GPU TECHNOLOGY CONFERENCE

GPU MEMORY BOOTCAMP BEST PRACTICES

TONY SCUDIERO - NVIDIA

OUTLINE

- Ratio analysis of hardware
  - FLOP:Byte ratios
- Loads in Flight
- Views of GPU memory
  - Thread, SM, L2/DRAM
  - Address coalescing
- Optimizing Transpose, guided by NVVP
  - occupancy
  - address coalescing
  - shared memory bank access
  - instruction level parallelism

PEAK RATIOS

Processor	Peak GFLOP/s	Peak GB/s	Ops/Byte	Ops/Word
K40 SP	4,290	288	~15	~60
K40 DP	1,430	288	~5	~40
E5-2690 v3 SP (SIMD)	416	68	~6*	~24*
E5-2690 v3 DP (SIMD)	208	68	~3*	~24*

- Without a lot of ops per word, codes are likely to be bound by operand delivery

*Ops must be SIMD instructions

TWO TYPES OF OPTIMIZATIONS

- Algorithmic Optimizations
  - Minimizing the total amount of work done to compute an answer
  - Generally approached from Big-O notation standpoint
- Execution Optimizations
  - Maximizing the use of computational resources in carrying out the computation
  - Usually done in mapping an algorithm or abstract implementation onto a specific piece of computing hardware
- Software optimization as a Min-Max game

ALGORITHM OPTIMIZATION

- Big-O notation
  - Great for publishing papers; infinitely scaling algorithms on abstract Turing machines
  - Algorithmic optimizations are always more important than execution optimization as N approaches infinity
- Flaws and Caveats
  - Ignores too many details when the code hits the silicon
  - Not the ideal metric for algorithm selection
    - Leading constants: O(N lg N) may be slower than $O(N^2)$ for small N
    - Parallelism: A parallelizeable $O(N^2)$ algorithm can outrun serial O(n lg n) on real hardware
  - Ignores data locality issues
- Useful to count memory operations

EXAMPLE

- Compare Matrix-vector multiply(GEMV) to Matrix-Matrix multiplication (GEMM)

	Matrix-Vector (GEMV)	Matrix-Matrix(GEMM)
Flop Count	$3N^2 - N$	$3N^3 - N^2$
Asymptotic Runtime	$O(N^2)$	$O(N^3)$
Memory Ops	$2N^2 + N$	$3N^2$
Asymptotic Accesses	$O(N^2)$	$O(N^2)$
Op per Access	1	O(N)

Assume NxN matrix and Nx1 vector

BANDWIDTH & LATENCY: NYC TO MIAMI

(64GB µSDs)
Packet Size:
15.7 Exabytes

Latency:
~1.87 hrs

Bandwidth:
2.32 Petabytes/s

LOADS IN FLIGHT

- Imagine one plane takes off every 2 minutes, per FAA rules
  - 15.7 Exabytes (1 packet) every 120 seconds = 130.83 petabytes/sec
  - Maximum “channel” bandwidth
- How many planes can we have in flight simultaneously?
  - Latency / take off rate = 56 planes
- What happens to bandwidth if a plane isn’t ready on time?
  - Delivered data drops, time carries on => utilized bandwidth lowered
  - Need all 56 aircraft “slots” used to maximize bandwidth

KEEP THE ANALOGY FLYING

- Send empty C5 back to NY from Miami with a specific request for more data.
  - Assume it takes 1 hour to load at NYC and 1 hour to unload at Miami
  - What is the latency to fill a data specific request?
    - (1.87 hrs flight time + 1 hr load/unload)*2 = 5.74 hrs
  - How many aircraft do we need now?
    - Answer: 5.74 * 60 / 2 = 172.2
    - # C5 Galaxys produced: 131, but reality need not spoil our analogy!

MEANWHILE IN MIAMI

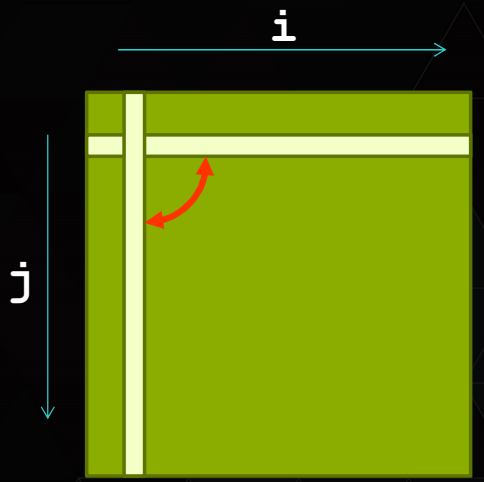
- When a request is sent from Miami, takes 5.7 hours to get a response
- Have workers in Miami doing other things while waiting for that data
  - Likely processing the data from previous deliveries.
- Thought Experiment: What happens when a data request isn't a full 15.7 Exabytes?
  - Plane still leaves, still _could_ carry 15.7 exabytes
    - Channel bandwidth is unchanged
  - Utilized bandwidth goes down

#ICYMI: HERE'S THE ANALOGY

- Analogy to the GPU memory system
  - NYC is DRAM
  - Miami is SM
  - Each C5 Galaxy is a memory transaction
    - It's contents are transaction size, usually a cache line
    - C5s currently working = loads in flight
  - High latency, extremely high bandwidth
- Thinking in terms of airplanes can help understand memory operations
  - Even if it doesn't help, it's more fun

MATRIX TRANSPOSE

- NxN Matrix, Rows ← → Columns
- Algorithmic Analysis
  - $O(N^2)$ Memory
  - No computation on data



SIMPLE SERIAL IMPLEMENTATION

```
void referenceTranspose(int rows, int cols, float * in, float * out)
{
  for(int i=0; i<rows; i++)
  {
    for(int j=0; j<cols; j++)
    {
      out[i*rows + j] = in[j*cols + i];
    }
  }
}
```

SIMPLE PARALLEL IMPLEMENTATION

- Observation: Outer (i) loop iterations are independent. Parallelize it!

```
__global__ void transpose1(int rows, int cols, float * in, float* out)
{
  int i;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  for(int j=0; j<cols; j++)
  {
    out[i*rows + j] = in[j*cols + i];
  }
}
```

- Launch 1D grid of 1D blocks, 256 threads/block

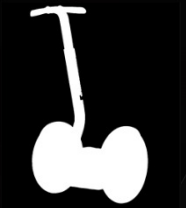
PERFORMANCE

Kernel	Float	Double
transpose1	16.96 GB/sec	33.94 GB/sec

this space intentionally

left blank

Tesla K40c, Boost Clock (875mhz), ECC off



SEGUE: MEASURING PERFORMANCE

- OpenMP Timers:

```
double start = omp_get_wtime();
    Thing you want to time goes here
double end = omp_get_time();
double duration = end-start; //in seconds
```

- CUDA events:

```
cudaEventRecord(start);
   Thing you want to time goes here
cudaEventRecord(end);
cudaEventSynchronize(end);
cudaEventGetElapsedTime(&time_ms, start, stop); // in milliseconds
```

- In General: Lower time is better

TIMING TO PERFORMANCE

- When theoretical bounds are known
  - Loads required for algorithm
  - flops required for algorithm
  - bytes/sec or flops/sec from this
- When theoretical bounds are not known
  - Macro operations per unit time (e.g. Atom interactions per second
  - Micro operations per unit time (e.g. Dot products per second)
  - Warning: Compare performance of identical workloads only
    - Unless comparability can be otherwise demonstrated

CAVEAT OF HARDWARE COUNTERS

- Profiler provides throughput metrics for dram and L2
  - dram_read_throughput
  - l2_l1_read_throughput
  - l2_tex_read_throughput
- These provide memory bandwidth numbers from the L2/DRAM perspective
  - Includes cache misses that could have been hits
  - Includes bandwidth used moving bytes not consumed by algorithm
  - Includes ECC overheads (when applicable)
- Not necessarily indicative of real performance

