CSE 4373/5373 - General Purpose GPU Programming

GPU Performance Basics

Alex Dillhoff

University of Texas at Arlington

Memory Coalescing

Memory Coalescing

- Global memory accesses are one of the largest bottlenecks in GPU applications.
- DRAM has high latency based on its design.
- Each cell has a transistor and a capacitor.
- If the capacitor is charged, it represents a 1.
- The process to detect the charges in these cells is on the order of 10s of nanoseconds.

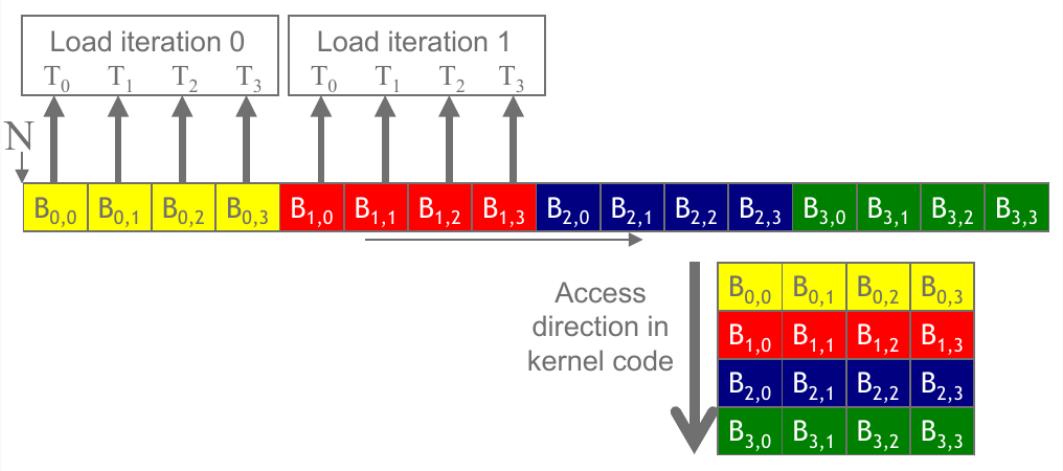
Memory Coalescing

- DRAM can read consecutive groups of cells via *bursts*.
- Data that is stored consecutively can be accessed within the same burst.
- With random access, the DRAM will have to make multiple bursts to read the required data.
- **Memory coalescing** refers to optimizing our global memory accesses to take advantage of DRAM bursts.

Memory Coalescing

- Some problems are naturally coalesced.
- Matrix multiplication, for example, is naturally coalesced since the data is stored in a contiguous manner.
- For other problems, we may have to reorganize our data to take advantage of coalescing.

Memory Coalescing



Memory accesses for a row-major matrix (NVIDIA DLI).

Memory Coalescing

Strategies to optimize code for memory coalescing are to rearrange the threads, the data, or transfer the data first to shared memory so that accesses are faster, referred to as **corner turning**.

Example: Matrix Multiplication

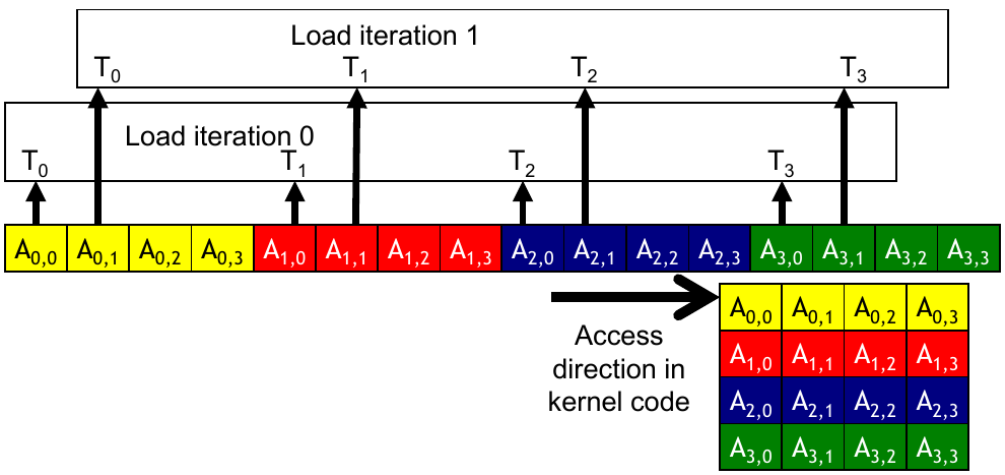
Example: Matrix Multiplication

Consider $C = AB$, where A is in row-major order and B is in column-major order.

