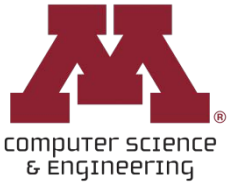
CSCI 5451: Introduction to Parallel Computing

Lecture 21: Additional Performance Considerations



UNIVERSITY OF MINNESOTA
Driven to Discover®

Announcements (11/17)

- HW3 has been released on the course website. You can find the zip file for the assignment here.
- The Project Description is on the course website. You can find it here. Be sure to review this in the coming days. It describes the requirements for the Final Project in detail
- The Project Planning Meeting is worth 4 points and is simply a meeting with me prior to Thanksgiving regarding your project. To get full credit, you must book a time with me here and meet on or before Nov 26
- The remaining due dates are :
  - HW3 -->. Nov 28
  - HW4 (Released Nov 21) --> Dec 5
  - HW5 (Released Nov 28) --> Dec 18
  - Project Planning Meeting --> Nov 26
  - Final Project & Report --> Dec 18

Overview

- HBM Optimizations
  - Coalescing
  - HBM in Depth
  - Request Pipeline
  - Coalescing in Practice
- Thread Coarsening
- General Notes on Optimization

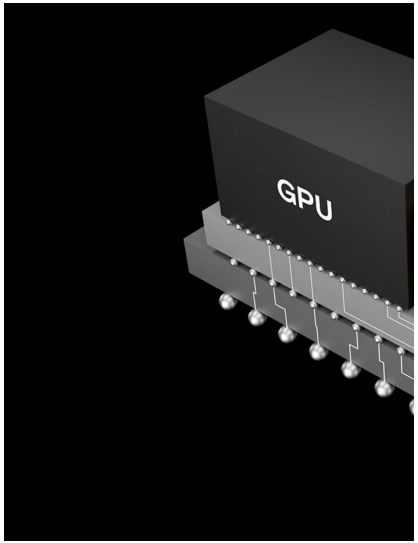
Overview

- **HBM Optimizations**
  - **Coalescing**
  - HBM in Depth
  - Request Pipeline
  - Coalescing in Practice
- Thread Coarsening
- General Notes on Optimization

HBM Optimizations

- Last lecture, we focused on how to utilize shared memory
- This allowed us to utilize cache on the GPU itself
- Today we will be discussing how we can utilize High Bandwidth Memory in the GPU more effectively
- To do so, we will be going in depth on how exactly DRAM-style memory works in general, and GPU HBM in particular

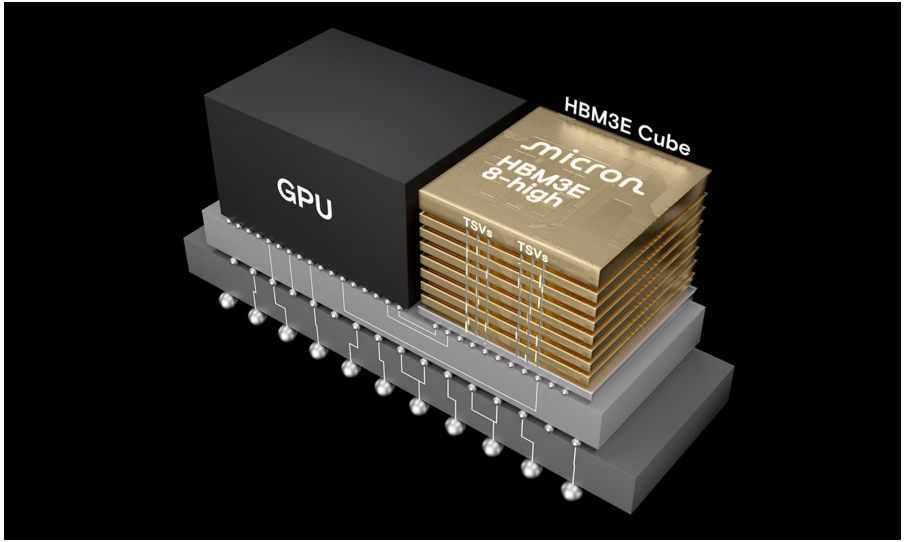
Shared Cache



HBM Optimizations

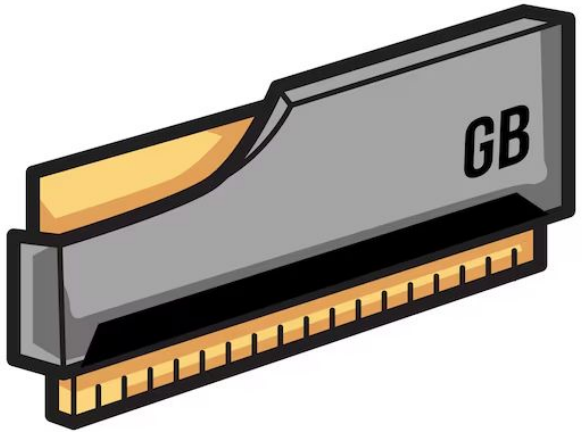
- Last lecture, we focused on how to utilize shared memory
- This allowed us to utilize cache on the GPU itself
- Today we will be discussing how we can utilize High Bandwidth Memory in the GPU more effectively
- To do so, we will be going in depth on how exactly DRAM-style memory works in general, and GPU HBM in particular

High Bandwidth Memory



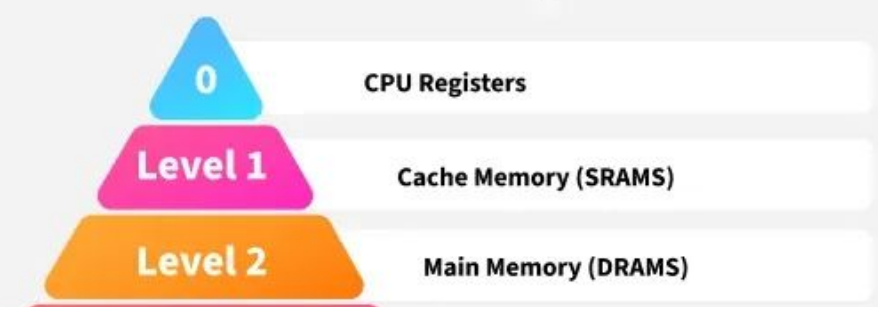
DRAM basics

- DRAM is quite slow
- DRAM can be highly parallelized (we will see how)
- When reading from DRAM, a sequence of data is returned → Not just the single entry (think cache lines, not single values)
  - This range of values is retrieved in parallel from DRAM
- These consecutive locations are called DRAM *bursts*



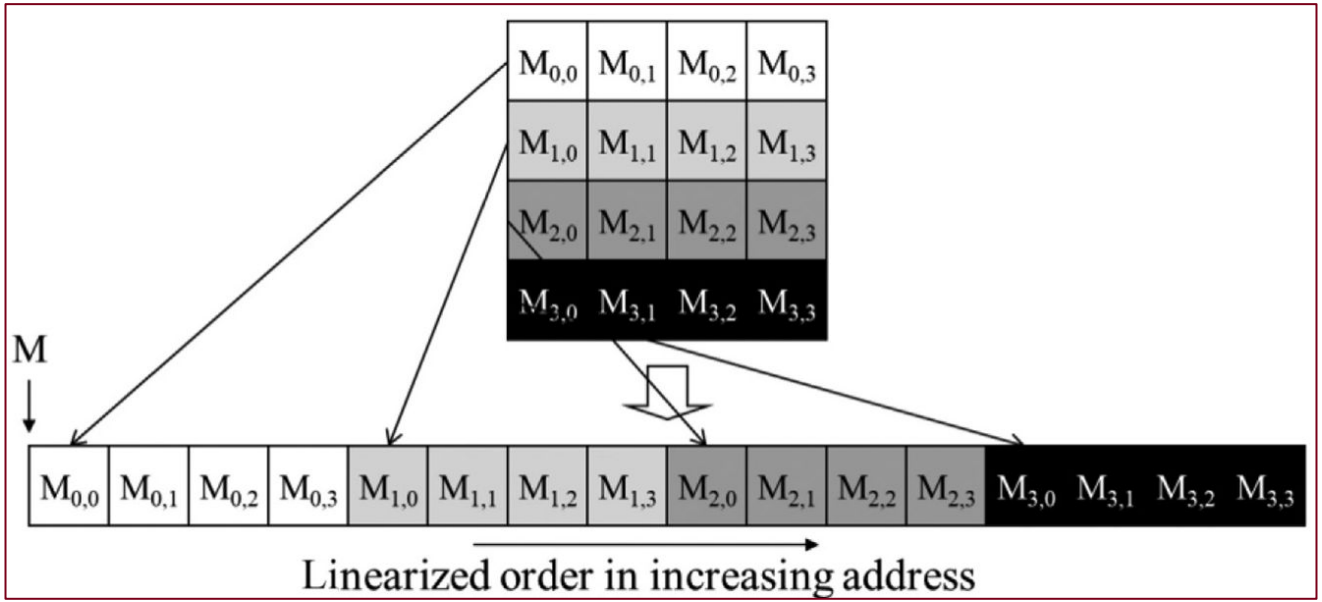
The Tension of Size vs Speed in Memory

- Faster memory requires
  - Larger transistors
  - Larger capacitors
  - More Sensing Circuitry
- All of the above reduce memory density and increase cost
- Thus, you can have faster memory only if you have less of it.
- SRAM is faster than DRAM in this regard



