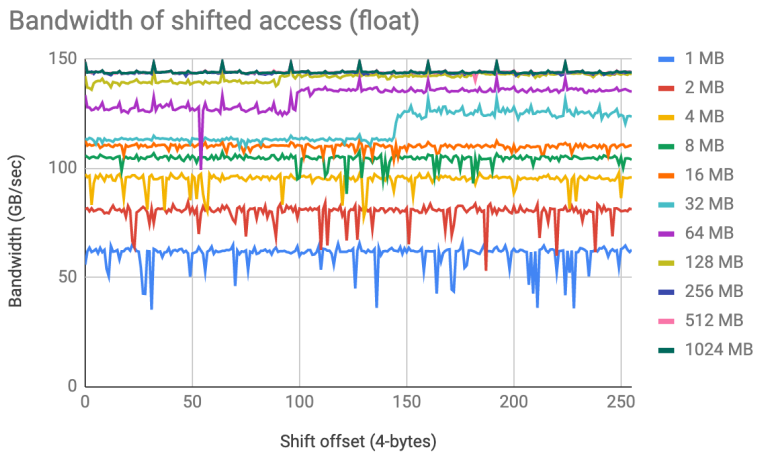
Benchmarks

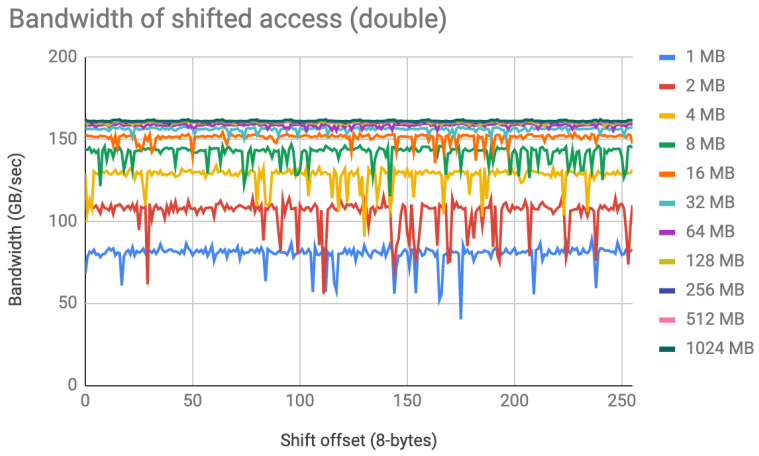
```
template <typename T>
__global__ void offsetCopy(T* odata, const T* __restrict__ idata, int offset, int len) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    for (uint i = xid; i < len; i += gridDim.x * blockDim.x)
    {
        odata[i] = idata[i];
    }
}
```

Offset copy for different data types

Offset







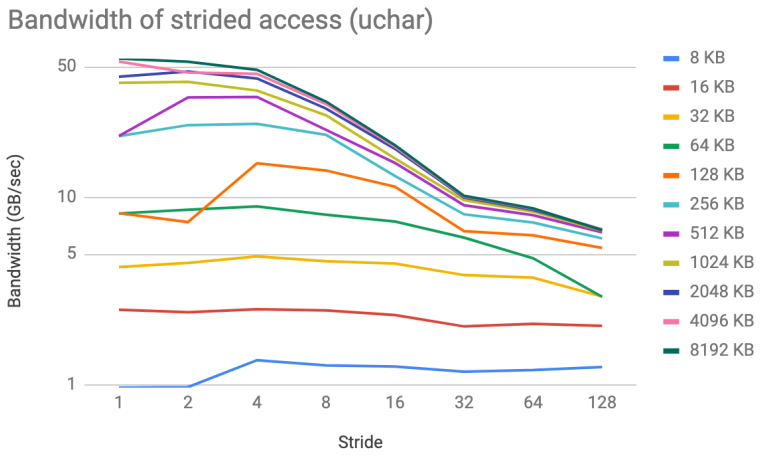
Peak is 165 GB/sec

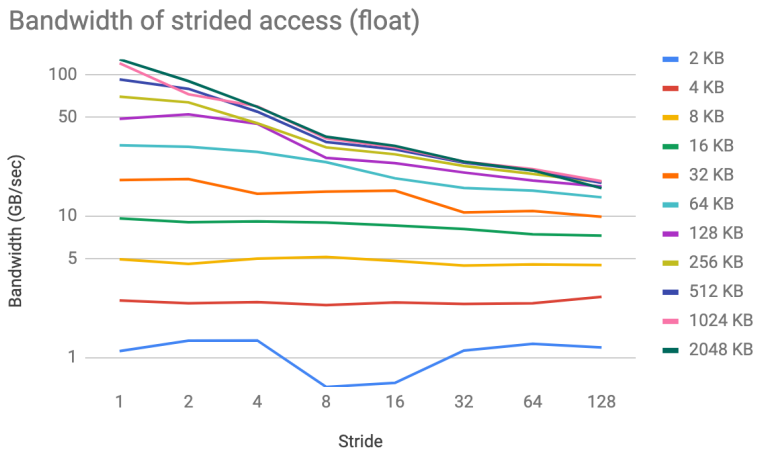
- Small impact from offset
- Appears as soon as offset is not 0 or multiple of 32/128 bytes.
- Effect decreases with longer types.

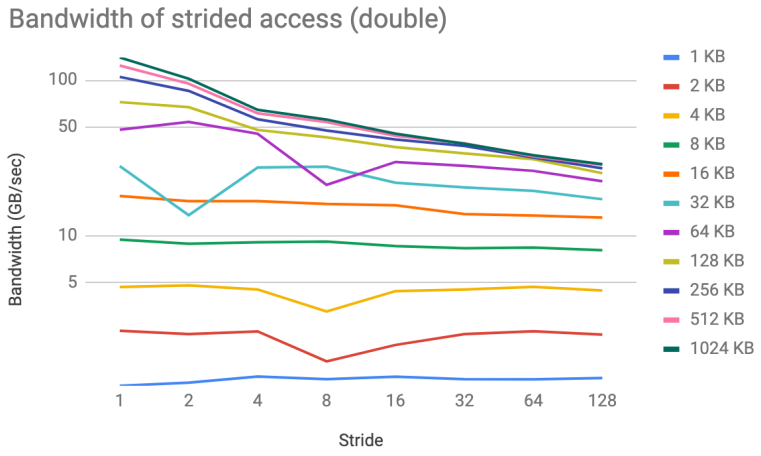
Stride

```
template <typename T>
__global__ void stridedCopy(T* odata, const T* __restrict__ idata, int stride, int len) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x;
    for (uint i = xid; i < len; i += gridDim.x * blockDim.x)
    {
        odata[i] = idata[i * stride];
    }
}
```

