Memory Access Coalescing

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

March 25, 2020

Recap: Memory Spaces

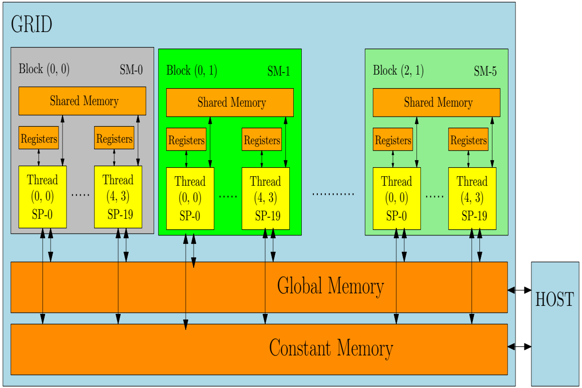


Figure: Global Memory Accesses

Access Scopes

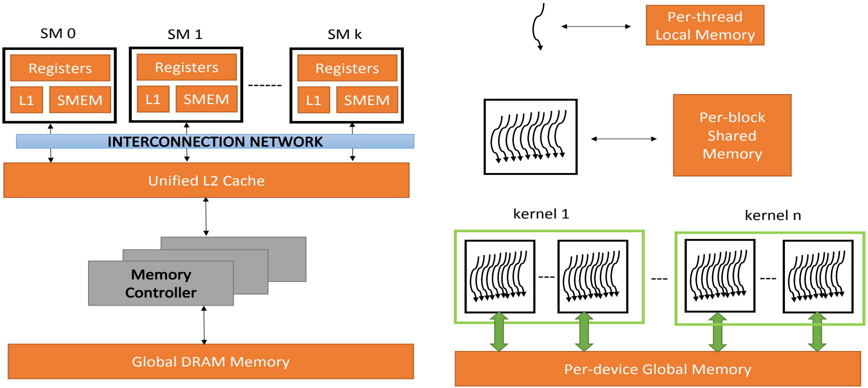


Figure: Types of Memory Accesses

Memory Access Types

Latency of accesses differ for different memory spaces

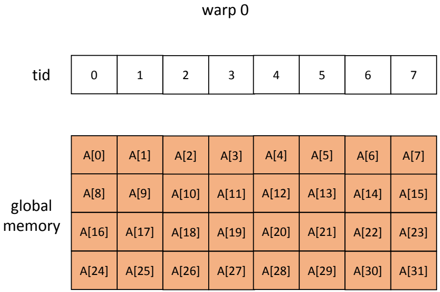
- Global Memory (accessible by all threads) is the slowest
- Shared Memory (accessible by threads in a block) is very fast.
- Registers (accessible by one thread) is the fastest.

Warp Requests to Memory

- The GPU coalesces global memory loads and stores requested by a warp of threads into global memory transactions.
- A warp typically requests 32 aligned 4 byte words in one global memory transaction.
- Reducing number of global memory transactions by warps is one of the keys for optimizing execution time
- Efficient memory access expressions must be designed by the user for the same.

Coalescing Examples

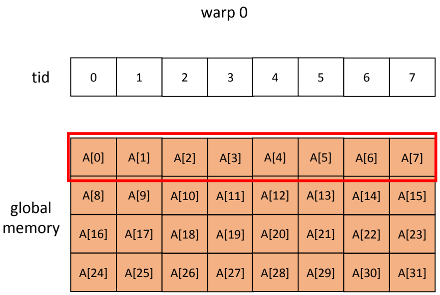
```
__global__ void memory_access(float* a)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid] = a[tid] + 1;
}
```



Coalescing Examples

```
__global__ void memory_access(float* a)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid] = a[tid] + 1;
}
```

1 global memory transaction for read
1 global memory transaction for write



Coalescing Examples

```
__global__ void memory_access(float* a)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid] = a[tid] + 1;
}
```

1 global memory transaction for read
1 global memory transaction for write

Coalescing Examples

```
__global__ void memory_access(float* a)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid] = a[tid] + 1;
}
```

1 global memory transaction for read
1 global memory transaction for write

Coalescing Examples: Offset

```
__global__ void offset_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid+s] = a[tid+s] + 1;
}
```

warp 0

tid	0	1	2	3	4	5	6	7

global memory	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]
	A[16]	A[17]	A[18]	A[19]	A[20]	A[21]	A[22]	A[23]
	A[24]	A[25]	A[26]	A[27]	A[28]	A[29]	A[30]	A[31]

Coalescing Examples: Offset

```
__global__ void offset_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid+s] = a[tid+s] + 1;
}
```

Misaligned offset access: **s=1**

2 global memory transactions for read
2 global memory transactions for write

warp 0

tid								
	0	1	2	3	4	5	6	7

global memory

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]
A[16]	A[17]	A[18]	A[19]	A[20]	A[21]	A[22]	A[23]
A[24]	A[25]	A[26]	A[27]	A[28]	A[29]	A[30]	A[31]

Coalescing Examples: Offset


```
__global__ void offset_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid+s] = a[tid+s] + 1;
}
```

Misaligned offset access: **s=1**

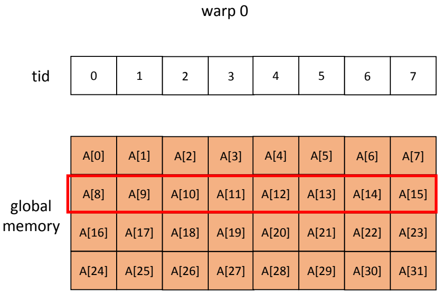
2 global memory transactions for read
2 global memory transactions for write

