CSE 4373/5373 - General Purpose GPU Programming

CUDA Memory Architecture

Alex Dillhoff

University of Texas at Arlington

CUDA Memory Architecture

- All of our kernels have been using global memory
- More complex kernels will end up being memory bound
- CUDA devices offer a variety of memory types

Memory Access

Memory Access

- Transferring memory is one of the biggest bottlenecks in GPU programming.
- Companies like NVIDIA devote a lot of resources to improving the bandwidth and latency of memory transfers.
- When training a deep learning model, the datasets used are far too large to fit on the GPU.
- This means that the data must be transferred to the GPU before the actual training code can execute on the device.

Memory Access

- In matrix multiplication, data accesses are limited to a single line of code in the inner-most loop.
- The memory access pattern is very regular and predictable.

Memory Access

```
for (int k = 0; i k < numCols; k++) {
    Cvalue += A[row * numCols + k] * B[k * numCols + col];
}
```

- This line consists of a floating-point multiplication, floating-point addition, and two memory accesses.
- The result is not stored yet, so there is no access to the C matrix.

Memory Access

- The operation effiency can be described in terms of floating-point operations per second (FLOP/s) and the accesses can be measured in the number of bytes transferred.
- In this case, we have 2 FLOPs and 8 bytes transferred.
- This means that the ratio of FLOPs to bytes transferred is 0.25 FLOP/B.
- This is described as **computational intensity**.

Memory Access

- If our kernel relies on too many memory accesses, then the computational intensity will be low.
- The GPU will be spending more time waiting for data to be transferred than actually performing computations.
- The goal is to increase the computational intensity as much as possible.

Memory Access

- The H100 SXM5 has 3TB/s of memory bandwidth.
- This global memory bandwidth limits the kernel to 3000 * 0.25 = 750 GFLOP/s.
- The peak performance of the H100 is 66.9 TFLOPS.
- If the specialized Tensor cores are utilized, the peak performance is 494.7 TFLOPS.

Memory Access

- That means that are kernel is only using 0.15% of the peak performance of the GPU.
- This program is certainly **memory bound**.
- Our theoretical limit to computational intensity is the peak performance of the GPU.
- Programs that achieve this peak are called **compute bound**.

Memory Access

- Based on the tools we have discussed so far, it is not clear how we can optimize this kernel.
- The only way to improve the computational intensity is to reduce the number of memory accesses.
- Modern GPUs have more than just global memory.

Memory Types

Memory Types

- **Global Memory** - The largest memory space on the GPU. It is accessible to all threads and is the slowest memory type.
- **Local Memory** - Resides on global memory, but is not shared between threads. It includes local variables and function arguments.

Memory Types

- **Shared Memory** - A small, fast memory space that is shared between threads in a block.
- **Constant Memory** - A small, read-only memory space that is cached.
- **Registers** - The fastest memory space on the GPU. It is private to each thread.

Memory Types

Data in CPU registers are swapped depending on the context of the program.

GPU registers are consistent even when other threads are launched to hide latency.

This results in a larger register file on the GPU.

Memory Types

Following the von Neumann architecture, memory that is closer to the chip is faster but more expensive.

Data residing on registers is the most ideal for performance since the processor can work directly with the register values.

Memory Types

This benefit comes in the form of energy consumption as well.

Transferring data from global memory to the chip requires additional cycles resulting in more energy used.

Variable Declarations

Variable declaration	Memory	Scope	Lifetime
Automatic variables (not arrays)	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
`__device__ __shared__ int SharedVar;`	Shared	Block	Grid
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

Variable Declarations

- Automatic array variables should seldom be used.
- Each variable will be allocated to a register resulting in faster access times.
- Global variables are more commonly used to pass information to another kernel that is being launched.

Tiling

Tiling

These memory types serve as tools that we can use to increase efficiency.

The first pattern discussed is **tiling**.

Tiling is a well-described technique that has a fitting analogy.

Tiling

If a wall needs to be tiled, it is more efficient to use many small tiles that are lighter and easier to handle.