The naive implementation of this algorithm will have poor memory coalescing since the values required are not consecutive in memory.

The DRAM will have to make multiple bursts to read the data.

Memory Coalescing



Memory accesses for a column-major matrix (NVIDIA DLI).

Example: Matrix Multiplication

- The accesses to the elements in B will be slower since the data is not coalesced.
- Accessing the elements *is* efficient if we assign N consecutive threads to load N consecutive elements from the same column of B.
- This works in conjunction with tiling.

Example: Matrix Multiplication

- The original loads to shared memory pull from consecutive elements in *B* which allows the application to take advantage of DRAM bursts.
- Once the data is in shared memory, the rest of the algorithm can be performed with coalesced accesses.
- Shared memory uses SRAM instead of DRAM, so coalescing is not an issue.

Hiding Memory Latency

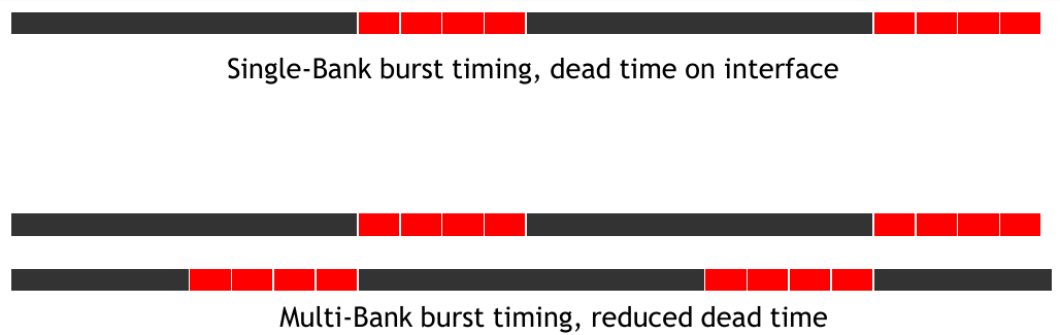
Hiding Memory Latency

- DRAMS have *banks* and *channels*.
- A controller has a bus that connects banks to the processor.
- When the DRAM accesses data, the decoder enables the cells so that they can share the information stored with the sensing amplifier.

Hiding Memory Latency

- This presents a high latency relative to the time it takes to actually transfer the data.
- This is why there are multiple banks per channel.
- The controller can initiate accesses on other banks instead of sitting and waiting for a single bank to finish.

Hiding Memory Latency



Single versus multi-bank burst timings (NVIDIA DLI).

Hiding Memory Latency

- It is possible that the controller will initiate a request to a bank that is already busy.
- This is called a *bank conflict*.
- The controller will have to wait for the bank to finish its current request before it can service the new request.
- The more banks that are available, the less likely it is that a bank conflict will occur.

Example: Matrix Multiplication

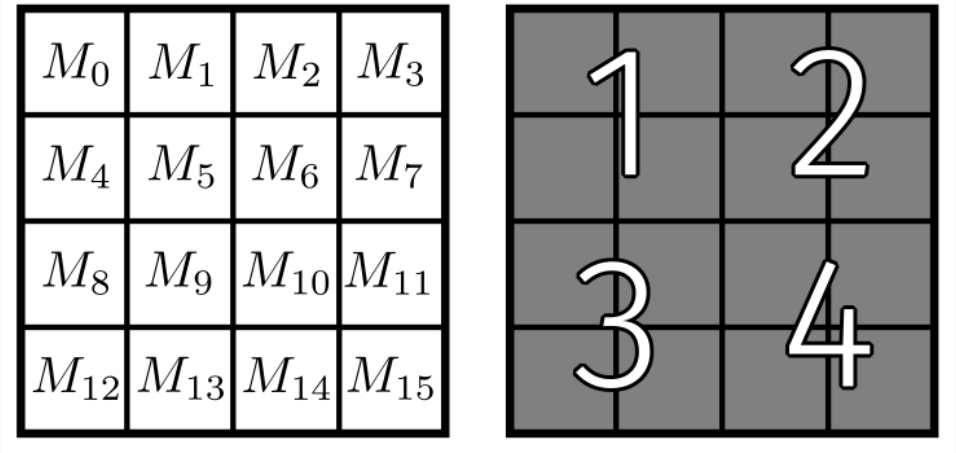
Example: Matrix Multiplication

Consider DRAM with 4 channels and 2 banks per channel.