Coalescing Examples: Offset

```
__global__ void offset_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid+s] = a[tid+s] + 1;
}
```

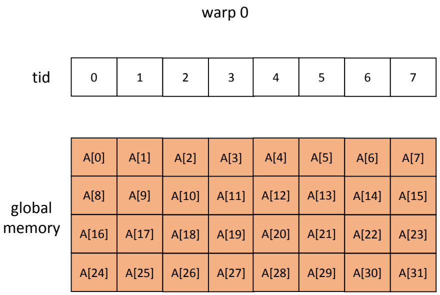
Aligned offset access: **s=8**

1 global memory transaction for read
1 global memory transaction for write



Coalescing Examples: Strided

```
__global__ void strided_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid*s] = a[tid*s] + 1;
}
```



Coalescing Examples: Strided

```
__global__ void strided_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid*s] = a[tid*s] + 1;
}
```

Misaligned strided access: **s=2**

warp 0

tid	0	1	2	3	4	5	6	7

global memory

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]
A[16]	A[17]	A[18]	A[19]	A[20]	A[21]	A[22]	A[23]
A[24]	A[25]	A[26]	A[27]	A[28]	A[29]	A[30]	A[31]

Coalescing Examples: Strided

```
__global__ void strided_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid*s] = a[tid*s] + 1;
}
```

Misaligned strided access: **s=2**

2 global memory transactions for read
2 global memory transactions for write

warp 0

tid								
	0	1	2	3	4	5	6	7

global memory

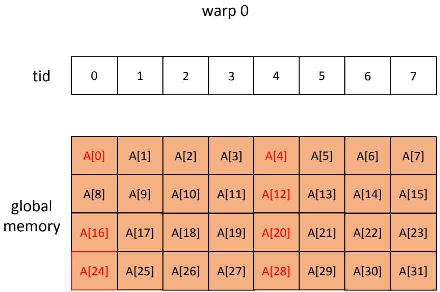
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]
A[16]	A[17]	A[18]	A[19]	A[20]	A[21]	A[22]	A[23]
A[24]	A[25]	A[26]	A[27]	A[28]	A[29]	A[30]	A[31]

Coalescing Examples: Strided

```
__global__ void strided_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid*s] = a[tid*s] + 1;
}
```

Misaligned strided access: **s=4**

2 global memory transactions for read
2 global memory transactions for write

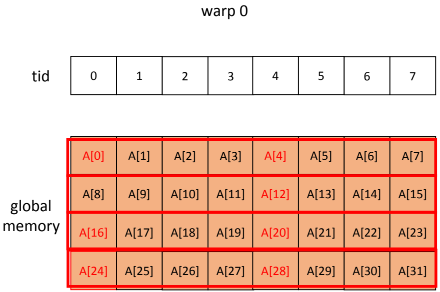


Coalescing Examples: Strided

```
__global__ void strided_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid*s] = a[tid*s] + 1;
}
```

Misaligned strided access: **s=4**

4 global memory transactions for read
4 global memory transactions for write



Profiling

- Profiling can be performed using the CUDA event API.
- CUDA events are of type cudaEvent_t
- Events are created using `cudaEventCreate()` and destroyed using `cudaEventDestroy()`
- Events can record timestamps using `cudaEventRecord()`
- The time elapsed between two recorded events is done using `cudaEventElapsedTime()`

Driver Code: Offset Access

```
cudaEvent_t startEvent, stopEvent;
float ms;
int blockSize = 1024;
int n = nMB*1024*1024/sizeof(float); //nMB=128
cudaMalloc(&d_a, n * sizeof(float));
for (int i = 0; i <= 32; i++)
{
        cudaMemset(d_a, 0.0, n * sizeof(float));
        cudaEventRecord(startEvent);
        offset_access<<n/blockSize,blockSize>>(d_a, i);
        cudaEventRecord(stopEvent);
        cudaEventSynchronize(stopEvent);
        cudaEventElapsedTime(&ms, startEvent, stopEvent);
        printf("%d, %fn", i, 2*nMB/ms);
}
```

Source:
https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/

Driver Code: Strided Access

```
cudaEvent_t startEvent, stopEvent;
float ms;
int blockSize = 1024;
int n = nMB*1024*1024/sizeof(float); //nMB=128
cudaMalloc(&d_a, n * 33 * sizeof(float));
for (int i = 0; i <= 32; i++)
{
        cudaMemset(d_a, 0.0, n * sizeof(float));
        cudaEventRecord(startEvent);
        offset_access<<n/blockSize,blockSize>>(d_a, i);
        cudaEventRecord(stopEvent);
        cudaEventSynchronize(stopEvent);
        cudaEventElapsedTime(&ms, startEvent, stopEvent);
        printf("%d, %fn", i, 2*nMB/ms);
}
```

Source:
https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/

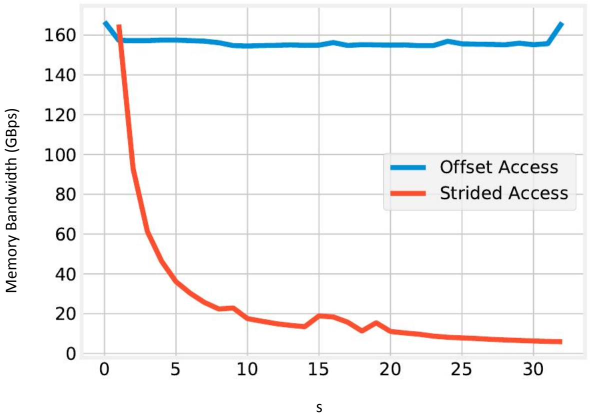


Figure: Memory Bandwidth Plot

Using Shared Memory

- Applications typically require different threads to access the same data over and over again (data reuse)
- Redundant global memory accesses can be avoided by loading data into shared memory.