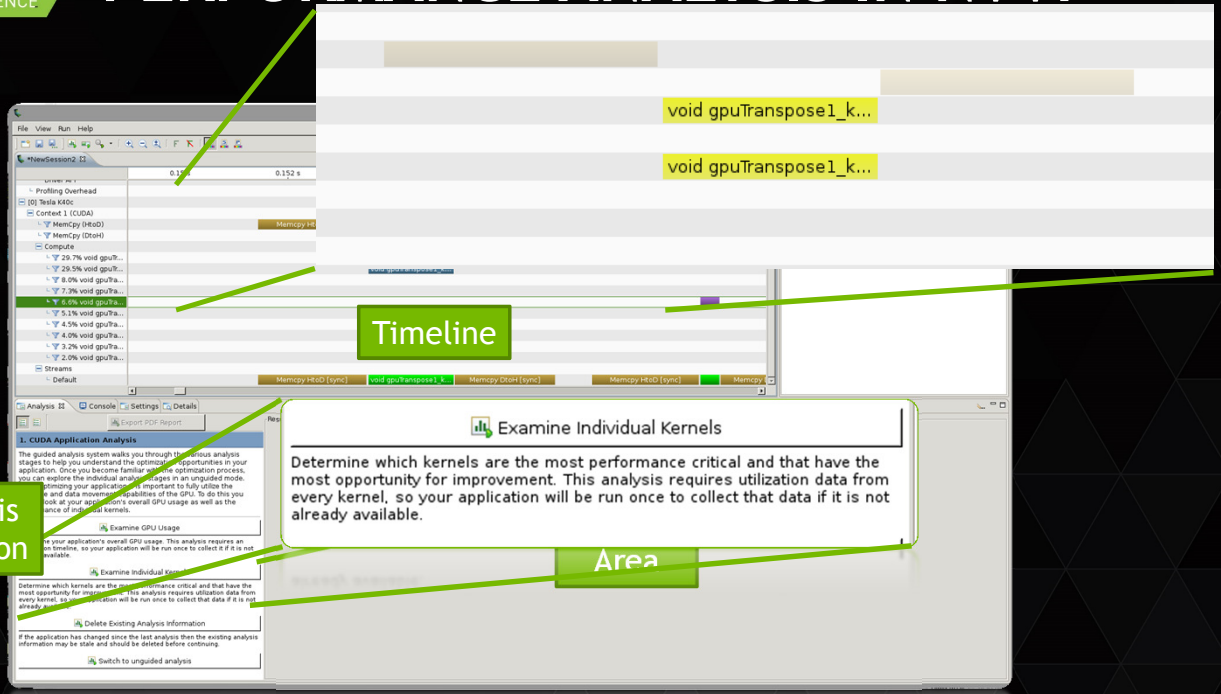
CAVEAT OF HARDWARE COUNTERS

- Example from a real Kernel:
  - Hardware Counters
    - NVPROF reported dram_read_throughput: 80.613GB
  - Algorithm Analysis
    - 4400 bytes/op, 12,140,242 ops/second = 53,417,064,800 bytes/second
  - Memory Efficiency = 66.26%
  - Hardware counters misrepresent performance
    - dram_read_throughput/4400 => 18,312,136 ops/second
    - 1.5x overestimate of performance

- MentalStack.pop(); //Back to Transpose

Tesla K40c, Boost Clock (875mhz), ECC off

PERFORMANCE ANALYSIS IN NVVP



KERNEL OPTIMIZATION PRIORITIES

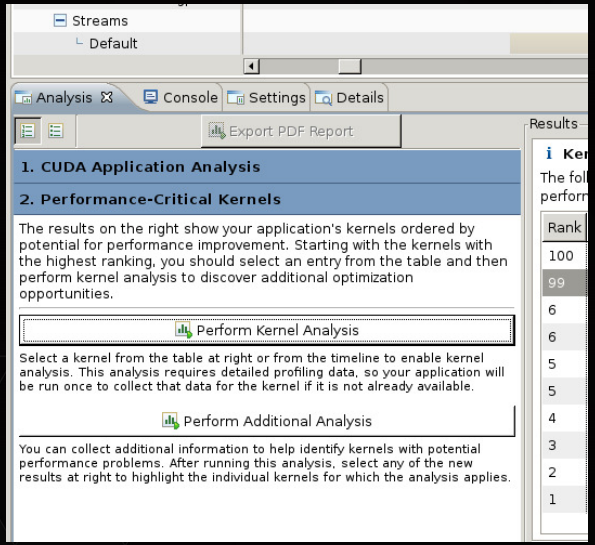
Results

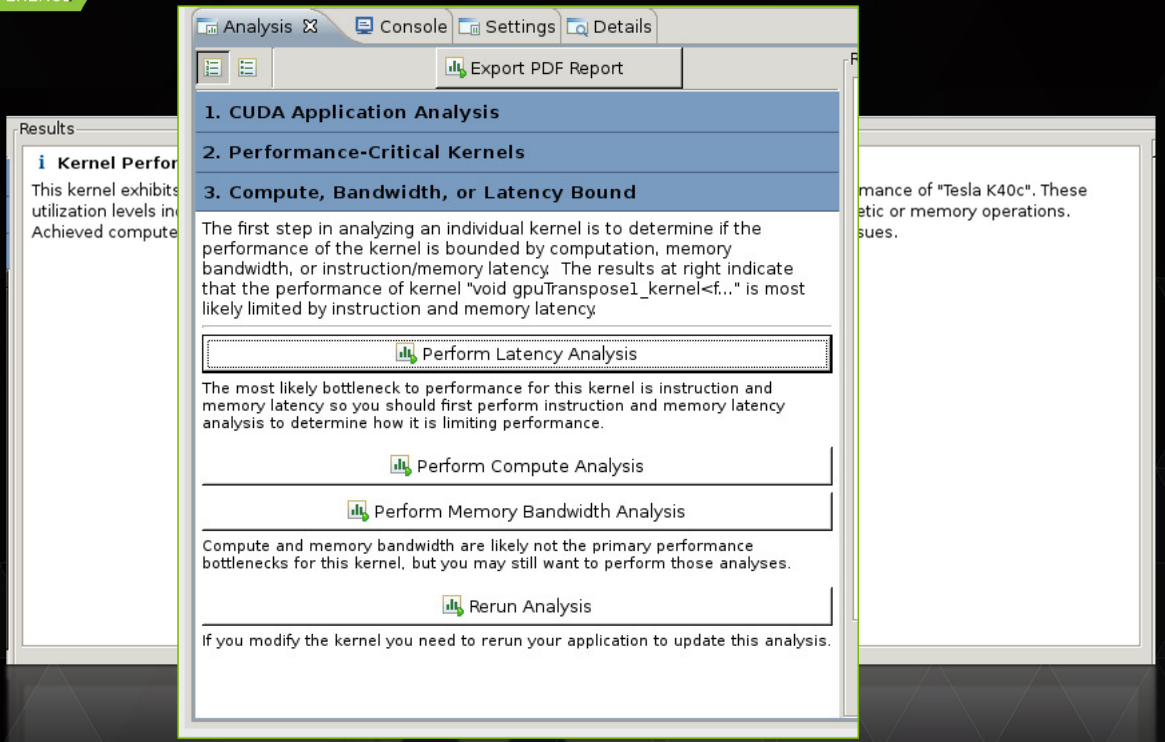
Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

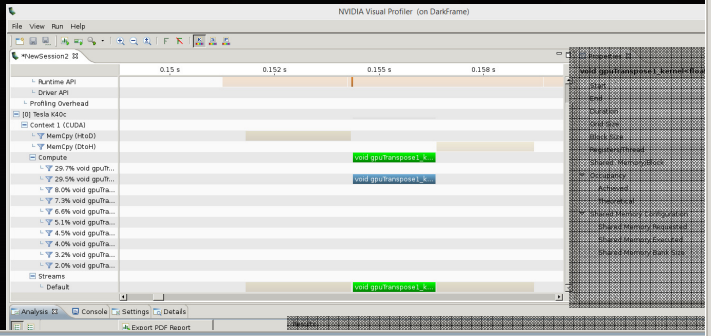
Rank	Description
100	[1 kernel instances] void gpuTranspose1_kernel<double>(int, int, double*, double*)
99	[1 kernel instances] void gpuTranspose1_kernel<float>(int, int, float*, float*)
6	[1 kernel instances] void gpuTranspose2_kernel<double>(int, int, double*, double*)
6	[1 kernel instances] void gpuTranspose3_kernel<double, int=32>(int, int, double*, double*)
5	[1 kernel instances] void gpuTranspose2_kernel<float>(int, int, float*, float*)
5	[1 kernel instances] void gpuTranspose5_kernel<double, int=32, int=8>(int, int, double*, double*)
4	[1 kernel instances] void gpuTranspose3_kernel<float, int=32>(int, int, float*, float*)
3	[1 kernel instances] void gpuTranspose4_kernel<double, int=32>(int, int, double*, double*)
2	[1 kernel instances] void gpuTranspose4_kernel<float, int=32>(int, int, float*, float*)
1	[1 kernel instances] void gpuTranspose5_kernel<float, int=32, int=8>(int, int, float*, float*)

PERFORM KERNEL ANALYSIS





LATENCY ANALYSIS



Results

Grid Size Too Small To Hide Compute And Memory Latency

The kernel does not execute enough blocks to hide memory and operation latency enough to fill the GPU with multiple "waves" of blocks. Based on theoretical occupa execute 8 blocks on each of the 15 SMs, so the kernel may need to execute a mul memory latency. If the kernel is executing concurrently with other kernels then few sharing the SMs with those kernels.