Strided access



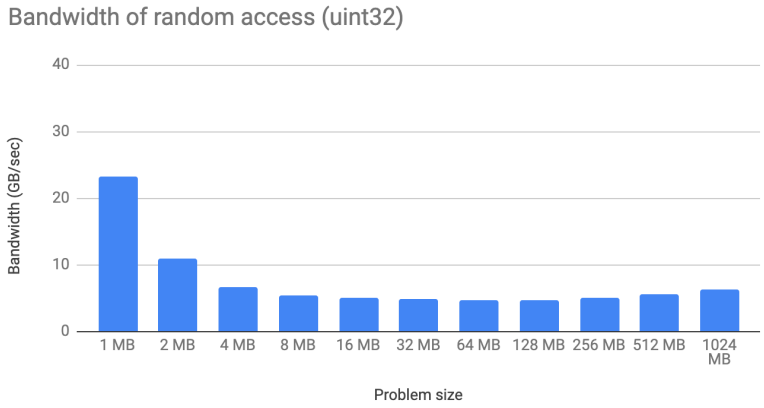


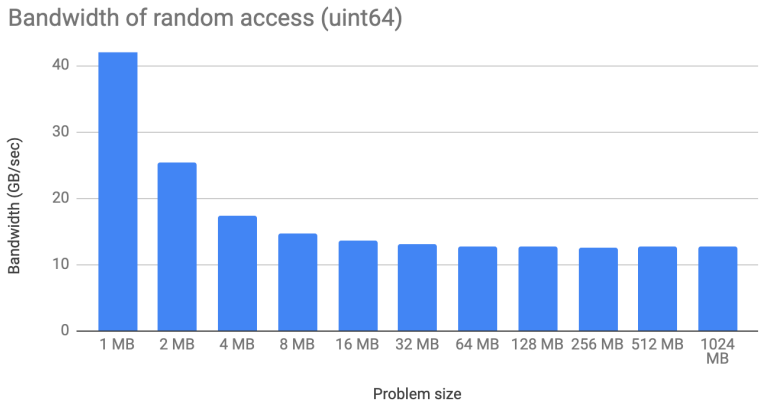


- Decrease as stride increases
- Slight “plateau” around 32 bytes

Random copy

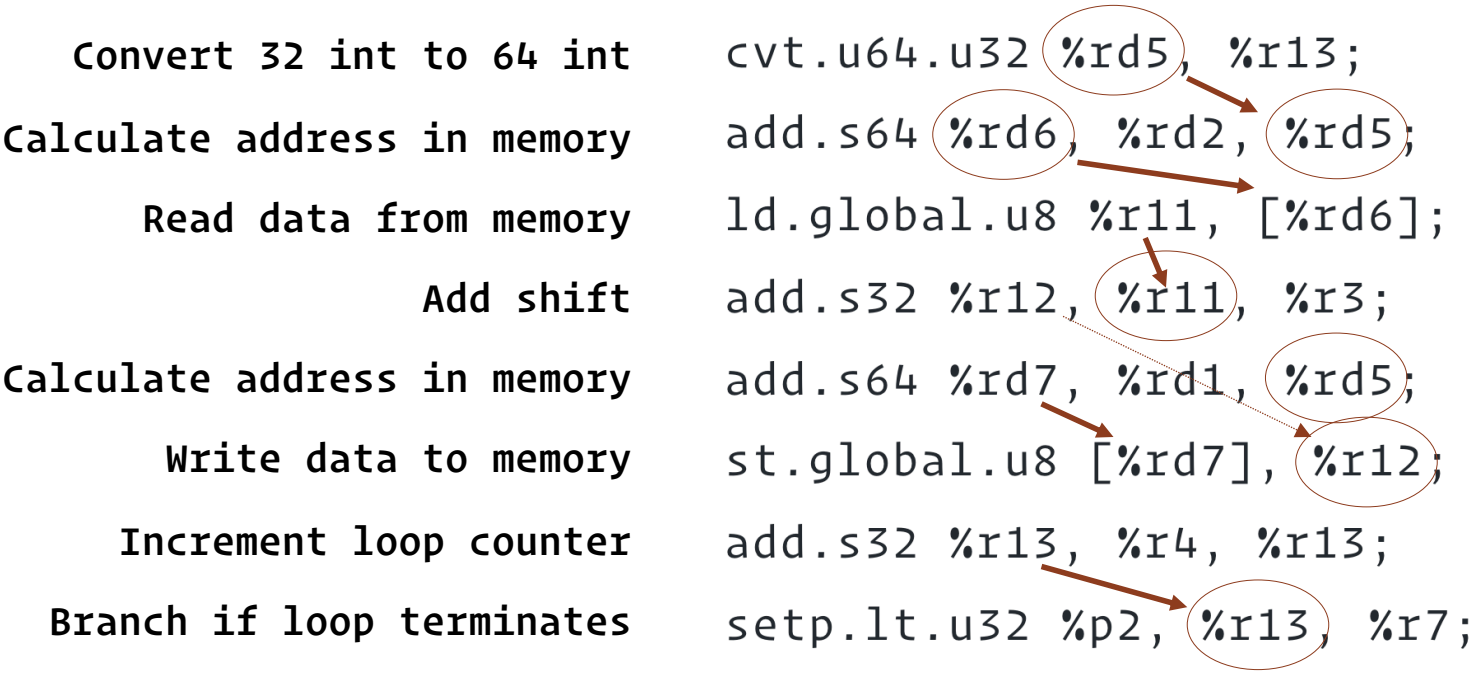
```
template <typename T>
__global__ void randomCopy(T* odata, const T* __restrict__ idata, int len) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x;
    for (uint i = xid; i < len; i += gridDim.x * blockDim.x)
    {
        odata[idata[i]] = idata[i];
    }
}
```


Bandwidth of random access (uint32)
Bandwidth (GB/sec)
40
30
20
10
0
1 MB
2 MB
4 MB
8 MB
16 MB
32 MB
64 MB
128 MB
256 MB
512 MB
1024 MB
Problem size


Bandwidth of random access (uint64)
Bandwidth (GB/sec)
40
30
20
10
0
1 MB
2 MB
4 MB
8 MB
16 MB
32 MB
64 MB
128 MB
256 MB
512 MB
1024 MB
Problem size