The burst size of this DRAM is 8 bytes, or 2 elements.

When data is written to DRAM in the first place, it is distributed in an interleaved fashion across the different channels and banks.

Hiding Memory Latency

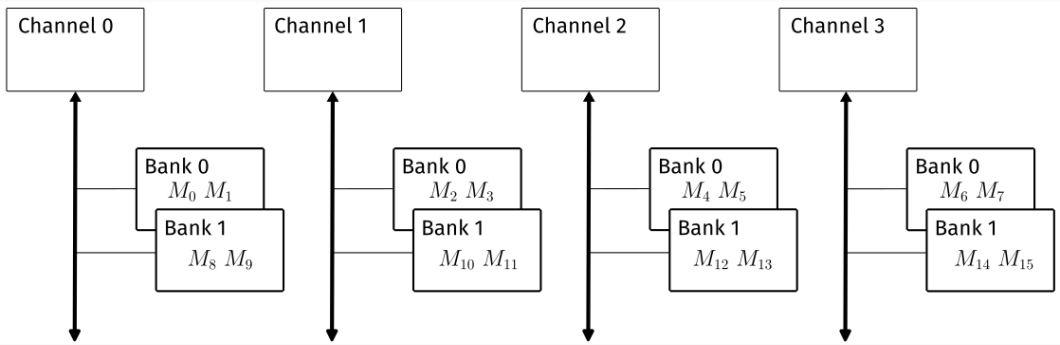


Matrix M with linearized indices and P split into 4 blocks.

Example: Matrix Multiplication

- M is loaded into DRAM in an interleaved fashion.
- The first 8 bytes are loaded into bank 0 of channel 0.
- The next 8 bytes go into bank 0 of channel 1, and so on.
- Each burst returns 8 bytes.
- While the first access is being performed on bank 0 channel 0, the controller can initiate a request to bank 0 channel 1.

Hiding Memory Latency



DRAM distribution for matrix M.

Example: Matrix Multiplication

- Given the distribution visualized above, we can see that the data accesses for the blocks of P will be coalesced.
- Output tile 1 in matrix P requires M_0, M_1, M_4, and M_5.
- The first two are available in a single burst from channel 0 bank 0, and the second two are available in a single burst from channel 2 bank 0.

Thread Coarsening

Thread Coarsening

The *price of parallelism* may refer to the cost of

- launching threads
- synchronization
- redundant work
- redundant memory accesses

Thread Coarsening

It there are enough hardware resources available, parallelism at the finest level is ideal.

If there are not enough resources, there is a price to pay for this parallelism.

The hardware will need to serialize the work into blocks of threads that can be executed in parallel.

Thread Coarsening

If this is the case for a particular application, it may be beneficial to apply some form of **thread coarsening**.

If the hardware would serialize the work due to inefficient resources, the price of parallelism was paid for nothing.

As the programmer, you have the ability to coarsen the threads to alleviate the price of parallelism.

Example: Coarsened Tiled Matrix Multiplication

Coarsened Tiled Matrix Multiplication

- In tiled matrix multiplication, it is possible that two separate blocks work with the same tile of data from an input matrix.
- We pay a price for this redundancy, but the benefit is that we can parallelize the work.
- If the hardware does not have sufficient resource, it will serialize these two blocks.
- This results in paying the price of data redundancy without the benefit of parallelism.

Coarsened Tiled Matrix Multiplication

Let $A, B \in \mathbb{R}^{6\times 6}$, then $C = AB \in \mathbb{R}^{6\times 6}$.

If we use a 2×2 tile size, then we have 9 blocks of work that can be executed concurrently.

For argument's sake, let's say that the hardware can only execute 3 blocks of work at a time.