Optimization: Increase the number of blocks executed by the kernel.

Properties

void gpuTranspose1_kernel<float>(int, int, float*, float*)

Start	154.375 ms (154
End	156.352 ms (156
Duration	1.976 ms (1.976
Grid Size	[8,1,1]
Block Size	[256,1,1]
Registers/Thread	16
Shared Memory/Block	0 B
Occupancy	
Achieved	12.5%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

OCCUPANCY AND BANDWIDTH

- The memory controller can handle some number of transactions per second
  - How many airplanes can air traffic control land per minute?
    - Need the next plane “ready to land” to saturate landing pattern
- “Fully utilize the bottleneck resource”
  - Never leave the critical resource waiting for more work to do (wasted capacity)
- More Parallelism == More threads == more simultaneous loads
  - == more loads in flight
- Hiding Latency
  - GPU hides latency behind parallelism
  - Occupancy is one way to hide latency

IMPROVING THE SITUATION

- NVVP says grid is too small
  - This means insufficient parallelism
  - Consequence: Not enough loads in flight (airplanes!)
- How can we make the grid larger?
  - Less work per thread, more threads
- Initial version parallelized 1 loop, let's parallelize both loops

"You must expose additional parallelism"

TRANSPOSE WITH MORE THREADS

- Use 2D threadblock and grid
  - N total threads in each dimension
  - Each thread transposes one value of the matrix

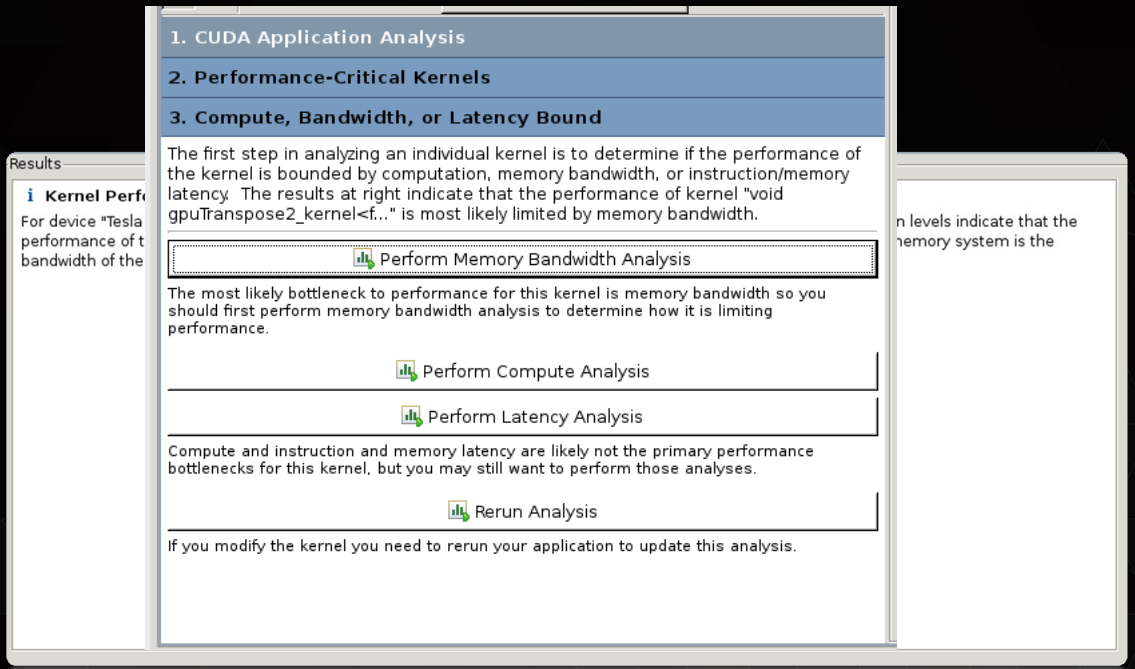
```
__global__ void transpose2(int rows, int cols, float * in, float * out)
{
  int i, j;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;

  out[i*rows + j] = in[j*cols + i];
}
```

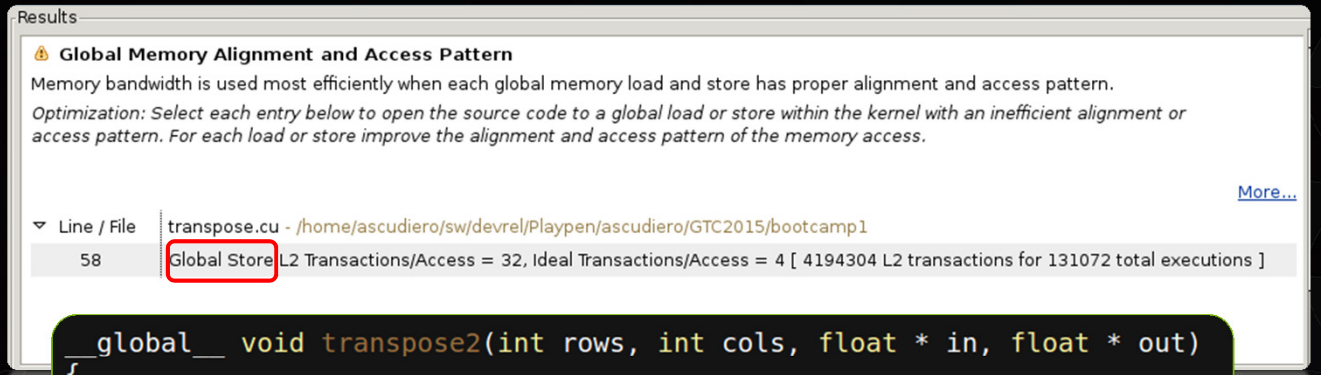
PERFORMANCE

Kernel	Float	Double
transpose1	16.96 GB/sec	33.94 GB/sec
transpose2 (2D)	76.23 GB/sec	137.878 GB/sec
Improvement:	4.5x	4.06x

Tesla K40c, Boost Clock (875mhz), ECC off



BANDWIDTH ANALYSIS




```
__global__ void transpose2(int rows, int cols, float * in, float * out)
{
  int i, j;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;

  out[i*rows + j] = in[j*cols + i];
}
```

Line 58

SIMT EXECUTION MODEL

- Launch a grid of threadblocks
  - Each thread within the entire grid is running the same kernel
- Each threadblock is assigned to a Streaming Multiprocessor (SM)
  - Within the threadblock, threads are organized into groups of 32 threads called “warps”
    - Warp groupings are predictable, based on threadIdx
  - All threads within a warp execute the same instruction simultaneously

WARP EXECUTION: CONCEPTUALLY

- 1 Program counter per warp
  - i.e. one “next” instruction
- The next instruction is issued to 32 parallel functional units
  - 32 separate register sets, one for each functional unit
  - Each functional unit ends up with different operands; same operation
  - Each functional unit generates its own output
- Repeat until all instructions are exhausted

Conceptual Model Only

SIMT, WARPS, & MEMORY

- GPU executes 32 threads simultaneously
- LD.E R2, [R6];

Half of warp
shown for clarity

VIEWS OF MEMORY ACCESS

- From the thread's perspective
  - Thread asks for a small chunk of data (4-8 bytes) from memory
    - f = array[offset + threadIdx.x];
  - This is a request or access
- From the SM's perspective
  - Issues a load for the cache line containing array[offset + threadIdx.x]
    - May need to issue many loads to satisfy a single warp request
    - Each cache line load is a replay
- From the L2 and DRAM perspective
  - If the segment is valid in L2, return it from L2
  - Otherwise, fill L2 from DRAM and send value back to SM