Homework assignment 3

Convert 32 int to 64 int	`cvt.u64.u32 %rd5, %r13;`
Calculate address in memory	`add.s64 %rd6, %rd2, %rd5;`
Read data from memory	`ld.global.u8 %r11, [%rd6];`
Add shift	`add.s32 %r12, %r11, %r3;`
Calculate address in memory	`add.s64 %rd7, %rd1, %rd5;`
Write data to memory	`st.global.u8 [%rd7], %r12;`
Increment loop counter	`add.s32 %r13, %r4, %r13;`
Branch if loop terminates	`setp.lt.u32 %p2, %r13, %r7;`
	`@%p2 bra BB6_2;`



```
for (uint i = tid; i < array_length; i += gridDim.x * blockDim.x)
{
    output_array[i] = input_array[i] + shift_amount;
}
```

```
darve@hw3:~/cuda_samples/NVIDIA_CUDA-10.0_Samples/1_Utilities/deviceQuery$ nvidia-smi  -q -i 0 -d CLOCK

==============NVSMI LOG==============

Timestamp                           : Mon Apr 29 18:06:28 2019
Driver Version                      : 418.40.04
CUDA Version                        : 10.1

Attached GPUs                       : 1
GPU 00000000:00:04.0
    Clocks
        Graphics                    : 324 MHz
        SM                          : 324 MHz
        Memory                      : 324 MHz
        Video                       : 405 MHz
    Applications Clocks
        Graphics                    : 562 MHz
        Memory                      : 2505 MHz
    Default Applications Clocks
        Graphics                    : 562 MHz
        Memory                      : 2505 MHz
    Max Clocks
        Graphics                    : 875 MHz
        SM                          : 875 MHz
        Memory                      : 2505 MHz
        Video                       : 540 MHz
    Max Customer Boost Clocks
        Graphics                    : N/A
    SM Clock Samples
        Duration                    : 686.56 sec
        Number of Samples           : 43
        Max                         : 875 MHz
        Min                         : 324 MHz
        Avg                         : 444 MHz
    Memory Clock Samples
        Duration                    : 686.54 sec
        Number of Samples           : 43
        Max                         : 2505 MHz
        Min                         : 324 MHz
        Avg                         : 1089 MHz
    Clock Policy
        Auto Boost                  : On
        Auto Boost Default          : On
```

Possible bottlenecks: instructions

Instructions throughput rate:

- Frequency: 875 MHz
- SMX: 13
- LD/ST units per SMX: 32
- Throughput (for `char`): 875 10^6 * 13 * 32 = 364 GB/sec

- But LD/ST units are not issuing instructions every cycle. Their utilization may be around 25%.
- This gives an effective rate of 90 GB/s.

Possible bottlenecks: memory

Memory bandwidth:

- Frequency: 2505 MHz
- Memory bus width: 384-bit
- DDR memory: 2x
- Throughput: 2505 10^6 x 2 x 384 / 8 = 240 GB/sec
- With ECC (error-correcting code) on: 180 GB/sec (~ 75%)
- Measured: 156 GB/sec

Performance

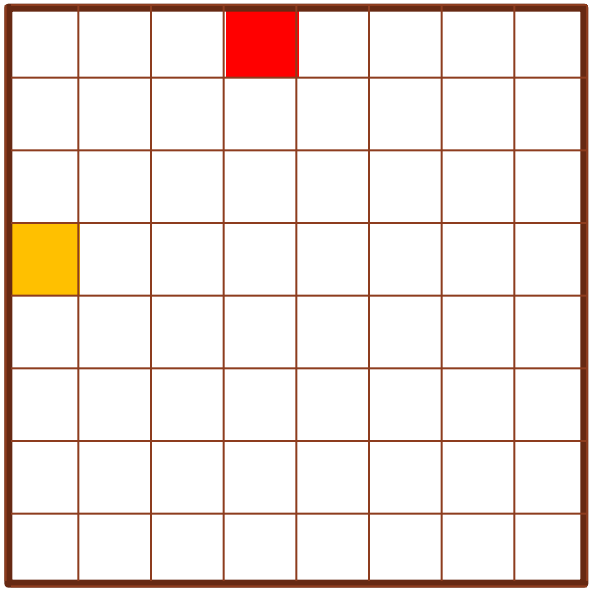
- For `char`, the rate of instructions is the limiting factor.
- For `int`, each instruction requests 4 times more memory. Now we can saturate the memory bus and get close to full bandwidth.

Matrix transpose

Let’s put all these concepts into play through a specific example: a matrix transpose.

It’s all about bandwidth!

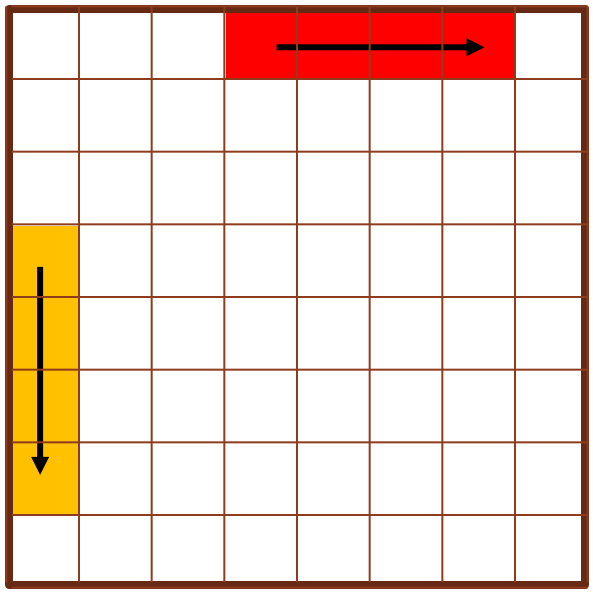
Even for such a simple calculation, there are many optimizations.



simpleTranspose

```
__global__
void simpleTranspose(int* array_in, int* array_out, int n_rows, int n_cols) {
  const int tid = threadIdx.x + blockDim.x * blockIdx.x;

  int col = tid % n_cols;
  int row = tid / n_cols;

  if(col < n_cols && row < n_rows) {
    array_out[col * n_rows + row] = array_in[row * n_cols + col];
  }
}
```

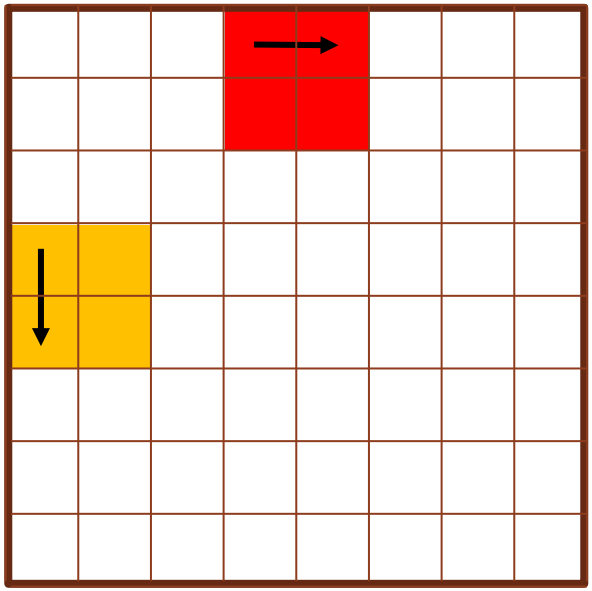
Memory access pattern



Coalesced reads

Strided writes

2D kernel



```
__global__
void simpleTranspose2D(int* array_in, int* array_out, int n_rows, int n_cols) {
  const int col = threadIdx.x + blockDim.x * blockIdx.x;
  const int row = threadIdx.y + blockDim.y * blockIdx.y;

  if(col < n_cols && row < n_rows) {
    array_out[col * n_rows + row] = array_in[row * n_cols + col];
  }
}
```