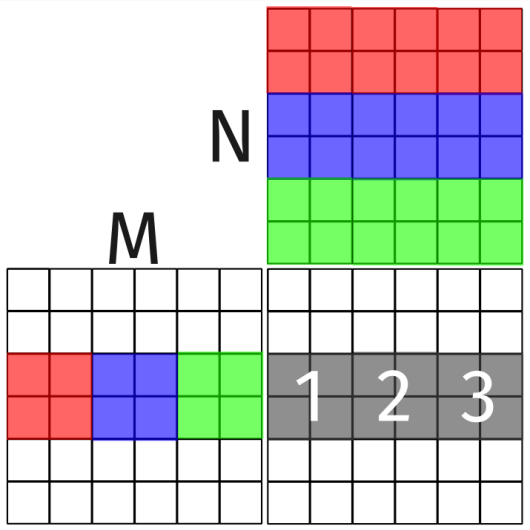
Coarsened Tiled Matrix Multiplication

We can use thread coarsening to reduce the number of blocks to 3.

Each block will be responsible for a single row of the tiled output matrix.

If the output matrix is 6×6, then each block will be responsible for a 2×6 tile of the output matrix.

Coarsened Tiled Matrix Multiplication



Coarsened Tiled Matrix Multiplication

- The block itself will perform a similar function as the implementation of tiled matrix multiplication we saw previously.
- The kernel should be modified so that it processes values to fill for 3 blocks of work, spanning each row.
- Although each block uses a different column from matrix *N*, they all use the same row from matrix *M*.
- A coarsened solution will take advantage of this reuse of data.

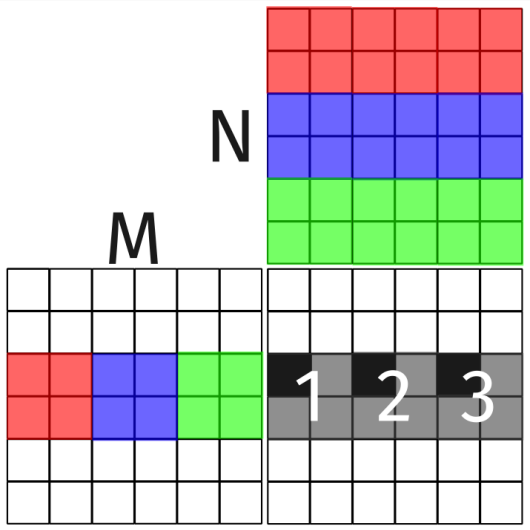
Coarsened Tiled Matrix Multiplication

Consider the thread that computes the value for the top left entry of block 1.

This thread will compute the output value as normal before looping to compute the corresponding relative position in blocks 2 and 3.

If the first entry computed is $(0, 0)$ of block 1, then the next entry computed will be $(0, 0)$ of block 2, and so on.

Coarsened Tiled Matrix Multiplication



Coarsened Tiled Matrix Multiplication

The kernel for this starts by establishing shared memory.

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

Coarsened Tiled Matrix Multiplication

The row is computed as normal, but the column is offset by a factor of coarsening.

```
int row = by * TILE_WIDTH + ty;
int colStart = bx * TILE_WIDTH * COARSE_FACTOR;
```

Coarsened Tiled Matrix Multiplication

An intermediate array is used to store the results of the coarsened computation.

```
float Pvalue[COARSE_FACTOR];
for (int i = 0; i < COARSE_FACTOR; i++) {
    Pvalue[i] = 0.0f;
}
```

Coarsened Tiled Matrix Multiplication

```
for (int ph = 0; ph < width / TILE_WIDTH; ph++) {
    Mds[ty][tx] = M[row * width + ph * TILE_WIDTH + tx];

    for (int c = 0; c < COARSE_FACTOR; c++) {
        int col = colStart + c * TILE_WIDTH;

        Nds[ty][tx] = N[(ph * TILE_WIDTH + ty) * width + col];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; k++) {
            Pvalue[c] += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
    }
}
```

Coarsened Tiled Matrix Multiplication

The results are then written to the output matrix.

```
for (int c = 0; c < COARSE_FACTOR; c++) {
    int col = colStart + c * TILE_WIDTH;
    P[row * width + col] = Pvalue[c];
}
```

Applying Coarsening

Applying Coarsening

- Thread coarsening is yet another technique that can be applied to optimize your parallel programs.
- The previous section demonstrated *how* it can be applied, but *when* should it be applied.
- Deciding on whether to apply this technique is largely determined by careful analysis of your current application.
- This analysis should include benchmarking and profiling.

Applying Coarsening

- We can at least discuss when *not* to apply coarsening.
- The most obvious instance is when coarsening is completely unnecessary.
- Consider the vector addition kernel.
- Each thread performs an independent computation that has no overlapping data with other thread.

