CSE 160

Taanish Reja

Winter 2025

1 Basics of Parallelism

1.1 CPU vs GPU design

1.1.1 CPU

Latency oriented design

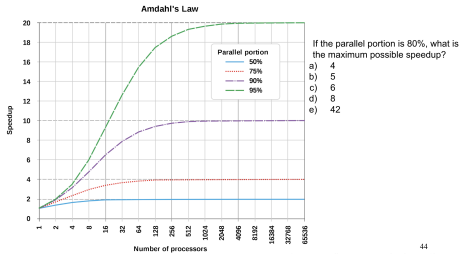
- High clock frequency
- Large Caches (reduces memory latency)
- Complex control pathways (CPUs reduce branch latency via branch prediction (predicting what a conditional statement will evaluate to))
- Have sophisticated ALUs

1.1.2 GPU

- Moderate clock frequency
- Small caches
- Simple control (no branch prediction and no data forwarding (data forwarding -¿ instead of CPUs finishing an instruction, writing its result to a register and having later instructions read the register, CPUs can substitute the data directly into the second instruction; writing to a register is a long process and the second instruction may execute before the write-back, so data forwarding prevents such stalls))
- Small, efficient ALUs (long latency but optimized for throughput)

1.2 Amdahl's Law

Speedup $= \frac{1}{(1-p)+\frac{p}{s}}$
Speedup: theoretical speedup of whole task
s: speedup of the part of the original task that is being sped up
p: proportion of the part of original task that is being sped up



Example
Answer
$\lim_{s\to\infty} \frac{1}{(1-0.8)+\frac{0.8}{s}} = 5$

1.3 Execution Model

1.3.1 Kernel

- OpenCL executes an instance of a kernel for each point in the domain
- Kernels execute as a set of parallel **work-items**
  - Work-items may not execute at the same rate

1.3.2 ND-Range

- **Global Size**: Total number of work-items (threads) executing the kernel.
- **Local Size**: Number of work-items grouped into work-groups (blocks in CUDA).
- **Dimensionality**: It can be 1D, 2D, or 3D
- **Example**: ND-range dimensions (12,6) = 72 work-items per ND-range
- **Example**: Work-group Dimensions 4 x 3 = 12 work-items/work-group

1.3.3 Global ID vs. Local ID

- **Number of Work-groups**: $(W_x, W_y) = ceil(\frac{G_x}{S_x}), ceil(\frac{G_y}{S_y})$
- **Work-group ID**: $(w_x, w_y) = (\frac{g_x - s_x}{S_x}, \frac{g_y - s_y}{S_y})$

1.4 General

- Synchronization between **work-items** possible only within **work-groups**: **barriers** and **memory fences**
- Cannot synchronize between work-groups within a kernel

- Kernels execute as a set of work-items
- Threads are grouped into work-groups

1.4.1 ND-range

  - Blocks are grouped into grids
  - Each kernel launch creates a eparate grid

- work-item $\in$ work-group $\in$ nd-range

2 Memory Model

2.1 Host vs. Device

- Host: CPU and associated memory space (configures the kernel)
- Device: the device that runs the kernel (GPU and associated memory space; can also be the CPU or another CPU)

2.2 Memory Hierarchies

- 1. Private memory (per work item)
- 2. Local memory (shared within a work-group)
- 3. Global memory/constant memory (visible to all work-groups)
- 4. Host memory (On the CPU)
  NOTE: Memory management is EXPLICIT. You are responsible for moving memory (host-¿global-¿local then bacK) and freeing it. Memory CANNOT be dynamically allocated in the kernel, but you can use stack-allocated arrays (__**local** defines local memory, no modifier is private)

Example Assuming that: Transferring data between CPU and GPU memory takes 1 ms per transaction, regardless of data size, Performing vectorScale using one GPU work-item requires 5 ms, and vectorScale is perfectly parallelizable, i.e., 2 work-items takes 2.5 ms, 4 work-items 1.25 ms, etc. Changing the number of work-items from 1 to 100 will result in an overall speed up of?