Access pattern

```
dim3 block_dim(8, 32);
dim3 grid_dim(n / 8, n / 32);
```

For a given warp:

- column: 0 to 7
- row: 0 to 3

Reads are partially coalesced

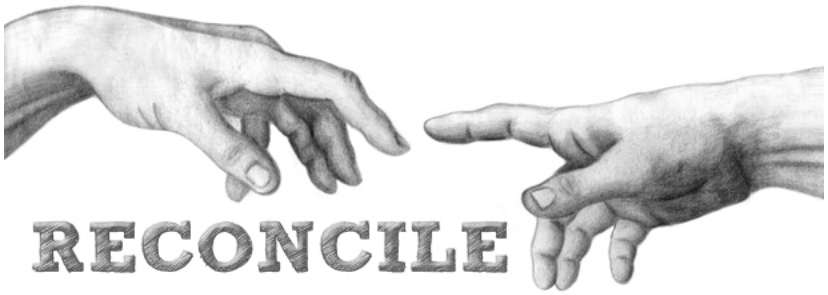

Writes are partially coalesced


Performance

```
Bandwidth bench
GPU took 1.02581 ms
Effective bandwidth is 130.841 GB/s

simpleTranspose
GPU took 41.3072 ms
Effective bandwidth is 35.7418 GB/s

simpleTranspose2D
GPU took 16.9288 ms
Effective bandwidth is 87.2122 GB/s
```

Time(%)	Time	Calls	Avg	Min	Max	Name
32.38%	41.547ms	11	3.7770ms	3.7636ms	3.7859ms	simpleTranspose(int*, int*, int, int)
26.61%	34.143ms	3	11.381ms	10.890ms	11.920ms	[CUDA memcpy DtoH]
13.29%	17.052ms	11	1.5501ms	1.5453ms	1.5533ms	simpleTranspose2D(int*, int*, int, int)

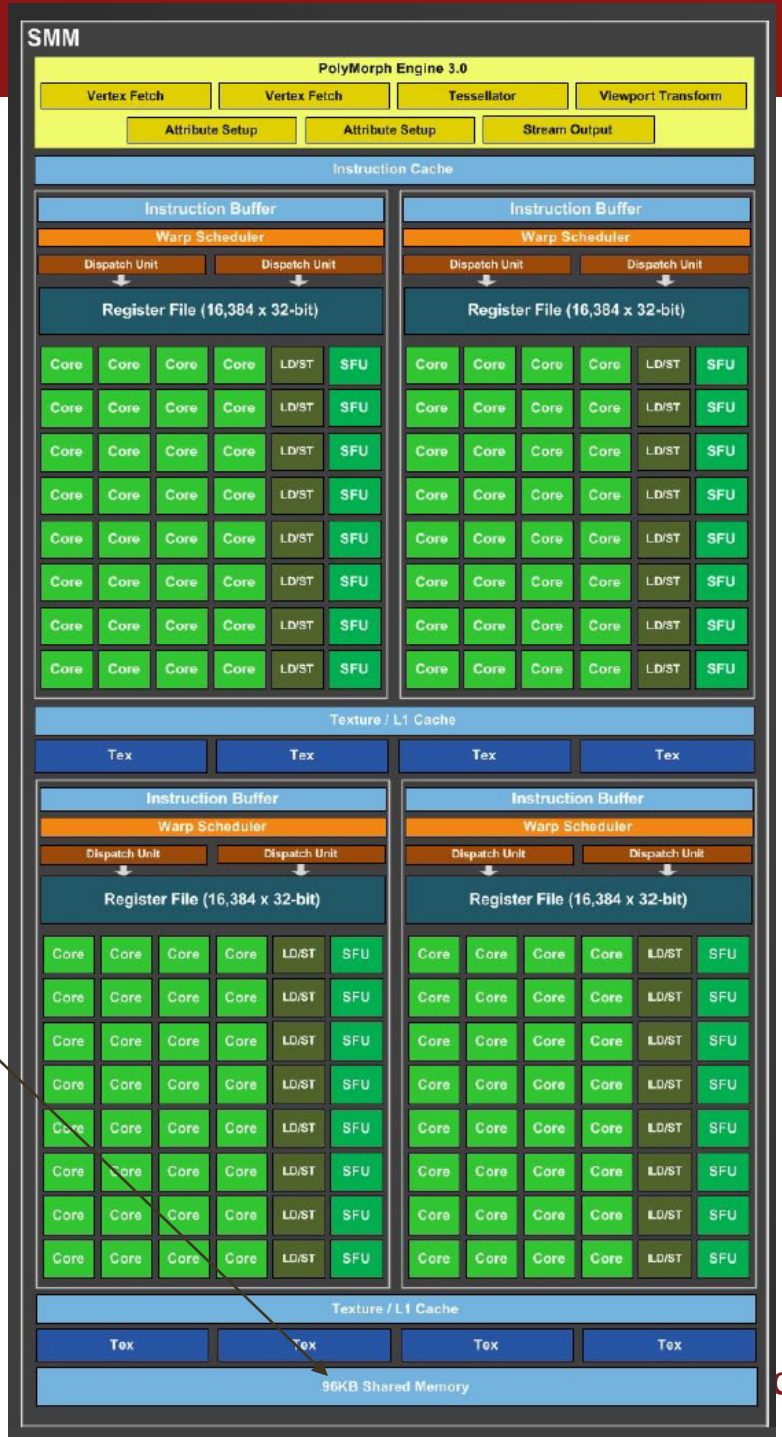
How can you reconcile the reads and writes?



Shared memory

- You need to read data in a coalesced way.
- Transpose locally using fast memory.
- Write data in a coalesced way.
- For this, we need to use shared memory!

It's here!


SMM
PolyMorph Engine 3.0
Vertex Fetch
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Attribute Setup
Stream Output
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache
Tex
96KB Shared Memory

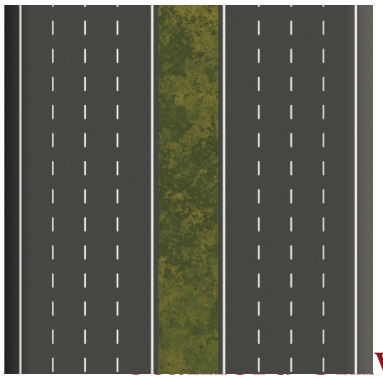
Shared memory

- On-chip: high bandwidth, low latency
- Data in shared memory is only accessible by threads in the same thread block!

```
template<int num_warps>
__global__
void fastTranspose(int* array_in, int* array_out, int n_rows, int n_cols) {
  const int warp_id  = threadIdx.y;
  const int lane     = threadIdx.x;

  __shared__ int block[warp_size][warp_size];
```

Shared memory variable

Imagine a highway with 32 lanes



versity

Load data from global memory

```
// Load 32x32 block into shared memory
int gc = bc * warp_size + lane; // Global column index

for(int i = 0; i < warp_size / num_warps; ++i) {
  int gr = br * warp_size + i * num_warps + warp_id; // Global row index
  block[i * num_warps + warp_id][lane] = array_in[gr * n_cols + gc];
}

__syncthreads();
```

Shared memory variable

Global memory

All threads in block participate in this; so we need to wait that all threads are done before continuing.