Using Shared Memory

- Each SM typically has 64KB of on-chip memory that can be partitioned between L1 cache and shared memory.
- Settings are typically 48KB shared memory / 16KB L1 cache, and 16KB shared memory / 48KB L1 cache. By default the 48KB shared memory setting is used.
- This can be configured during runtime API from the host for all kernels using `cudaDeviceSetCacheConfig()` or on a per-kernel basis using `cudaFuncSetCacheConfig()`

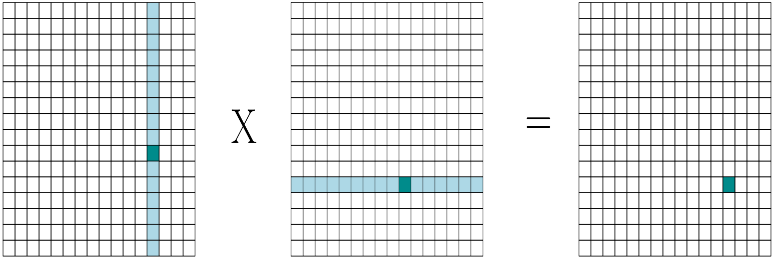
Recap: Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int N){
int i=blockIdx.y*blockDim.y+threadIdx.y;
int j=blockIdx.x*blockDim.x+threadIdx.x;
if ((i<N) && (j<N)) {
  float Pvalue = 0.0;
  for (int k = 0; k < N; ++k) {
     Pvalue += d_M[i*N+k]*d_N[k*N+j];
  }
  d_P[i*N+j] = Pvalue;
 }
}
```

Recap Matrix Multiplication Kernel

- Number of threads launched is equal to the number of elements in the matrix
- The same row and column is accessed multiple times by different threads.
- Redundant global memory accesses are a bottleneck to performance

Recap: Matrix Multiplication Kernel



\# Total Mem. accesses required

$$= N^2\,(\mathrm{N} + \mathrm{N}/32)$$

$$\approx N^3$$

Matrix Multiplication Kernel using Tiling

An alternative strategy is to use shared memory for reducing global memory traffic

- Partition the data into subsets called tiles so that each tile fits into shared memory
- Threads in a block collaboratively load tiles into shared memory before they use the elements for the dot-product calculation

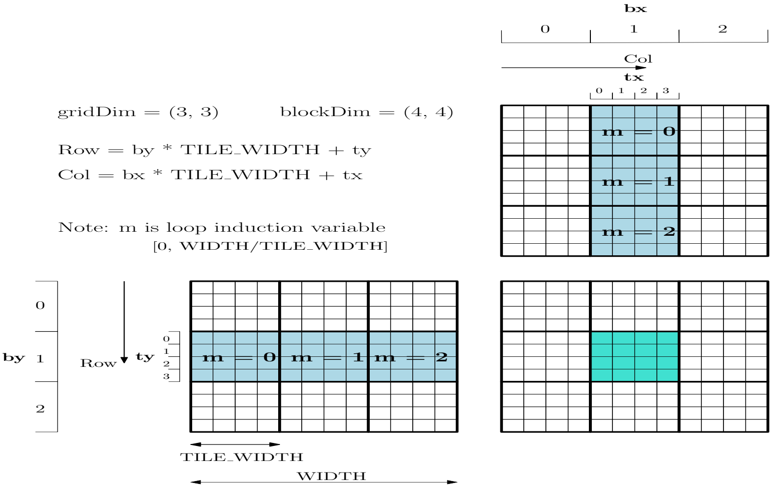


Figure: Access Expressions

Matrix Multiplication Kernel using Tiling

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P,int Width) {.

        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

         int bx = blockIdx.x;
         int by = blockIdx.y;
         int tx = threadIdx.x;
         int ty = threadIdx.y;
```

```
int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;
float Pvalue = 0;
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
 Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
 __syncthreads();
 for (int k = 0; k < TILE_WIDTH; ++k)
  Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
}
d_P[Row*Width + Col] = Pvalue;
}
```

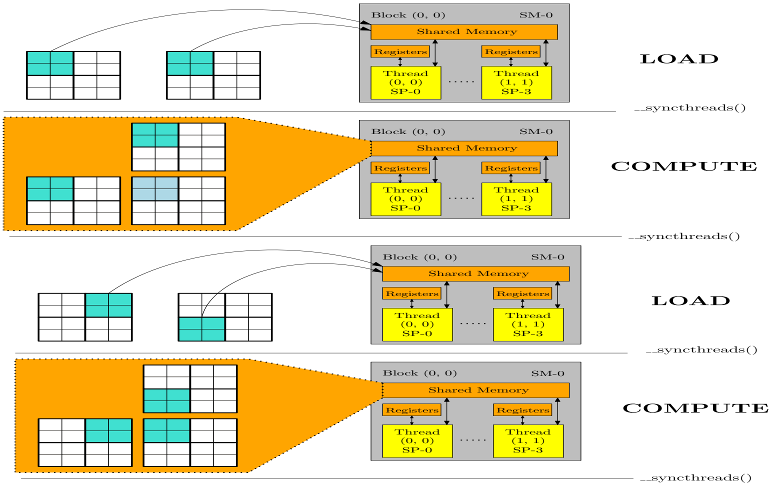


Figure: Load and compute tiles in shared memory

Mem. accesses for computing a tile in C = (# Mem. accesses to load a tile) **x** (# Tiles to load from A & B)= (W/32 x W) x (2N/W)