2.3 OpenCL Pipeline

- Get Platform and Device
- Create Context and Command Queue
- Compile and Create Kernel
- Create Input and Outpu Device Buffers
- Write Input Data to Input Device Buffers
- Setup Kernel Arguments
- Setup NDRange and Call the Kernel
- Read the Output Device Buffer to Host Memory
- Free the Device Buffers
- Free command queue and context

Memory transfer from CPU to GPU: 1ms
Amdahl speedup: $\frac{1}{(1-1)+\frac{\frac{1}{5}}{100}}$

- Perfectly parallelizable (proportion is 1, can parallelize every part)
- One work item requires 5 ms, so the speed up is 5/100 if perfectly parallelizeable

Memory transfer from GPU to CPU: 1ms
Sped up time: 2.05ms
Original time: 7ms **Speedup:** 3.4 times

2.4 GPU Coarsening

Making each work items do more work
Coarsening is important for reducing memory loads (less threads, less threads loading memory); increasing work per thread is called increasing **occupancy**

Example

Work-item Coarsening

```
__kernel void vectorScale(
    __global float* input,
    __global float* output,
    __global float  scale)
{
   int i = 2*get_global_id(0);
   output[i] = scale * input[i];
   output[i+1] = scale * input[i+1];
}
```

What changes to ND-Range are necessary?

a) Nothing

b) ND-range size should double

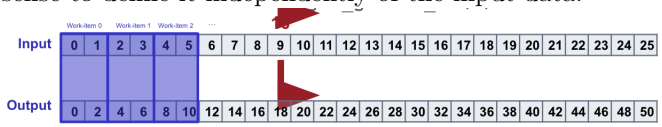
c) ND-range size should half

d) ND-range should add another dimension (i.e., 1 or y)

e) None of the above

[h]

Answer: We should halve the ND Range (it is one dimensional, so halving makes sense here). This is because each work items does two adjacent global memory loads (the next work item does the next two). Note the ND range is unrelated to the input data size (it can differ), however it doesn't usually make sense to define it independently of the input data.



[h]

3 Parallel Work-items

3.1 Kernel in ND range

- All work-items in an ND range run the same kernel code
- Single Program Multiple Data (SPMD)
- Each thread has a unique index that it uses to compute memory addresses, make control decisions, etc.

3.2 Work-groups

- Divide work-items array into multiple work-groups
- **Work-items** within a **work-group** cooperate via **local memory**, **atomic operations**, and **barrier synchronization**
- Work-items in different blocks cooperate less

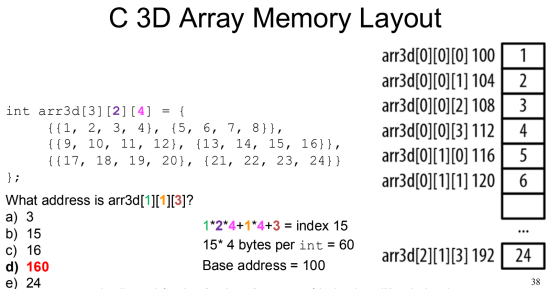
- **Example**:
  int i = get_global_id(0);

  // is logically equivalent to

  int work_group_id = floor((float) get_global_id(0)/get_local_size(0));
  int i = work_group_id*get_local_size(0) + get_local_id(0);

- **Note 1**: Some devices have restrictions on work-group (local) size and ND-range (global) sizes

- **Note 2**: Total amount of time to complete a parallel job is limited by the thread that takes the longest to finish



4 Matrix Multiplication

4.1 Simple Parallel

Matrix multiplication can be parallelized by assigning each work-item (thread) to compute a single element of the output matrix. This approach maps naturally to NDRange in OpenCL.

- We assign one work-item per output element P(i,j).