Write is the same but transpose

```
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];
}
```

This works now!

```
lane = threadIdx.x
gr = ... + lane
array_out[gc * n_rows + gr]
```

Stride is 1 with respect to **`threadIdx.x`**

Performance

```
Bandwidth bench
GPU took 1.0261 ms
Effective bandwidth is 130.804 GB/s

simpleTranspose
GPU took 41.2702 ms
Effective bandwidth is 35.7739 GB/s

simpleTranspose2D
GPU took 17.024 ms
Effective bandwidth is 86.7243 GB/s

fastTranspose
GPU took 17.1863 ms
Effective bandwidth is 85.9053 GB/s
```

Yeah!

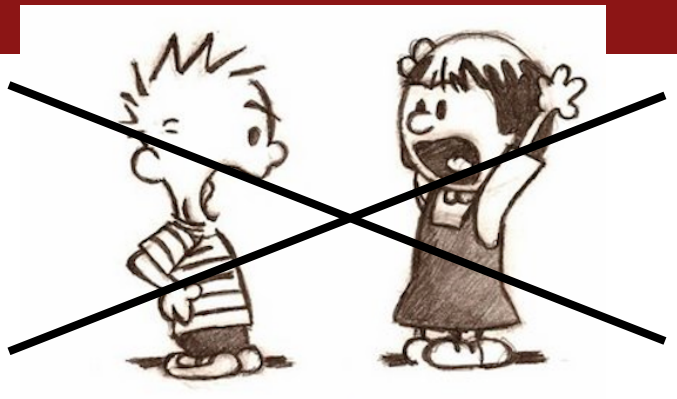
Wait…

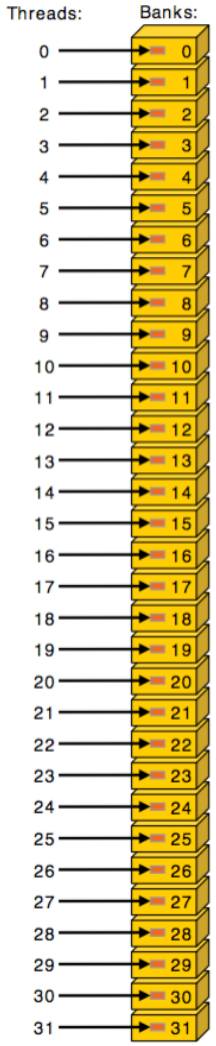
- We went from 87 GB/s down to 86 GB/s
- What happened?

Banks

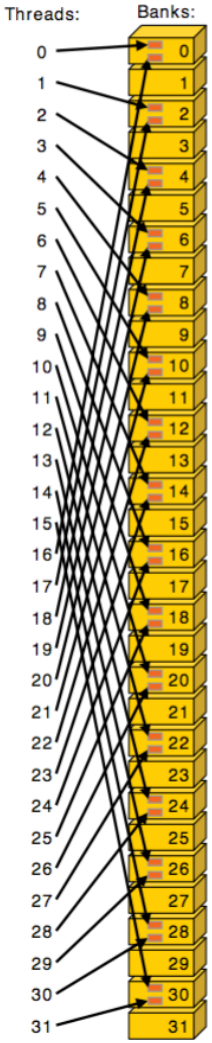
- Shared memory suffers from bank conflicts.
- The shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously.
- Any memory read or write request made of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.
- However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized.
- Each bank has a bandwidth of 4 bytes per two clock cycles.