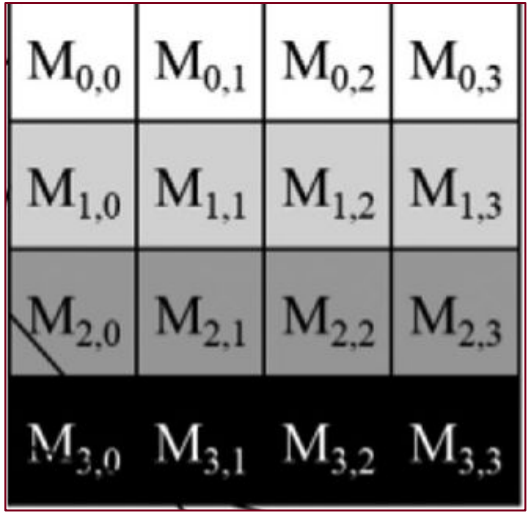
Data Loading in Warps

Oftentimes threads load similar data at a given instruction within a warp → how can we utilize this?



Coalescing

- When multiple threads within a warp all issue loads to consecutive locations in memory, the GPU will *coalesce* them into one request
- We combine multiple requests into a single request
- This reduces redundant loads → don't reload from global memory more than once



Coalescing in Action

In this code stub, assuming both matrices are row-major, which memory accesses will be coalesced?

Code

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
if (row < Width && col < Width) {
    float Pvalue = 0.0f;
    for(unsigned int k = 0; k < Width; ++k) {
        Pvalue += N[row*Width + k]*M[k*Width + col];
    }
    P[row*Width + col] = Pvalue;
}
```

Coalescing in Action

In this code stub, assuming both matrices are row-major, which memory accesses will be coalesced?

M

Note that this is different from serial accesses in that N would normally be viewed as better - it takes advantage of cache locality.

Code

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
if (row < Width && col < Width) {
    float Pvalue = 0.0f;
    for(unsigned int k = 0; k < Width; ++k) {
        Pvalue += N[row*Width + k]*M[k*Width + col];
    }
    P[row*Width + col] = Pvalue;
}
```

Coalescing in Action

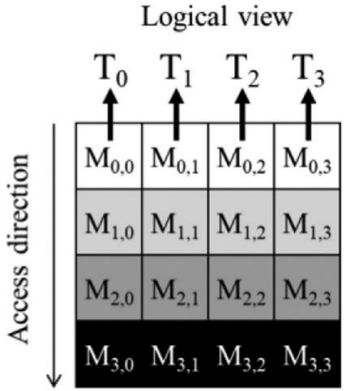
In this code stub, assuming both matrices are row-major, which memory accesses will be coalesced?

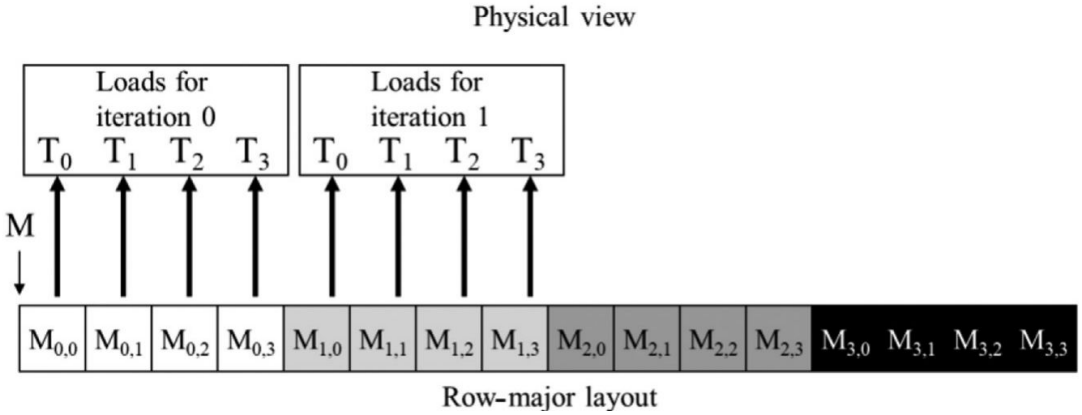
M

Note that this is different from serial accesses in that N would normally be viewed as better - it takes advantage of cache locality.

Code

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
if (row < Width && col < Width) {
    float Pvalue = 0.0f;
    for(unsigned int k = 0; k < Width; ++k) {
        Pvalue += N[row*Width + k]*M[k*Width + col];
    }
    P[row*Width + col] = Pvalue;
}
```





Coalescing in Action

Would this code stub still exhibit memory coalescing for M if M was column-major instead of row-major?

Code

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
if (row < Width && col < Width) {
    float Pvalue = 0.0f;
    for(unsigned int k = 0; k < Width; ++k) {
        Pvalue += N[row*Width + k]*M[col*Width + k];
    }
    P[row*Width + col] = Pvalue;
}
```

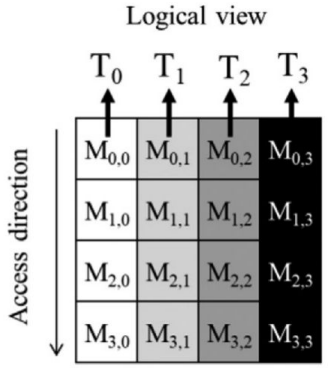
Coalescing in Action

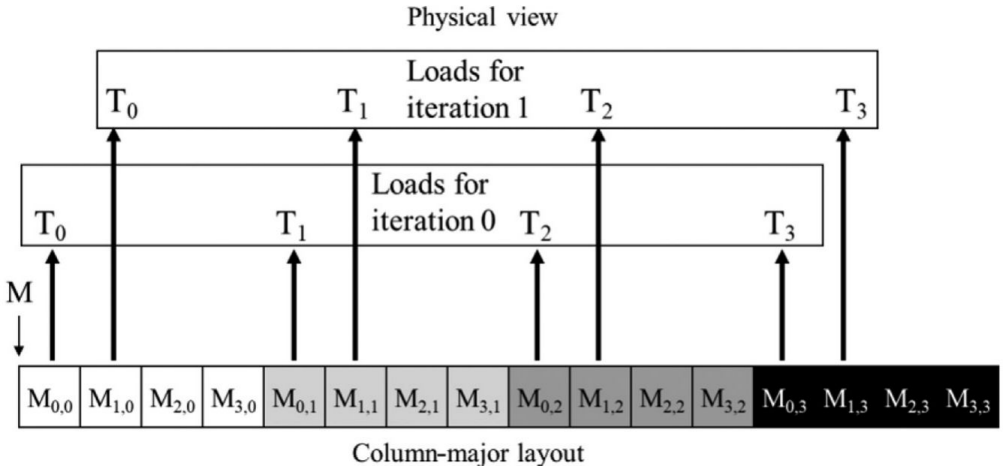
Would this code stub still exhibit memory coalescing for M if M was column-major instead of row-major?