Total Mem. Accesses = (# Mem. accesses for computing a tile in C) x (# Tiles)
= (W/32 x W) x (2N/W) x (N^2/W^2)
= $(N^3/16W)$

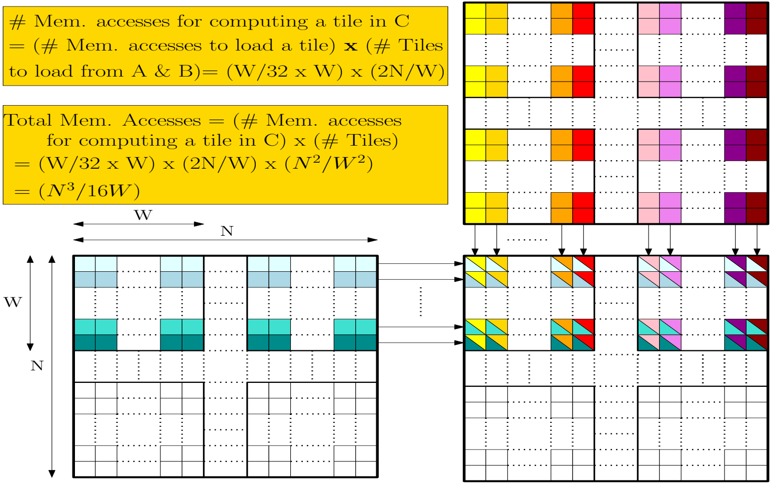


Figure: Number of memory accesses

Tranpose Operation

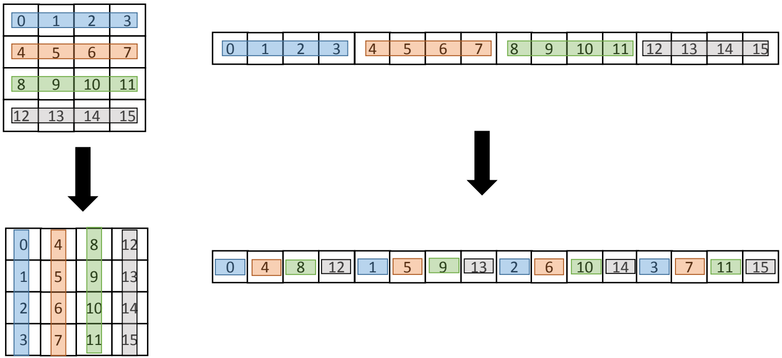


Figure: Transposing a Matrix

Matrix Transpose CPU only

```
void transposeHost(float *out, float *in, const int nx, const int ny)
{
        for (int iy = 0; iy < ny; ++iy)
        {
                for (int ix = 0; ix < nx; ++ix)
                {
                        out[ix*ny+iy] = in[iy*nx+ix];
                }
        }
}
```

Professional CUDA C Programming by Cheng et al.

Matrix Transpose GPU Kernel- Naive Row

```
__global__ void transposeNaiveRow(float *out, float *in, const int nx, int ny)
{
  unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
  if (ix < nx && iy < ny) {
    out[ix * ny + iy] = in[iy * nx + ix];
  }
}
```

Loads by rows and stores by columns

Matrix Transpose GPU Kernel- Naive Col

```
__global__ void transposeNaiveRow(float *out, float *in, const int nx,int ny)
{
  unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
  if (ix < nx && iy < ny) {
    out[iy*nx + ix] = in[ix*ny + iy];
  }
}
```

Loads by columns and stores by rows

Driver Code

```
#define CHECK(call)
{
         cudaError_t err = call;
         if (err != cudaSuccess)
         {
           fprintf(stderr , " Failed with error code %s\n", cudaGetErrorString
                (err));
           exit(EXIT_FAILURE) ;
         }
}
```

Driver Code

```
int main(int argc, char **argv)
{
    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("%s starting transpose at ", argv[0]);
    printf("device %d: %s ", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // set up array size 8192*8192
    int nx = 1 << 13;
    int ny = 1 << 13;

    // select a kernel and block size
    int iKernel = 0;
    int blockx = 32;
    int blocky = 32;

    if (argc > 1) iKernel = atoi(argv[1]);
```

Driver Code

```
size_t nBytes = nx * ny * sizeof(float);
// execution configuration
dim3 block (blockx, blocky);
dim3 grid  ((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y
    );
// allocate host memory
float *h_A = (float *)malloc(nBytes);
float *hostRef = (float *)malloc(nBytes);
float *gpuRef  = (float *)malloc(nBytes);
// initialize host array
initialData(h_A, nx * ny);
// allocate device memory
float *d_A, *d_C;
CHECK(cudaMalloc((float**)&d_A, nBytes));
CHECK(cudaMalloc((float**)&d_C, nBytes));
// copy data from host to device
CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
```

Driver Code

```
// kernel pointer and descriptor
void (*kernel)(float *, float *, int, int);
char *kernelName;
// set up kernel
switch (iKernel)
{
   case 0:
     kernel = &transposeNaiveRow; kernelName = "NaiveRow";  break;
   case 1:
     kernel = &transposeNaiveCol; kernelName = "NaiveCol"; break;
}
   // run kernel

   kernel<<<grid, block>>>(d_C, d_A, nx, ny);
        CHECK(cudaGetLastError());
        CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));
}
```

Profile using NVPROF

- nvprof is a command-line profiler available for Linux, Windows, and OS X.
- nvprof is able to collect statistics pertaining to multiple events/metrics at the same time.
- nvprof is a standalonetool and does not require the programmer to use the CUDA events API.