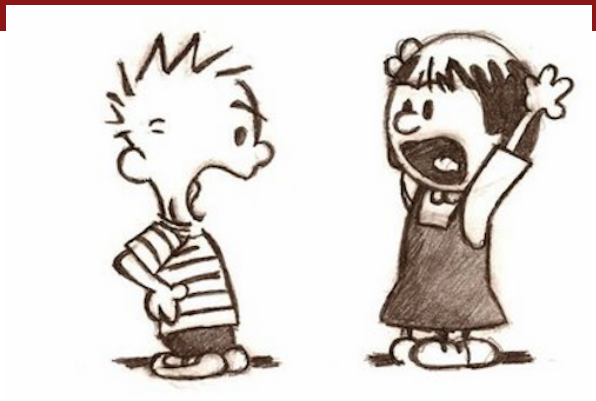
Conflict free



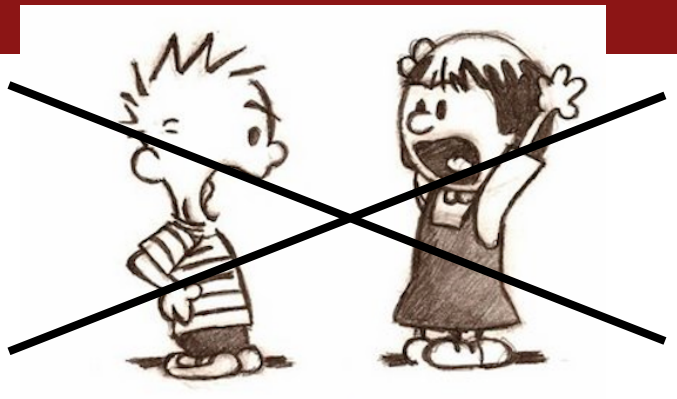


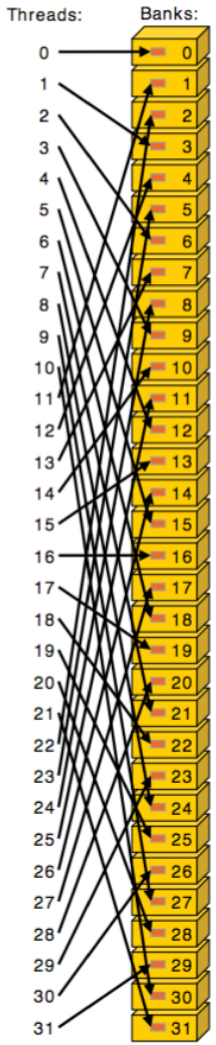
Two way conflict





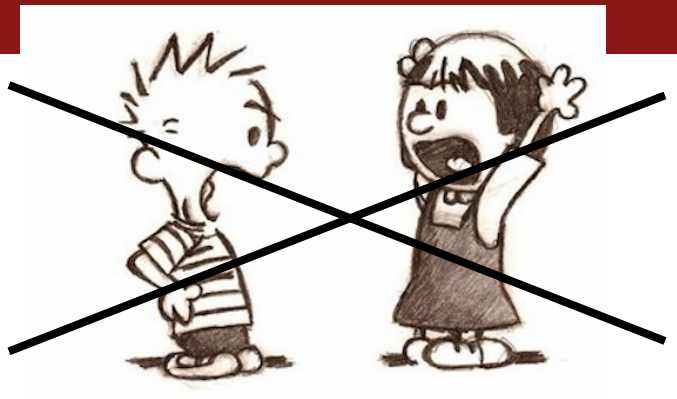
Conflict free



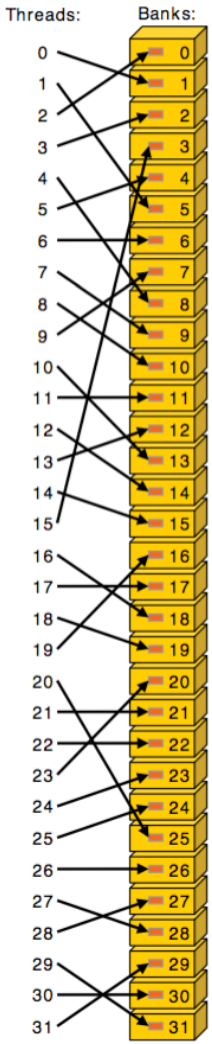


Stride of 3

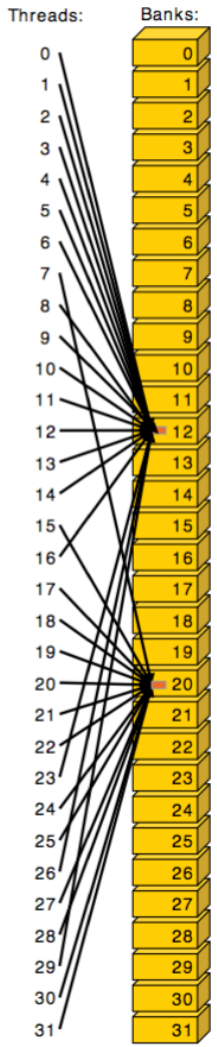
Conflict free

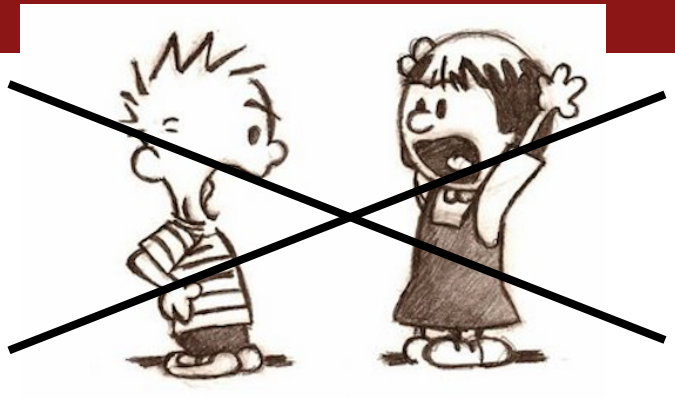


General permutation



Conflict free



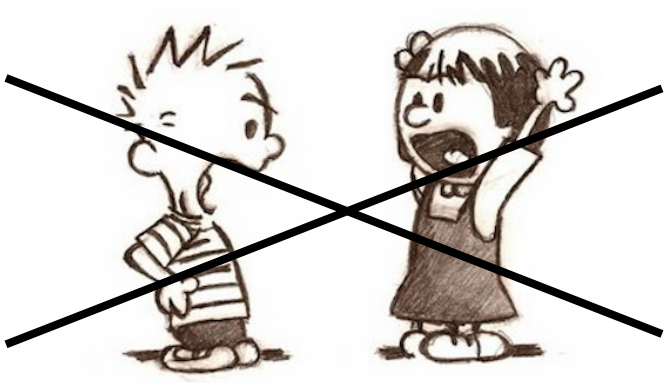


Broadcast capability
Accessing the same word inside the bank

Shared memory writes

```
block[i * num_warps + warp_id][lane] = array_in[gr * n_cols + gc];
```

Stride of 1



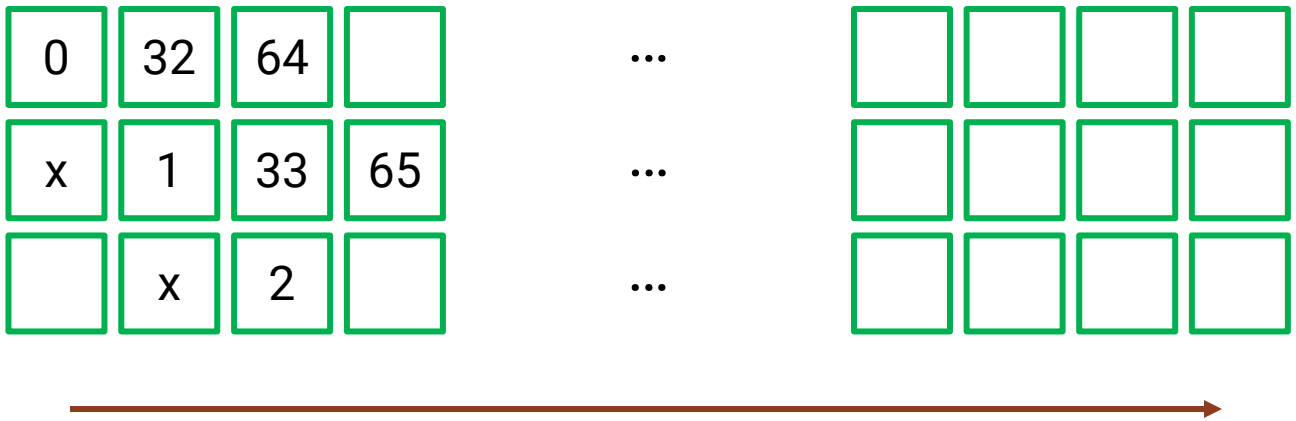
Shared memory reads

```
__shared__ int block[warp_size][warp_size];
```

```
array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
```

Stride of 32!

```
__shared__ int block[warp_size][warp_size+1];
```

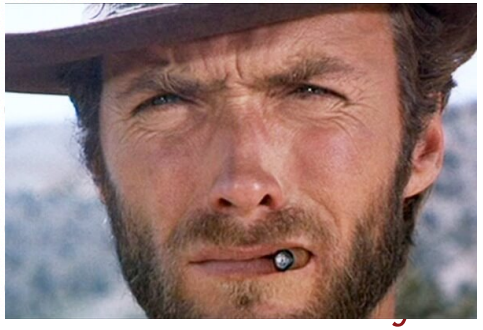


32 banks
Contiguous memory space

```
array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
```