- NDRange = (width, width) → Each work-item performs dot product of a row of M and a column of N for each element of P

```
void mat_mul(float* M, float* N, float* P, int width)
{
    for (int i = 0; i < width; ++i)
        for (int j = 0; j < width; ++j) {
            float sum = 0;
            for (int k = 0; k < width; ++k) {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * width + j] = sum;
        }
}
```

4.1.1 Setup:

- Declare and initialize data
- Setup the platform and load program
- Setup buffers and write A and B matrices to the device to the device memory
- Create the kernel functor
- Run the Kernel and collect results

4.2 Execution types

- Sequential (single instruction operates on one piece of data)
- Vector sequential (single instruction operates on multiple pieces of data)
- Pipeline execution (each instruction is broken into phases; different instructions can be in different phases, which increases instruction throughput; single element per instruction)
- Vector pipeline execution (multiple elements are processed by each instruction, and each instruction is pipelined; thus, multiple instructions can be run at once in different stages)

4.3 Optimizing matrix multiplication

Problem: $N \times M = P$

4.3.1 Pre-tiling

One work item per row
Each work item computes a row of the output matrix P

- Reduces overhead caused from switching between work-items and work groups

Note: Consider making workgroups multiples of 32 threads

- Utilizes hardware warps

Code:

Matrix Multiplication: One Work-item Per Row

```
__kernel void mat_mul(__global float *P, __global float *M, __global float
*N, const int width){
          int i = get_global_id(0);
          for (int j = 0; j < width; ++j) {
            float sum = 0;
            for (int k = 0; k < width; ++k) {
              float a = M[i * width + k];
              float b = N[k * width + j];
              sum += a * b;
            }
          P[i * width + j] = sum;
          }
}
```

One Work-item Per Row + Private Memory
Aims to optimize performance by using private memory (local memory) within each work-item. It focuses on assigning one work-item per row of matrix M and optimizing memory accesses using local storage for part of the matrix.

- Index i is determined by get_global_id(0). Each work-item is responsible for one row of P and corresponding row of M.
- The array local_M[1024] is private memory (local memory), meaning that it is only accessible by the current work-item.
- The entire row is loaded into local memory once (in the first loop) and reused in the inner loops

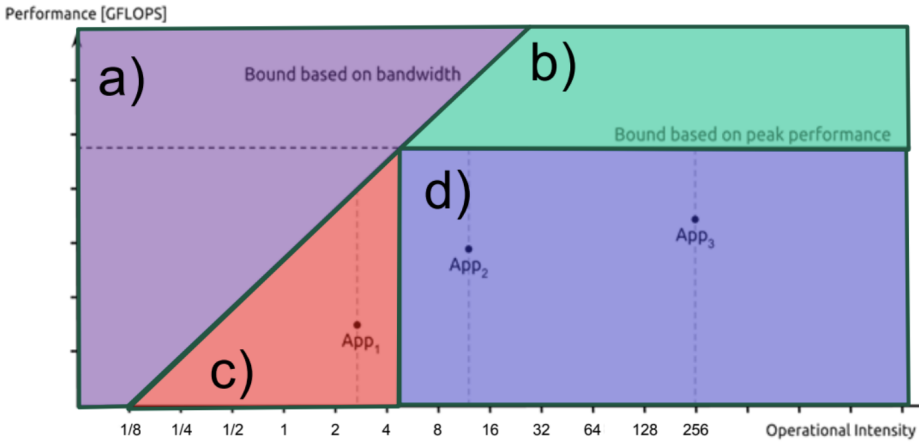
One Work-item Per Row + Private Memory

```
__kernel void mat_mul(__global float *P, __global float *M, __global float
*N, const int width){
          int i = get_global_id(0);
          int k;
          float local_M[1024]; //sizeof private variables must be known at compile time
          for (k = 0; k < width; k++) //assumes width < 1024 (poor portability!)
            local_M[k] = M[i * width + k];
          for (int j = 0; j < width; ++j) {
            float sum = 0;
            for (k = 0; k < width; ++k) {
              float a = local_M[k];
              float b = N[k * width + j];
              sum += a * b;
            }
          P[i * width + j] = sum;
          }
}
```

