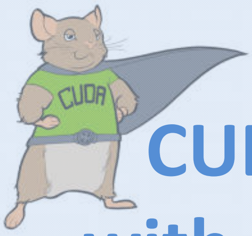


CUDA Optimization with Parallel Transpose

이 정 근 (Jeong-Gun Lee)

한림대학교 소프트웨어융합대학, 임베디드 SoC 연구실

www.onchip.net

Email: Jeonggun.Lee@hallym.ac.kr

Before Start ~

- How to use a **Google Drive** in a **Colab** environment ?

Google Drive를 활용한 CUDA 프로그래밍
직접 Google Drive에서 프로그램을 수행할 수 있습니다.
이경우 mount가 필요하고 여러개의 화일로 이루어진 코드들을 수행할 수 있습니다.
또한 직접 구글 드라이브로 git clone하여 소스를 살펴보고 실 데이터와 함께 수행 시킬 수 있습니다.

```
In [0]: from google.colab import drive
Enter your authorization code:
```
지정된 URL을 클릭하면 인증 코드를 볼수 있으며, 이를 복사하여 입력합니다.

```
In [2]: drive.mount('/content/gdrive')
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3Aietf%3Awg%3Aoauth%3A2.0%3Aoob&scope=email%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdocs.test%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive.photos.readonly%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fpeopleapi.readonly&response_type=code Enter your authorization code: ·········· Mounted at /content/gdrive

In [3]: %cd /content
/content

In [4]: !ls
gdrive sample_data

In [5]: %cd gdrive
/content/gdrive
```

https://github.com/jeonggunlee/CUDATeaching/blob/master/02_cuda_lab/00_googleDrive_CUDAExam.ipynb

CUDA Optimization – Matrix Transpose

- 최적화의 예로 "**Optimizing Parallel Transpose in CUDA**"를 살펴보도록 해요!

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

$$\begin{bmatrix} 6 & 4 & 24 \\ 1 & -9 & 8 \end{bmatrix}^T = \begin{bmatrix} 6 & 1 \\ 4 & -9 \\ 24 & 8 \end{bmatrix}$$

Matrix Transpose 1

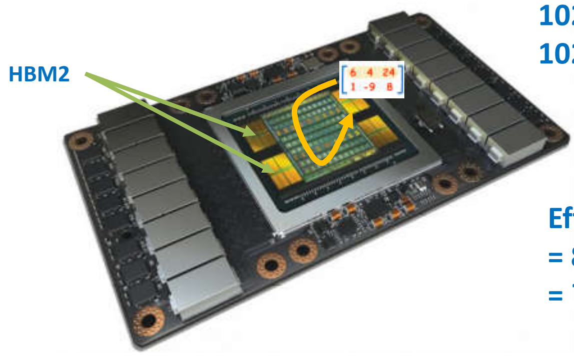
$\begin{bmatrix} 6 & 4 & 24 \\ 1 & -9 & 8 \end{bmatrix}^T = \begin{bmatrix} 6 & 1 \\ 4 & -9 \\ 24 & 8 \end{bmatrix}$

- Consider an **n×n matrix** where **32 divides n**.
- Focus on the GPU device code:
  - the host code performs typical tasks: data allocation and transfer between host and device, the launching and timing of several kernels, result validation, and the deallocation of host and device memory.
- Benchmarks illustrate this section:
  - **Compare** our matrix transpose kernels against a **matrix copy kernel**,
  - for each kernel, we compute the "**effective bandwidth**", calculated in[**GB/s**] as **twice the size of the matrix (once for reading the matrix and once for writing) divided by the time of execution**

Matrix Transpose 2

$\begin{bmatrix}6 & 4 & 24\\1 & -9 & 8\end{bmatrix}^T = \begin{bmatrix}6 & 1\\4 & -9\\24 & 8\end{bmatrix}$

- Benchmarks illustrate this section:
  - for each kernel, we compute the [*effective bandwidth*], calculated in [GB/s] as **twice the size of the matrix (once for reading the matrix and once for writing) divided by the time of execution**

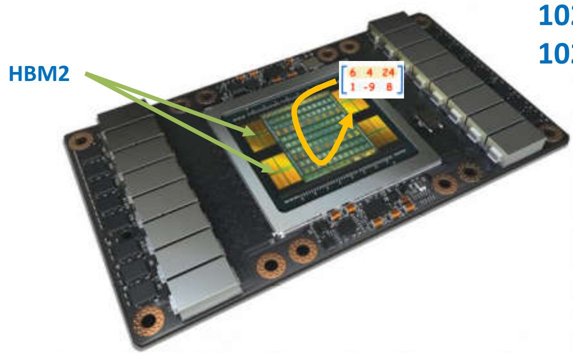


1024x1024 size float matrix
1024x1024 x4B x2 = 8MB

Effective Bandwidth
= 8MB/t_{exe}
= ?