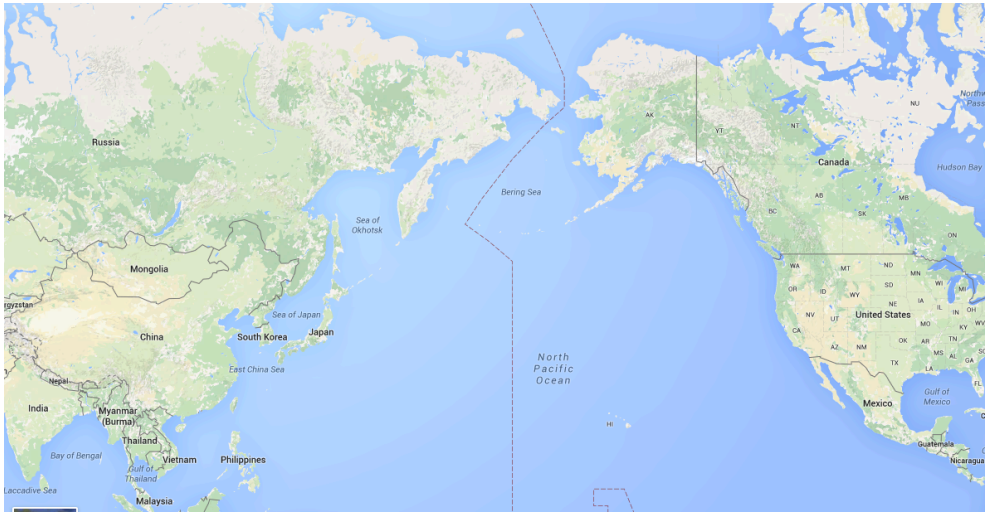
```
Bandwidth bench
GPU took 1.02511 ms
Effective bandwidth is 130.93 GB/s

simpleTranspose
GPU took 41.2939 ms
Effective bandwidth is 35.7533 GB/s

simpleTranspose2D
GPU took 17.0411 ms
Effective bandwidth is 86.6374 GB/s

fastTranspose
GPU took 12.1581 ms
Effective bandwidth is 121.433 GB/s
```

```
      Avg        Min        Max  Name
3.7378ms   3.7260ms   3.7520ms  simpleTranspose
10.720ms   10.600ms   10.942ms  [CUDA memcpy Dt
1.5568ms   1.5534ms   1.5586ms  simpleTranspose
12.335ms   12.335ms   12.335ms  [CUDA memcpy Ht
1.0944ms   1.0891ms   1.0993ms  void fastTransp
```



Concurrency, latency

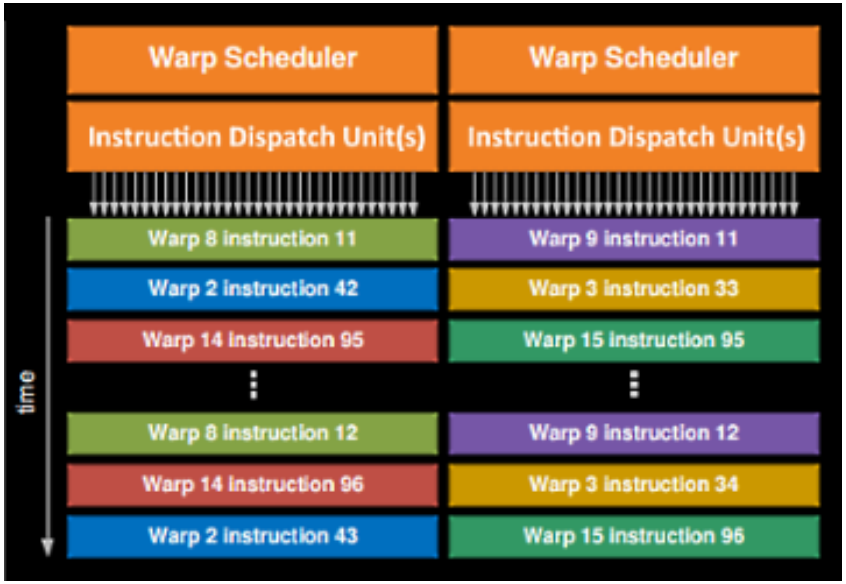
- Imagine you are a pencil manufacturer.
- You outsource your manufacturing plants to China but your market is in the US.
- How do you organize the logistics of the transport?



- You may have great and cheap bandwidth.
- But the latency is very long.
- This works as long as you have enough parallel work to do, i.e., lots of boxes of pencils to ship.
- Then they can be put on slow but big ships.
- Once the shipping pipeline is full, you get cheap and frequent deliveries in large quantities.
- This is the same on a GPU.

Concurrency!

- Concurrency is used to hide memory latency.
- Concurrency is also important for other pipelines in the system, for example in the floating point units.
- The system keeps switching between warps, to avoid being idle.



How is concurrency managed?

The key is therefore to have as many threads live on an SM as possible.
(That's not 100% true all the time but that's a good rule of thumb.)

This is constrained by 2 requirements:

- We can only fit a whole number of blocks on an SM.
- There are hardware limits.

Main ones:

- Max dim. of grid: y/z 65,535 (x is huge, 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: 64
- Max threads per SM: 2,048
- # of 4-byte registers per block: 65,536 (128K per block)
- Max shared mem per block: 49,152 (112 KB per block)

How can we make sense of this?

Use the spreadsheet: CUDA occupancy calculator.

Use CUDA API:

cudaOccupancyMaxActiveBlocksPerMultiprocessor

- Occupancy prediction based on the block size and shared memory usage of a kernel

cudaOccupancyMaxPotentialBlockSize

cudaOccupancyMaxPotentialBlockSizeVariableSMem

- Returns grid and block size that would achieve maximum occupancy for a device function.

