Effective Use of the Memory Subsystem

Now that you can write correct CUDA kernels, and understand the importance of launching grids that give the GPU sufficient opportunity to hide latency, you are going to learn techniques to effectively utilize GPU memory subsystems. These techniques are widely applicable to a variety of CUDA applications, and some of the most important when it comes time to make your CUDA code go fast.

You are going to begin by learning about memory coalescing. To challenge your ability to reason about memory coalescing, and to expose important details relevent to many CUDA applications, you will then learn about 2-dimensional grids and thread blocks. Next you will learn about a very fast, user-controlled, on-demand memory space called shared memory, and will use shared memory to facilitate memory coalescing where it would not have otherwise been possible. Finally, you will learn about shared memory bank conflicts, which can spoil the performance possibilities of using shared memory, and a technique to address them.

Objectives

By the time you complete this section, you will be able to:

- Write CUDA kernels that benefit from coalesced memory access patterns.
- Work with multi-dimensional grids and thread blocks.
- Use shared memory to coordinate threads within a block.
- Use shared memory to facilitate coalesced memory access patterns.
- Resolve shared memory bank conflicts.

The Problem: Uncoalesced Memory Access Hurts Performance

Before you learn the details about what **coalesced memory access** is, run the following cells to observe the performance implications for a seemingly trivial change to the data access pattern within a kernel.

Imports

```
import numpy as np
from numba import cuda
```

Data Creation

In this cell we define `n` and create a grid with threads equal to `n`. We also create an output vector with length `n`. For the inputs we create vectors of size `stride * n` for reasons that will be made clear below:

```
n = 1024*1024 # 1M

threads_per_block = 1024
blocks = int(n / threads_per_block)

stride = 16

# Input Vectors of length stride * n
a = np.ones(stride * n).astype(np.float32)
b = a.copy().astype(np.float32)

# Output Vector
out = np.zeros(n).astype(np.float32)

d_a = cuda.to_device(a)
d_b = cuda.to_device(b)
d_out = cuda.to_device(out)
```

Kernel Definition

In `add_experiment`, every thread in the grid will add an item in `a`, and an item in `b` and write the result to `out`. The kernel has been written such that we can pass a `coalesced` value of either `True` or `False` to affect how it indexes into the `a` and `b` vectors. You will see the performance comparison of the two modes below.

```
@cuda.jit
def add_experiment(a, b, out, stride, coalesced):
    i = cuda.grid(1)
    # The above line is equivalent to
    # i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if coalesced == True:
        out[i] = a[i] + b[i]
    else:
        out[i] = a[stride*i] + b[stride*i]
```

Launch Kernel Using Coalesced Access

Here we pass `True` as the `coalesced` value, and observe the performance of the kernel over several runs:

```
%timeit add_experiment[blocks, threads_per_block](d_a, d_b, d_out,
stride, True); cuda.synchronize
```

Here we make sure the kernel ran as expected:

```
result = d_out.copy_to_host()
truth = a[:n] + b[:n]

np.array_equal(result, truth)
```

Launch Kernel Using Uncoalesced Access

In this cell we pass `False`, to observe the perfomance of the uncoalesced data access pattern for `add_experiment`:

```
%timeit add_experiment[blocks, threads_per_block](d_a, d_b, d_out,
stride, False); cuda.synchronize
```

Here we make sure the kernel ran as expected:

```
result = d_out.copy_to_host()
truth = a[::stride] + b[::stride]

np.array_equal(result, truth)
```

Results

The performance of the uncoalesced data access pattern was far worse. Now you will learn why, and how to think about data access patterns in your kernels to obtain high performing kernels.