No

Code

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
if (row < Width && col < Width) {
    float Pvalue = 0.0f;
    for(unsigned int k = 0; k < Width; ++k) {
        Pvalue += N[row*Width + k]*M[col*Width + k];
    }
    P[row*Width + col] = Pvalue;
}
```

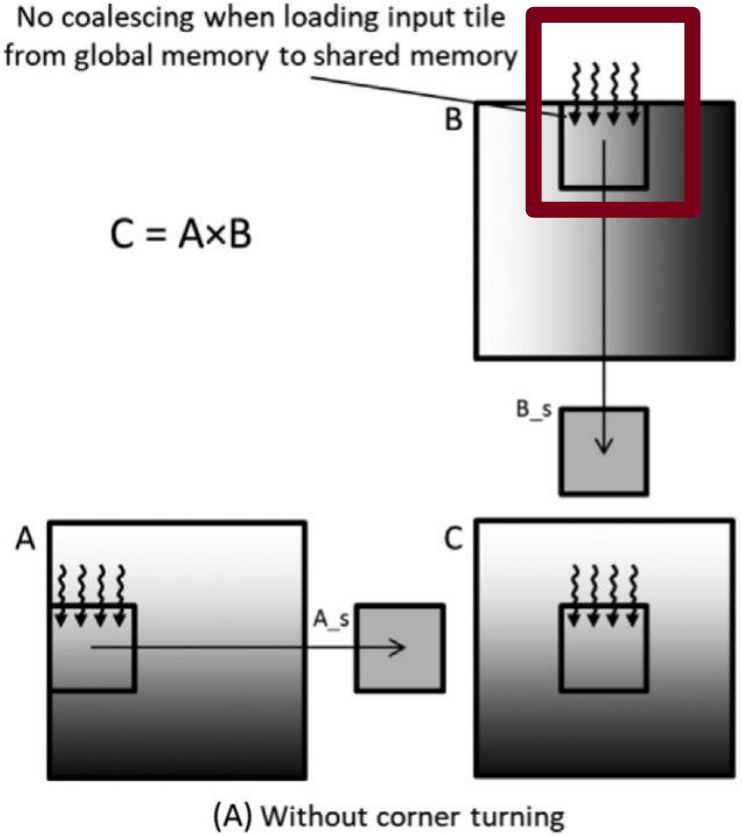




Coalescing Strategies

- Two strategies for coalescing
  - Change thread mapping so threads access continuous data
  - Change data layout
- Corner turning is an example of this → Switch orientation of threadblock caching to match data layout
  - Example → A row-major, B column major

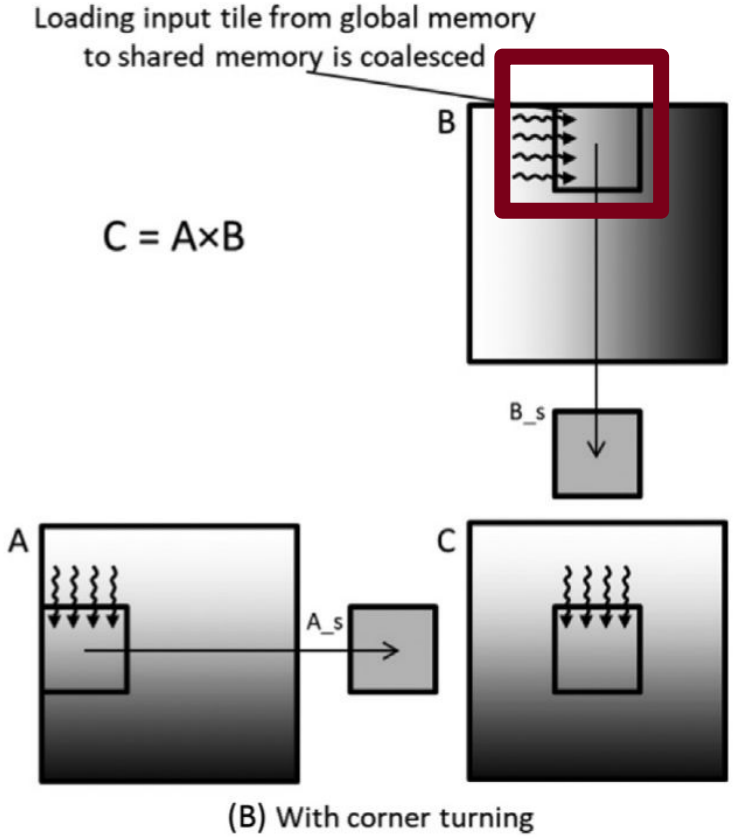
This loading strategy does not coalesce B accesses



Coalescing Strategies

- Two strategies for coalescing
  - Change thread mapping so threads access continuous data
  - Change data layout
- Corner turning is an example of this → Switch orientation of threadblock caching to match data layout
  - Example → A row-major, B column major

But this one does



Coalescing Overview

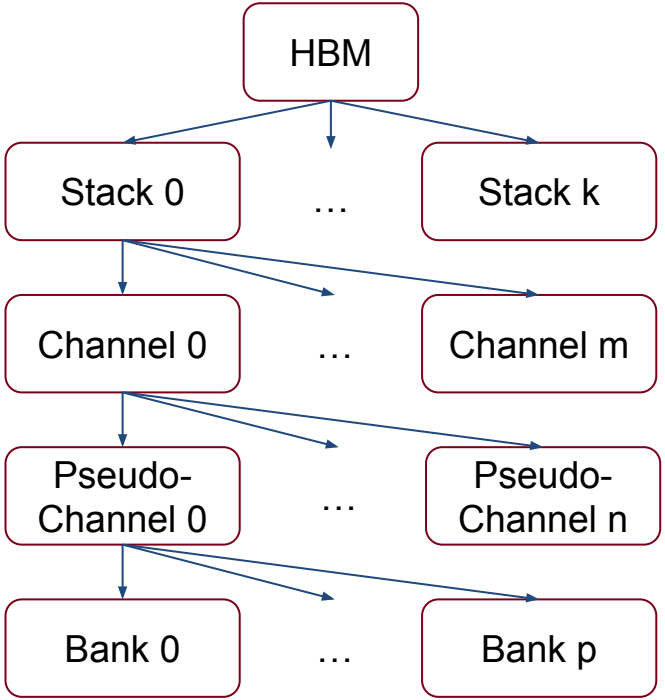
Combine multiple accesses to memory which will result in accessing the same continuous region of values from global memory

Overview

- **HBM Optimizations**
  - Coalescing
  - **HBM in Depth**
  - Request Pipeline
  - Coalescing in Practice
- Thread Coarsening
- General Notes on Optimization

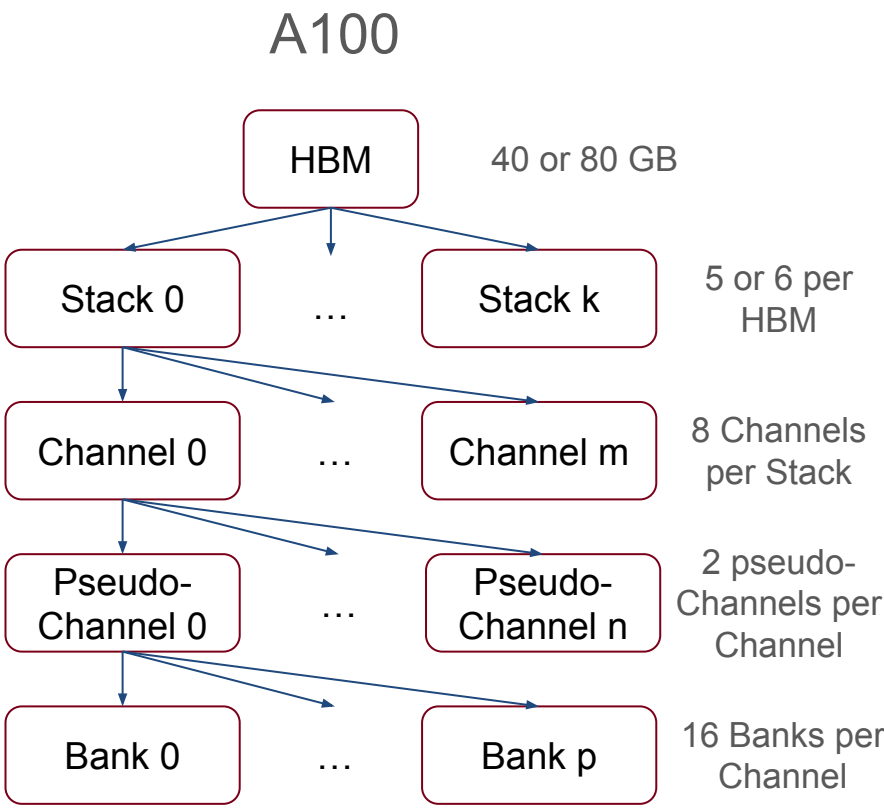
HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (**A100**, H100, B100)
  - Stack
  - Channels
  - Pseudo-Channels
  - Banks



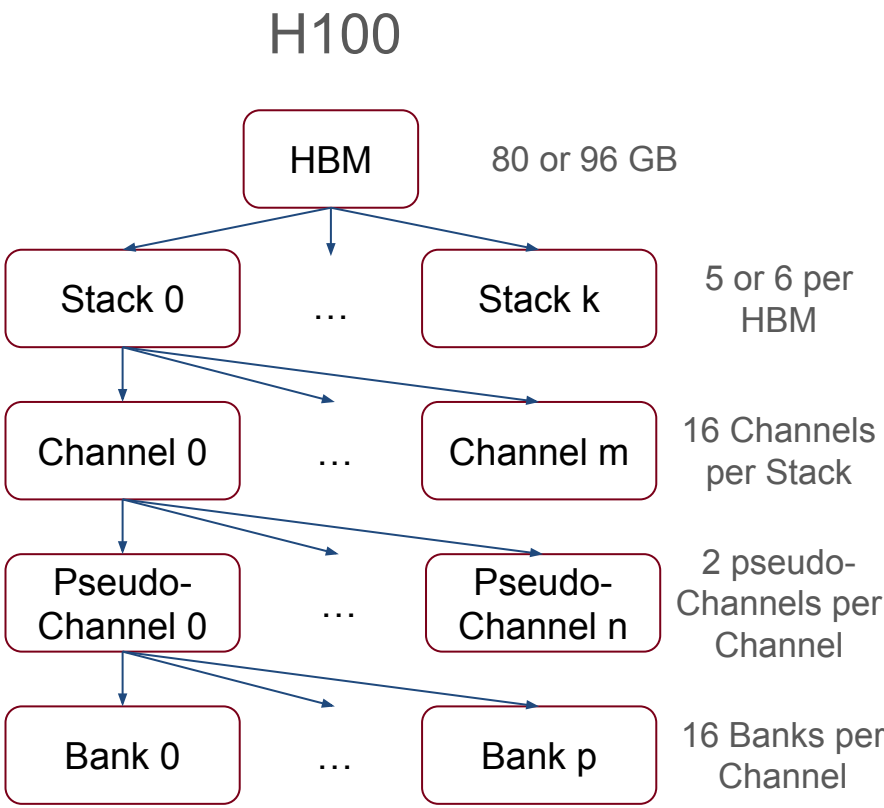
HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (**A100**, H100, B100)
  - Stack
  - Channels
  - Pseudo-Channels
  - Banks



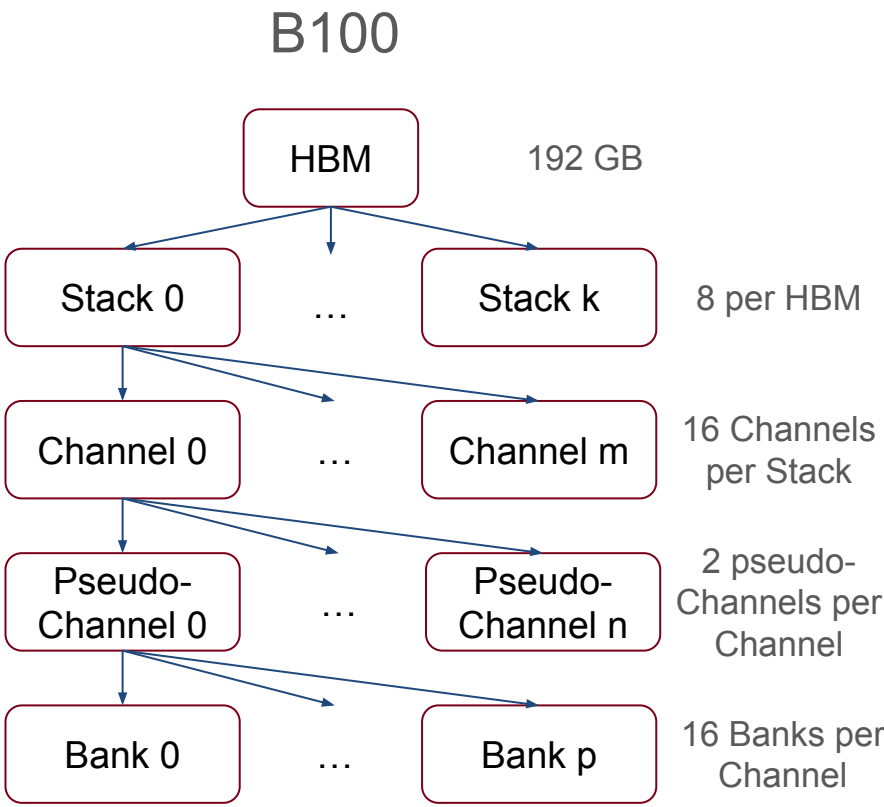
HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, **H100**, B100)
  - Stack
  - Channels
  - Pseudo-Channels
  - Banks



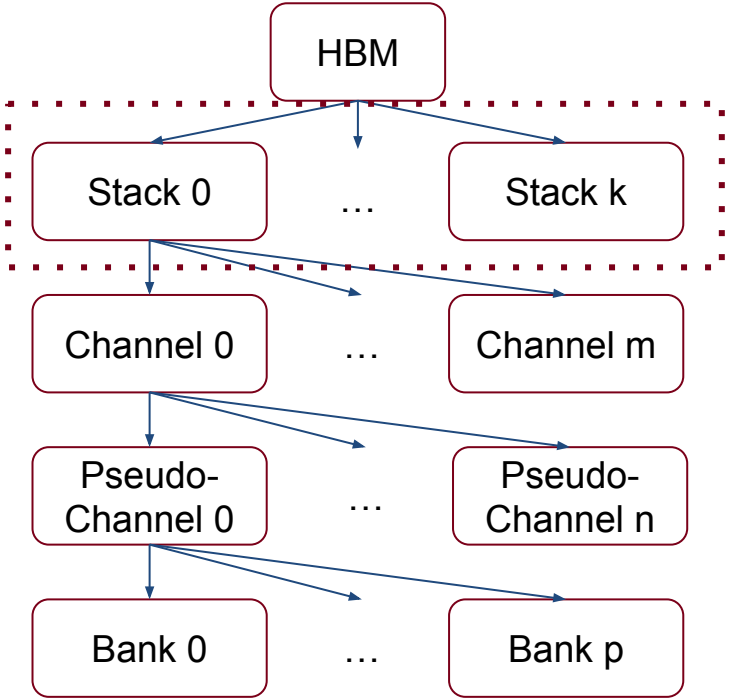
HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, H100, **B100**)
  - Stack
  - Channels
  - Pseudo-Channels
  - Banks



HBM in Greater Depth

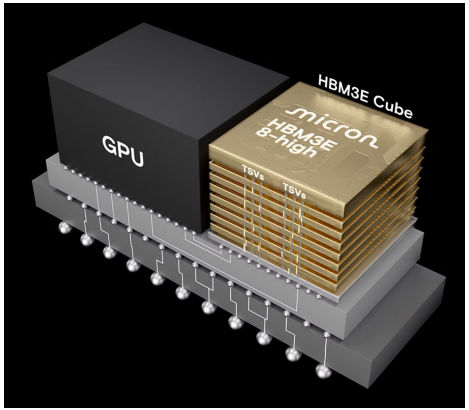
- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, H100, B100)
  - **Stack**
  - Channels
  - Pseudo-Channels
  - Banks



HBM in Greater Depth

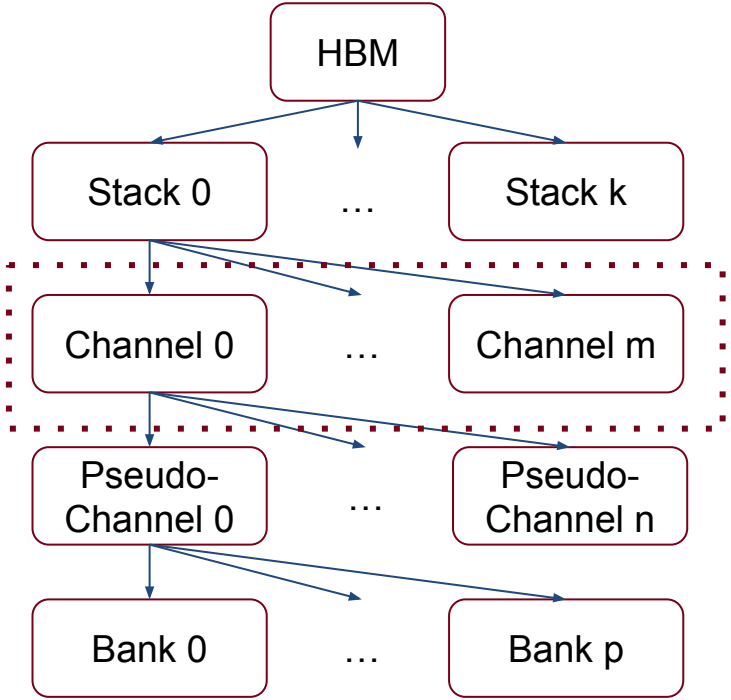
- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, H100, B100)
  - **Stack**
  - Channels
  - Pseudo-Channels
  - Banks

Multiple stacks make up an HBM → They can handle concurrent requests from different SMs.



HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, H100, B100)
  - Stack
  - **Channels**
  - Pseudo-Channels
  - Banks



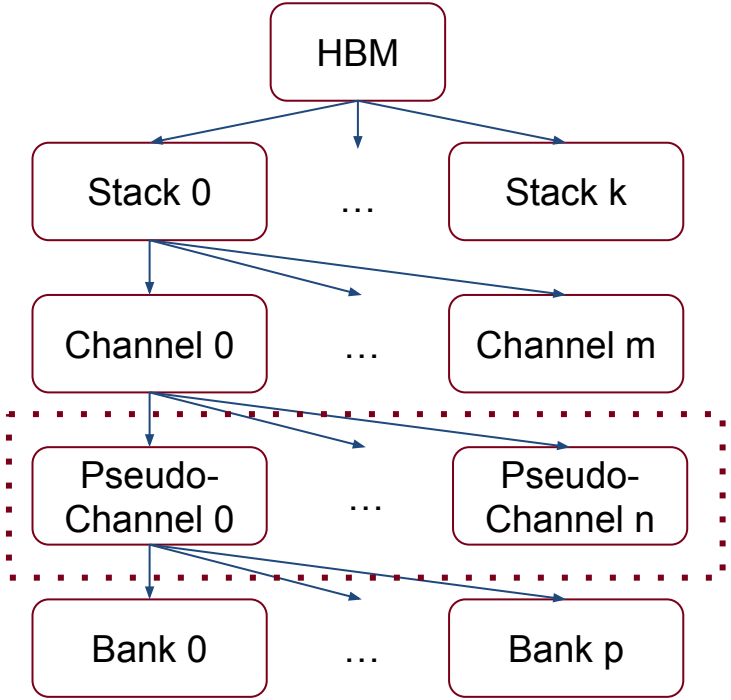
HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, H100, B100)
  - Stack
  - **Channels**
  - Pseudo-Channels
  - Banks

- Each Channel in a stack is independent
  - Requests to Channel A are independent of those to Channel B
  - Channels direct requests to appropriate memory banks (discussed in next slides)
  - Each Channel can return only 1 request at a time back to the GPU once retrieved from memory
  - Each Channel can have many concurrent requests to different memory banks it is responsible for
- Each individual channel has a memory controller
  - Queues up requests, can combine requests to same region if they are in the queue at the same time

HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, H100, B100)
  - Stack
  - Channels
  - **Pseudo-Channels**
  - Banks



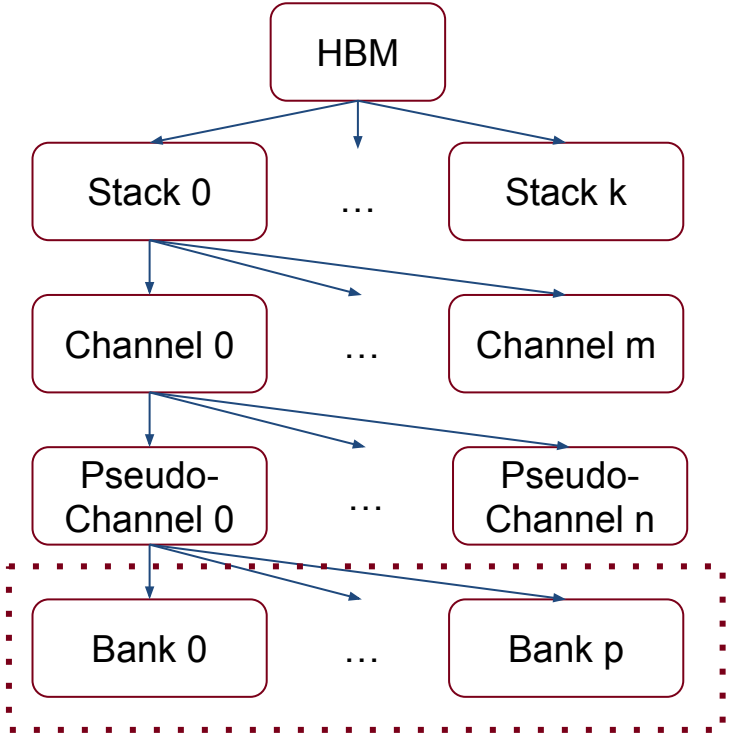
HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, H100, B100)
  - Stack
  - Channels
  - **Pseudo-Channels**
  - Banks

Always 2 per Channel. Helpful for interleaving requests

HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, H100, B100)
  - Stack
  - Channels
  - Pseudo-Channels
  - **Banks**



HBM in Greater Depth

- In general, modern HBM allows us to achieve much more parallelism in memory access
- The maximum parallelism of these data accesses dictates the bandwidth
- Hierarchy of modern HBM (A100, H100, B100)
  - Stack
  - Channels
  - Pseudo-Channels
  - **Banks**

- Banks are where the memory actually resides
- Retrieving from a bank in a single burst is serial - a given bank can handle one and only one request at a time
- It must completely finish this request before handling a new one (no pipelining here)

Overview

- HBM Optimizations
  - Coalescing
  - HBM in Depth
  - **Request Pipeline**
  - Coalescing in Practice
- Thread Coarsening
- General Notes on Optimization

Request Pipeline

A request starts from a thread in an SM

Request Pipeline

A request starts from a thread in an SM

Request Pipeline

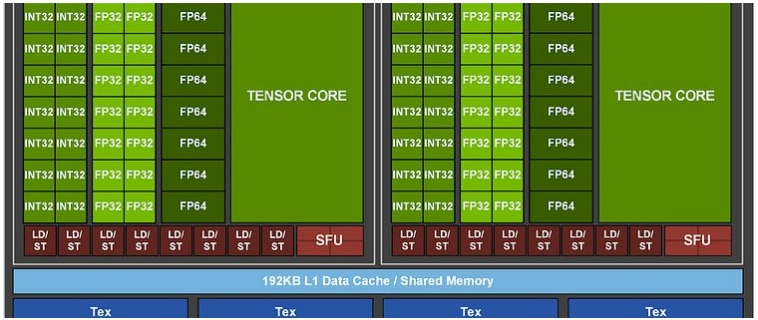
A request starts from a thread in an SM

1. The hardware checks to see if the request can be coalesced

Request Pipeline

A request starts from a thread in a Subpartition

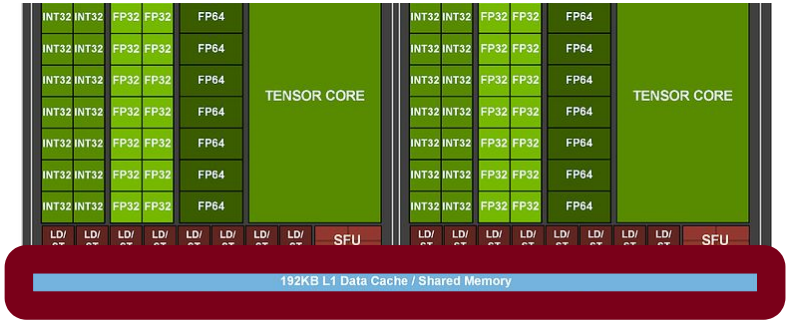
1. The hardware checks to see if the request can be coalesced
2. L1 cache is checked



Request Pipeline

A request starts from a thread in a Subpartition

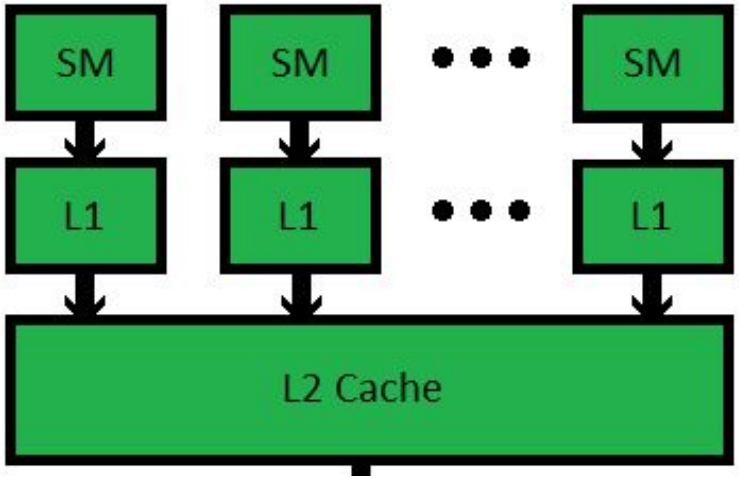
1. The hardware checks to see if the request can be coalesced
2. L1 cache is checked



Request Pipeline

A request starts from a thread in an SM

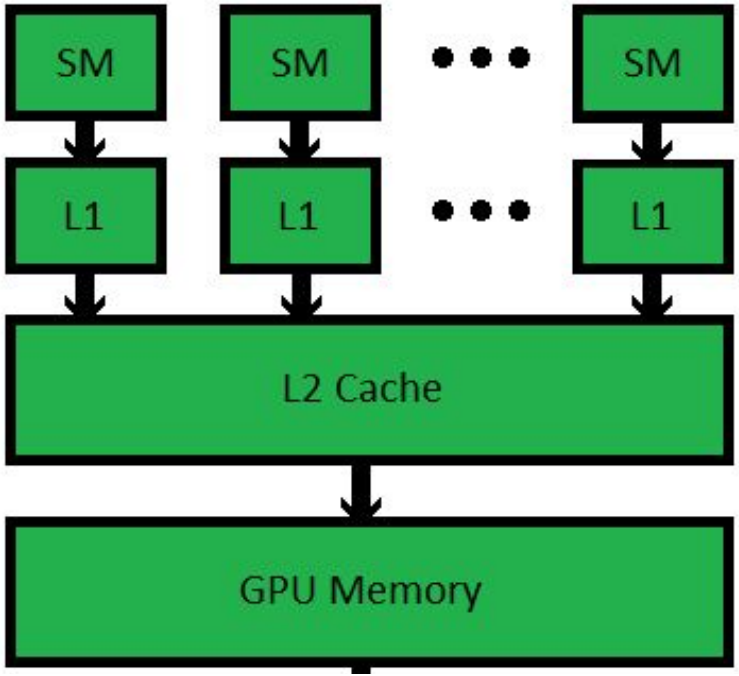
1. The hardware checks to see if the request can be coalesced
