

Hands-on CUDA Optimization

CUDA Workshop



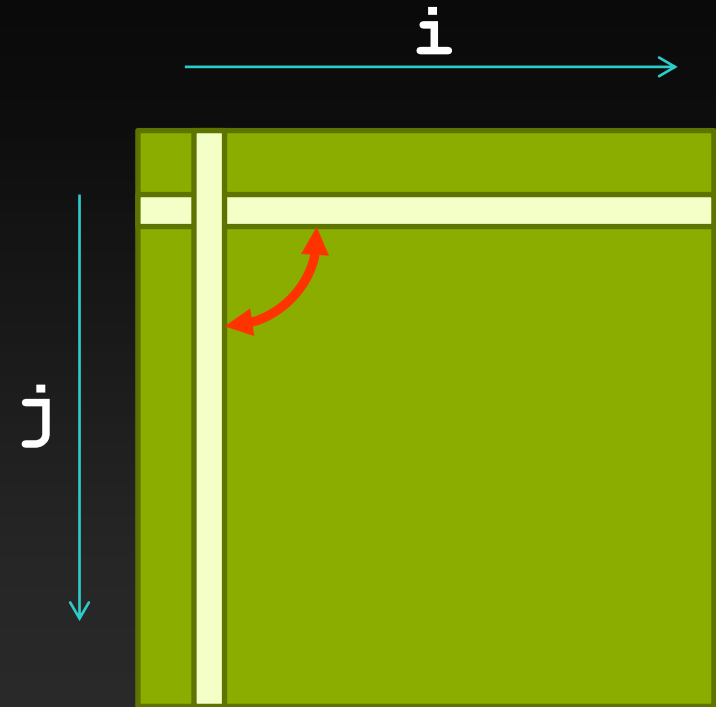
Exercise

- Today we have a progressive exercise
- The exercise is broken into 5 steps
- If you get lost you can always catch up by grabbing the corresponding directory
- If you need to peek at the solution for each step it is found in the directory named “solution”
- To start make a copy of the step1 directory
- We will now review the code

Case Study: Matrix Transpose

```
void transpose(float in[][], float out[][], int N)
{
    for(int j=0; j < N; j++)
        for(int i=0; i < N; i++)
            out[j][i] = in[i][j];
}
```

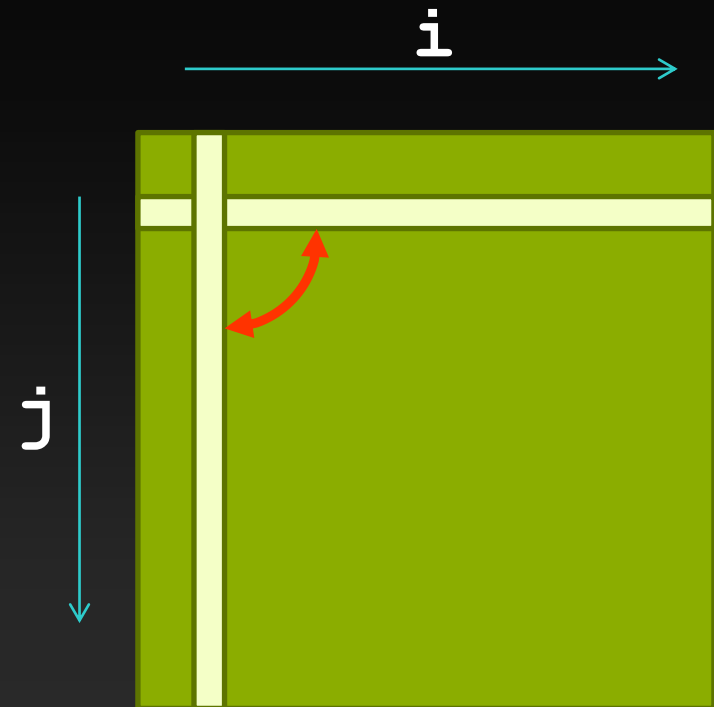
- Commonly used in applications
 - BLAS and FFT
- Stresses memory systems
 - Strided reads or writes



2D to 1D indexing

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

- This indexing is often used in numerical codes
- We will use this indexing during this presentation



Parallelization for CPU

```
void transpose(float in[], float out[], int N)
{
    #pragma omp parallel for
    for(int j=0; j < N; j++)
        #pragma omp parallel for
        for(int i=0; i < N; i++)
            out[i*N+j] = in[j*N+i];
}
```

```
%> export OMP_NUM_THREADS=16
%> aprun -n 1 -d 16 ./transpose
```

Kernel	Throughput
CPU+OMP	4.9 GB/s

Exercise: Compile with NVCC

- **Modify make file to build with nvcc**
 - For CUDA filenames must end in `.cu`
 - Specify architecture
 - `-arch=sm_35`
 - Pass an argument to the host compiler using `-Xcompiler`
 - `-Xcompiler -fopenmp`
- **Recompile and run**