Matrix Transpose 2

$\begin{bmatrix}6 & 4 & 24\\1 & -9 & 8\end{bmatrix}^T = \begin{bmatrix}6 & 1\\4 & -9\\24 & 8\end{bmatrix}$



1024x1024 size float matrix
1024x1024 x4B x2 = 8MB

Effective Bandwidth
= 8MB/t_{exe}
If (t_{exe}=0.02 ms) ?
= 8MB/0.02ms = 800MB/2ms = 400GB/s

Transpose 'C' code: CPU version

$O(n^2)$

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

https://github.com/udacity/cs344/blob/master/Lesson%20Code%20Snippets/Lesson%205%20Code%20Snippets/transpose.cu

Transpose 'C' code: GPU version 1

$O(n^2)$

```
// to be launched on a single thread
// transpose_serial<<<1,1>>>(d_in, d_out);
__global__ void transpose_serial(float in[], float out[])
{
        for(int j=0; j < N; j++)
            for(int i=0; i < N; i++)
                out[j + i*N] = in[i + j*N]; // out(j,i) = in(i,j)
}
```

https://github.com/udacity/cs344/blob/master/Lesson%20Code%20Snippets/Lesson%205%20Code%20Snippets/transpose.cu

Transpose 'C' code: GPU version 2

```
// to be launched on a single thread
// transpose_parallel_per_row<<<1,N>>>(d_in, d_out);
__global__ void
transpose_parallel_per_row(float in[], float out[])
{
        int i = threadIdx.x;

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

O(n)

https://github.com/udacity/cs344/blob/master/Lesson%20Code%20Snippets/Lesson%205%20Code%20Snippets/transpose.cu

Transpose 'C' code: GPU version 3

```
// dim3 blocks(N/K,N/K); // blocks per grid
// dim3 threads(K,K);    // threads per block
// transpose_parallel_per_element<<<blocks,threads>>>(d_in, d_out);
__global__ void
transpose_parallel_per_element(float in[], float out[])
{
        int i = blockIdx.x * K + threadIdx.x;
        int j = blockIdx.y * K + threadIdx.y;

        out[j + i*N] = in[i + j*N]; // out(j,i) = in(i,j)
}
```

O(1)

https://github.com/udacity/cs344/blob/master/Lesson%20Code%20Snippets/Lesson%205%20Code%20Snippets/transpose.cu

Deep Dive: *Performance-Aware Opt.*

Transpose 'C' code: GPU version 3

```
// dim3 blocks(N/K,N/K); // blocks per grid
// dim3 threads(K,K);      // threads per block
// transpose_parallel_per_element<<<blocks,threads>>>(d_in, d_out);
__global__ void
transpose_parallel_per_element(float in[], float out[])
{
        int i = blockIdx.x * K + threadIdx.x;
        int j = blockIdx.y * K + threadIdx.y;