Execute Code: NaiveRow

nvprof --devices 0 --metrics gst_throughput, gld_throughput ./transpose 0

```
==108029== NVPROF is profiling process 108029, command: ./transpose 0
./transpose starting transpose at device 0: Tesla K40m  with matrix nx 8192 ny
     8192 with kernel 0
==108029== Some kernel(s) will be replayed on device 0 in order to collect all
     events/metrics.
==108029== Replaying kernel "transposeNaiveRow(float*, float*, int, int)" (
    done)
==108029== Metric result:
Invocations  Metric Name      Metric Description         Min         Max
Device "Tesla K40m (0)"
Kernel: transposeNaiveRow(float*, float*, int, int)
1          gst_throughput   Global Store Throughput 249.37GB/s  249.37GB/s
1          gld_throughput   Global Load Throughput  31.171GB/s  31.171GB/s
```

Execute Code: NaiveCol

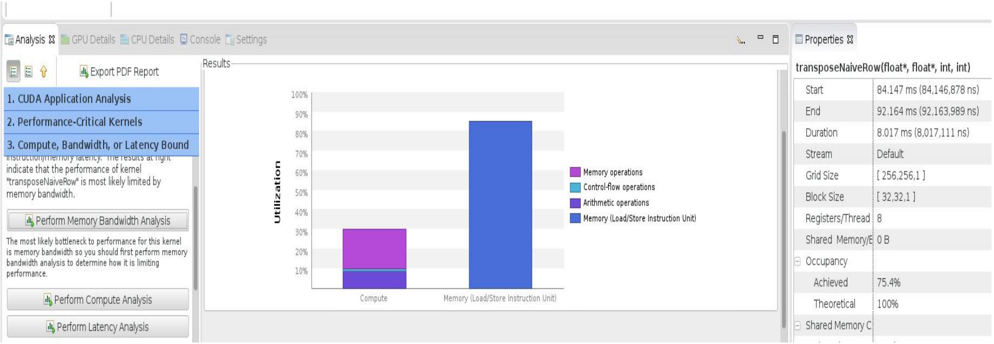
nvprof --devices 0 --metrics gst_throughput, gld_throughput ./transpose 1

```
==108037== NVPROF is profiling process 108037, command: ./transpose 1
./transpose starting transpose at device 0: Tesla K40m  with matrix nx 8192 ny
     8192 with kernel 1
==108037== Some kernel(s) will be replayed on device 0 in order to collect all
     events/metrics.
==108037== Replaying kernel "transposeNaiveCol(float*, float*, int, int)" (
    done)
==108037== Metric result:
Invocations    Metric Name Metric Description        Min         Max
Device "Tesla K40m (0)"
Kernel: transposeNaiveCol(float*, float*, int, int)
1          gst_throughput  Global Store Throughput  17.421GB/s  17.421GB/s
1          gld_throughput  Global Load Throughput   139.37GB/s  139.37GB/s
```

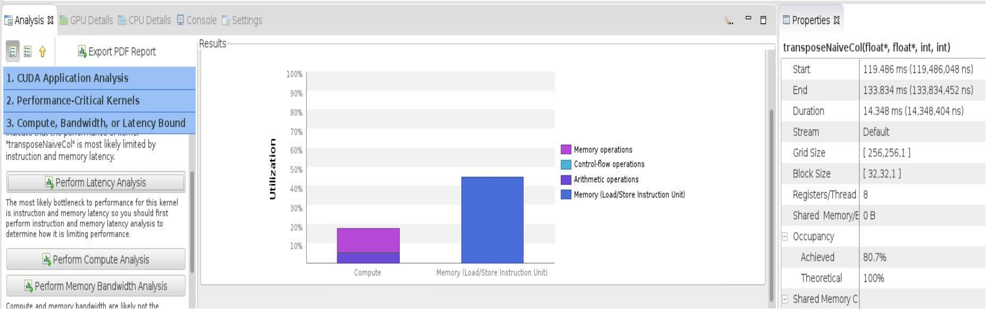
Using Nvidia Visual Profiler

- The nvvp software provides a GUI based tool for analyzing CUDA applications and supports a guided analysis mode for optimizing kernels.
- nvprof provides a *–analysis-metrics* option to capture all GPU metrics for use by NVIDIA Visual Profiler software during its guided analysis mode.
- The -o flag can be used with nvprof to dump a logs file that can be imported into nvvp.