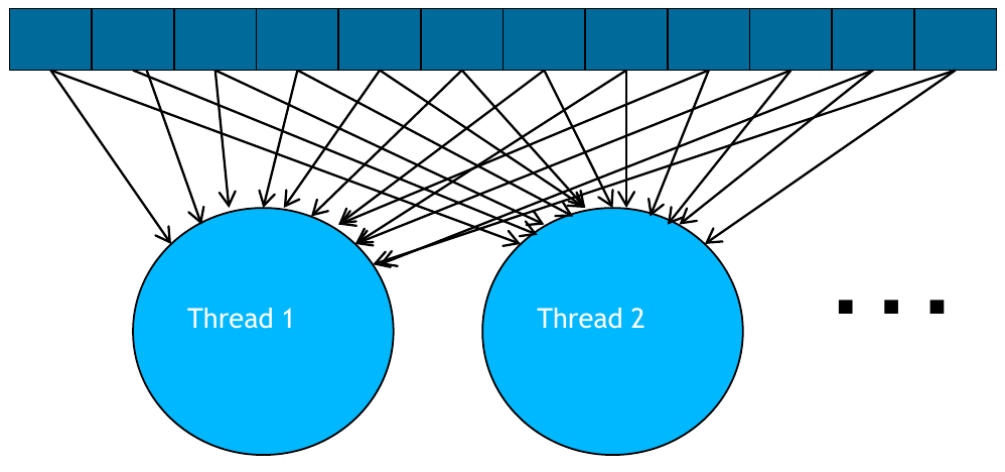
In GPU programming, the wall represents the entire global memory space.

The individual tiles are local memory that is allocated to each thread block.

Tiling

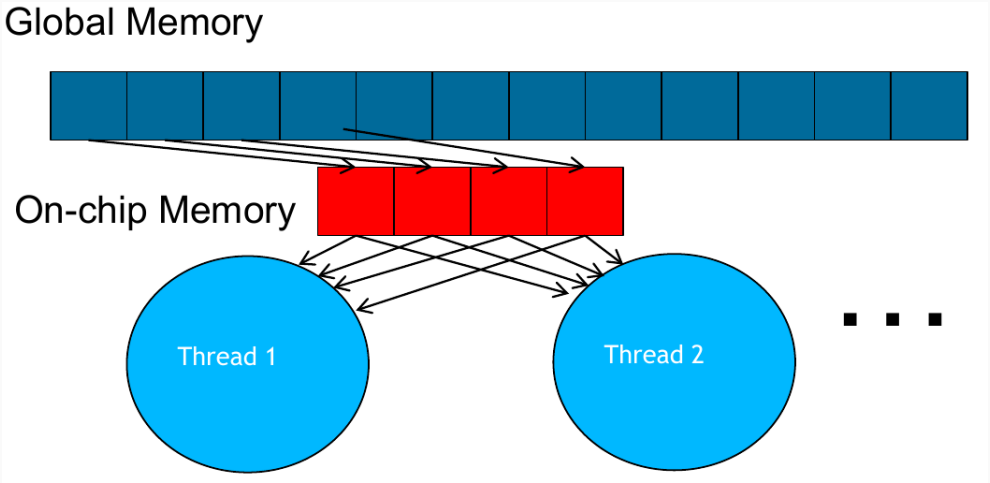
- The kernels we have seen so far have used a *global memory access pattern*.
- In this pattern, all threads have access to every data point from the input.
- Using a *tiling pattern*, we can optimize memory accesses by moving shared resources to local memory that is faster to access.

Tiling



Global memory pattern (source: NVIDIA DLI).

Tiling



Tiling pattern (source: NVIDIA DLI).

Tiling

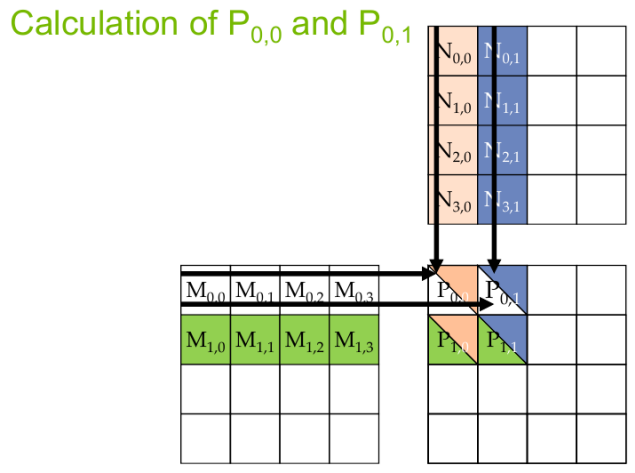
Tiling itself is quite simple in concept...

The challenge will be identifying when the tool can be properly applied.