        out[j + i*N] = in[i + j*N]; // out(j,i) = in(i,j)
}
```

O(1)

N = Height or width (square matrix: height == width)
K = size of a thread block in x-dimension or y-dimension

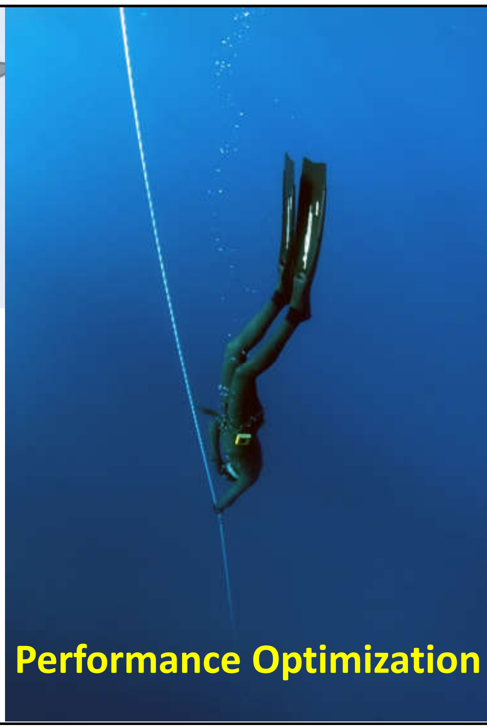
Deep Dive: *Performance-Aware Opt.*

ubuntu@tegra-ubuntu:~/TRANSPOSE/cs344/Lesson Code Snippets/Lesson 5 Code Snippets$ ./**transpose**

transpose_serial: **963.833 ms.**
Verifying transpose...Success
transpose_parallel_per_row: **13.0229 ms.**
Verifying transpose...Success
transpose_parallel_per_element: **11.4738 ms.**
Verifying transpose...Success
transpose_parallel_per_element_tiled 32x32: **9.39575 ms.**
Verifying ...Success
transpose_parallel_per_element_tiled 16x16: **4.44258 ms.**
Verifying ...Success
transpose_parallel_per_element_tiled_padded 16x16: **4.073 ms.**
Verifying...Success

~1/74 @ *N* threads

성능 최적화~



Matrix Transpose - CUDA

$\begin{bmatrix} 6 & 4 & 24 \\ 1 & -9 & 8 \end{bmatrix}^T = \begin{bmatrix} 6 & 1 \\ 4 & -9 \\ 24 & 8 \end{bmatrix}$

256 threads

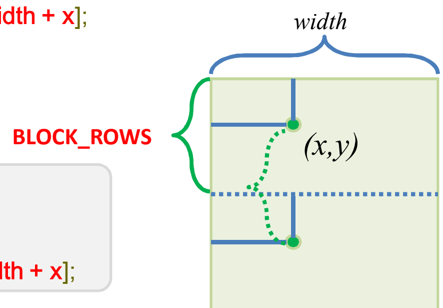
- All kernels launch **thread blocks** of dimension **"32x8"**, where each block transposes (or copies) a tile of dimension 32x32.

launch blocks of 32×8 threads!

- As such, the parameters TILE_DIM and BLOCK_ROWS are set to 32 and 8, respectively.

A simple copy kernel 1 – for Comparison

```
__global__ void copy(float *odata, float* idata)
{
   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) {
        odata[(y+j)*width + x] = idata[(y+j)*width + x];
   }
}
```




```
j = 0
  >> odata[y*width + x] = idata[y*width + x];
j = 1
  >>odata[(y+1)*width + x] = idata[(y+1)*width + x];
```

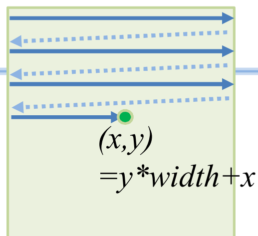
https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/

A simple copy kernel 2 – for Comparison

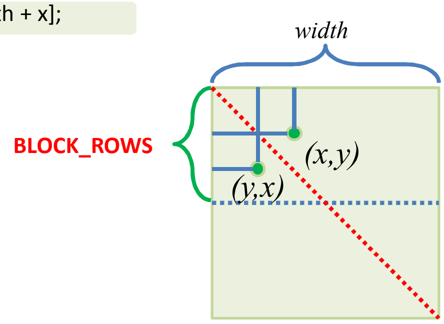
- **odata** and **idata** are pointers to the input and output matrices,
- **width** = gridDim.x*TILE_DIM
- In this kernel, **xIndex** and **yIndex** are global 2D matrix indices and they used to calculate index, the 1D index used to access matrix elements.