2. L1 cache is checked
3. L2 cache is checked



Request Pipeline

A request starts from a thread in an SM

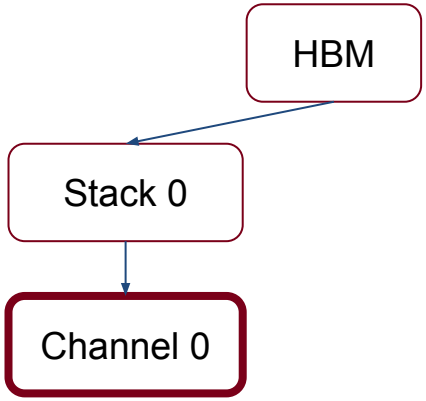
1. The hardware checks to see if the request can be coalesced
2. L1 cache is checked
3. L2 cache is checked
4. The Request reaches a memory controller (associated with one & only one channel)



Request Pipeline

A request starts from a thread in an SM

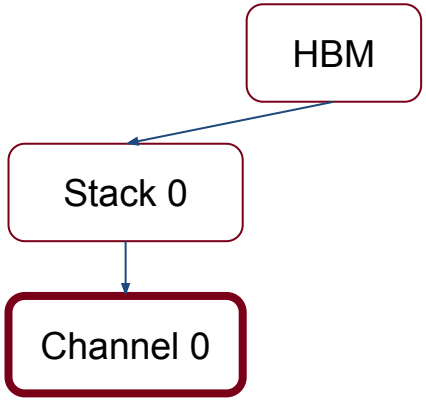
1. The hardware checks to see if the request can be coalesced
2. L1 cache is checked
3. L2 cache is checked
4. The Request reaches a memory controller (associated with one & only one channel)
5. The request traverses over to the HBM



Request Pipeline

A request starts from a thread in an SM

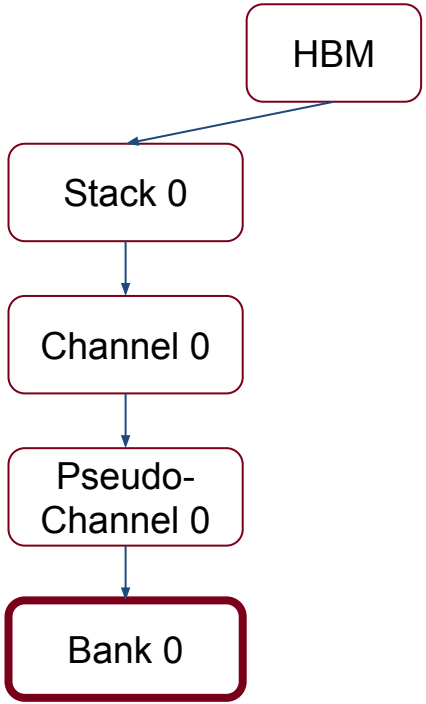
1. The hardware checks to see if the request can be coalesced
2. L1 cache is checked
3. L2 cache is checked
4. The Request reaches a memory controller (associated with one & only one channel)
5. The request traverses over to the HBM
6. The request is queued until ready



Request Pipeline

A request starts from a thread in an SM

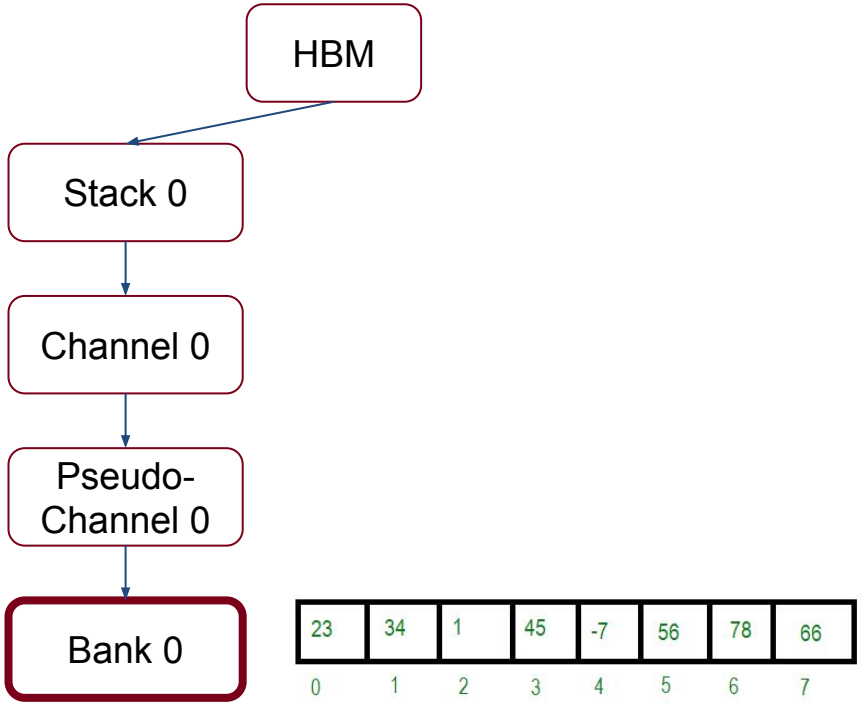
1. The hardware checks to see if the request can be coalesced
2. L1 cache is checked
3. L2 cache is checked
4. The Request reaches a memory controller (associated with one & only one channel)
5. The request traverses over to the HBM
6. The request is queued until ready
7. The request is forwarded through the pseudo-channel, then to the appropriate bank



Request Pipeline

A request starts from a thread in an SM

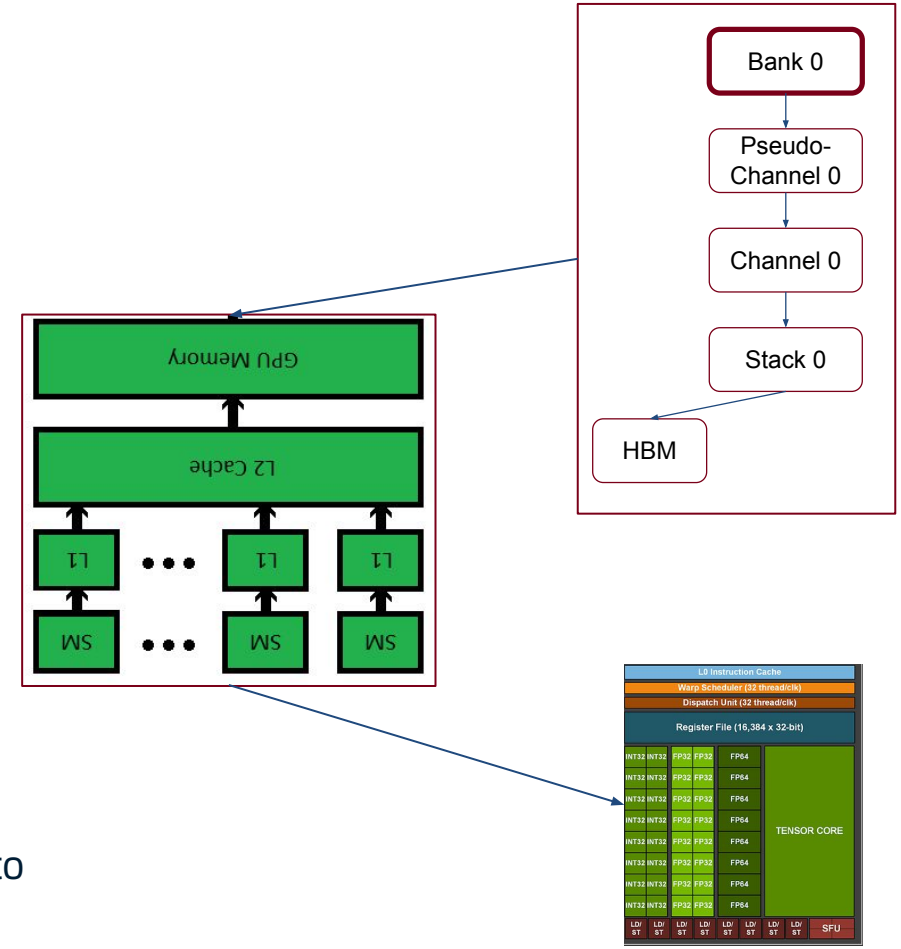
1. The hardware checks to see if the request can be coalesced
2. L1 cache is checked
3. L2 cache is checked
4. The Request reaches a memory controller (associated with one & only one channel)
5. The request traverses over to the HBM
6. The request is queued until ready
7. The request is forwarded through the pseudo-channel, then to the appropriate bank
8. Burst of data is retrieved



Request Pipeline

A request starts from a thread in an SM

1. The hardware checks to see if the request can be coalesced
2. L1 cache is checked
3. L2 cache is checked
4. The Request reaches a memory controller (associated with one & only one channel)
5. The request traverses over to the HBM
6. The request is queued until ready
7. The request is forwarded through the pseudo-channel, then to the appropriate bank
8. Burst of data is retrieved
9. Steps 2-8 occur in reverse order to return to the original thread of execution



Calculating Bandwidth

- How can we estimate the bandwidth?
  - Each channel is capable of moving 128 bits twice for every clock cycle (this is called Double Data Rate - DDR)