Tiling

- Consider matrix multiplication: the naive kernel we explored previously uses each thread to compute one value of the output matrix.
- This kernel uses a global memory access pattern, and we can identify that many of the computations require the same input.
- They key to introducing tiling for matrix multiplication will be **identifying which data use reused.**

Tiling



Memory accesses for matrix multiplication(source: NVIDIA DLI).

Tiling

- In the previous example, the block size is 2×2.
- Each row of the block relies on the same input row from the matrix on the left.
- $P_{0,0}$ and $P_{0,1}$ will use the same data from the first row of M.

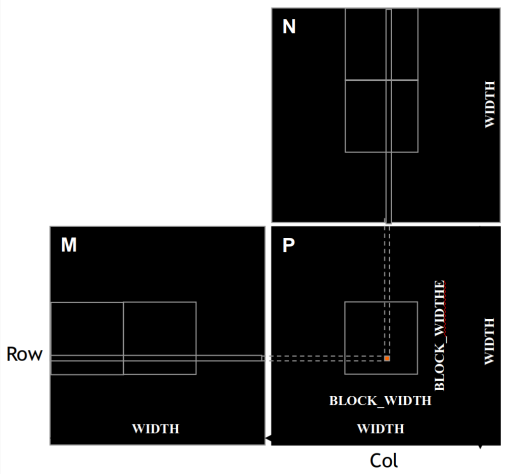
Tiling

- In our original kernel, this requires 8 global memory accesses.
- If we placed this row in shared memory, each output thread could access the values much quicker.
- We can see a similar pattern for the column values in N.

Tiling

- Since we are using tiling with a block size of B, we will consider working with $2B$ values from the input at a time.
- If the number of values we need to compute an output entry exceeds $2B$, then we can synchronize the threads before moving to the next section.

Tiling



Tiled matrix multiplication overview (source: NVIDIA DLI).

Tiled Matrix Multiplication

Tiled Matrix Multiplication

The concept of tiled matrix multiplication is this: load a subset of data from M and N into shared memory before using that data to perform the dot product.

We have a few limitations to think about here...

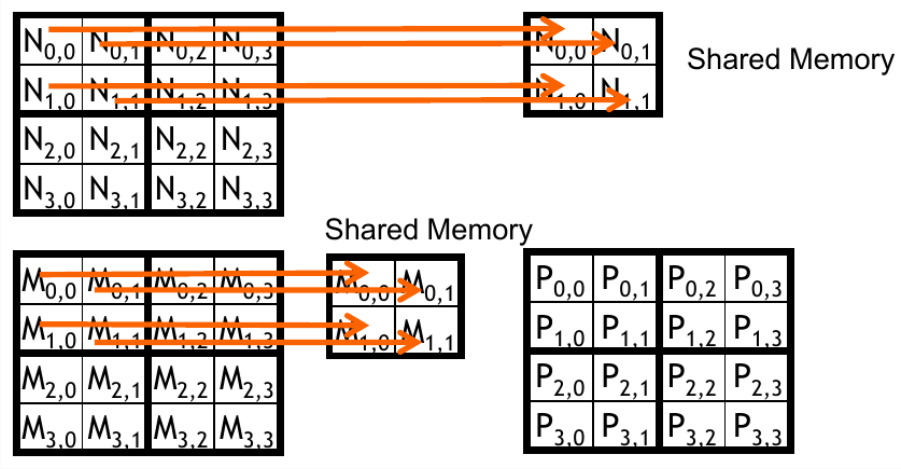
Tiled Matrix Multiplication

- The amount of shared memory is much smaller than global memory; we cannot fit all the data at once.
- The block size will limit how many elements can be loaded into shared memory at once.
- As suggested by tiling, we are only working with a small chunk at a time.