VIEWS OF BANDWIDTH

- Thread's view of bandwidth
  - How much data the thread receives per unit time
  - Bandwidth ~ 1 / Exposed Latency
    - Airplane Analogy: Travel latency hurts less if you can work/sleep through it
- SM's view of bandwidth
  - How much data is delivered to all threads running on the SM in unit time
  - Bandwidth ~ 1/ non-covered Latency
    - Airplane Analogy: Miami working on other projects while waiting for C5 to return
- L2 / DRAM's view of bandwidth
  - How much data is moving between DRAM and Cache/SMs per second
    - Airplane Analogy: Total planes in and out of NYC

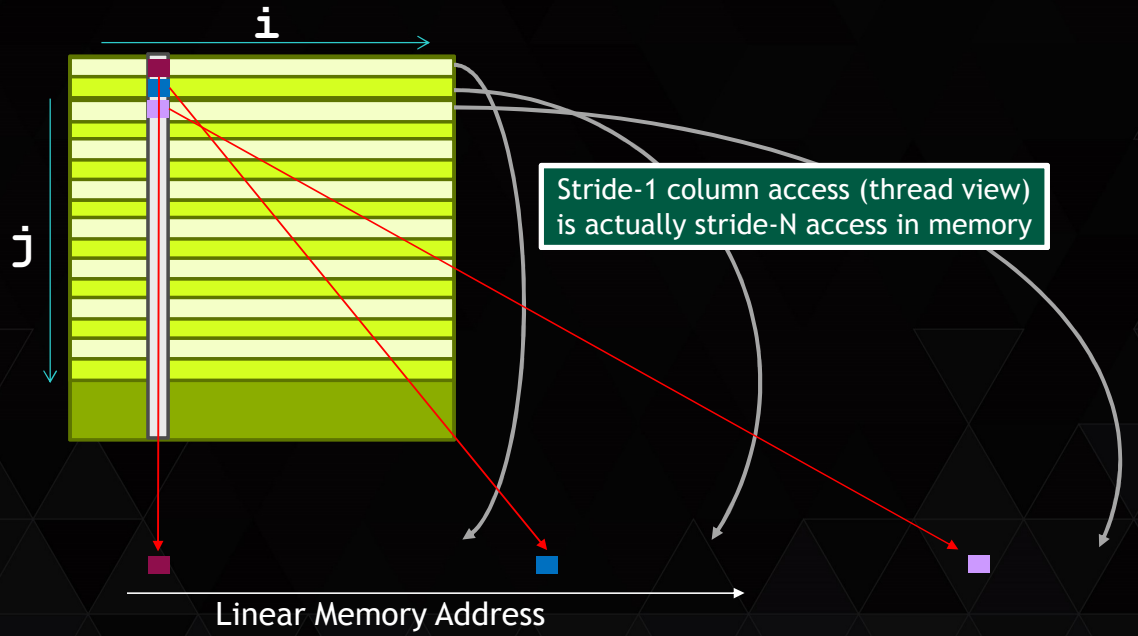
THREE VIEWS OF A LOAD

- Thread view:
  - LD.E R2, [R6];
  - This is one “Request” or “Access”
- SM view:
  - 32 memory addresses generated by the issuing warp
  - Replay load instruction for each cache line need to “cover” all 32 addresses
  - One L1 transaction per replay
- L2/DRAM view:
  - Bring in all segments for loaded cache lines (1 L2/DRAM transaction per segment)

FINALLY: COALESCING

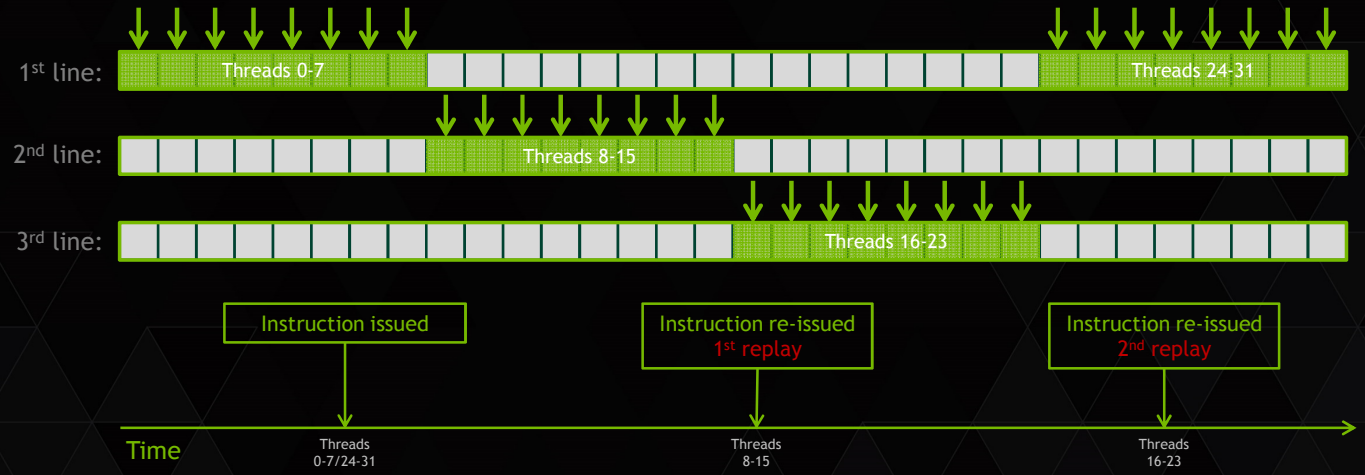
- Coalescing is the attempt to minimize the number of operations the memory system needs to do to satisfy all addresses from a single warp's memory instruction
- If multiple addresses lie within the same cache line, that line is moved only once
  - Best case: All addresses lie in a single cache line (128 Bytes)
    - SM View: Issue load for only 1 cache line
    - L2/DRAM View: 4 L2/DRAM Transactions
  - Worst case: 32 separate L1 transactions are necessary to satisfy one request
    - SM View: Issue loads for 32 cache lines (31 Replays)
    - L2/DRAM View : 128 L2/DRAM Transactions
- Applies to Reads and Writes

UNRAVELING THE MATRIX



TRANSACTIONS AND REPLAYS

- A warp reads from addresses spanning 3 lines of 128B



ACCESS PATTERNS IN DEPTH

- Traditional Optimization advice: “Stride -1 access”

```
for(int i=0; i<n; i++){
    r += data[i];
```

- Increases likelihood of cache hits
  - If sizeof(cache line) > sizeof(data type), spatial reuse
- Repeated access to the same line in DRAM
  - Access per activate

ACCESS PATTERNS IN DEPTH

- Stride-1 access looks a bit different in GPU code (thread's view)

```
// Block Stride across an array
for(int i=threadIdx.x; i<n; i+= blockDim.x) {
    r += data[i];
```

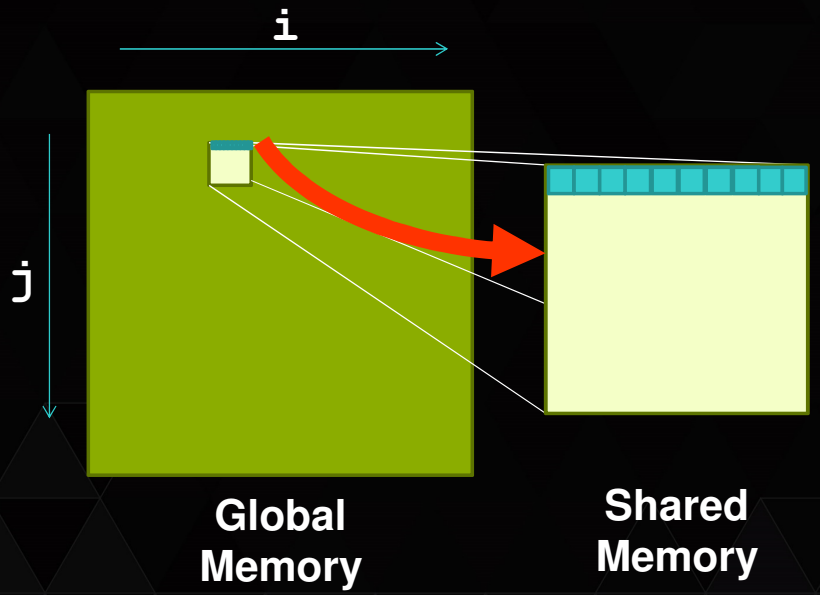
```
// Grid stride across an array
int idx = blockIdx.x*blockDim.x + threadIdx.x;
int stride = blockDim.x * gridDim.x;
for(int i=idx; i<n; i+=stride){
    r += data[i];
```