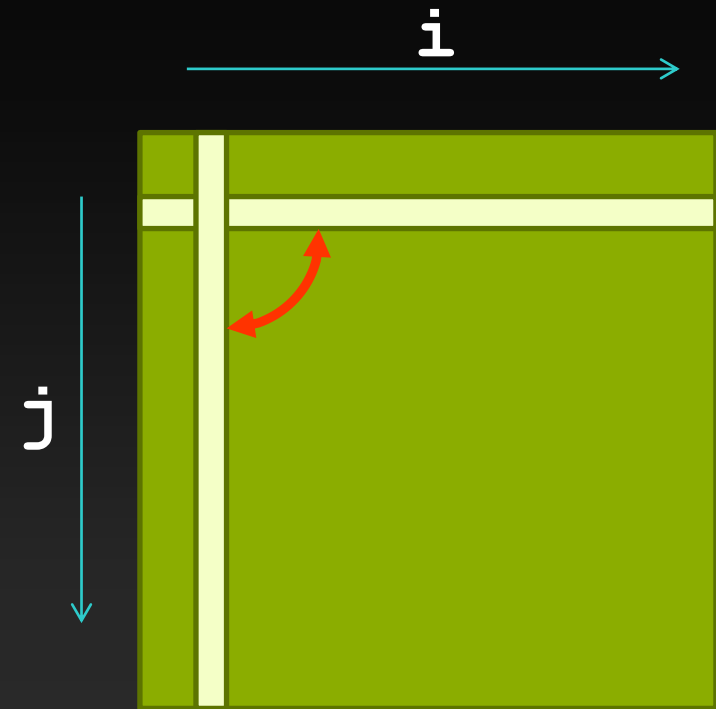
```
%> module load cudatoolkit
%> make clean
%> make
%> aprun -n 1 -d 16 ./transpose
```
- **Notice nvcc can build CPU only applications**
- **It actually passes host code through to the host compiler**

Exercise: Add CUDA APIs

- Search for “TODO” and fill in cuda code
- Start with the host code
 - Create separate pointers for CUDA memory
 - Allocate & free memory device memory
 - `cudaMalloc(**ptr, size_t size)`
 - `cudaFree(*ptr)`
 - Copy data between CPU and GPU
 - `cudaMemcpy(*dst, *src, size_t size, cudaMemcpyKind)`
 - `cudaMemcpyKind`: `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`
 - Synchronize the device to ensure timing is correct
 - `cudaDeviceSynchronize()`
 - Pass device pointers into transpose function

Exercise: Write Our First Kernel

- **Create transpose kernel**
 - `__global__` says this is a kernel
 - **Parallelize over rows**
 - 1 thread per row
 - Replace outer loop with index calculation
 - 1D indexing
 - `blockDim.x*blockIdx.x+threadIdx.x`
- **Launch kernel**
 - `<<<gridDim,blockDim>>>`
 - `blockDim = 256 threads`



Results

- Initial implementation 1.5x faster
- K20X theoretical bandwidth is 250 GB/s
 - Low percent of peak
 - Why?

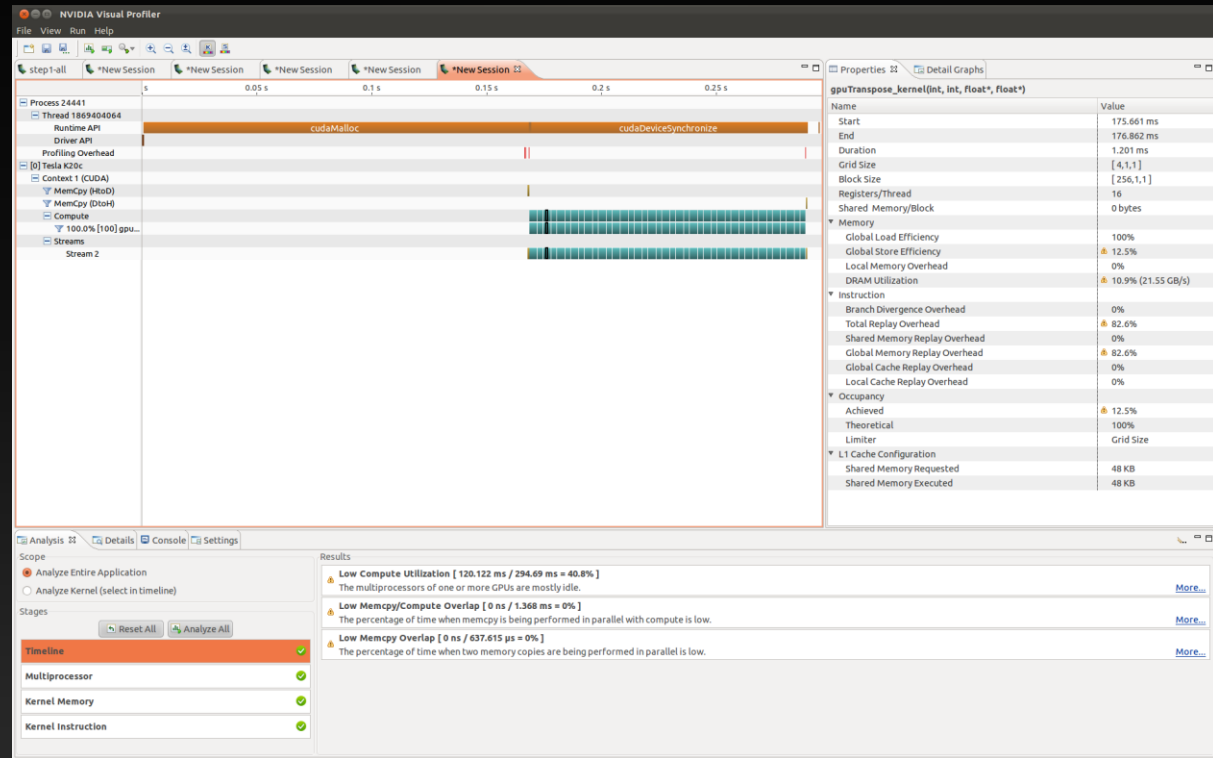
Kernel	Throughput
CPU+OMP	4.9 GB/s
CUDA-1D	7.2 GB/s