```
__global__ void copy(float *odata, float* idata)
{
    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) {
        odata[(y+j)*width + x] = idata[(y+j)*width + x];
    }
}
```

A naive transpose kernel




```
__global__ void transposeNaive(float *odata, float* idata)
{
    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) {
        odata[x*width + (y+j)] = idata[(y+j)*width + x];

    }
}
```



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

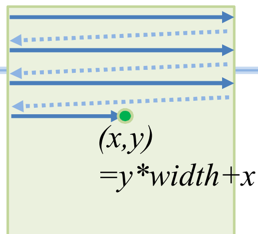
Naive transpose kernel vs copy kernel

- The performance of these two kernels on a 1024x1024 matrix using a Tesla GPUs is given in the following table:

	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
transposeNaive	18.8	55.3

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

Problem ?



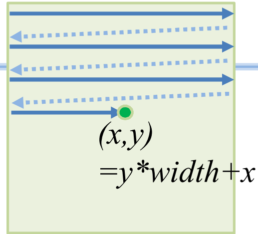

```
__global__ void transposeNaive(float *odata, float* idata)
{
    int x = blockIdx.x*TILE_DIM + threadIdx.x;
    int y = blockIdx.y*TILE_DIM + threadIdx.y;
    int width = gridDim.x*TILE_DIM;

    for (int j=0; j<TILE_DIM; j+=BLOCK_ROWS) {
        odata[x*width + (y+j)] = idata[(y+j)*width + x];

    }
}
```

(xindex, yindex): (0, 0) → (1, 0) → (2, 0) → (3, 0) → ⋯ (31, 0)

→ (0, 1) → (1, 1) → (2, 1) → (3, 1) → ⋯ (31, 1)

→ ⋯

Problem ?




```
__global__ void transposeNaive(float *odata, float* idata)
{
    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) {
        odata[x*width + (y+j)] = idata[(y+j)*width + x];