- Avoids repeated access to the same area in DRAM
  - Same intention as L1-stride access in CPU

COALESCING TRANSPOSE WRITES

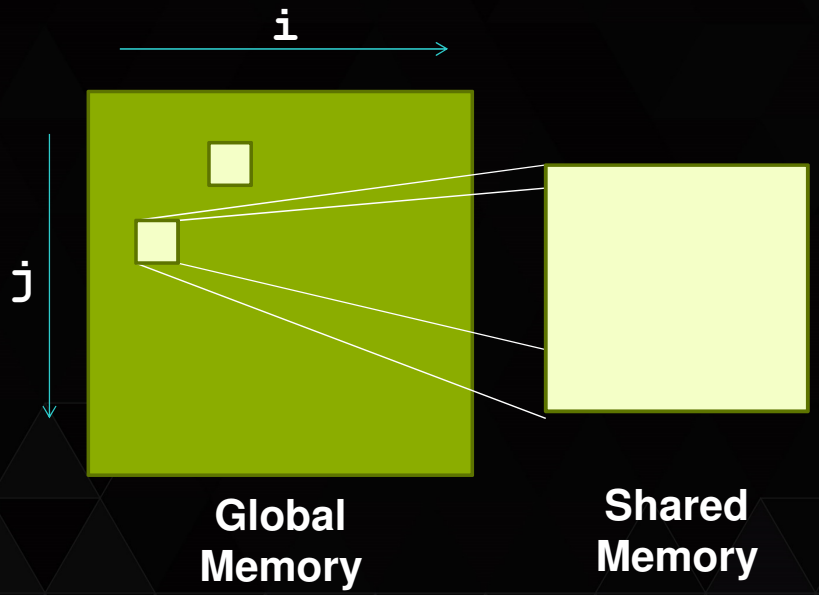
- What if we loaded a 2D “tile” of the matrix by rows into a ‘near-core’ memory
  - Transpose it in ‘near-core’ memory
  - Write the columns out as rows
- Row read / Row write allows coalescing
- And we have a programmer-controlled, near-core memory
  - “Shared Memory”

USING SHARED IN TRANSPOSE



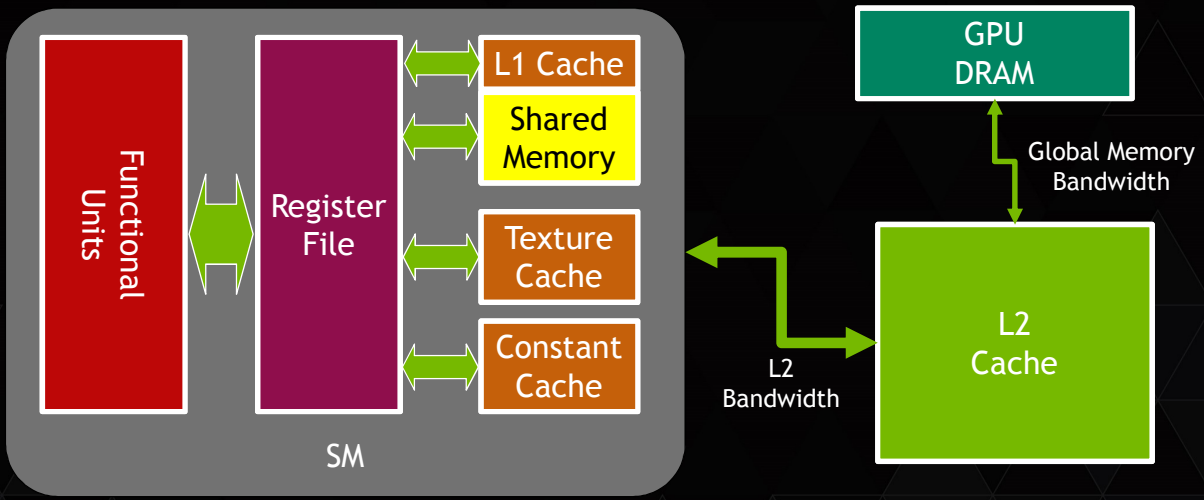
- Row is loaded fully coalesced

USING SHARED IN TRANSPOSE

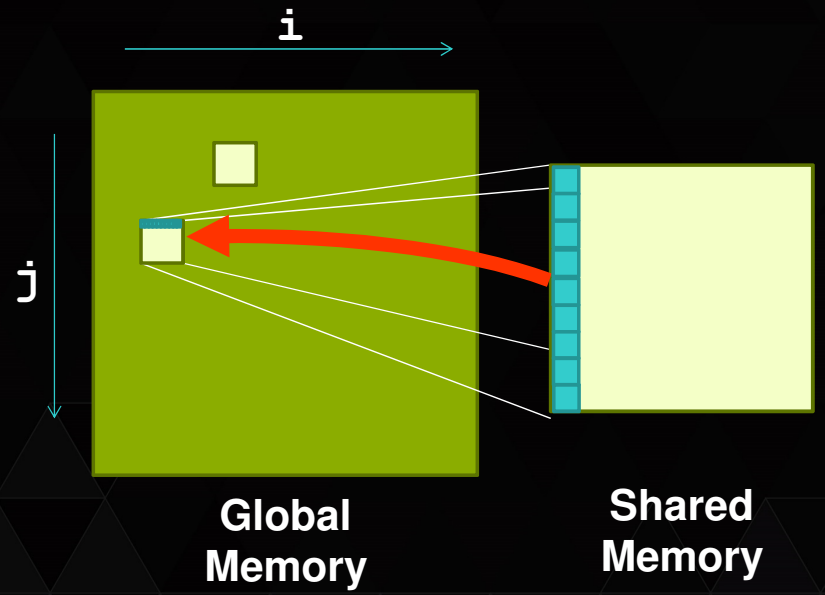


- Row is loaded fully coalesced
- Indexing: location of block is flipped across the diagonal

ON-CORE SCRATCH MEMORY



USING SHARED IN TRANSPOSE



- Row is loaded fully coalesced
- Indexing: location of block is flipped across the diagonal
- Column of shared is written to a row of the matrix
  - Transposes the Block
  - Coalesced Write

SHOW ME THE CODE

```
#define TILE_DIM 32
__global__ void transpose3(int rows, int cols, float * in, float * out)
{
  int i, j;
  __shared__ float tile[TILE_DIM][TILE_DIM];

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;
  // Stage Matrix tile to Shared memory
  tile[threadIdx.y][threadIdx.x] = in[j*cols + i];

  __syncthreads();

  i = blockIdx.y * blockDim.y + threadIdx.x;
  j = blockIdx.x * blockDim.x + threadIdx.y;
  // Write matrix tile back out, transposed
  out[j*rows + i] = tile[threadIdx.x][threadIdx.y];
}
```

Reads

Writes

SYNCTHREADS

- Thread (i,j) in the block
  - loads value (i,j) of the tile from memory
  - writes value(j,i) of the tile back to memory

- Think of the write as “consuming” the loaded value
  - A different thread “consumes” the value in shared
  - That thread must ensure the value is loaded before consuming (writing)
  - Thus, a barrier (__syncthreads();) is necessary between load and consume

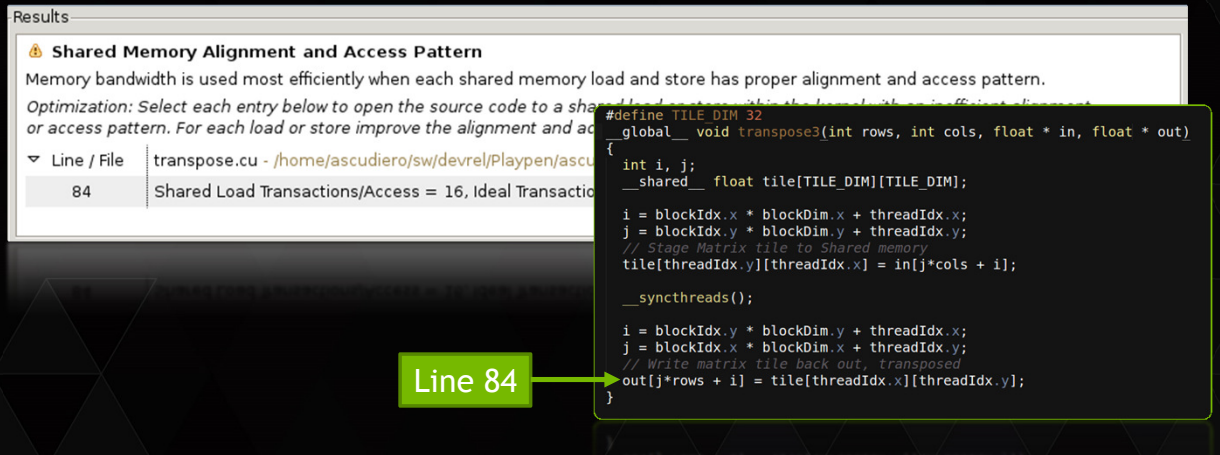
PERFORMANCE

Kernel	Float	Double
transpose1	16.96 GB/sec	33.94 GB/sec
transpose2 (2D)	76.23 GB/sec	137.878 GB/sec
transpose3 (coalesced)	101.03 GB/sec	127.86 GB/sec
Improvement	1.32x	0.92x