  - Channels can only return one block of 128 bits at a time
  - Compute bandwidth by multiplying the total number of channels by the amount a channel moves in a given clock cycle, then by the clock rate
- Examples (A100/H100)

- A100 Specs
  - 6 stacks
  - 8 channels/stack
  - 128-bit per channel
  - 1215 MHz clock
  - Bandwidth = 6 * 8 * (128/8) * 1215e6 * 2 = 1.87TB/s
- H100 Specs
  - 6 stacks
  - 16 channels/stack
  - 128-bit per channel
  - 1.4+ GHz clock
  - Bandwidth = 6 * 16 * (128/8) * 1.4e9 * 2 = 4.3TB/s

Calculating Bandwidth

- How can we estimate the bandwidth?
  - Each channel is capable of moving 128 bits twice for every clock cycle (this is called Double Data Rate - DDR)
  - Channels can only return one block of 128 bits at a time
  - Compute bandwidth by multiplying the total number of channels by the amount a channel moves in a given clock cycle, then by the clock rate
- Examples (A100/H100)

- A100 Specs
  - 6 stacks
  - 8 channels/stack
  - 128-bit per channel
  - 1215 MHz clock
  - Bandwidth = 6 * 8 * (128/8) * 1215e6 * 2 = 1.87TB/s
- H100 Specs
  - 6 stacks
  - 16 channels/stack
  - 128-bit per channel
  - 1.4+ GHz clock
  - Bandwidth = 6 * 16 * (128/8) * 1.4e9 * 2 = 4.3TB/s

There are other constraints, this is only an approximation

Calculating Concurrency

- What is the maximum number of requests which can all be getting handled at the same time (not queued, but actually in the process of being retrieved by HBM)?
  -

Calculating Concurrency

- What is the maximum number of requests which can all be getting handled at the same time (not queued, but actually in the process of being retrieved by HBM)?
  - The total number of banks in HBM
  - # stacks * # channels/stack * #pseudo-channels/channel * #banks /pseudo-channel
- A100/H100/B100 Examples

A100

5 stacks × 8 channels/stack × 2 × 16 = 1280 independent bank units

H100

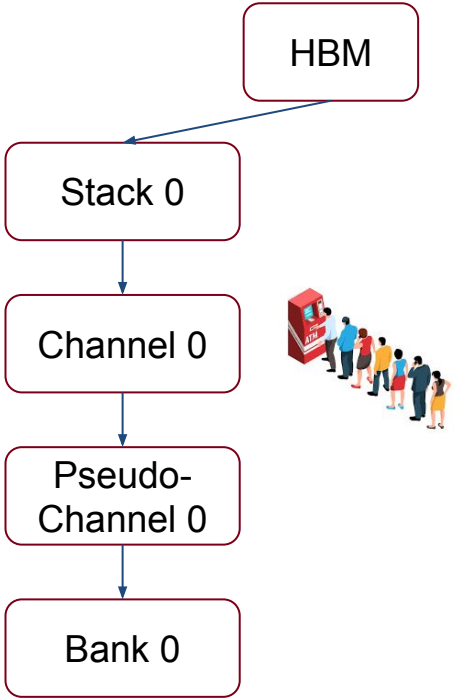
5 stacks × 16 channels/stack × 2 × 16 = 2560 independent bank units

B100

8 stacks × 16 channels/stack × 2 × 16 = 4096 independent bank units

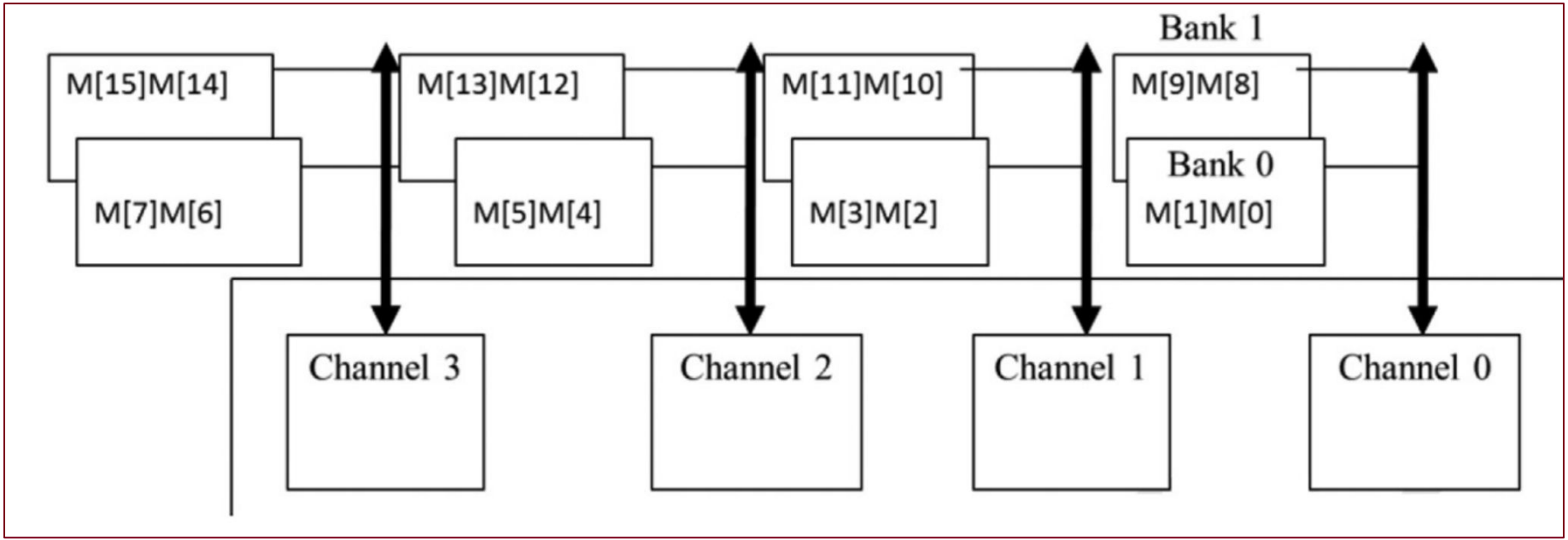
Intelligent Memory Layout

- Suppose the HBM laid out data in memory such that all continuous regions fill up a single bank before going to the next bank (e.g. I put a 1000x1000 matrix of floats entirely on one HBM bank).
- What is the problem with this approach?
  - Too many bank conflicts
  - Many threads will attempt to read from same bank → only one can at a time
  - Large queue at corresponding channel



Intelligent Memory Layout

- Solution → HBM memory is interleaved
  - This is handled by the hardware