Tiled Matrix Multiplication

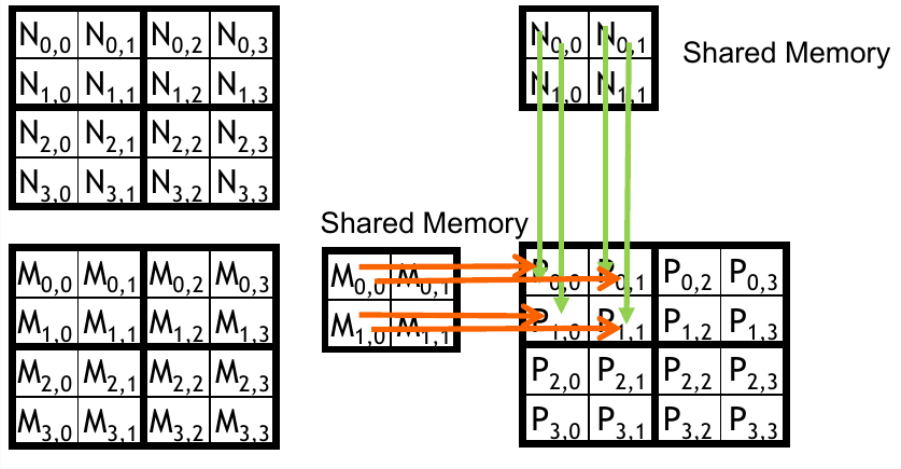
- Using a 2×2 block gives us 4 threads to work with.
- Overlaying that block on the input only allows us to grab 2 values each from the first 2 rows in M and 2 values each from the first 2 columns in M.
- For each tile, the subset of data will be loaded in followed by adding the dot product of the subset to the current value.

Tiled Matrix Multiplication



Loading the first tile (source: NVIDIA DLI).

Tiled Matrix Multiplication



Computing the dot product on the first subset (source: NVIDIA DLI).

Tiled Matrix Multiplication

- The block will move to the next subset of data to finish computing the first block of the output matrix.
- This process can be arbitrarily scaled up to support larger matrices without necessarily increasing the block size.
- We would want to increase the block size to take advantage of the additional threads.

Tiled Matrix Multiplication

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Tiled matrix multiplication computations (source: NVIDIA DLI).

Implementation in CUDA

Implementation in CUDA

Our implementation should follow these steps:

1. Establish shared memory for the input from matrix M and matrix N.
2. Load the first subset of data from M and N into shared memory (remember to synchronize threads).
3. Compute the dot product of the subset (remember to synchronize threads).
4. Repeat steps 2 and 3 until all subsets have been computed.

Implementation in CUDA

Step 1 is obvious: we need to establish the shared memory for this solution.

Steps 2 and 3 are the same as described above, but we do need to remember to synchronize the threads.

Without synchronization, the computation may continue before all the data is properly loaded.

Step 4 implies that each thread will loop through the subsets until all values have been computed.

Preparation

```
__global__ void MatMulKernel(float* M, float* N, float* P, int Width) {
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

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

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
```

Execution

```
    float Pvalue = 0;
    for (int ph = 0; ph < Width / TILE_WIDTH; ++ph) {
        Mds[ty][tx] = M[Row * Width + ph * TILE_WIDTH + tx];
        Nds[ty][tx] = N[(ph * TILE_WIDTH + ty) * Width + Col];
        __syncthreads();

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

    P[Row * Width + Col] = Pvalue;
}
```

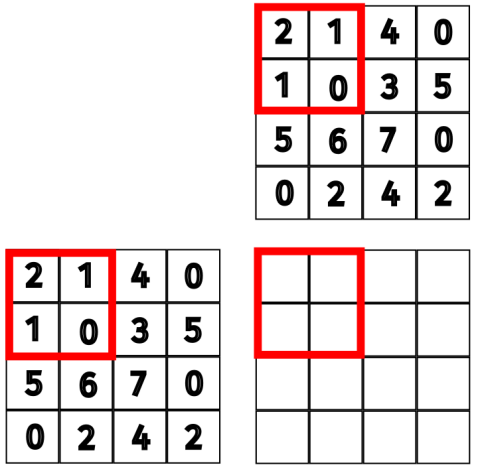
Example

Consider multiplying two 4×4 matrices with a block size of 2×2.

Our block will compute the top left submatrix of the output, $P_{0,0}$, $P_{0,1}$, $P_{1,0}$, and $P_{1,1}$.

We will view the computation from the perspective of the thread for $P_{0,0}$.

Example



Setup of tiled matmul example.

Computing the Indices

```
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;
```

First iteration

- The row and column of the output computed by the current thread is calculated using the block and thread indices.
- This is simply $(0, 0)$ for the first thread.
- It gets slightly more complicated when computing the input subset in the loop.

