In the lecture, you learned about GPUs. In this exercise, we will study practical considerations when programming GPUs for specific operations.

- **Up Front**: GPU programming is often focused on optimizing and hiding memory accesses. Most kernels (with notable exceptions like matrix multiplication) are primarily memory-bandwidth bound. This means you often cannot fully 'hide' memory latency with computation because the compute portion takes relatively few cycles compared to the memory access time.

Exercise 1: Memory Coalescing and Caching

Recap:
A GPU consists of global High Bandwidth Memory (HBM, also called Global Memory), L2 cache, and L1 cache/shared memory. For this exercise, it's important to understand the following: When threads within a warp issue memory instructions (e.g., each thread issues one load instruction targeting HBM), the hardware coalesces these requests. It attempts to satisfy them using the minimum number of 32-byte transactions from global memory. A key constraint is that these hardware transactions must be aligned to 32-byte boundaries.

Also note: the GPU's L1 cache also operates with a 32-byte line size and is significantly faster (orders of magnitude) than global memory.

Exercise 1.1:

We will study a very simple kernel that loads an array A from global memory, computes $B = A^2$ element-wise, and writes the result B back to global memory. Notice that this kernel's performance will be limited primarily by global memory throughput, not by the computational (FLOPs) intensity.

1. Navigate to the `exercise_1` directory and examine the file: `exercise1_coalesced.cu`. Read the code and comments carefully; they explain the process. REMINDER: Threads within a block are grouped into warps. REMINDER: Threads of a block are chunked into warps by simply sorting and chunking into bins of 32 elements!
2. Compile this with: `nvcc -arch=sm_90a -O3 -o exercise1_coalesced exercise1_coalesced.cu`
3. Run the compiled executable and record its runtime.
4. Now profile it using Nsight Compute. For this run: `ncu --set full ./exercise1_coalesced` exercise1_coalesced.
5. In the nsight dump you should find a section called: `Memory Workload Analysis` under which you will find the % of achieved HBM throughput: `Max Bandwidth`. Make sure that this number is around 70-80% or above before you continue to the next exercise.

6. If this was all successfull then make a file called exercise_1.1_solution.txt and copy the ncu output into there.

Exercise 1.2:

Now that you've seen a coalesced kernel, we will examine what happens when memory accesses are *not* coalesced.

1. Head to the file called `exercise1_uncoalesced`. This kernel performs the same computation as before but accesses memory in a non-coalesced pattern. Try to understand why this access pattern prevents coalescing before reading the explanation below.

   Lets look at what happens for warp0 and its threads the first time it enters the for loop:
   - thread 0: loads element 0 at byte-adressing location 0 (4bytes)
   - thread 1: loads element 4 at byte-adressing location 16 (4bytes)
   - thread 2: loads element 8 at byte-adressing location 32 (4bytes)
   - ....
   - Now we do a bunch of compute
   - thread 0: stores element 0 at byte-adressing location (4bytes)
   - thread 1: stores element 4 at byte-adressing location 16 (4bytes)
   - thread 2: stores element 8at byte-adressing location 32 (4bytes)
   - ...

     As mentioned, the hardware issues memory transactions in 32-byte aligned, 32-byte wide chunks. Consider the load operation for the first warp: Thread 0 requests bytes 0-3. Thread 1 requests bytes 16-19. Thread 2 requests bytes 32-35, and so on. To satisfy thread 0 and thread 1, the hardware must issue a 32-byte transaction covering addresses 0-31. To satisfy thread 2, it must issue *another* 32-byte transaction covering addresses 32-63. This continues for the entire warp. Notice how multiple 32-byte transactions are required to fetch data that, if accessed sequentially (coalesced), could potentially fit into fewer transactions.
2. Assuming no caching, calculate the memory bandwidth overhead created by this access pattern. Hint: For each 32-byte segment fetched from global memory, how much of that data is actually used by the threads in the warp during that specific warp-wide instruction?
3. Compile and run this uncoalesced kernel as before, recording its runtime. Does the observed runtime difference roughly align with your calculated overhead? Don't worry if it doesn't match: Caching (especially L2) can significantly influence performance, mitigating some overhead. However, you should still observe a substantial slowdown (potentially >100%) compared to the coalesced version.
4. Make a file called exercise_1.2_solution.txt and write your calculated memory bandwidth overhead in there.

5. (Optional) Profile this kernel again with Nsight Compute and check the `Memory Workload Analysis` section. You should see a higher L2 cache hit rate and a comment on the `L1TEX Global Access Pattern`. Does the information provided by Nsight Compute align with your understanding of the overhead?