- Assume only 1 stack and 1 pseudochannel/channel for simplicity



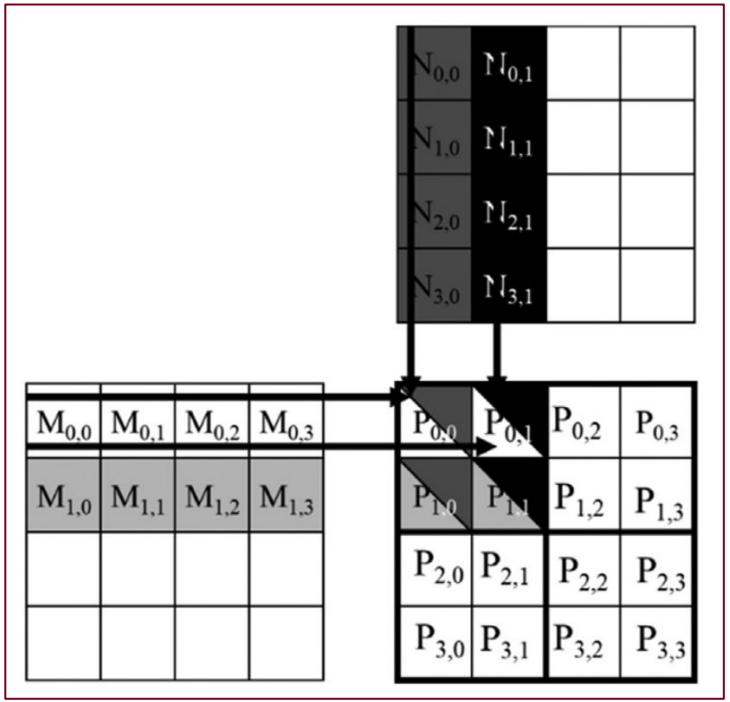
Overview

- **HBM Optimizations**
  - Coalescing
  - HBM in Depth
  - Request Pipeline
  - **Coalescing in Practice**
- Thread Coarsening
- General Notes on Optimization

Coalescing in Tiled Matmul

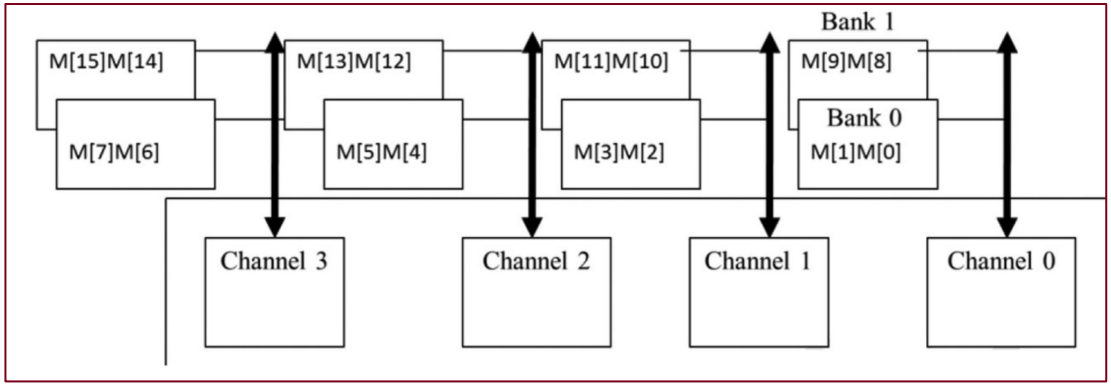
```
02    __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07        int bx = blockIdx.x;  int by = blockIdx.y;
08        int tx = threadIdx.x; int ty = threadIdx.y;
09
10        // Identify the row and column of the P element to work on
11        int Row = by * TILE_WIDTH + ty;
12        int Col = bx * TILE_WIDTH + tx;
13
14        // Loop over the M and N tiles required to compute P element
15        float Pvalue = 0;
16        for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18            // Collaborative loading of M and N tiles into shared memory
19            Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20            Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21            __syncthreads();
22
23            for (int k = 0; k < TILE_WIDTH; ++k) {
24                Pvalue += Mds[ty][k] * Nds[k][tx];
25            }
26            __syncthreads();
27
28        }
29        P[Row*Width + Col] = Pvalue;
30
31    }
```

Tiled MatMul Kernel from previous lecture



Coalescing in Tiled Matmul

This layout of data minimizes contention across threads in different blocks



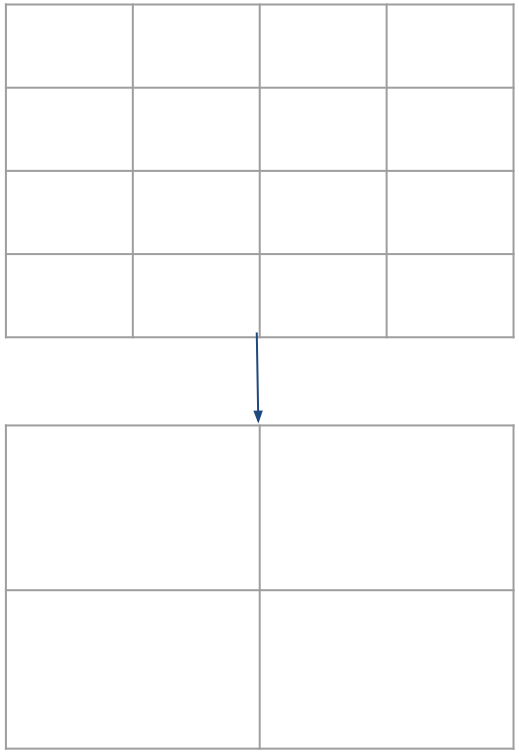
Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0 (2D index)	M[0][0],M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase 0 (linearized index)	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[19], M[12], M[13]	M[8], M[9], M[12], M[13]

Overview

- HBM Optimizations
  - Coalescing
  - HBM in Depth
  - Request Pipeline
  - Coalescing in Practice
- **Thread Coarsening**
- General Notes on Optimization

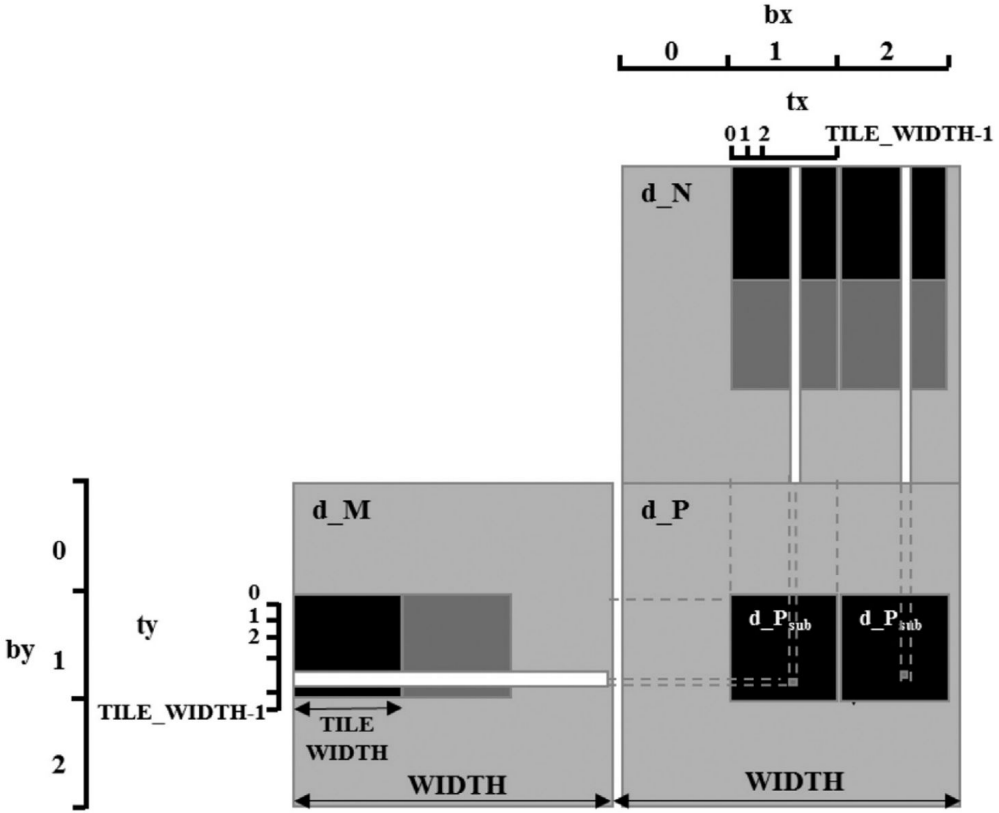
Granularity

- In the code examples we have examined so far (vector addition, color conversion, blurring, matrix multiplication), each thread is responsible for one entry in the output
- We can choose to update this such that each thread has more than one output location to write to
- This is called *thread coarsening*
- Useful when there are some redundant loads, excess synchronization, and redundant work



Coarse Matrix Multiplication

- We can apply coarsening to the tiled matrix multiplication algorithm from last lecture
- 1-d coarsening results with a Coarsening factor of 2 results in each thread computing two different output locations in the output matrix



Thread Coarsened Implementation