Presentation: Global Memory Coalescing

Execute the following cell to load the slides, then click on "Start Slide Show" to make them full screen.

```
from IPython.display import IFrame
IFrame('https://view.officeapps.live.com/op/view.aspx?src=https://
developer.download.nvidia.com/training/courses/C-AC-02-V1/coalescing-
v3.pptx', 800, 450)
```

> ***Footnote**: for additional details about global memory segment size across a variety of devices, and with regards to caching, see The CUDA Best Practices Guide.*

Exercise: Column and Row Sums

For this exercise you will be asked to write a column sums kernel that uses fully coalesced memory access patterns. To begin you will observe the performance of a row sums kernel that makes uncoalesced memory accesses.

Row Sums

Imports

```
import numpy as np
from numba import cuda
```

Data Creation

In this cell we create an input matrix, as well as a vector for storing the solution, and transfer each of them to the device. We also define the grid and block dimensions to be used when we launch the kernel below. We set an arbitrary row of data to some arbitrary value to facilitate checking for correctness below.

```
n = 16384 # matrix side size
threads_per_block = 256
blocks = int(n / threads_per_block)

# Input Matrix
a = np.ones(n*n).reshape(n, n).astype(np.float32)
# Here we set an arbitrary row to an arbitrary value to facilitate a
check for correctness below.
a[3] = 9

# Output vector
sums = np.zeros(n).astype(np.float32)

d_a = cuda.to_device(a)
d_sums = cuda.to_device(sums)
```

** Kernel Definition**

`row_sums` will use each thread to iterate over a row of data, summing it, and then store its row sum in `sums`.

```
@cuda.jit
def row_sums(a, sums, n):
    idx = cuda.grid(1)
    sum = 0.0

    for i in range(n):
        # Each thread will sum a row of `a`
        sum += a[idx][i]

    sums[idx] = sum
```

Row Sums Performance

```
%timeit row_sums[blocks, threads_per_block](d_a, d_sums, n);
cuda.synchronize()
```

Check for Correctness

```
result = d_sums.copy_to_host()
truth = a.sum(axis=1)

np.array_equal(truth, result)
```

Column Sums

Imports

```
import numpy as np
from numba import cuda
```

Data Creation

In this cell we create an input matrix, as well as a vector for storing the solution, and transfer each of them to the device. We also define the grid and block dimensions to be used when we launch the kernel below. We set an arbitrary column of data to some arbitrary value to facilitate checking for correctness below.

```
n = 16384 # matrix side size
threads_per_block = 256
blocks = int(n / threads_per_block)

a = np.ones(n*n).reshape(n, n).astype(np.float32)
# Here we set an arbitrary column to an arbitrary value to facilitate
a check for correctness below.
a[:, 3] = 9
sums = np.zeros(n).astype(np.float32)

d_a = cuda.to_device(a)
d_sums = cuda.to_device(sums)
```

** Kernel Definition**

`col_sums` will use each thread to iterate over a column of data, summing it, and then store its column sum in `sums`. Complete the kernel definition to accomplish this. If you get stuck, feel free to refer to the solution.

```
@cuda.jit
def col_sums(a, sums, ds):
    # TODO: Write this kernel to store the sum of each column in
matrix `a` to the `sums` vector.
    pass
```

Check Performance

Assuming you have written `col_sums` to use coalesced access patterns, you should see a significant (almost 2x) speed up compared to the uncoalesced `row_sums` you ran above:

```
%timeit col_sums[blocks, threads_per_block](d_a, d_sums, n);
cuda.synchronize()
```

Check Correctness

Confirm your kernel is working as expected.

```
result = d_sums.copy_to_host()
truth = a.sum(axis=0)

np.array_equal(truth, result)
```

2 and 3 Dimensional Blocks and Grids

Both grids and blocks can be configured to contain a 2 or 3 dimensional collection of blocks or threads, respectively. This is done mostly as a matter of convenience for programmers who often work with 2 or 3 dimensional datasets. Here is a very trivial example to highlight the syntax. You may need to read *both* the kernel definition and its launch before the concept makes sense.

```
import numpy as np
from numba import cuda

A = np.zeros((4,4)) # A 4x4 Matrix of 0's
d_A = cuda.to_device(A)

# Here we create a 2D grid with 4 blocks in a 2x2 structure, each with
4 threads in a 2x2 structure
# by using a Python tuple to signify grid and block dimensions.
blocks = (2, 2)
threads_per_block = (2, 2)
```