Tesla K40c, Boost Clock (875mhz), ECC off

BACK TO NVVP

- Perform Kernel Analysis
- Perform Memory Bandwidth Analysis



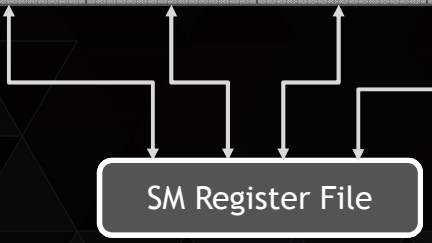
ABOUT SHARED MEMORY

- On-SM, fast local memory
  - Near register file == Low latency
  - 128 byte/clock bandwidth
- Programmer controlled “scratch” space
  - Allocate stack data in kernel with __shared__ keyword
- Programmer-visible bank layout
  - 32 Banks, new bank every 4 or 8 bytes
    - Bank = (address/4)%32
  - Can read one 32-bit word per bank per clock
  - Warp access: read 32 words from per instruction

SHARED MEMORY THOUGHT EXPERIMENT

Shared Memory

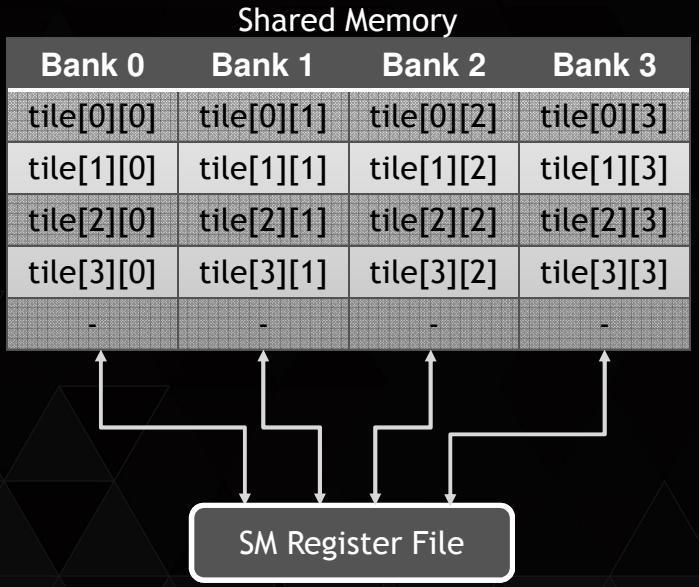
Bank 0	Bank 1	Bank 2	Bank 3
0	4	8	12
16	20	24	28
32	36	40	44
48	52	56	60
64	68	72	76



Imagine
* we have 4 banks (instead of 32)
* warps are 4 threads (instead of 32)
* 4 byte access mode

Makes this fit nicely on a slide

SHARED MEMORY THOUGHT EXPERIMENT



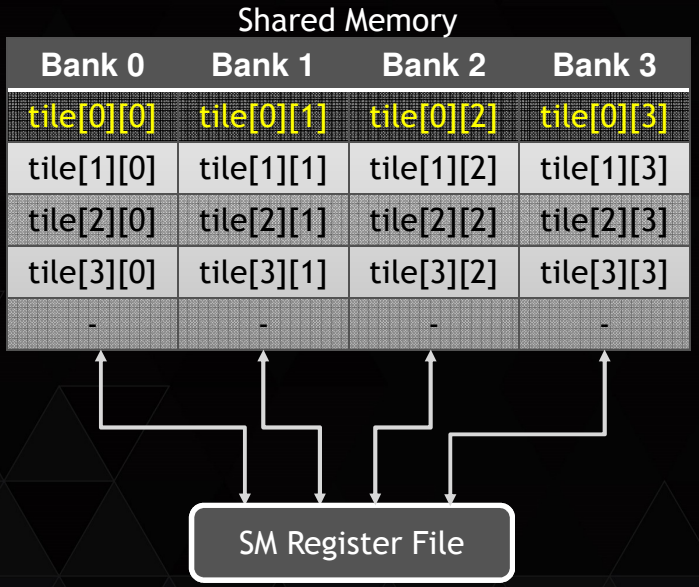

```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][4];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

SHARED MEMORY THOUGHT EXPERIMENT



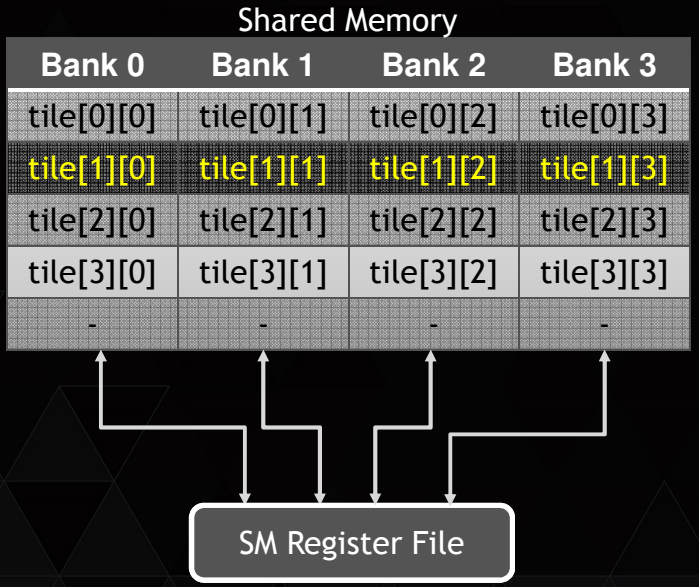

```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][4];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

SHARED MEMORY THOUGHT EXPERIMENT



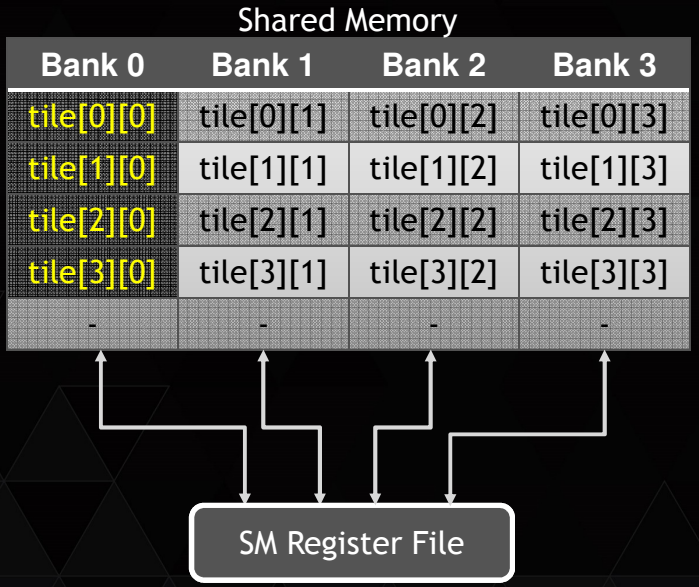

```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][4];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

SHARED MEMORY THOUGHT EXPERIMENT



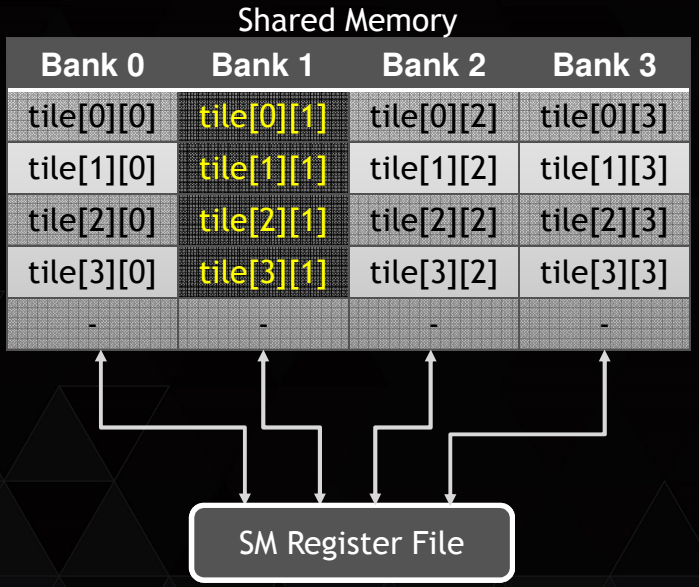