Tools for Profiling

- Profile-driven optimization
- Tools:
 - **nsight**: Visual Studio Edition or Eclipse Edition
 - **nvvp**: NVIDIA Visual Profiler
 - **nvprof**: Command-line profiling

Introducing NVVP

- **Cuda profiling tool**
 - Analyzes performance
 - Identifies hotspots
 - Suggests improvements
- **Let's open NVVP**
 - Import profiles
 - Interpret results



NVVP: Step1

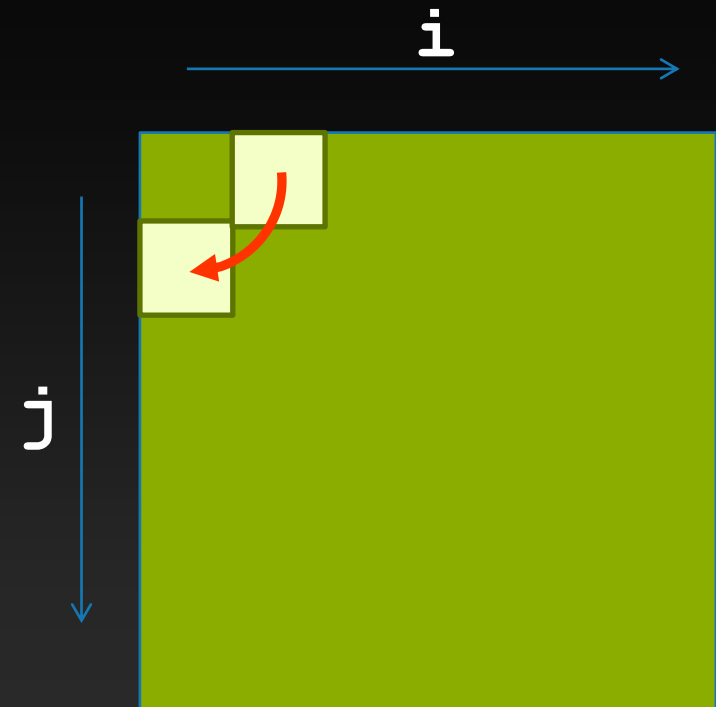
- Always look at occupancy first!
- Each block is scheduled on an SM
 - There are 14 SMs on K20X
 - Only 4 blocks!
- Bottleneck
 - Grid size
 - Most of the GPU is idle
- Solution
 - Express more parallelism

Duration	1.201 ms
Grid Size	[4,1,1]
Block Size	[256,1,1]
Registers/Thread	16
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	100%
Global Store Efficiency	⚠ 12.5%
Local Memory Overhead	0%
DRAM Utilization	⚠ 10.9% (21.55 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 82.6%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	⚠ 82.6%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	⚠ 12.5%
Theoretical	100%
Limiter	Grid Size

profiles/step1.nvvp

Exercise: Express More Parallelism

- The CPU version parallelizes over rows and columns
- Lets do the same on the GPU
 - Replace columns loop with an index calculation
 - Change launch configuration to 2D
 - `blockSize = 32x32`
 - `<<<gridDim,blockDim>>>`
 - `dim3(xdim,ydim)`
 - Don't forget to update both `gridDim` and `blockDim`



Results

- We are now at a 12x speedup over the parallel CPU version
- But how are we doing overall?
 - Peak for K20X is 250 GB/s
 - ~24% of peak
- Why is bandwidth utilization low?
- Back to NVVP