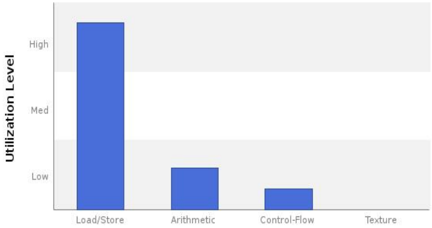
Naive Row Kernel Profiling Analysis



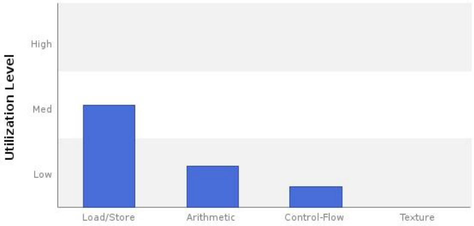
Naive Col Kernel Profiling Analysis



Compute Analysis



Naive Row



Naive Col

Memory Bandwidth Analysis: Naive Row

L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	2097152	33.483 GB/s	
Global Stores	67108864	267.863 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	69206016	301.345 GB/s	Idle Low Medium High Max

L2 Cache

L1 Reads	8388608	33.483 GB/s	
L1 Writes	67108864	267.863 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	75497472	301.345 GB/s	Idle Low Medium High Max

Memory Bandwidth Analysis: Naive Col

L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	67108864	149.667 GB/s	
Global Stores	2097152	18.708 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	69206016	168.375 GB/s	Idle Low Medium High Max

L2 Cache

L1 Reads	67108864	149.667 GB/s	
L1 Writes	8388608	18.708 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	75497472	168.375 GB/s	Idle Low Medium High Max

Texture Cache

Latency Analysis in NVVP

Instruction stalls prevents warps from executing on any given cycle and are of the following types.

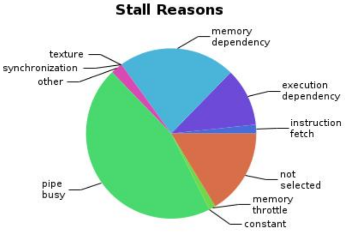
- **Pipeline busy:** The compute resources required by the instruction is not available.
- **Constant:** A constant load is blocked due to a miss in the constants cache.
- **Memory Throttle:** Large number of pending memory operations prevent further forward progress.
- **Texture:** The texture subsystem is fully utilized or has too many outstanding requests.
- **Synchronization:** The warp is blocked at a __syncthreads() call.

Latency Analysis in NVVP

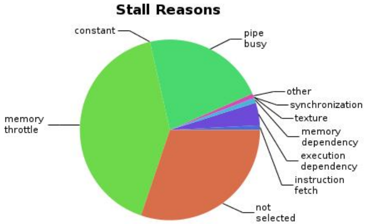
Instruction stalls prevents warps from executing on any given cycle and are of the following types.

- **Instruction Fetch:** The next assembly instruction has not yet been fetched.
- **Execution Dependency:** An input required by the instruction is not yet available.
- **Memory Dependency:** A load/store cannot be made because the required resources are not available, or are fully utilized, or too many requests of a given type are oustanding.
- **Not Selected:** Warp was ready to issue, but some other warp was issued instead.

Latency Analysis



Naive Row



Naive Col

Transpose using Shared Memory

```
#define TILE_DIM 32
#define BLOCK_ROWS 32
__global__ void transposeCoalesced(float *odata, float *idata, const int nx,
    const int ny)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();
```

Source: https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/

Transpose using Shared Memory

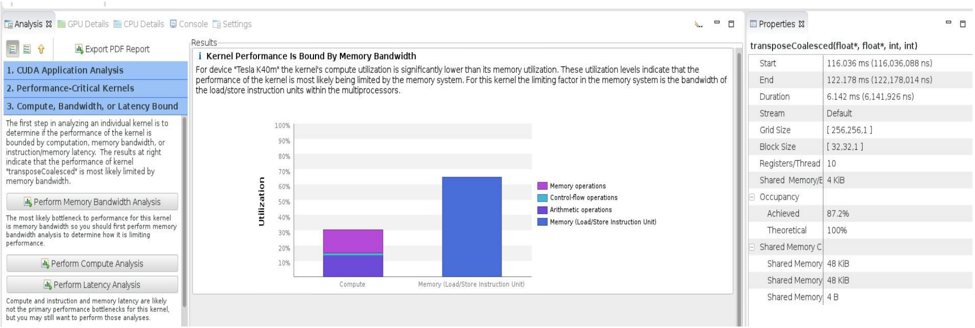
```
    x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
    y = blockIdx.x * TILE_DIM + threadIdx.y;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
      odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Execute Code: TransposeCoalesced

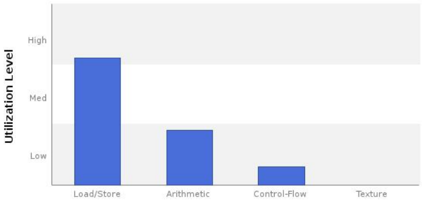
nvprof –devices 0 –metrics shared_store_throughput,shared_load_throughput ./transpose 2

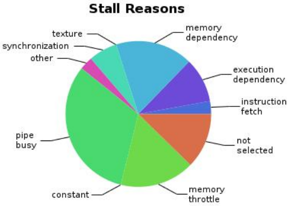
```
==108373== NVPROF is profiling process 108373, command: ./transpose 2
./transpose starting transpose at device 0: Tesla K40m  with matrix nx 8192 ny
     8192 with kernel 2
==108373== Metric result:
Invocations  Metric Name       Metric Description           Min        Max
Device "Tesla K40m (0)"
Kernel: transposeCoalesced(float*, float*, int, int)
1  shared_store_throughput Shared Memory Store Throughput 81.40GB/s 81.40GB/s
1  shared_load_throughput  Shared Memory Load Throughput  1e+03GB/s  1e+03GB/s
```

Kernel Analysis



Compute and Latency Analysis





Memory Bandwidth Analysis

L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	33554432	1,398.573 GB/s	
Shared Stores	2097502	87.425 GB/s	
Global Loads	2097152	43.705 GB/s	
Global Stores	2097152	43.705 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	39846238	1,573.41 GB/s	Idle Low Medium High Max

L2 Cache			
L1 Reads	8388608	43.705 GB/s	
L1 Writes	8388608	43.705 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	16777216	87.411 GB/s	Idle Low Medium High Max

Using Shared Memory: Simple Copy