Exercise 1.3:

Using the file `exercise1.3`. Your task is to change only the array indexing within the loop so that memory accesses become fully coalesced. Each thread must still process 4 elements in total, but the specific elements which are accessed by each thread should be changed to achieve coalescing.
The provided code includes checks for numerical correctness. The runtime of your modified kernel should be significantly faster than the uncoalesced version and comparable to (or potentially even slightly faster than) the original `exercise1_coalesced` results in the first part of this exercise.

Make a file called exercise_1.2_solution.txt and copy the entire new .cu file into there for grading.

Exercise 1.4 (Optional, dealing with L1 Caching):

CUDA performance can sometimes be counterintuitive. We will now examine a kernel that demonstrates the potential impact (and pitfalls) of L1 caching.

1. As you did in the previous exercises go and have a look at `exercise1_weird_2.cu`. It is a slightly modified version of the un-coalesced kernel from before. Think about what will happen.
2. Compile and run this kernel. You will likely observe a surprisingly fast execution time.
3. Now use nsight compute to profile this kernel and check that you are still seeing the `L1TEX Global Access Pattern` message that you saw when profiling the uncolesced kernel from exercise 1.2.
4. In the same section you should see a message about L1 cache hit rate which should give you a hint about why we are seeing very fast kernel execution times.
5. What you should take away from this quick detour:
    - Under specific circumstances (often involving immediate data reuse within a warp or block), the L1 cache *can* sometimes mitigate the penalty of non-coalesced global memory accesses.
    - The GPU L1 cache is relatively small. If there are intervening operations or if the data footprint exceeds the cache capacity between the fetch and reuse, L1 cache hits become unlikely. Performance then relies on the much slower L2 cache or global memory.
    - Do not rely on this caching behavior. It is often fragile and not guaranteed across different GPU architectures or workloads.

- If you want to have the maximum memory throughput, have a look at the reference kernel execution that uses vectorized HBM loads instead of looping in a coalesced manner. This reduces the instruction overhead and is the safest way to get optimal throughput, as it relies less on hardware coalescing.

Exercise 2: Shared Memory

(Reference: This exercise is inspired by material demonstrating shared memory banking, originally developed by Mark Harris of NVIDIA.)

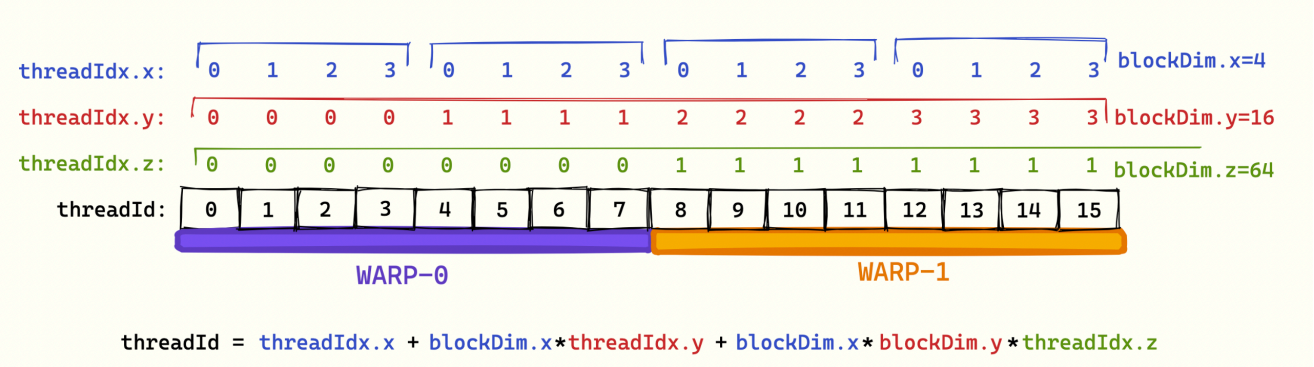
We will now explore different methods for transposing a matrix on the GPU. An efficient approach leverages a key GPU feature: shared memory. If you are unfamiliar with shared memory or need a refresher, please read this introduction: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/)
TLDR:

- Shared memory is allocated on a per-block basis. All threads within a block can access that block's shared memory. It serves as a fast, user-managed cache and enables efficient communication and data sharing among threads within the same block.
- Shared memory is physically organized into banks (32). Each bank can service one request per cycle. If multiple threads within a warp access *different* addresses that fall into the *same* bank simultaneously, a *bank conflict* occurs. This conflict serialises the accesses to that bank, reducing effective bandwidth. Accesses are conflict-free if all threads in a warp access addresses in different banks or if all threads access the exact same address within a bank (broadcast).