33

- This reduces global memory access time

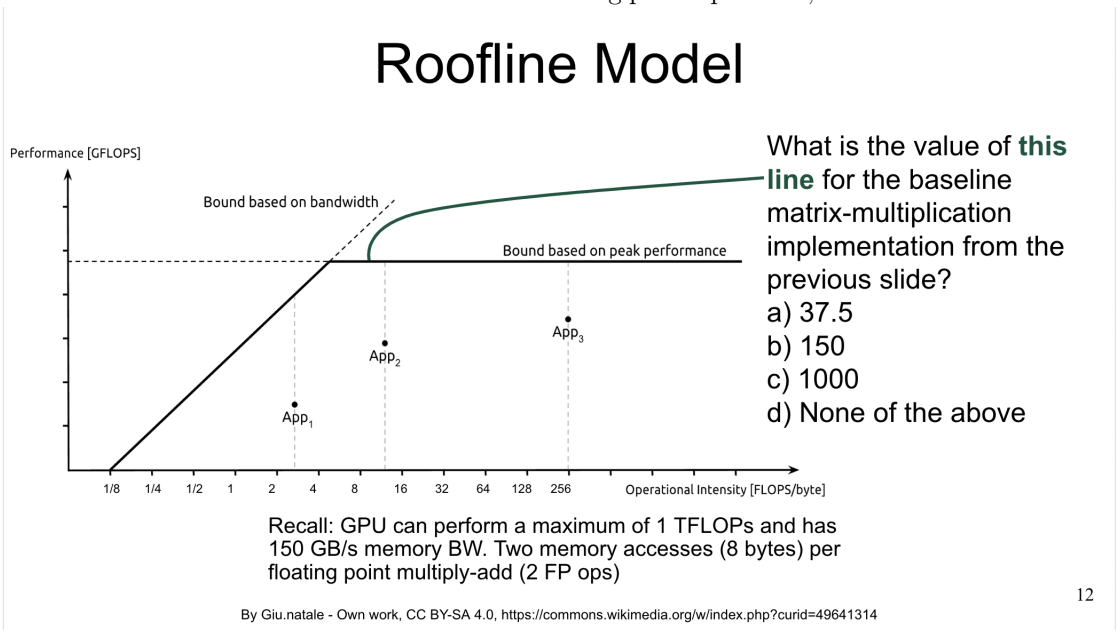
4.4 Roofline Model



- A: More bandwith than computational power
- B: More computational power than bandwith
- C: Max bandwith usage, low computational power

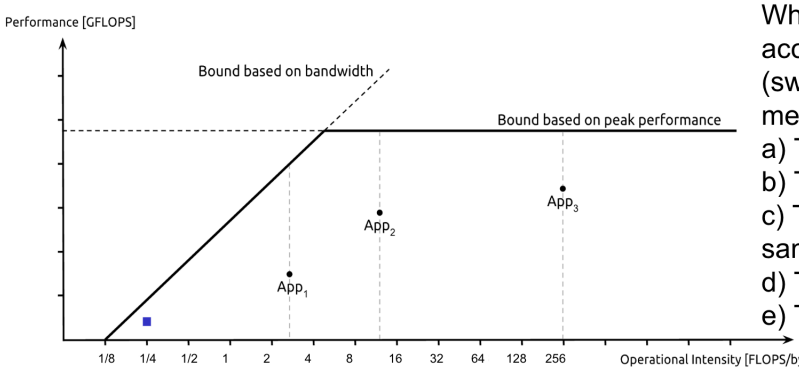
- D: More computational power than bandwithd, limited by the physical limits of the device
- Slope: Ideal performance

x-axis: Flops PER byte: This is the amount of work (flops) we can do per byte loaded and stored. This can be optimized by more work per thread **y-axis:** Flops: This is the amount of actual floating point operations we do per second. This can be optimized by loading memory faster (if we load it faster, we can do the floating point operations faster, thus increasing performance), special instructions that reduce the time we need to do a floating point operation, etc...