This kernel will take an input matrix of 0s and write to each of its elements, its (x,y) coordinates within the grid in the format of `X.Y`:

```
@cuda.jit
def get_2D_indices(A):
    # By passing `2`, we get the thread's unique x and y coordinates
in the 2D grid
    x, y = cuda.grid(2)
    # The above is equivalent to the following 2 lines of code:
    # x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    # y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    # Write the x index followed by a decimal and the y index.
    A[x][y] = x + y / 10

get_2D_indices[blocks, threads_per_block](d_A)

result = d_A.copy_to_host()
result
```

Exercise: Coalesced 2-Dimensional Matrix Add

Imports

```
import numpy as np
from numba import cuda
```

Data Creation

In this cell we define 2048x2048 elmement input matrices `a` and `b`, as well as a 2048x2048 0-initialized output matrix. We copy these matrices to the device.

We also define the 2-dimensional block and grid dimensions to be used below. Note that we are creating a grid with the same number of total threads as there are input and output elements, such that each thread in the grid will calculate the sum for a single element in the output matrix.

```
n = 2048*2048 # 4M

# 2D blocks
threads_per_block = (32, 32)
# 2D grid
blocks = (64, 64)

# 2048x2048 input matrices
a = np.arange(n).reshape(2048,2048).astype(np.float32)
b = a.copy().astype(np.float32)

# 2048x2048 0-initialized output matrix
out = np.zeros_like(a).astype(np.float32)

d_a = cuda.to_device(a)
d_b = cuda.to_device(b)
d_out = cuda.to_device(out)
```

2D Matrix Add

Your job is to complete the TODOs in `matrix_add` to correctly sum `a` and `b` into `out`. As a challenge to your understanding of coalesced access patterns, `matrix_add` will accept a `coalesced` boolean indicating whether the access patterns should be coalesced or not. Both modes (coalesced and uncoalesced) should produce correct results, however, you should observe significant speedups below when running with `coalesced` set to `True`.

If you get stuck, feel free to check out the solution.

```
@cuda.jit
def matrix_add(a, b, out, coalesced):
    # TODO: set x and y to index correctly such that each thread
    # accesses one element in the data.
    x, y = pass
```

```
    if coalesced == True:
        # TODO: write the sum of one element in `a` and `b` to `out`
        # using a coalesced memory access pattern.
    else:
        # TODO: write the sum of one element in `a` and `b` to `out`
        # using an uncoalesced memory access pattern.
```

Check Performance

Run both cells below to launch `matrix_add` with both the coalesced and uncoalesced access patterns you wrote into it, and observe the performance difference. Additional cells have been provided to confirm the correctness of your kernel.

Coalesced

```
%timeit matrix_add[blocks, threads_per_block](d_a, d_b, d_out, True);
cuda.synchronize

result = d_out.copy_to_host()
truth = a+b

np.array_equal(result, truth)
```

Uncoalesced

```
%timeit matrix_add[blocks, threads_per_block](d_a, d_b, d_out, False);
cuda.synchronize

result = d_out.copy_to_host()
truth = a+b

np.array_equal(result, truth)
```

Shared Memory

So far we have been differentiating between host and device memory, as if device memory were a single kind of memory. But in fact, CUDA has an even more fine-grained memory hierarchy. The device memory we have been utilizing thus far is called **global memory** which is available to any thread or block on the device, can persist for the lifetime of the application, and is a relatively large memory space.

We will now discuss how to utilize a region of on-chip device memory called **shared memory**. Shared memory is a programmer defined cache of limited size that depends on the GPU being used and is **shared** between all threads in a block. It is a scarce resource, cannot be accessed by threads outside of the block where it was allocated, and does not persist after a kernel finishes executing. Shared memory however has a much higher bandwidth than global memory and can be used to great effect in many kernels, especially to optimize performance.