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):	3.7	(Help)
1.b) Select Shared Memory Size Config (bytes)	114688	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	18	
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:	3.7
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	131072
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	114688
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4

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	84	20
Shared Memory (Bytes)	4352	49152	26

Note: SM is an abbreviation for (Streaming) Multiprocessor

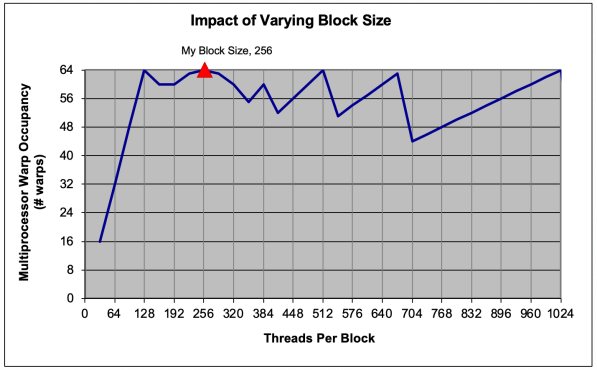
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	**8**	**8**	**64**
Limited by Registers per Multiprocessor	20		
Limited by Shared Memory per Multiprocessor	26		

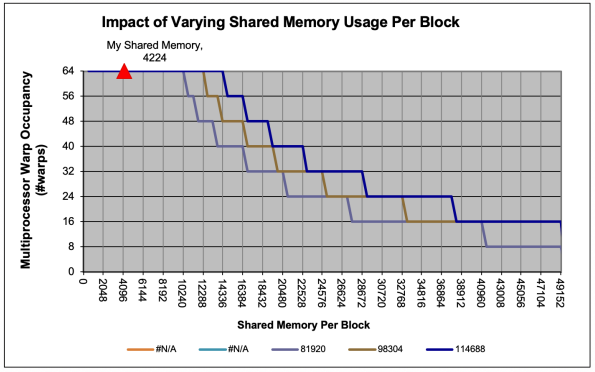
Note: Occupancy limiter is shown in orange

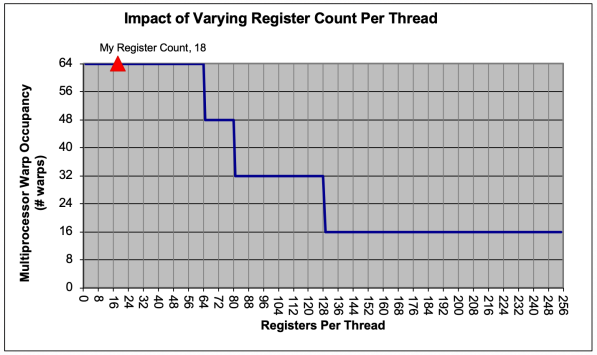
Physical Max Warps/SM = 64

Occupancy = 64 / 64 = 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.







Stanford University

How to get kernel information

```
darve@hw3:~/cuda2$ nvcc --ptxas-options=-v -arch=sm_35 -O3 transpose.cu -o transpose
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z13fastTransposeILi8EEvPiS0_ii' for 'sm_35'
ptxas info    : Function properties for _Z13fastTransposeILi8EEvPiS0_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 18 registers, 4224 bytes smem, 344 bytes cmem[0]
ptxas info    : Compiling entry function '_Z17simpleTranspose2DPiS_ii' for 'sm_35'
ptxas info    : Function properties for _Z17simpleTranspose2DPiS_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 6 registers, 344 bytes cmem[0]
ptxas info    : Compiling entry function '_Z15simpleTransposePiS_ii' for 'sm_35'
ptxas info    : Function properties for _Z15simpleTransposePiS_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 9 registers, 344 bytes cmem[0]
```

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

Default cache configuration: 115 KB of shared memory and 16 KB of L1 cache.

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.

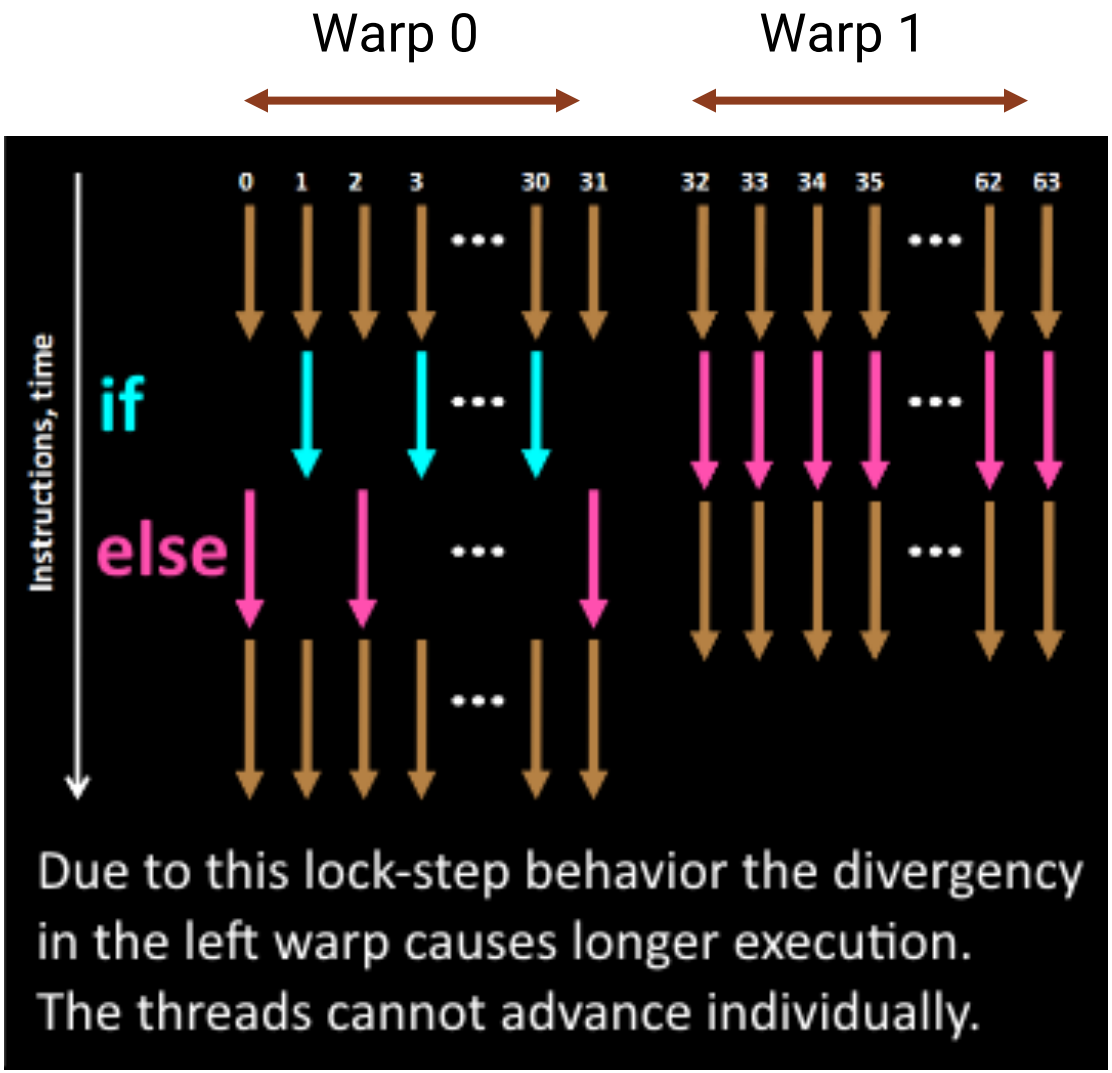
Here that number is 8 warps = 256 threads / block.

```
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

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{
    ...;
  }
}
```

Divergence!

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

No divergence!