Kernel	Throughput
CPU+OMP	4.9 GB/s
GPU-1D	7.2 GB/s
GPU-2D	59 GB/s

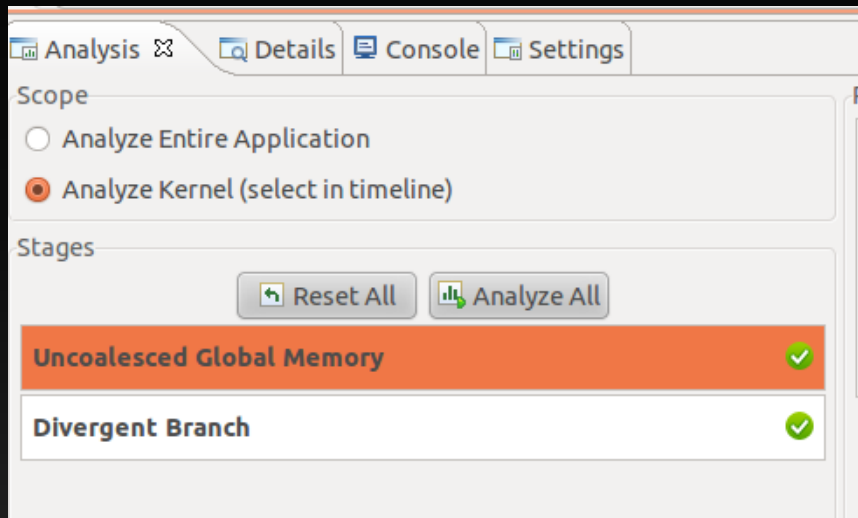
NVVP Profile: Step2

- Occupancy is now much better
- All SMs have work
- DRAM utilization is low
- Global store efficiency is low
- Global memory replay overhead is high
- Bottleneck
 - Uncoalesced stores

Duration	157.508 μ s
Grid Size	[32,32,1]
Block Size	[32,32,1]
Registers/Thread	8
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	100%
Global Store Efficiency	⚠ 12.5%
Local Memory Overhead	0%
DRAM Utilization	⚠ 35.3% (70.06 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 64.5%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	⚠ 64.5%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	75%
Theoretical	100%

profiles/step2.nvvp

Use NVVP to Find Coalescing Problems



● Compile with `-lineinfo`

```
__global__ void gpuTranspose_kernel(int rows, int cols)
{
    int i; int j;

    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;
    out[i*cols + j] = in[j*cols + i];
}
```

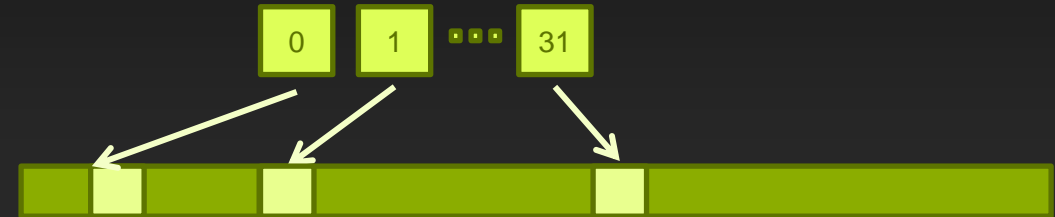
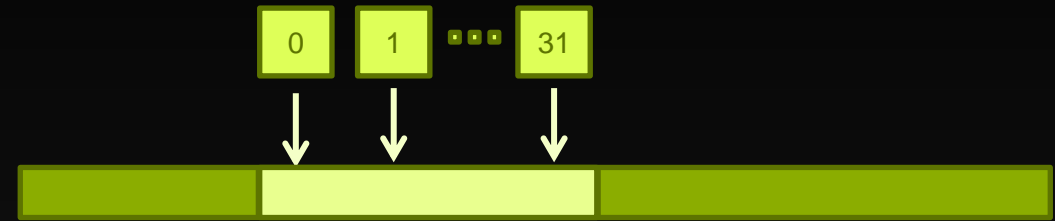
Uncoalesced Global Memory Accesses

⚠ Global memory loads and stores have poor access patterns, leading to inefficient use of global memory bandwidth. Select from the table below to see the source code which generates the inefficient global loads and stores.

Location	Description
File: main.c	
Line: 41	Global Store L2 Transactions/Access = 32.0 [1048576 L2 transactions for 32768 total executions]

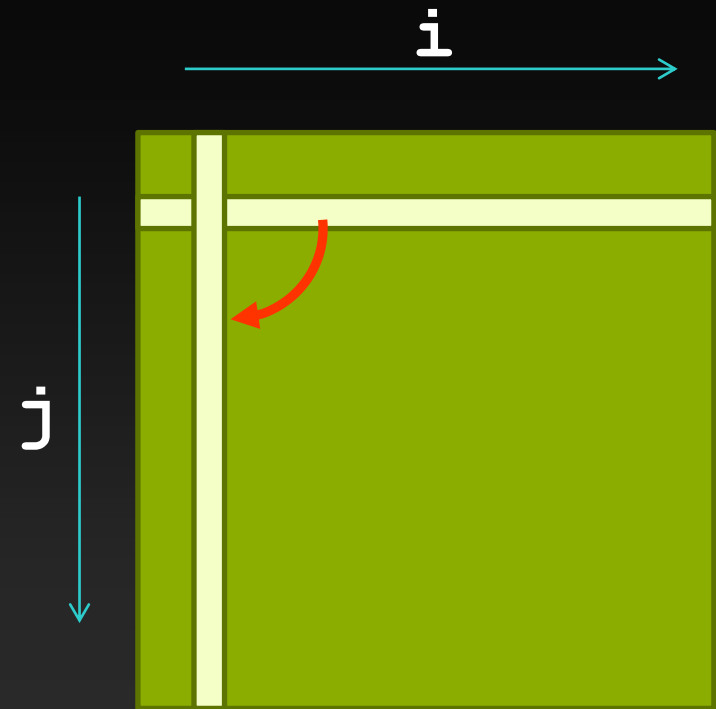
What is an Uncoalesced Global Store?