Execution

```
float Pvalue = 0;
for (int ph = 0; ph < Width / TILE_WIDTH; ++ph) {
    Mds[ty][tx] = M[Row * Width + ph * TILE_WIDTH + tx];
    Nds[ty][tx] = N[(ph * TILE_WIDTH + ty) * Width + Col];
    __syncthreads();
```

First iteration

- The input needs to be transferred to shared memory.
- The loop will skip over a tile at a time.
- At this point, we already know which row of M and column of N we need to access.
- We need to compute the column index for M and the row index for N.

First iteration

- For M, we start with `Row * Width`.
- This needs to be offset by the tile offset index `ph` of the main loop, yielding `Row * Width + ph * TILE_WIDTH`.
- Finally, we need to add the thread index `tx` to get the final index `Row * Width + ph * TILE_WIDTH + tx`.
- The same process is applied to N.

First iteration

Note that this only transfers a single value from each matrix to shared memory, but our computation relies on 2 values from each matrix.

Each thread in the block is collaboratively loading the data into shared memory.

This is why the call to `__syncthreads()` is necessary.

Computing the dot product

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

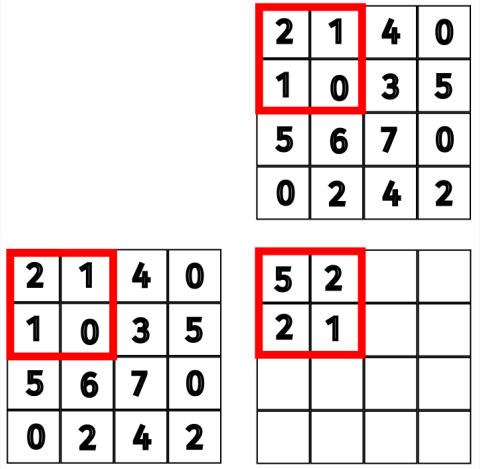
Computing the Dot Product

- The next step is to compute the dot product of the subset.
- Again, we see a call to `__syncthreads()`.
- Without this synchronization, the loop may be allowed to continue and overwrite the data in shared memory before a thread has finished.

Computing the Dot Product

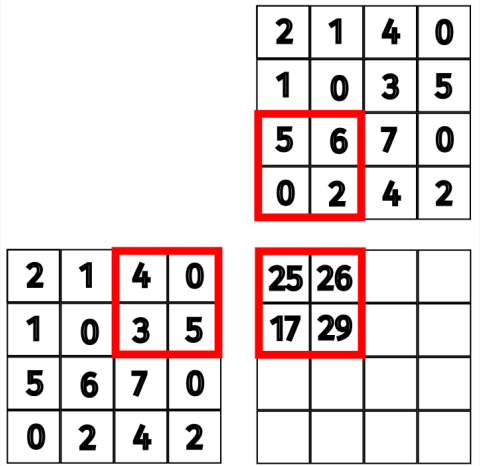
- Once the final value is computed, each thread can freely write it back to global memory.
- Since each thread is computing a different value, there is no need to synchronize the threads before writing to global memory.

Intermediate Output



Updated values for the first subset.

Intermediate Output



Final values for this tile.

Performance Analysis

Performance Analysis

In the naive implementation, we had a computational intensity of 0.25 FLOP/B.

With a 16×16 tile, the number of global memory accesses is reduced by a factor of 16.

This gives us a computational intensity of 4 FLOP/B.

Performance Analysis

We previously stated that the H100 has a global memory bandwidth of 3TB/s.

This means that the theoretical limit for the performance of this kernel is 3000 * 4 = 12000 GFLOP/s which is much better than the 750 GFLOP/s we had before.

Boundary Checking

Boundary Checking

The previous implementation assumed that the width of the matrices was a multiple of the tile width and that the input would always be square matrices.

Consider changing our 2×2 block to a 3×3 block using the same input sizes.

Boundary Checking Solution 1

An issue arises when computing the second tile offset since the block exceeds the boundaries of our input and output.

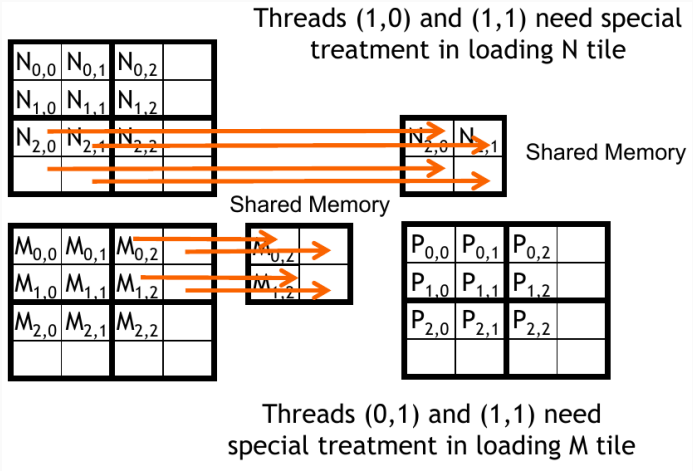
One solution would be to check the boundary condition on both the input, when transferring the data to shared memory, and the output, when reading the data from shared memory.