    }
}
```

(xindex, yindex): (0, 0) → (1, 0) → (2, 0) → (3, 0) → ⋯ (31, 0)

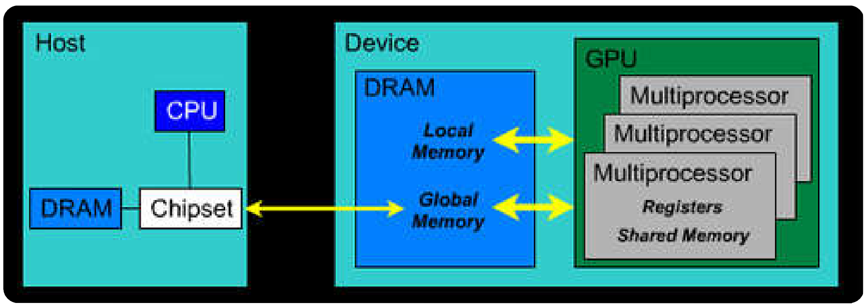
→ (0, 1) → (1, 1) → (2, 1) → (3, 1) → ⋯ (31, 1)

→ ⋯

One Transaction .vs. 32 Transactions !

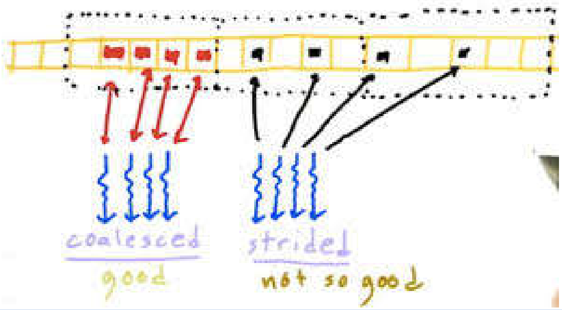
Coalesced Transpose 1

- Because **device memory [GPU DDR Memory]** has a much higher latency and lower bandwidth than **on-chip memory [shared memory]**, special attention must be paid to: **how global memory accesses are performed**?



Coalesced Transpose 2

- The ***simultaneous global memory accesses*** by each thread of a during the execution of a single read or write instruction will be ***coalesced*** into a single access if:
  - The size of the memory element accessed by each thread is either 4, 8, or 16 bytes.
  - The elements form a ***contiguous block of memory***.
  - The i-th element is accessed by the i-th thread in the warp.

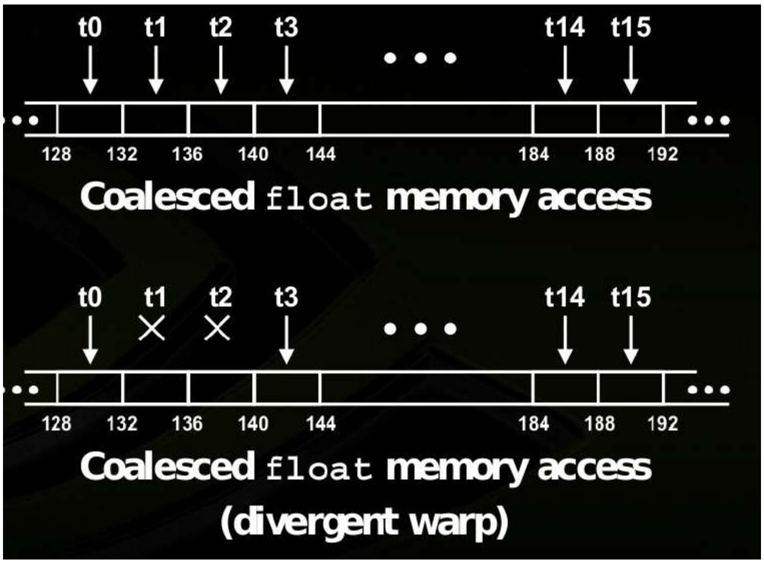


Coalesced Transpose 3

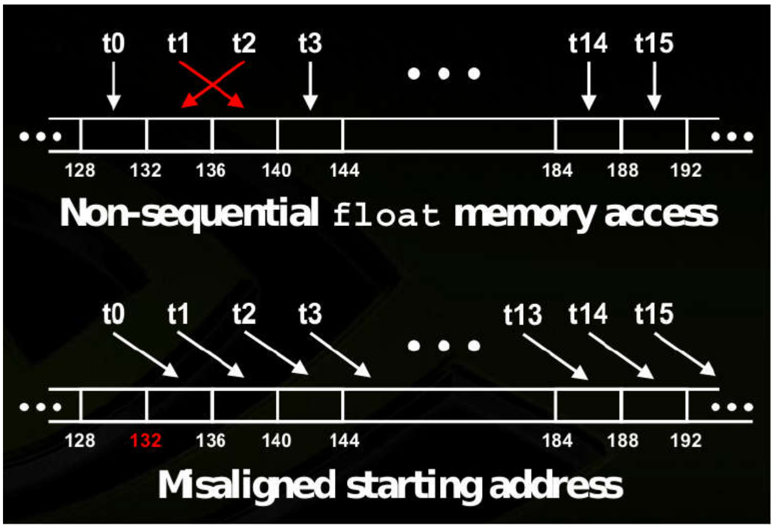
- The simultaneous global memory accesses by each thread of a during the execution of a single read or write instruction will be ***coalesced*** into a single access if:
  - The size of the memory element accessed by each thread is either 4, 8, or 16 bytes.
  - The elements form a *contiguous block of memory*.
  - The i-th element is accessed by the i-th thread in the warp.

- ***Last two requirements can be relaxed*** (compiler optimization) with compute capabilities of 1.2.
- **Coalescing happens** even if some threads do not access memory (**divergent warp**)

Coalesced Transpose 4



Coalesced Transpose 5



Coalesced Transpose 6

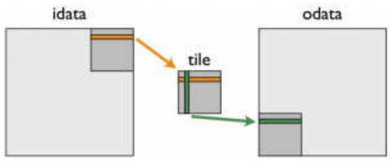
- **Basically,** "**all loads from idata are coalesced**".

- Coalescing behavior differs between the simple copy and naïve transpose kernels when **writing to odata**.
  - Simple copy – coalesced
  - Naïve Transpose – non-coalesced

Coalesced Transpose 7

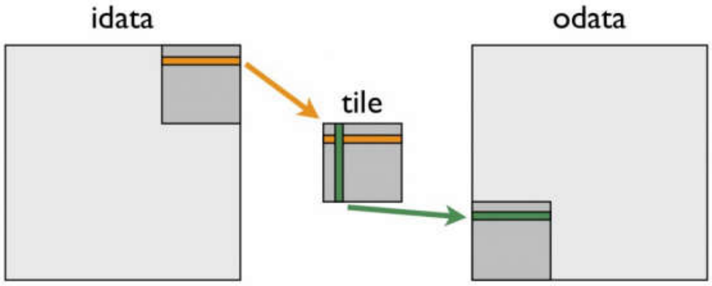
- The way to avoid **uncoalesced** global memory access is
  - to read the data into **shared memory** and,
  - have each warp access noncontiguous locations in shared memory in order to write contiguous data to odata.
- There is no performance penalty for noncontiguous access patterns in shared memory as there is in global memory.
- a **__synchthreads()** call is required to ensure that all reads from idata to shared memory have completed before writes from shared memory to odata.

Coalesced Transpose 8