```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][4];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

SHARED MEMORY THOUGHT EXPERIMENT




```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][4];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

WARP-WISE

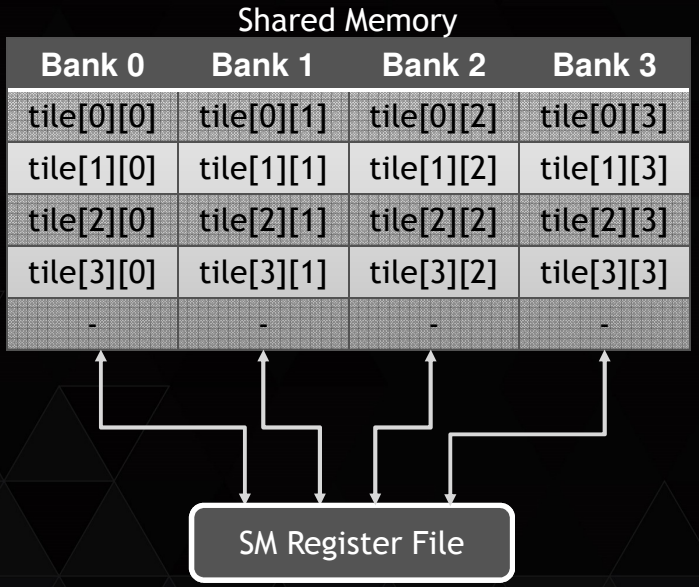
- Access by different warps will be executed on different clocks
  - No conflicts
- Bank conflicts can only occur within a warp
  - Single instruction trying to load more than one distinct value from the same shared memory bank
  - Instruction will “Replay” as many times as necessary
  - Replays are serializations
  - Replays only use a small fraction (e.g. 1/32) of available bandwidth

EFFECTS ON DRAM BANDWIDTH

- Instruction replays occupy SM and warp for many cycles
  - No useful work getting done by that warp until completed
  - Including issuing memory operations
    - Increased delay between memory operations can decrease loads in flight
    - Insufficient loads in flight will result in lower memory utilization

- Note: Row access performs, column access replays
  - You may note: This is similar to what we observed with coalescing
  - This is a coincidence
  - Shared memory bank conflicts and coalescing are disjoint issues

AVOIDING BANK CONFLICTS



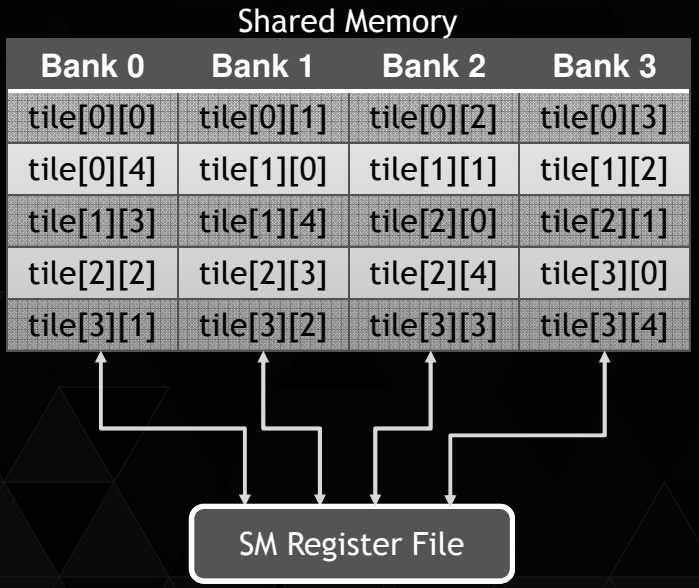

```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][4];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

AVOIDING BANK CONFLICTS



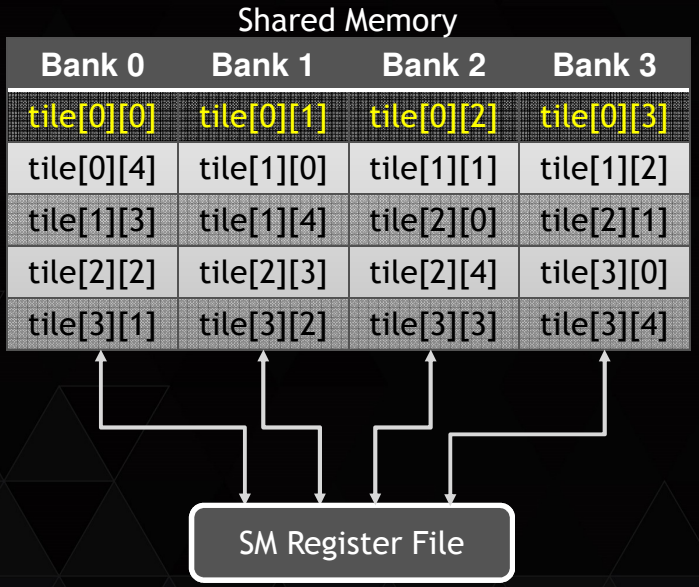

```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][5];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

AVOIDING BANK CONFLICTS



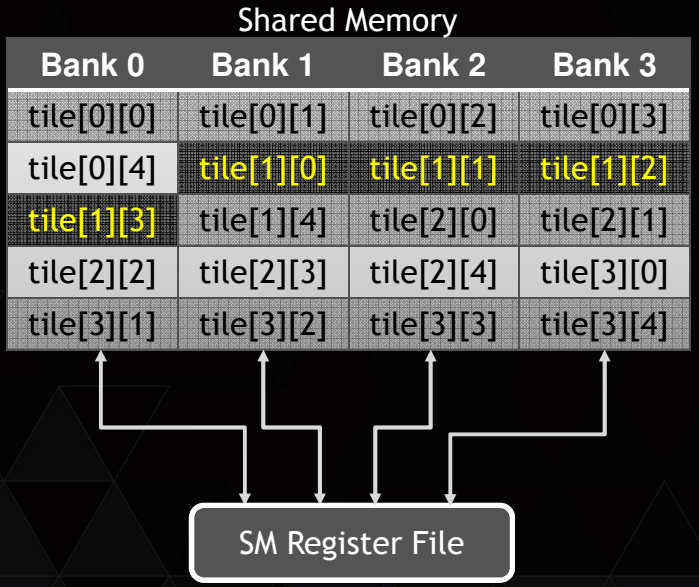

```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][5];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

AVOIDING BANK CONFLICTS



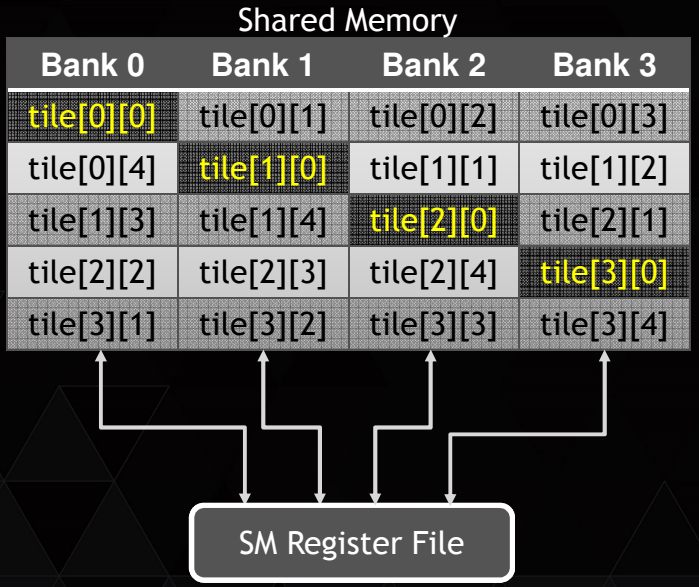

```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][5];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

AVOIDING BANK CONFLICTS



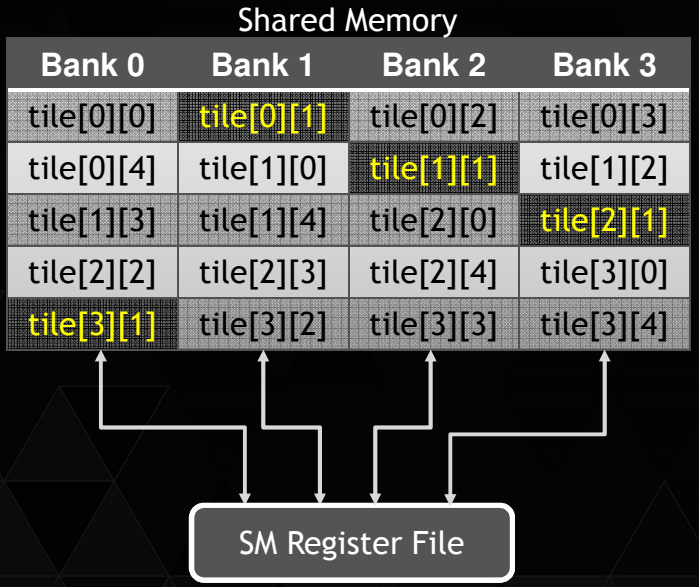