```
__global__ void copySharedMem(float *odata, float *idata, const int nx, const
    int ny)
{
  __shared__ float tile[TILE_DIM * TILE_DIM];
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] = idata[(y+j)*width + x];
  __syncthreads();
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x];
}
```

Profiling Results: CopySharedMem

Results

i Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	2097152	139.383 GB/s	
Shared Stores	2108698	140.15 GB/s	
Global Loads	2097152	69.691 GB/s	
Global Stores	2097152	69.691 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	8400154	418.915 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	8388608	69.691 GB/s	
L1 Writes	8388608	69.691 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	16777216	139.383 GB/s	Idle Low Medium High Max

copySharedMem(float*, float*, int, int)

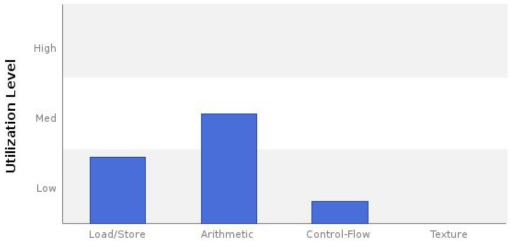
Start	83.947 ms (83,94
End	87.798 ms (87,79
Duration	3.852 ms (3,851,7
Stream	Default
Grid Size	[256,256,1]
Block Size	[32,32,1]
Registers/Thread	8
Shared Memory/Block	4 KiB
Occupancy	
Achieved	87.8%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

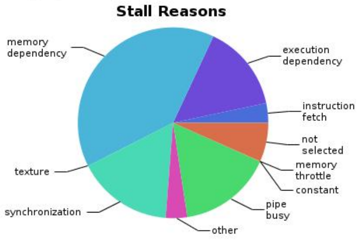
No Bank Conflicts

```
__global__ void transposeNoBankConflicts(float *odata, float *idata, const int
     nx, const int ny)
{
   __shared__ float tile[TILE_DIM][TILE_DIM+1];
   int x = blockIdx.x * TILE_DIM + threadIdx.x;
   int y = blockIdx.y * TILE_DIM + threadIdx.y;
   int width = gridDim.x * TILE_DIM;
   for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];
   __syncthreads();
   x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
   y = blockIdx.x * TILE_DIM + threadIdx.y;
   for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

No Bank Conflicts

Profiling Results: No bank conflicts





Profiling Results: No bank conflicts

Export PDF Report

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
4. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel. The results at right indicate that the kernel is limited by the bandwidth available to the device memory.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Results

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	2097152	131.19 GB/s	
Shared Stores	2108993	131.931 GB/s	
Global Loads	2097152	65.595 GB/s	
Global Stores	2097152	65.595 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	8400449	394.31 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	8388608	65.595 GB/s	
L1 Writes	8388608	65.595 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	16777216	131.19 GB/s	Idle Low Medium High Max

transposeNoBankConflicts(float*, float*, int, int)

Start	122.132 ms (122,
End	126.225 ms (126,
Duration	4.092 ms (4,092,3
Stream	Default
Grid Size	[256,256,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4.125 KiB
Occupancy	
Achieved	88.6%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

Transpose Fine Grained

```
__global__ void transposeFineGrained(float *odata, float *idata, int width,
    int height)
{
    __shared__ float block[TILE_DIM][TILE_DIM+1];
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index = xIndex + (yIndex)*width;
        for (int i=0; i < TILE_DIM; i += BLOCK_ROWS)
       block[threadIdx.y+i][threadIdx.x]=idata[index+i*width];
    __syncthreads();
    for (int i=0; i < TILE_DIM; i += BLOCK_ROWS)
            odata[index+i*height] = block[threadIdx.x][threadIdx.y+i];
}
```

Profiling Results: Transpose FineGrained

Results

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	2097152	136.746 GB/s	
Shared Stores	2107510	137.421 GB/s	
Global Loads	2097152	68.373 GB/s	
Global Stores	2097152	68.373 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	8398966	410.912 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	8388608	68.373 GB/s	
L1 Writes	8388608	68.373 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	16777216	136.746 GB/s	Idle Low Medium High Max

transposeFineGrained(float*, float*, int, int)

Start	84.468 ms (84,4
End	88.394 ms (88,3
Duration	3.926 ms (3,926,
Stream	Default
Grid Size	[256,256,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4.125 KiB
Occupancy	
Achieved	88%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

Transpose Coarse Grained

```
__global__ void transposeCoarseGrained(float *odata, float *idata, int width,
    int height)
{
  __shared__ float block[TILE_DIM][TILE_DIM+1];
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;
  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;
  for (int i=0; i<TILE_DIM; i += BLOCK_ROWS)
        block[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  __syncthreads();
  for (int i=0; i<TILE_DIM; i += BLOCK_ROWS)
    odata[index_out+i*height] = block[threadIdx.y+i][threadIdx.x];
}
```

Profiling Results: Transpose CoarseGrained

Results

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	2097152	132.358 GB/s	
Shared Stores	2105761	132.902 GB/s	
Global Loads	2097152	66.179 GB/s	
Global Stores	2097152	66.179 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	8397217	397.618 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	8388608	66.179 GB/s	
L1 Writes	8388608	66.179 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	16777216	132.358 GB/s	Idle Low Medium High Max

transposeCoarseGrained(float*, float*, int, int)