Here are a few common use cases for shared memory:

- Caching memory read from global memory that will need to be read multiple times within a block.
- Buffering output from threads so it can be coalesced before writing it back to global memory.
- Staging data for scatter/gather operations within a block.

Shared Memory Syntax

Numba provides functions for allocating shared memory as well as for synchronizing between threads in a block, which is often necessary after parallel threads read from or write to shared memory.

When declaring shared memory, you provide the shape of the shared array, as well as its type, using a Numba type. **The shape of the array must be a constant value**, and therefore, you cannot use arguments passed into the function, or, provided variables like `numba.cuda.blockDim.x`, or the calculated values of `cuda.griddim`. Here is a convoluted example to demonstrate the syntax with comments pointing out the movement from host memory to global device memory, to shared memory, back to global device memory, and finally back to host memory:

Imports

We will use `numba.types` to define the types of values in shared memory.

```
import numpy as np
from numba import types, cuda
```

Swap Elements Using Shared Memory

The following kernel takes an input vector, where each thread will first write one element of the vector to shared memory, and then, after syncing such that all elements have been written to shared memory, will write one element out of shared memory into the swapped output vector.

Worth noting is that each thread will be writing a swapped value from shared memory that was written into shared memory by another thread.

```
@cuda.jit
def swap_with_shared(vector, swapped):
    # Allocate a 4 element vector containing int32 values in shared
memory.
    temp = cuda.shared.array(4, dtype=types.int32)

    idx = cuda.grid(1)

    # Move an element from global memory into shared memory
    temp[idx] = vector[idx]

    # cuda.syncthreads will force all threads in the block to
synchronize here, which is necessary because...
    cuda.syncthreads()
```

```
    #...the following operation is reading an element written to
shared memory by another thread.

    # Move an element from shared memory back into global memory
    swapped[idx] = temp[3 - cuda.threadIdx.x] # swap elements
```

Data Creation

```
vector = np.arange(4).astype(np.int32)
swapped = np.zeros_like(vector)

# Move host memory to device (global) memory
d_vector = cuda.to_device(vector)
d_swapped = cuda.to_device(swapped)

vector
```

** Run Kernel**

```
swap_with_shared[1, 4](d_vector, d_swapped)
```

Check Results

```
# Move device (global) memory back to the host
result = d_swapped.copy_to_host()
result
```

Presentation: Shared Memory for Memory Coalescing

Execute the following cell to load the slides, then click on "Start Slide Show" to make them full screen.

```
from IPython.display import IFrame
IFrame('https://view.officeapps.live.com/op/view.aspx?src=https://
developer.download.nvidia.com/training/courses/C-AC-02-V1/
shared_coalescing.pptx', 800, 450)
```

Excercise: Used Shared Memory for Coalesced Reads and Writes With Matrix Transpose

In this exercise you will implement what was just demonstrated in the presentation by writing a matrix transpose kernel which, using shared memory, makes coalesced reads and writes to the output matrix in global memory.

Coalesced Reads, Uncoalesced Writes

As reference, and for performance comparison, here is a naive matrix transpose kernel that makes coalesced reads from input, but uncoalesced writes to output.

Imports

```
from numba import cuda
import numpy as np
```

Data Creation

Here we create a 4096x4096 input matrix `a` as well as a 4096x4096 output matrix `transposed`, and copy them to the device.