- **Global memory access happens in transactions of 32 or 128 bytes**
- **Coalesced access:**
 - A group of 32 contiguous threads (“warp”) accessing adjacent words
 - Few transactions and high utilization
- **Uncoalesced access:**
 - A warp of 32 threads accessing scattered words
 - Many transactions and low utilization

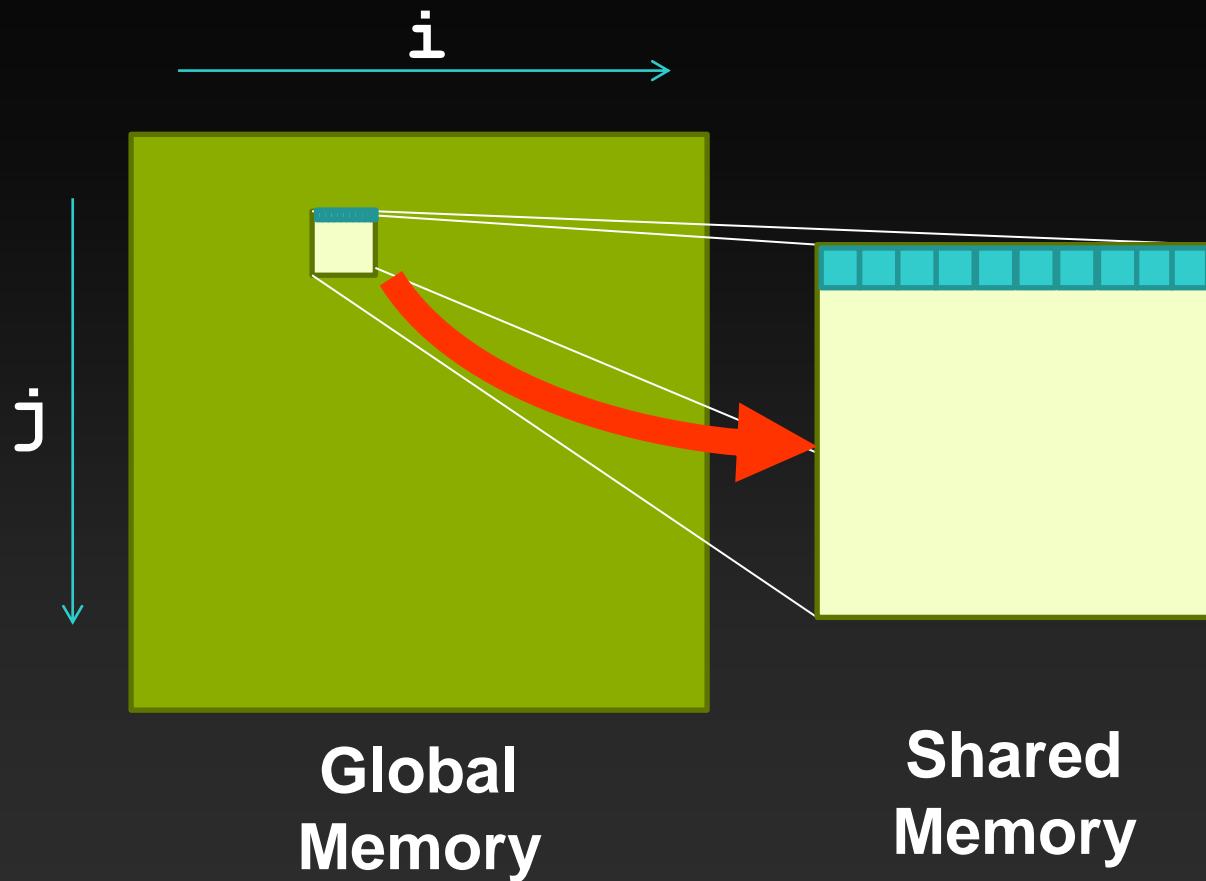


Memory Coalescing

- When we write column j memory access pattern is strided
- Solution
 - Read coalesced into shared memory
 - Transpose in shared memory
 - Write coalesced from shared memory