The limit is 1000 gflops. This is just a given hyperparameter (the gpu is limited at 1 tflops)

Roofline Model



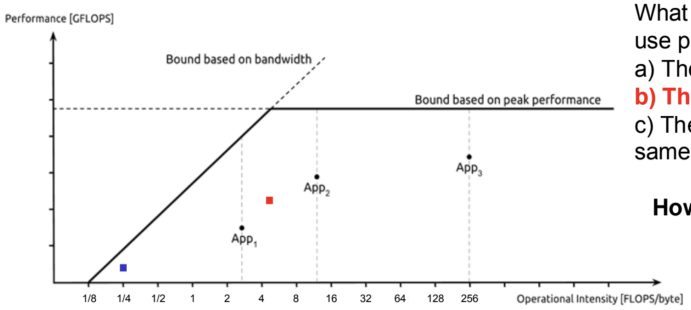
What happens if we account for memory stores (sw) in addition to the memory loads (lw)?
a) The point moves left
b) The point moves right
c) The point stays in the same position
d) The point moves up
e) The point moves down

Recall: GPU can perform a maximum of 1 TFLOPs and has 150 GB/s memory BW. Two memory accesses (8 bytes) per floating point multiply-add (2 FP ops)

15

If we consider stores in addition to loads, we deal with more bytes while doing the same amount of flops. This moves us left.

Roofline Model



What happens when we use private memory for M?
a) The point moves left
b) The point moves right
c) The point stays in the same position

How far right does the point move?

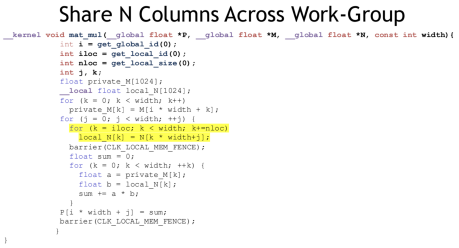
Recall: GPU can perform a maximum of 1 TFLOPs and has 150 GB/s memory BW. Two memory accesses (8 bytes) per floating point multiply-add (2 FP ops)

In this problem, we are talking about global memory loads, not private memory loads (assumed those are free). Since the matrix multiplication code which uses private memory reduces the amount of global memory loads, we are reducing bytes while maintaining flops, thus moving us right.

4.5 Private memory

- Private memory is (generally) hardware registers
- A kernel can allocate too many variables to private memory
- A compiler deals with excess variables by spilling them into global memory

4.6 Share N Columns in Local Memory and M Rows in Private Memory



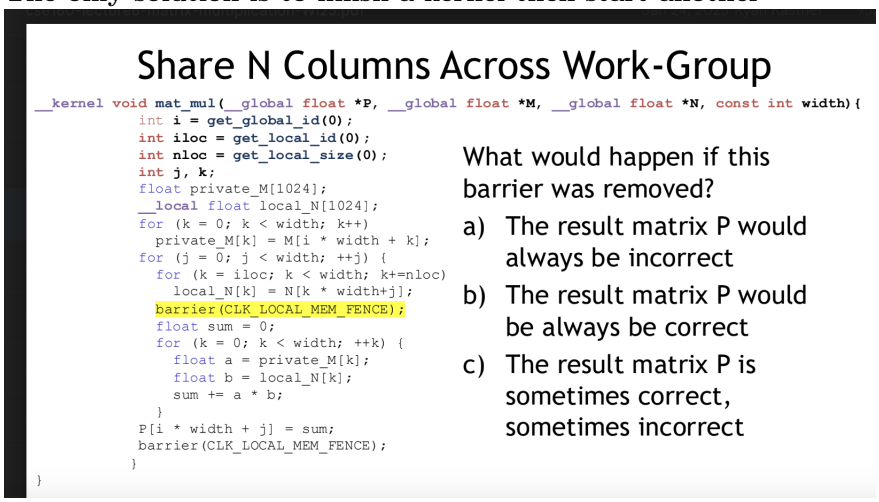
- Each work-item (thread) in a work-group loads one row of M into private memory (private_M), which is specific to that work-item.
- **private_M[1024]**: Each work-item still loads a row of M into its private memory.
- **local_N[1024]**: All work-items in a work-group share the same column of N loaded into local memory.
- By loading N's columns into local memory, all work-items in the work-group can access the data much faster than repeatedly fetching it from global memory.
- Matrix M's rows are still kept in private memory for each work-item; no work-item fetches the same data multiple times.