```
__global__ void transposeCoalesced(float *odata, const float *idata)
{
        __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();

        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];
}
```

Coalesced Transpose 9



- a warp of threads reads contiguous data from idata into rows of the shared memory tile.
- After recalculating the array indices, **a column of the shared memory tile is written to contiguous addresses in odata**.
- Because threads write different data to odata than they read from idata, we must use a block-wise barrier synchronization **__syncthreads()**.

Coalesced Transpose 10

Effective Bandwidth (GB/s, ECC enabled)		
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
copySharedMem	104.6	152.3
transposeNaive	18.8	55.3
transposeCoalesced	**51.3**	**97.6**

- There is a dramatic increase in effective bandwidth of the coalesced transpose over the naive transpose, **but there still remains a large performance gap** between the coalesced transpose and the copy:
  - One possible cause of this performance gap could be the **synchronization barrier** required in the coalesced transpose.
  - This can be easily assessed using the following copy kernel which utilizes shared memory and contains a __syncthreads() call.

Coalesced Transpose 11

```
__global__ void copySharedMem(float *odata, const float *idata)
{
  __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];
}
```

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

Coalesced Transpose 12

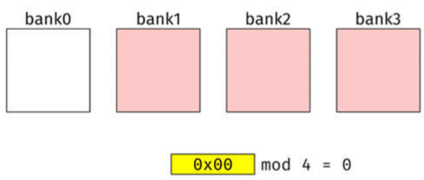
Effective Bandwidth (GB/s, ECC enabled)		
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
copySharedMem	104.6	152.3
transposeNaive	18.8	55.3
transposeCoalesced	**51.3**	**97.6**

- The shared memory copy results seem to suggest that **the use of shared memory with a synchronization barrier has little effect on the performance**.

Shared memory bank conflicts 1

- Shared memory is divided into 32 equally-sized memory modules, called **banks**, which are organized such that successive 32-bit words are assigned to successive banks.

Write



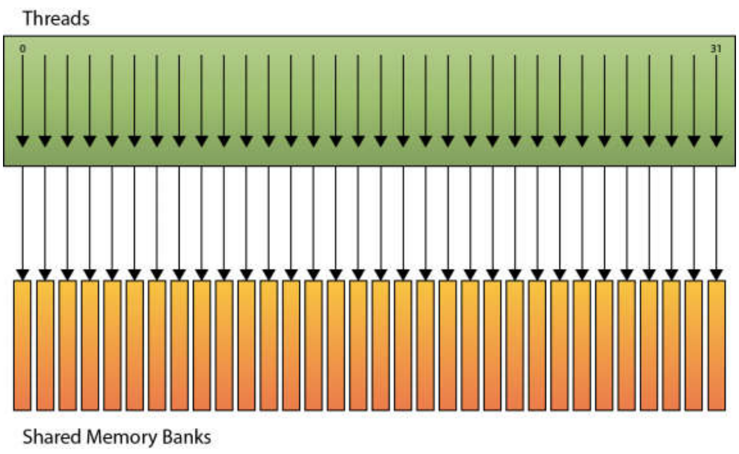
- These banks can be accessed simultaneously, and to achieve maximum bandwidth to and from shared memory, the [**threads in a warp should access shared memory associated with different banks**].

Shared memory bank conflicts 2

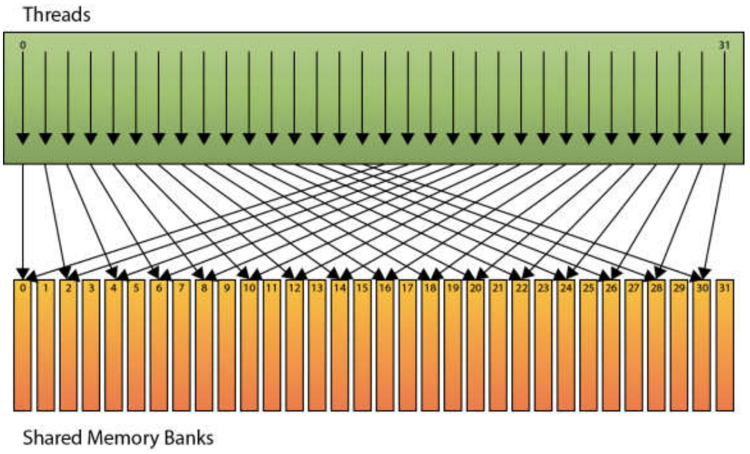