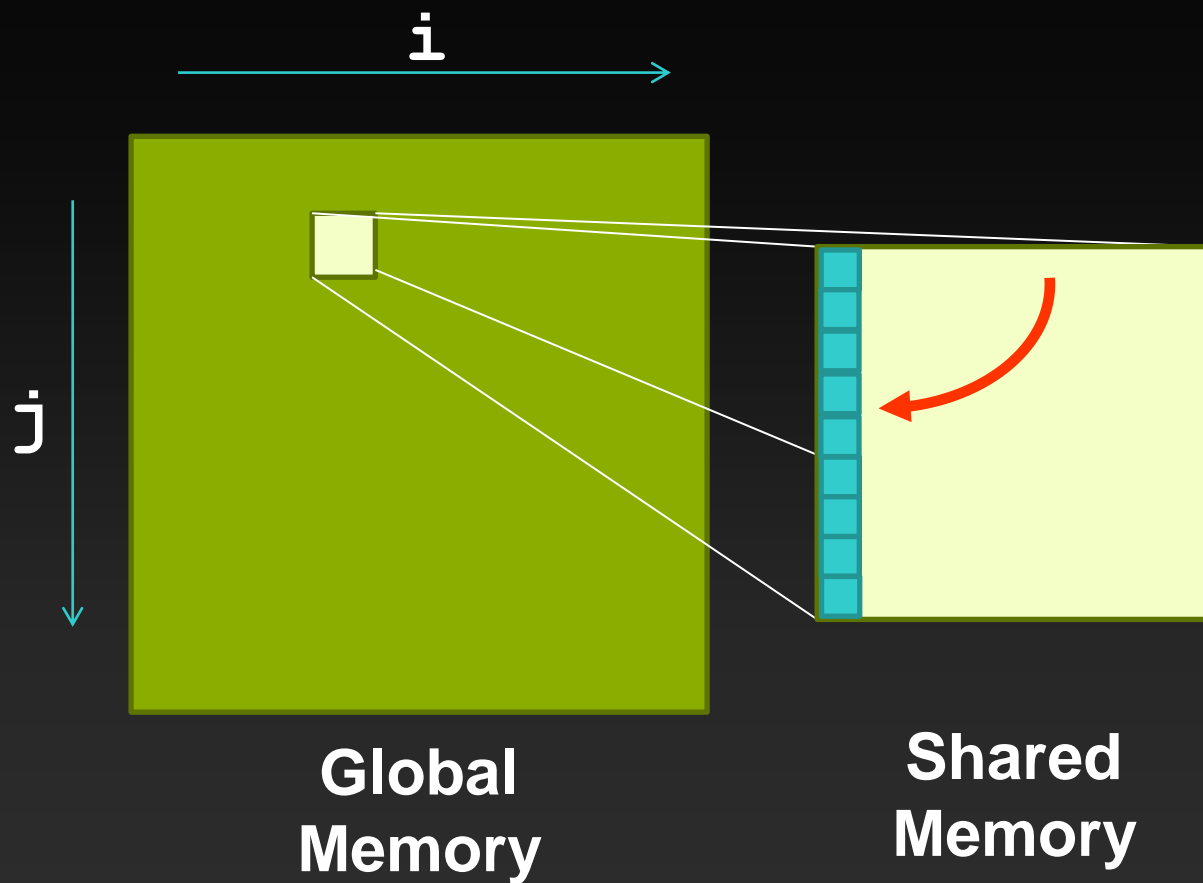


Transposing with Shared Memory



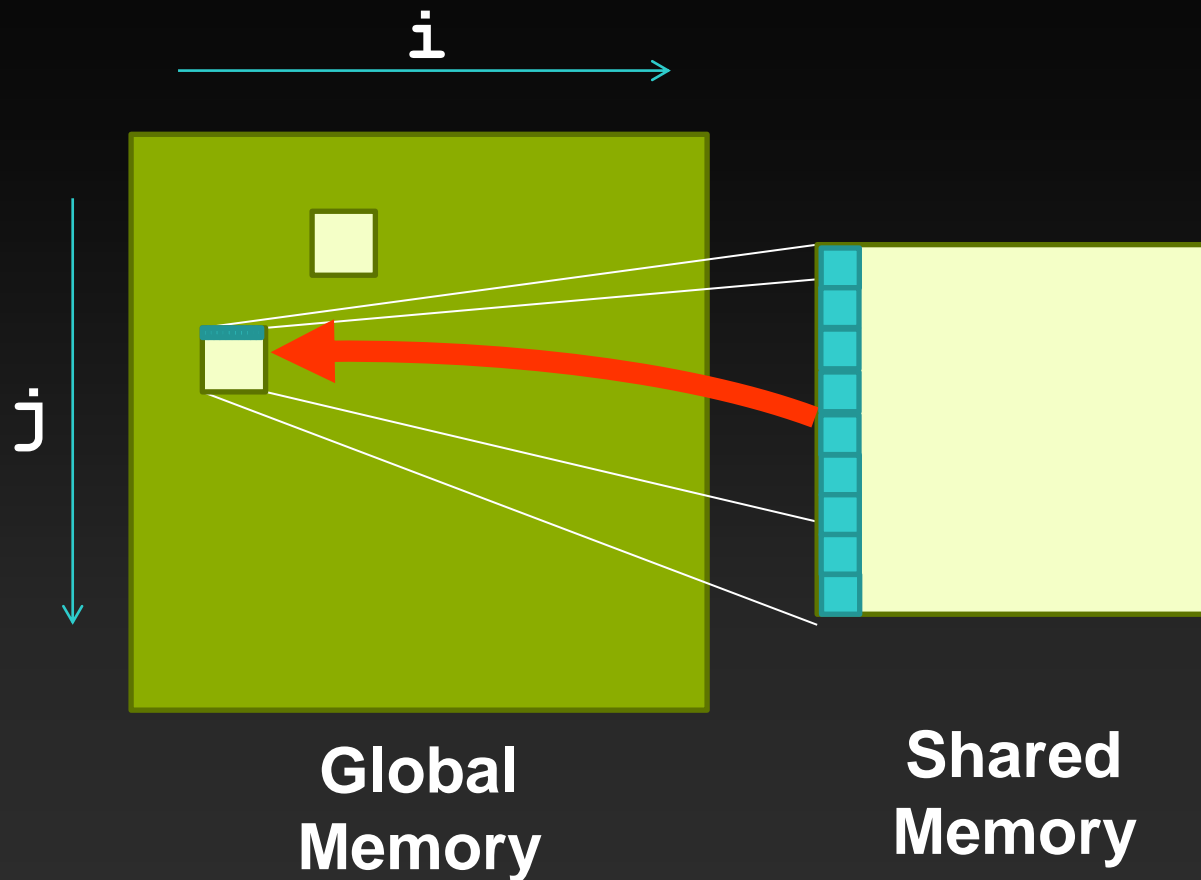
- Read block coalesced into shared memory