Boundary Checking Solution 1

This would require a conditional statement in the inner loop.

This is not ideal since the conditional statement would be executed for every thread in the block.

Boundary Checking Solution 1



Using a 3x3 block (source: NVIDIA DLI).

Boundary Checking Solution 2

Another solution is to pad the input with zeros.

If the index is outside our boundary, adding a 0 will not affect the result of the dot product.

This allows for a simpler implementation while still being flexible enough to handle matrices of any size.

Boundary Checking Solution 2

```
float Pvalue = 0;
for (int ph = 0; ph < ceil(Width/(float)TILE_WIDTH); ph++) {
    // Collaborative loading of M and N tiles into shared memory
    if (Row < Width && ph * TILE_WIDTH + tx < Width) {
        Mds[ty][tx] = M[Row * Width + ph * TILE_WIDTH + tx];
    } else {
        Mds[ty][tx] = 0.0;
    }
    if (ph * TILE_WIDTH + ty < Width && Col < Width) {
        Nds[ty][tx] = N[(ph * TILE_WIDTH + ty) * Width + Col];
    } else {
        Nds[ty][tx] = 0.0;
    }
    __syncthreads();
}
```

Memory Use and Occupancy

Memory Use and Occupancy

Just like exceeding the number of registers per thread can negatively affect occupancy, so can over allocating shared memory.

The H100 can have up to 228 KB per SM.

If we are maximizing the 2048 threads available per SM, each block cannot exceed 228 KB / 2048 threads = 112 B/thread.

Memory Use and Occupancy

How much shared memory is used by each block?

Each block has 2 arrays of size $\texttt{TILE\_WIDTH} \times \texttt{TILE\_WIDTH}$ of type `float`.

This gives us a total of

$$2 \times \texttt{TILE\_WIDTH} \times \texttt{TILE\_WIDTH} \times 4 = 8(\texttt{TILE\_WIDTH})^2\text{B}.$$

Memory Use and Occupancy

Each block uses $\texttt{TILE\_WIDTH}^2$ threads, resulting in 8 B/thread.

This is well below the limit of 112 B/thread.

Dynamically Changing the Block Size

Dynamically Changing the Block Size

The solution presented previously uses a constant to determine the tile size.

What if this tile size was not optimal for a given hardware configuration?

We would surely want to adjust this dynamically to maximize performance.

Dynamically Changing the Block Size

In CUDA, we can support this by using the `extern` keyword.

First, we need to define our shared memory as one array:
`extern __shared__ float Mds_Nds[];`.

This is a 1D array that represents the shared memory for both input matrices.

Dynamically Changing the Block Size

When launching this kernel, we need some way to inform it of the tile size.

We can query the device properties and determine the optimal tile size based on the hardware.

Dynamically Changing the Block Size

This size can be used as a third launch configuration input.

```
// Determine optimal tile size
size_t size = compute_optimal_size();
MatMulKernel<<<dimGrid, dimBlock, size>>>(M_d, N_d, P_d,
                                          Width, size/2, size/2);
```

Dynamically Changing the Block Size

- The kernel will need to be modified to use the new shared memory array.
- The first step is to determine the offset for each matrix.
- This is done by multiplying the tile size by the thread index.
- The second step is to use the offset to access the correct value in the shared memory array.

Dynamically Changing the Block Size

```
void MatMulKernel(float* M, float* N, float* P,
                  int Width, int Mds_offset, int Nds_offset) {
    extern __shared__ float Mds_Nds[];

    float *Mds = (float *)Mds_Nds;
    float *Nds = (float *)Mds_Nds + Mds_offset;

    // Rest of the kernel
}
```

Summary

Summary

- Tiling is a powerful tool that can be used to improve the performance of a kernel.
- It is important to understand the memory access pattern of your kernel and identify which data is reused.
- Moving data to shared memory will reduce the number of global memory accesses.