We also define a 2-dimensional grid with 2-dimensional blocks to be used below. Note that we have created a grid with a total number of threads equal to the number of elments in the input matrix.

```
n = 4096*4096 # 16M

# 2D blocks
threads_per_block = (32, 32)
#2D grid
blocks = (128, 128)

# 4096x4096 input and output matrices
a = np.arange(n).reshape((4096,4096)).astype(np.float32)
transposed = np.zeros_like(a).astype(np.float32)

d_a = cuda.to_device(a)
d_transposed = cuda.to_device(transposed)
```

Naive Matrix Transpose Kernel

This kernel correctly transposes `a`, writing the transposition to `transposed`. It makes reads from `a` in a coalesced fashion, however, its writes to `transposed` are uncoalesced.

```
@cuda.jit
def transpose(a, transposed):
    x, y = cuda.grid(2)

    transposed[x][y] = a[y][x]
```

Check Performance

```
%timeit transpose[blocks, threads_per_block](d_a, d_transposed);
cuda.synchronize()
```

Check Correctness

```
result = d_transposed.copy_to_host()
expected = a.T

np.array_equal(result, expected)
```

Refactor for Coalesced Reads and Writes

Your job will be to refactor the `transpose` kernel to use shared memory and make both reads to and writes from global memory in a coalesced fashion.

Imports

```
import numpy as np
from numba import cuda, types as numba_types
```

Data Creation

```
n = 4096*4096 # 16M

# 2D blocks
threads_per_block = (32, 32)
#2D grid
blocks = (128, 128)

# 4096x4096 input and output matrices
a = np.arange(n).reshape((4096,4096)).astype(np.float32)
transposed = np.zeros_like(a).astype(np.float32)

d_a = cuda.to_device(a)
d_transposed = cuda.to_device(transposed)
```

Write a Transpose Kernel that Uses Shared Memory

Complete the TODOs inside the `tile_transpose` kernel definition.

If you get stuck, feel free to check out the solution.

```
@cuda.jit
def tile_transpose(a, transposed):
    # `tile_transpose` assumes it is launched with a 32x32 block
dimension,
    # and that `a` is a multiple of these dimensions.

    # 1) Create 32x32 shared memory array.

    # TODO: Your code here.

    # Compute offsets into global input array. Recall for coalesced
access we want to map threadIdx.x increments to
    # the fastest changing index in the data, i.e. the column in our
```

```
array.
    # Note: `a_col` and `a_row` are already correct.
    a_col = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    a_row = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    # 2) Make coalesced read from global memory (using grid indices)
    # into shared memory array (using thread indices).

    # TODO: Your code here.

    # 3) Wait for all threads in the block to finish updating shared
memory.

    # TODO: Your code here.

    # 4) Calculate transposed location for the shared memory array
tile
    # to be written back to global memory. Note that
blockIdx.y*blockDim.y
    # and blockIdx.x* blockDim.x are swapped (because we want to write
to the
    # transpose locations), but we want to keep access coalesced, so
match up the
    # threadIdx.x to the fastest changing index, i.e. the column./
    # Note: `t_col` and `t_row` are already correct.
    t_col = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.x
    t_row = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.y

    # 5) Write from shared memory (using thread indices)
    # back to global memory (using grid indices)
    # transposing each element within the shared memory array.

    # TODO: Your code here.
```

Check Performance

Check the performance of your refactored transpose kernel. You should see a speedup compared to the baseline transpose performance above.

```
%timeit tile_transpose[blocks, threads_per_block](d_a, d_transposed);
cuda.synchronize()
```

Check Correctness

```
result = d_transposed.copy_to_host()
expected = a.T

np.array_equal(result, expected)
```

Why Such a Small Improvement?

While this is a significant speedup for only a few lines of code, but you might think that the performance improvement is not as stark as you expected based on earlier performance improvements to use coalesced access patterns. There are 2 main reasons for this:

1. The naive transpose kernel was making coalesced reads, so, your refactored version only optimized half of the global memory access throughout the execution of the kernel.
2. Your code as written suffers from something called shared memory bank conflicts, a topic to which we will now turn our attention.

Presentation: Memory Bank Conflicts

Execute the following cell to load the slides, then click on "Start Slide Show" to make them full screen.

```
from IPython.display import IFrame
IFrame('https://view.officeapps.live.com/op/view.aspx?src=https://
developer.download.nvidia.com/training/courses/C-AC-02-V1/
bank_conflicts.pptx', 800, 450)
```

Assessment: Resolve Memory Bank Conflicts

As a final exercise, and to get credit towards a certificate in the course for this final section of the workshop, you will refactor the transpose kernel utilizing shared memory to be shared memory bank conflict free.