Transposing with Shared Memory



- Read block coalesced into shared memory
- **Transpose shared memory indices**

Transposing with Shared Memory



- Read `block_ij` coalesced into shared memory
- Transpose shared memory indices
- **Write transposed block to global memory**

Exercise: Stage Through Shared Memory

- Allocate a static 2D array using `__shared__` keyword
- Read from global to shared memory
 - Global read indices are unchanged
 - Shared write indices use `threadIdx.{x,y}`
- Write from shared to global memory
 - Global write indices: transpose block
 - Shared read indices: transpose threads
- Sync between read and write: `__syncthreads()`

Results

- We got a small improvement but we are still low compared to peak
- Back to NVVP

Kernel	Throughput
CPU+OMP	4.9 GB/s
GPU-1D	7.2 GB/s
GPU-2D	59 GB/s
GPU-Shared	73 GB/s

NVVP Profile: Step3

- Global Store Efficiency is now 100%
- Global memory replay are much lower
- Shared memory replays are much higher

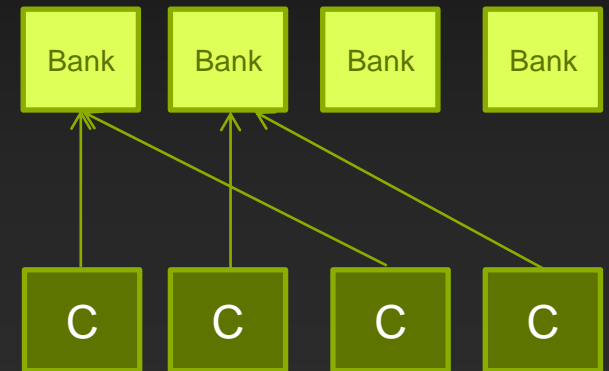
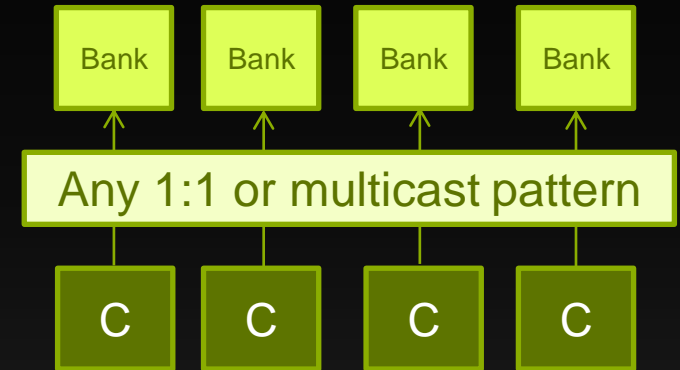
- Bottleneck
 - Shared memory bank conflicts

Duration	128.163 µs
Grid Size	[32,32,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	⚠ 37.9% (75.18 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	⚠ 36.5%
Shared Memory Replay Overhead	⚠ 30.7%
Global Memory Replay Overhead	5.8%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
Occupancy	
Achieved	86.2%
Theoretical	100%

profiles/step3.nvvp

Shared Memory Organization

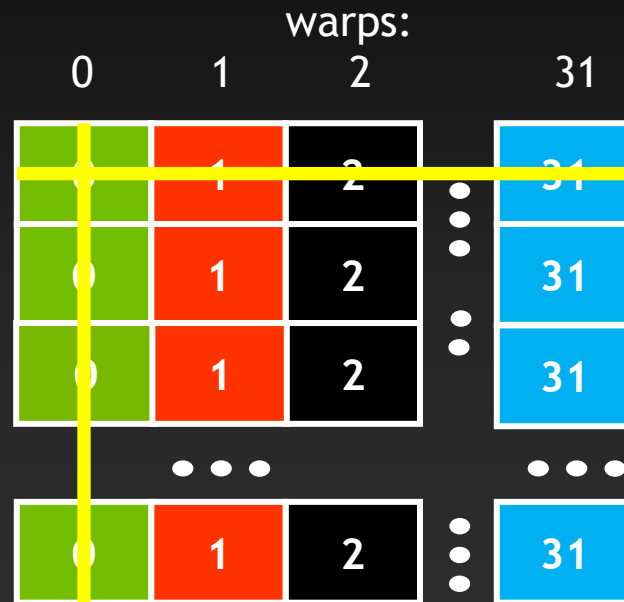
- **Organized in 32 independent banks**
- **Optimal access: all words from different banks**
 - Separate banks per thread
 - Banks can multicast
- **Multiple words from same bank serialize**



Shared Memory: Avoiding Bank Conflicts

- Example: **32x32** SMEM array
- Warp accesses a column:
 - 32-way bank conflicts (threads in a warp access the same bank)