Exercise 2.1:

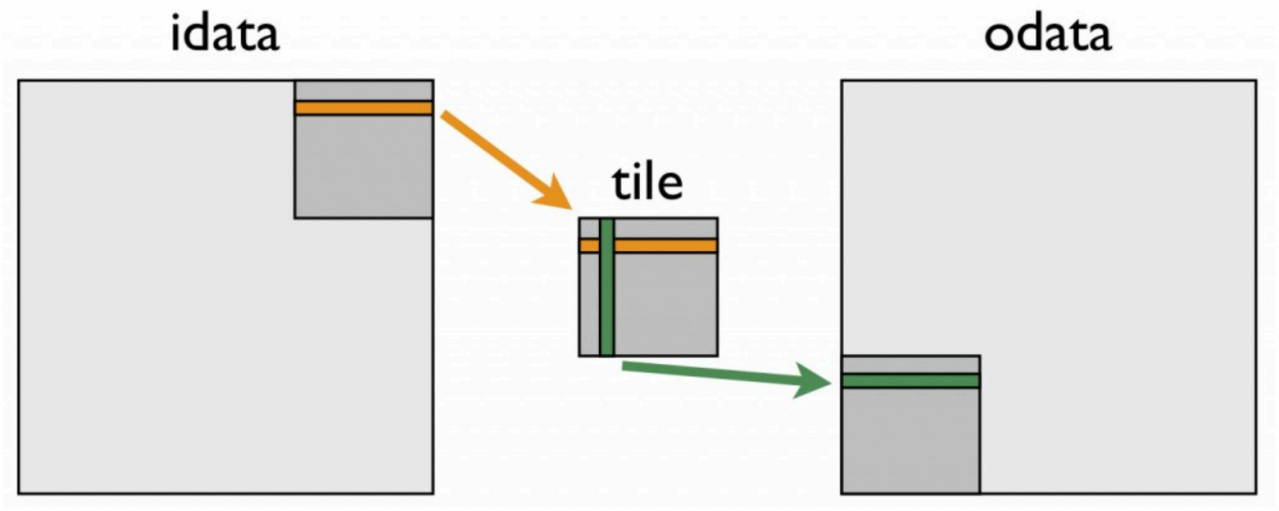
A simple, naive matrix transpose implementation (without using shared memory) is provided in the file:
`exercise_2/transpose_naive`.
This kernel introduces multi-dimensional block and grid dimensions (`dim3`). These are primarily a programmer convenience for organizing threads, especially when working with multi-dimensional data like matrices. It's important to understand how threads, identified by their multi-dimensional `threadIdx` and `blockIdx`, are linearly mapped to form warps. Threads are assigned a unique index calculated using their `threadIdx` and `blockIdx`. The linearization for 3D blocks within a grid is, which tell you how threads are grouped into warps:

$$threadId = threadIdx.x + blockDim.x * (threadIdx.y + blockDim.y * threadIdx.z)$$



1. Read this kernel and try to understand what it does.
2. Run and profile it using ncu as you learned before. Have a look at the memory metrics again and try to use what you learned in the last exercise to reason about wether this kernel is efficient or not? If it is not, then why is it not efficient?
3. Make a file called exercise_2.1_solution.txt and write your answer of `2.2`. One or two sentences is enough.

Exercise 2.2:



We will now look at a second kernel that solves the HBM issue of the naive kernel.

1. No need to code anything just yet but try to follow the kernel line for line and understand what is happening in the kernel in file `exercise_2/transpose_still_naive`
2. Compile and run this kernel. You should observe a change in runtime compared to the first naive version. Does the performance difference align with your understanding of the code's changes?
3. Profile the kernel with nsight compute and have a look at the `Memory Workload Analysis` section. Look for metrics or warnings indicating shared memory bank conflicts. If conflicts are present think about the degree of conflicts that happen and why (e.g., '2-way', '4-way').
4. There is a common technique to avoid shared memory bank conflicts during a transpose, often requiring a small amount of extra shared memory (padding). Think

about how you could modify the shared memory access pattern or layout to prevent these conflicts.

HINT: In the storing to smem phase a warp stores a row of data. When loading from smem the warp therefore loads a column! Since we have 32 banks the loading will be conflict-free since each row element goes into a different bank. But when loading a column then that entire column is in the same bank!

5. Make a file called exercise_2.2_solution.txt and copy the entire modified `.cu` file with the resolved bank conflicts.