4.7 Memory synchronization

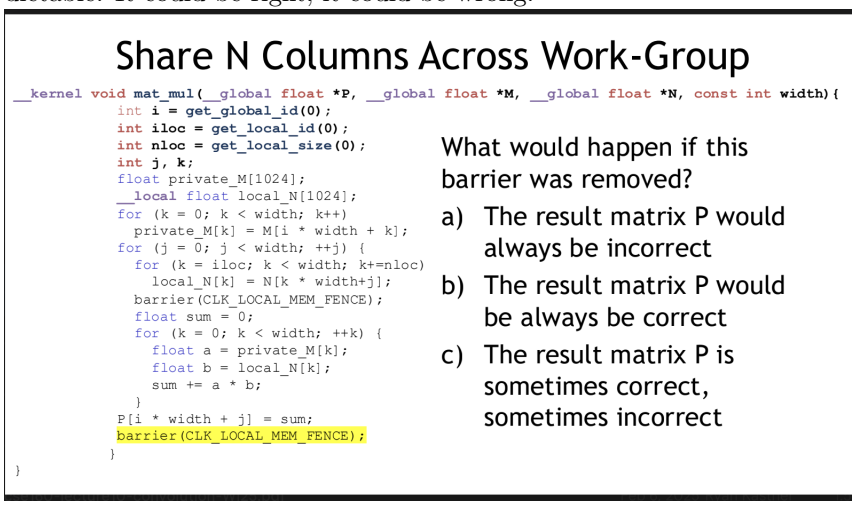
Work items within work groups can become synchronized, executing out of order. **Memory barriers** can force work items to halt execution at a certain point before proceeding until all work items in a work group have reached the barrier.

- Can make local memory fences
- Can make global memory fences

Work items between different work groups cannot be synchronized. The only solution is to finish a kernel then start another



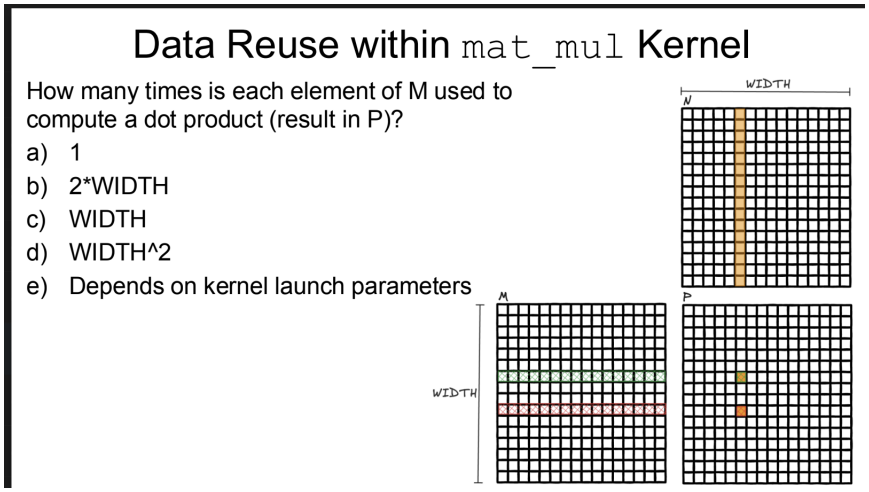
The answer is C. Removing the barrier just makes the kernels behavior unpredictable. It could be right, it could be wrong.



The answer is still C. Removing the barrier again makes the kernels behavior unpredictable. In this scenario, it is a little less obvious why. Lets say one work item is still calculating dot products while another work item has finished calculating sum. If it moves on (without the barrier), it will begin loading the next row, which could potentially corrupt the slow threads dot product calculations. However, the answer could still be right if the threads all worked at the exact same speed, which is why we can't say it will always be incorrect.