Bank 0
Bank 1
...
Bank 31



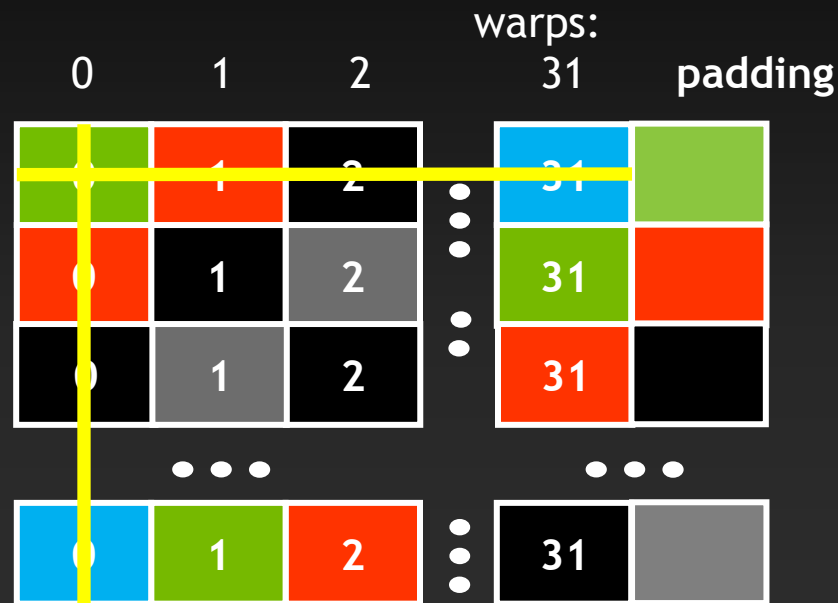
Accesses along row
produces 0 bank
conflicts

Accesses along
column produces 32
bank conflicts

Shared Memory: Avoiding Bank Conflicts

- Add a column for padding:
 - 32x33 SMEM array
- Warp accesses a column:
 - 32 different banks, no bank conflicts

Bank 0
Bank 1
...
Bank 31



Accesses along row
produces 0 bank
conflicts

Accesses along
column produces 0
bank conflicts

Exercise: Fix bank conflicts

- Add padding

Results

- Getting much better
- Back to NVVP

Kernel	Throughput
CPU+OMP	4.9 GB/s
GPU-1D	7.2 GB/s
GPU-2D	59 GB/s
GPU-Shared	73 GB/s
GPU-no-conflicts	114 GB/s

NVVP Profile: Step4

- Bank conflicts are fixed
- DRAM utilization is >50%

- Can we do better?

Duration	90.146 μ s
Grid Size	[32,32,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4.125 KB
▼ Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	57.1% (113.34 GB/s)
▼ Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	9.1%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	9.1%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
▼ Occupancy	
Achieved	86.5%
Theoretical	100%

profiles/step4.nvvp

NVVP Profile: Step4

- **DRAM Utilization is still a little low.**
 - Aim for 70%-80% of peak
- **Problem:**
 - Kepler requires 100+ lines in flight per SM to saturate DRAM
 - 1 line-in-flight per warp @ 100% occupancy = 64 lines in flight
- **Solution:**
 - Process multiple elements per thread
 - Instruction-level parallelism
 - More lines-in-flight
 - Less __syncthreads overhead
 - Amortize cost of indexing and thread launch

Duration	90.146 µs
Grid Size	[32,32,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4.125 KB
▼ Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	57.1% (113.34 GB/s)
▼ Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	9.1%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	9.1%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
▼ Occupancy	
Achieved	86.5%
Theoretical	100%

profiles/step4.nvvp

Exercise: Multiple Elements Per Thread

- **Change block size to 32 x 4**
 - **BLOCKY = 4**
 - **NUM_ELEMS_PER_THREAD = 8**
 - **Should the grid size also change?**
- **Loop over 8 elements on input**
 - **Update indexing whenever you see threadIdx.y and threadIdx.z**
- **Loop over 8 elements on output**
 - **Update indexing whenever you see threadIdx.y and threadIdx.z**
- **Unroll all loops using #pragma unroll**

NVVP Profile: Step5

- 80% of peak bandwidth
- Occupancy dropped
 - This is not a problem
 - ILP makes up for loss in occupancy
 - In general ILP is as good as high occupancy

Duration	56.13 μ s
Grid Size	[32,32,1]
Block Size	[32,4,1]
Registers/Thread	24
Shared Memory/Block	4.125 KB
▼ Memory	
Global Load Efficiency	100%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	79.9% (158.5 GB/s)
▼ Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	9.9%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	9.9%
Global Cache Replay Overhead	0%
Local Cache Replay Overhead	0%
▼ Occupancy	
Achieved	64.3%
Theoretical	68.8%

profiles/step5.nvvp

Final Results

- Use NVVP to identify bottlenecks
- Use optimization techniques to eliminate bottlenecks
- Refer to GTC archives for complete optimization techniques

Kernel	Throughput
CPU+OMP	4.9 GB/s
GPU-1D	7.2 GB/s
GPU-2D	59 GB/s
GPU-Shared	73 GB/s
GPU-no-conflicts	114 GB/s
GPU-multi-element	173 GB/s

- www.gputechconf.com/gtcnew/on-demand-gtc.php
- Search “GPU Performance Analysis and Optimization”