- These banks can be accessed simultaneously, and to achieve maximum bandwidth to / from shared memory the **threads in a warp should access a shared memory associated with different banks**.

- The **exception to this rule** is when **all threads in a warp read the same shared memory address**, which results in a **broadcast** where the data at that address is sent to all threads of the half warp in one transaction.

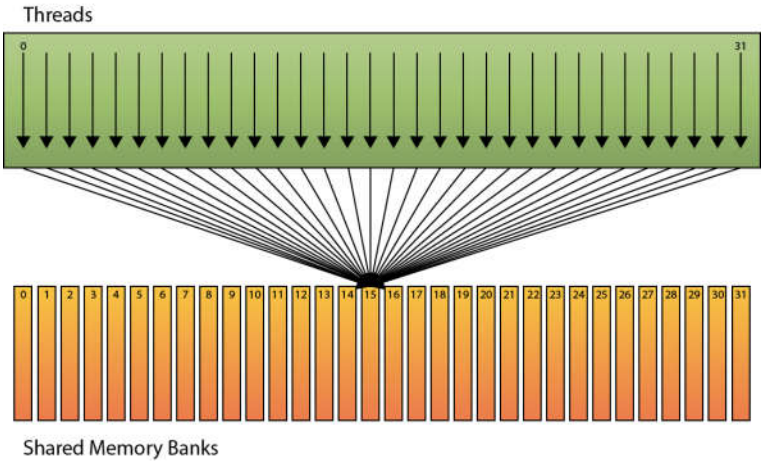
Shared memory bank conflicts 3



Shared memory bank conflicts 4



Shared memory bank conflicts 5



Shared memory bank conflicts 6

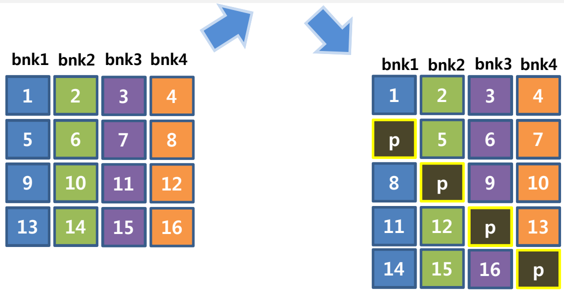
- The coalesced transpose uses a 32x32 shared memory array of floats.
- For a shared memory tile of 32 × 32 elements, **all elements in a column of data map to the same shared memory bank**
  - Resulting in a worst-case scenario for memory bank conflicts: reading a column of data results in a **32-way bank conflict**.
- A simple way to avoid this conflict is to **pad the shared memory array by one column**:

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```

Shared memory bank conflicts 7

- A simple way to avoid this conflict is to **pad the shared memory array by one column**:

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```



tile[tid % 4]:

→ 1, 5, 9 , 13 threads access a bank1

Shared memory bank conflicts 8

Effective Bandwidth (GB/s, ECC enabled)		
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
copySharedMem	104.6	152.3
transposeNaive	18.8	55.3
transposeCoalesced	51.3	97.6
transposeNoBankConflicts	**99.5**	**144.3**

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

Granularity of Parallelism

- **Size of a Tile ?**
  - We do test with a block of 32x8 threads with config. of "(32,8)"
  - What about **32x32** ?
    - "1024 threads wait at a barrier"
    - High Parallelism (?) but high synchronization overhead
  - What about **16x16** ?
    - "256 threads wait at a barrier"
    - Lower Parallelism (?) but lower synchronization overhead

Minimize timing waiting at a barrier !

https://github.com/jeonggunlee/cs344/blob/master/Lesson%20Code%20Snippets/Lesson%205%20Code%20Snippets/transpose.cu

Conclusion - Transpose

- Understand CUDA performance characteristics
  - **Memory coalescing**
  - **Bank conflicts**
  - **Granularity of parallelism**
- Use peak performance metrics to guide optimization