Applying Coarsening

- Another bad case for implementation would be when the coarsening factor causes hardware to be underutilized.
- Parallelization in hardware is scalable.
- If we take away the opporunity for scale, there may be unused compute.
- This is typically something we can determine via benchmarking.

Applying Coarsening

- In the coarsened version of matrix multiplication, additional private variables were used to store the coarsened values.
- These use additional registers per thread.
- If the application required more than the 32 registers available on our H100, for example, this would have a direct effect on occupancy.

Optimization Checklist

Optimization Checklist

So far we have discussed the following optimizations

- Maximizing occupancy
- Coalesced global memory accesses
- Minimizing control divergence
- Tiling
- Thread coarsening

Maximizing Occupancy

- Having more threads than physical cores available is beneficial, as it hides the latency required for other operations such as data fetching.

Maximizing Occupancy

- Having more threads than physical cores available is beneficial, as it hides the latency required for other operations such as data fetching.
- Instead of waiting on some operation to return, the hardware can switch to another thread to perform work.

Maximizing Occupancy

- Having more threads than physical cores available is beneficial, as it hides the latency required for other operations such as data fetching.
- Instead of waiting on some operation to return, the hardware can switch to another thread to perform work.
- This is implemented by adjusting the launch configurations or optimizing the number of registers used per thread, for example.

Coalesced Global Memory Accesses

- Random accesses to memory are less efficient than consecutive ones.

Coalesced Global Memory Accesses

- Random accesses to memory are less efficient than consecutive ones.
- This is a theme that is repeated through many themes of computer science, such as sorting.

Coalesced Global Memory Accesses

- Random accesses to memory are less efficient than consecutive ones.
- This is a theme that is repeated through many themes of computer science, such as sorting.
- Rearrange data to take advantage of DRAM bursts.

Coalesced Global Memory Accesses

- Random accesses to memory are less efficient than consecutive ones.
- This is a theme that is repeated through many themes of computer science, such as sorting.
- Rearrange data to take advantage of DRAM bursts.
- Load data from DRAM in a coalesced mannner into shared memory first.

Minimizing Control Divergence

- During SIMD execution, the hardware executes the same instructions on multiple data elements.

Minimizing Control Divergence

- During SIMD execution, the hardware executes the same instructions on multiple data elements.
- If a thread or group of threads would diverge from the others, the hardware would have to make multiple passes to cover all of the possible paths.

Minimizing Control Divergence

- During SIMD execution, the hardware executes the same instructions on multiple data elements.
- If a thread or group of threads would diverge from the others, the hardware would have to make multiple passes to cover all of the possible paths.
- In some cases, rearranging the data is feasible to minimize divergence.

Tiling

- Global memory accesses exhibit higher latency due to the nature of DRAM.

Tiling

- Global memory accesses exhibit higher latency due to the nature of DRAM.
- We can reduce the number of global memory accesses by using shared memory.

Tiling

- Global memory accesses exhibit higher latency due to the nature of DRAM.
- We can reduce the number of global memory accesses by using shared memory.
- This was exemplified in the tiled matrix multiplication examples, where there are many redundant data accesses.

Tiling

- Global memory accesses exhibit higher latency due to the nature of DRAM.
- We can reduce the number of global memory accesses by using shared memory.
- This was exemplified in the tiled matrix multiplication examples, where there are many redundant data accesses.
- Moving these data to shared memory reduces the number of global memory accesses.

Thread Coarsening

- In cases where the hardware would serialize execution of a kernel, thread coarsening can eliminate redundant work.

Thread Coarsening

- In cases where the hardware would serialize execution of a kernel, thread coarsening can eliminate redundant work.
- In the tiled matrix multiplication example, we saw that the hardware would serialize execution of the kernel if there were not enough resources available.

Thread Coarsening

- In this case, the same redundant loads to shared memory would be performed.
- To reduce this overhead, we coarsened the thread by having a single kernel perform the work of multiple blocks.

What’s Next?

What's Next?

Knowing when to apply each of these optimization techniques comes down to understanding your application.

The single most important step in optimizing your application is to identify the bottleneck.

Our next step is to learn the tools that will help us identify these bottlenecks.

What's Next

The first phase of the course has come to an end.

The next module of this course will focus on problems for which a straightforward solution is not obvious.

These are problems that come from other domains of computer science, such as graph theory and linear algebra.