```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][5];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

AVOIDING BANK CONFLICTS




```
kernel<<<N, 4>>>(...)

__global__ kernel(...)
{
  __shared__ tile[4][5];

  // Row Access
  tile[0][threadIdx.x] = memory[index];
  tile[1][threadIdx.x] = memory[index+k];

  // Column Access
  tile[threadIdx.x][0] = memory[index];
  tile[threadIdx.x][1] = memory[index+k];
}
```

SHARED MEMORY BANK CONFLICTS

- Pad the inner (fast moving) dimension by 1
  - Yes, waste a bit of shared memory
- Row access has no bank conflicts
- Column access has no bank conflicts
- Maximize utilization of bandwidth resource

SEGUE: RAW METAL PROGRAMMING

- Memory banking is usually hidden from programmers
- Single “memory” is spread across multiple memory controllers
  - Each controller addresses many banks of memory
  - Address to controller, address to bank within controller is carefully hashed
- CPU Memory:
  - Hashed for stride-one access (thread view)
- GPU Memory:
  - Hashed for block or grid-stride access (thread view)
    - i.e. coalesced
- Remember: As the programmer, you don’t see physical addresses of DRAM

TAKE IT BACK TO TRANSPOSE

```
#define TILE_DIM 32
__global__ void transpose4(int rows, int cols, float * in, float * out)
{
  int i, j;
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;
  // Stage Matrix tile to Shared memory
  tile[threadIdx.y][threadIdx.x] = in[j*cols + i];

  __syncthreads();

  i = blockIdx.y * blockDim.y + threadIdx.x;
  j = blockIdx.x * blockDim.x + threadIdx.y;
  // Write matrix tile back out, transposed
  out[j*rows + i] = tile[threadIdx.x][threadIdx.y];
}
```

Only Change

PERFORMANCE

Kernel	Float	Double
transpose1	16.96 GB/sec	33.94 GB/sec
transpose2 (2D)	76.23 GB/sec	137.878 GB/sec
transpose3 (coalesced)	101.03 GB/sec	127.86 GB/sec
transpose4 (no bank conflicts)	171.82 GB/sec	225.68112 GB/sec
Improvement	1.70x	1.76x

Tesla K40c, Boost Clock (875mhz), ECC off

OBSERVATION

	float	double
transpose4 (no bank conflicts)	171.82 GB/sec	225.681 GB/sec

- Double precision getting better bandwidth than single precision. Why?
- Single Precision:
  - 1 Warp: 32 loads of 4 bytes each, fully coalesced
  - 1 cache line; 4 DRAM transaction/warp
- Double Precision
  - 1 Warp: 32 loads of 8 bytes each, fully coalesced
  - 2 cache lines; 8 DRAM transactions/warp

HYPOTHESIS

	float	double
transpose4 (no bank conflicts)	171.82 GB/sec	225.681 GB/sec

- Hypothesis: Double precision performs better because it gets more transactions in flight to L2/DRAM than single precision
- To say another way:
  - Issuing 2N loads per warp in single precision should rival performance of N loads per warp in double precision
  - 2 loads in flight (SP) should get about 225 GB/sec
  - We can try this

TEST THE HYPOTHESIS

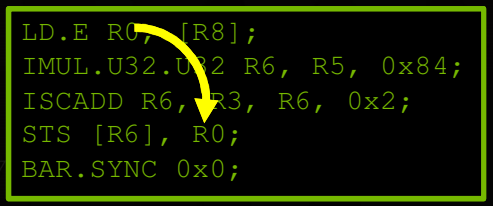
- Doubling DRAM transactions before stall for single precision should give same performance as double precision

Kernel	Float	Double
transpose4 (no bank conflicts)	171.82 GB/sec	225.681 GB/sec
transpose5 (2 elements / thread)	220.14 GB / sec	240.16 GB/sec
transpose5 (4 elements / thread)	237.42 GB / sec	240.31 GB/sec
transpose5 (8 elements / thread)	239.93 GB / sec	246.65 GB/sec
transpose5 (16 elements / thread)	225.02 GB/sec	225.84 GB/sec

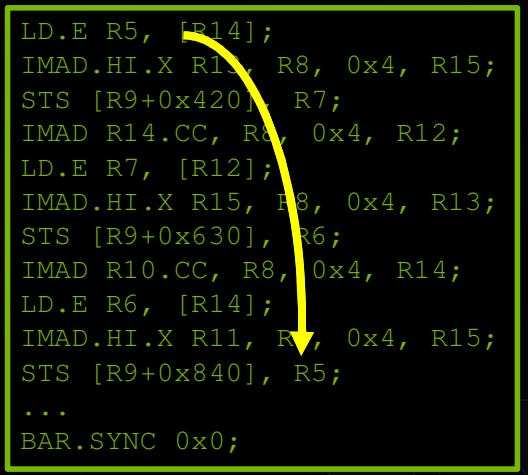
Loss of occupancy

MLP AND ILP

INSTRUCTION LEVEL PARALLELISM



Original Kernel



Kernel with ILP

WHAT IT MEANS: EXECUTION DEPENDENCY

```
#define TILE_DIM 32
__global__ void transpose4(int rows, int cols, float * in, float * out)
{
  int i, j;
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;
  // Stage Matrix tile to Shared memory
  tile[threadIdx.y][threadIdx.x] = in[j*cols + i];

  __syncthreads();

  i = blockIdx.y * blockDim.y + threadIdx.x;
  j = blockIdx.x * blockDim.x + threadIdx.y;
  // Write matrix tile back out, transposed
  out[j*rows + i] = tile[threadIdx.x][threadIdx.y];
}
```

```
LD.E R0, [R8];
IMUL.U32.U32 R6, R5, 0x84;
ISCADD R6, R3, R6, 0x2;
STS [R6], R0;

BAR.SYNC 0x0;
```

FINAL(!) PERFORMANCE

Kernel	Float	Double
transpose1	16.96 GB/sec	33.94 GB/sec
transpose2 (2D)	76.23 GB/sec	137.878 GB/sec
transpose3 (coalesced)	101.03 GB/sec	127.86 GB/sec
transpose4 (no bank conflicts)	171.82 GB/sec	225.681 GB/sec
transpose5 (multiple elements)	239.93 GB/sec	246.65 GB/sec
Speedup Achieved	**14.15x**	**7.27x**

Tesla K40c, Boost Clock (875mhz), ECC off

COME UP FOR AIR

- Ratio and Roofline analysis of hardware
  - FLOP:Byte ratios
- Loads in Flight
- Views of GPU memory
  - Thread, SM, L2/DRAM
- Optimizing Transpose, guided by NVVP
  - occupancy
  - address coalescing
  - shared memory bank access
  - instruction level parallelism

TAKEAWAYS

- Without large data reuse (ops/byte), codes are bound by operand delivery
  - Bandwidth and/or Latency to memory
- Bandwidth saturation requires many loads in flight (airplanes!)
- Best practices for GPU memory utilization
  - Address Coalescing: Efficient use of memory
    - Use shared memory to restructure load/store into coalesced patterns
  - Latency Hiding
    - Occupancy
    - Instruction Level Parallelism

FURTHER DETAILS

THANK YOU

JOIN THE CONVERSATION

#GTC15