Imports

```
import numpy as np
from numba import cuda, types as numba_types
```

Data Creation

```
n = 4096*4096 # 16M
threads_per_block = (32, 32)
blocks = (128, 128)

a = np.arange(n).reshape((4096,4096)).astype(np.float32)
transposed = np.zeros_like(a).astype(np.float32)

d_a = cuda.to_device(a)
d_transposed = cuda.to_device(transposed)
```

Make the Kernel Bank Conflict Free

The `tile_transpose_conflict_free` kernel is a working matrix transpose kernel which utilizes shared memory so that both reads from and writes to global memory are coalesced. Your job is to refactor the kernel so that it does not suffer from memory bank conflicts.

Note: Because this final exercise counts towards certification in the course, a solution will not be provided.

```
@cuda.jit
def tile_transpose_conflict_free(a, transposed):
    # `tile_transpose` assumes it is launched with a 32x32 block
dimension,
    # and that `a` is a multiple of these dimensions.

    # 1) Create 32x32 shared memory array.
    tile = cuda.shared.array((32, 32), numba_types.float32)

    # Compute offsets into global input array.
    x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    # 2) Make coalesced read from global memory into shared memory
array.
    # Note the use of local thread indices for the shared memory
write,
    # and global offsets for global memory read.
    tile[cuda.threadIdx.y, cuda.threadIdx.x] = a[y, x]

    # 3) Wait for all threads in the block to finish updating shared
memory.
    cuda.syncthreads()

    # 4) Calculate transposed location for the shared memory array
tile
    # to be written back to global memory.
    t_x = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.x
    t_y = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.y

    # 5) Write back to global memory,
    # transposing each element within the shared memory array.
    transposed[t_y, t_x] = tile[cuda.threadIdx.x, cuda.threadIdx.y]
```

Check Performance

Assuming you have correctly resolved the bank conflicts, this kernel should run significantly faster than both the naive transpose kernel, and, the shared memory (with bank conflicts) transpose kernel. In order to pass the assessment, your kernel will need to run on average in less than 840 µs.

The first value printed by running the following cell will give you the average run time of your kernel.

```
%timeit tile_transpose_conflict_free[blocks, threads_per_block](d_a,
d_transposed); cuda.synchronize()
```

Check Correctness

In order to pass the assessment, your kernel also needs to work correctly. Run the following 2 cells to confirm this is true.

```
result = d_transposed.copy_to_host()
expected = a.T

np.array_equal(result, expected)
```

Run the Assessment

If you have completed the refactor, observed it's run time to be less than 840 µs, and confirmed that it runs correctly, execute the following cells to run the assessment against your kernel definition.

```
from assessment import assess

assess(tile_transpose_conflict_free)
```

Get Credit for Your Work

After successfully passing the assessment above, revisit the webpage where you launched this interactive environment and click on the **"ASSESS TASK"** button as shown in the screenshot below. Doing so will give you credit for this part of the workshop that counts towards earning a **certificate of competency** for the entire course.

Run the assessment

Summary

Now that you have completed this session you are able to:

- Write CUDA kernels that benefit from coalesced memory access patterns.
- Work with multi-dimensional grids and thread blocks.
- Use shared memory to coordinate threads within a block.
- Use shared memory to facilitate coalesced memory access patterns.
- Resolve shared memory bank conflicts.

Download Content

To download the contents of this notebook, execute the following cell and then click the download link below. Note: If you run this notebook on a local Jupyter server, you can expect some of the file path links in the notebook to be broken as they are shaped to our own platform. You can still navigate to the files through the Jupyter file navigator.

```
!tar -zcvf section3.tar.gz .
```

Download files from this section.