```
#define TILE_WIDTH    32
#define COARSE_FACTOR 4
__global__ void matrixMulKernel(float* M, float* N, float* P, int width)


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

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the P element to work on
    int row = by*TILE_WIDTH + ty;
    int colStart = bx*TILE_WIDTH*COARSE_FACTOR + tx;

    // Initialize Pvalue for all output elements
    float Pvalue[COARSE_FACTOR];
    for(int c = 0; c < COARSE_FACTOR; ++c) {
        Pvalue[c] = 0.0f;
    }
```

```
    // Loop over the M and N tiles required to compute P element
    for(int ph = 0; ph < width/TILE_WIDTH; ++ph) {

        // Collaborative loading of M tile into shared memory
        Mds[ty][tx] = M[row*width + ph*TILE_WIDTH + tx];

        for(int c = 0; c < COARSE_FACTOR; ++c) {

            int col = colStart + c*TILE_WIDTH;

            // Collaborative loading of N tile into shared memory
            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();

        }

    }

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

}
```

When to Use Thread Coarsening

- Only apply when finest granularity results in excess costs
  - If there are redundant loads, redundant work, synchronization overhead which can be minimized with coarsening, then do so
- Don't apply such that resources are underutilized
  - Maximum coarsening is just single-threaded execution → we still want to use as many cores as possible
- Avoid excess occupancy resources
  - Thread coarsening often increases register/memory occupancy → Too much coarsening can overuse these resources

Overview

- HBM Optimizations
  - Coalescing
  - HBM in Depth
  - Request Pipeline
  - Coalescing in Practice
- Thread Coarsening
- **General Notes on Optimization**

Specific Performance Optimizations

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread

Specific Performance Optimizations

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/ cache lines	Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data

Specific Performance Optimizations

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Minimizing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	–	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data

Specific Performance Optimizations

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once

Specific Performance Optimizations

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily

General Performance Considerations

- When programming with CUDA, the goal is to utilize your resources to compute as much as possible
- To do this successfully, you need to be aware of the bottlenecks in your program (memory, compute, latency)
- Good kernels make efficient use of each resource to compute at as close to the maximum FLOPs for the target hardware

- Resources
  - Registers
  - SRAM/SMEM/l2 cache/
  - L1 Cache
  - Cores (Tensor/etc.)
  - Bandwidth