Start	84.367 ms (84,36
End	88.423 ms (88,42
Duration	4.056 ms (4,056,1
Stream	Default
Grid Size	[256,256,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4.125 KiB
Occupancy	
Achieved	88.3%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

Partition Camping

- Just as shared memory performance can be degraded via bank conflicts, an analogous performance degradation can occur with global memory access through 'partition camping'.
- Global memory is divided into either 6 partitions (on 8- and 9-series GPUs) or 8 partitions (on 200-and 10-series GPUs) of 256-byte width.
- To use global memory effectively, concurrent accesses to global memory by all active warps should be divided evenly amongst partitions.
- partition camping occurs when: global memory accesses are directed through a subset of partitions, causing requests to queue up at some partitions while other partitions go unused.

Partition Camping

- Since partition camping concerns how active thread blocks behave, the issue of how thread blocks are scheduled on multiprocessors is important.
- When a kernel is launched, the order in which blocks are assigned to multiprocessors is determined by the one-dimensional block ID defined as:
  `bid = blockIdx.x + gridDim.x*blockIdx.y;`
  – a row-major ordering of the blocks in the grid.
- Ref: “Optimizing Matrix Transpose in CUDA" - Greg Ruetsch, Paulius Micikevicius
- Ref: “High-Performance Computing with CUDA" - Marc Moreno Maza

Partition Camping

- Once maximum occupancy is reached, additional blocks are assigned to multiprocessors as needed
- How quickly and the order in which blocks complete cannot be determined
- So active blocks are initially contiguous but become less contiguous as execution of the kernel progresses.

Partition Camping

idata

0	1	2	3	4	5
64	65	66	67	68	69
128	129	130	...		

odata

0	64	128			
1	65	129			
2	66	130			
3	67	...			
4	68				
5	69				

- With 8 partitions of 256-byte width, all data in strides of 2048 bytes (or 512 floats) map to the same partition.
- Any float matrix with $512 \times k$ columns, such as our 2048×2048 matrix, will contain columns whose elements map to a single partition.
- With tiles of 32×32 floats whose one-dimensional block IDs are shown in the figures, the mapping of idata and odata onto the partitions is depicted next.

Partition Camping

idata

0	1	2	3	4	5
64	65	66	67	68	69
128	129	130	...		

odata

0	64	128			
1	65	129			
2	66	130			
3	67	...			
4	68				
5	69				

- Concurrent blocks will be accessing tiles row-wise in idata which will be roughly equally distributed amongst partitions
- However these blocks will access tiles column-wise in odata which will typically access global memory through just a few partitions.
- Just as with shared memory, padding would be an option (potentially expensive) but there is a better one ...

Diagonal block reordering

Cartesian Coordinate

0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1
0,2	1,2	2,2	3,2
0,3	1,3	2,3	3,3

Diagonal Coordinate

0,0	0,1	0,2	0,3
1,3	1,0	1,1	1,2
2,2	2,3	2,0	2,1
3,1	3,2	3,3	3,0

blockIdx.x + gridDim.x*blockIdx.y

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	4	8	12
13	1	5	9
10	14	2	6
7	11	15	3

Diagonal block reordering

- The key idea is to view the grid under a diagonal coordinate system. If blockIdx.x and blockIdx.y represent the diagonal coordinates, then (for block-square matrixes) the corresponding cartesian coordinates are given by: `blockIdx_y = blockIdx.x; blockIdx_x = (blockIdx.x+blockIdx.y)% gridDim.x;`
- One would simply include the previous two lines of code at the beginning of the kernel, and write the kernel assuming the cartesian interpretation of blockIdx fields, except using blockIdx_x and blockIdx_y in place of blockIdx.x and blockIdx.y, respectively, throughout the kernel.

Diagonal block reordering

```
__global__ void transposeDiagonal(float *odata,
float *idata, int width, int height)
{
        __shared__ float tile[TILE_DIM][TILE_DIM+1];
        int blockIdx_x, blockIdx_y;
        // diagonal reordering
        if (width == height) {
                blockIdx_y = blockIdx.x;
                blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
        } else {
                int bid = blockIdx.x + gridDim.x*blockIdx.y;
                blockIdx_y = bid%gridDim.y;
                blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
        }
```

Diagonal block reordering

```
int xIndex = blockIdx_x*TILE_DIM + threadIdx.x;
int yIndex = blockIdx_y*TILE_DIM + threadIdx.y;
int index_in = xIndex + (yIndex)*width;
xIndex = blockIdx_y*TILE_DIM + threadIdx.x;
yIndex = blockIdx_x*TILE_DIM + threadIdx.y;
int index_out = xIndex + (yIndex)*height;
for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
       tile[threadIdx.y+i][threadIdx.x] =
       idata[index_in+i*width];
}
__syncthreads();
for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
       odata[index_out+i*height] =
       tile[threadIdx.x][threadIdx.y+i];
}
}
```

Diagonal block reordering

Cartesian

idata

0	1	2	3	4	5
64	65	66	67	68	69
128	129	130	…		

odata

0	64	128			
1	65	129			
2	66	130			
3	67	…			
4	68				
5	69				

Diagonal

0	64	128			
	1	65	129		
		2	66	130	
			3	67	…
				4	68
					5

0					
64	1				
128	65	2			
	129	66	3		
		130	67	4	
			…	68	5

Partition Camping

-
-
-
-