4.8 Key Optimizations for Fastest Matrix Multiplication

- Map work group sizes to a multiple of warp/wavefront size (32/64 respectively)
- Use tiling (each work group computes a tile of the output matrix; Instead of grabbing an entire row or column, we grab an input "tile" from A and B and use this to compute a partial product of the output tile. We continually grab A and B tiles until we traverse A's columns and B's rows. In this model, one work item still computes one dot product
- Produce blocked result. Instead of regular tiled matrix multiplication, where one work item computes one dot product, one work item produces a block of dot products. This allows us to optimize operational intensity



The answer is width. To compute the entire matrix P, each element of N is used width times (multiplied by width number of columns)

4.9 Minimizing global memory usage

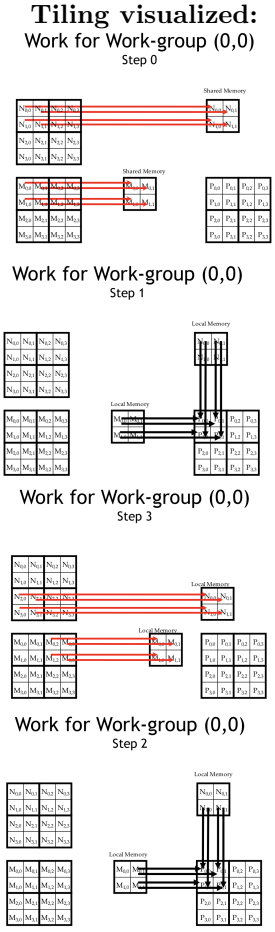
Global memory is dense but slow. We want to minimize its usage
Here is how we do so:

- Partition data into tiles
- Perform the computation on the tile from local memory so each work-item can efficiently access data. Store the result in local memory
- Copy shared memory to global memory

4.10 Tiling Phases

- Phase 1: Loading a Tile
  - Each work-item loads one M element and one N element in basic tiling code
- Phase 2: Calculate partial dot product for tile of P

Tiling visualized:



4.10.1 Tiling indexing (transferring global memory to local memory)

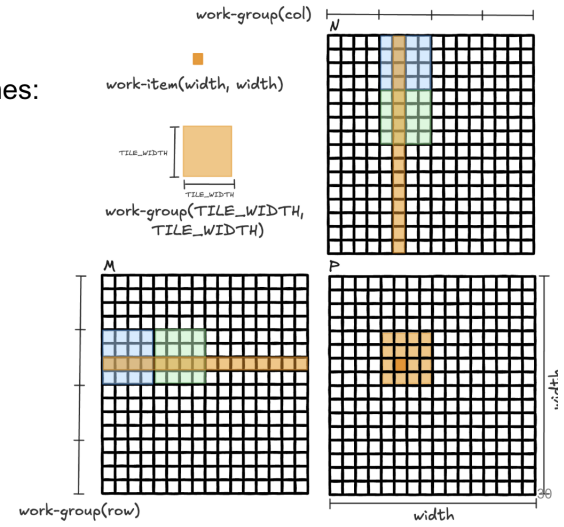
2D tiling for tile q, $M \times N = P$:
row = get_global_id(0) (note, this is inefficient; threads are organized by their 0th dimension. If we set row to your 0th dimension, we cannot coalesce memory accesses row wise)
col = get_global_id(1)

lr = get_local_id(0)
lc = get_local_id(1)
__local localM = M[row][q*TILE_WIDTH+lc]
__local localN = N[q*TILE_WIDTH+lr][col]
1D tiling for tile q:
same setup
__local localM = M[row*M_WIDTH+q*TILE_WIDTH+lc]
__local localN = N[(q*TILE_WIDTH+lr)*N_WIDTH + col

Memory Bandwidth Consumption

16x16 tiles reuse each tile element __ times:

a) 1
b) 16
c) 32
d) 256
e) WIDTH



Answer: 16 times (each row of a tile from M is used 16 times, each column of a tile from N is used 16 times)
Back to roofline model:
Recall our example GPU: 1,000 GFLOP/s, 150 GB/s mem BW, 4 bytes loaded per floating point operation
16x16 tiles reuse each operand 16 times:

- Reduce the global memory by a factor of 16
- 150GB/s bandwidth can now support (150/4)*16 = 600 GFLOPS!

32x32 tiles reuse each operand 32 times

- Reduce the global memory by a factor of 32
- 150GB/s bandwidth can now support (150/4)*32 = 1200 GFLOPS!

4.11 Local Memory Constraints

Device dependent
E.g., 64kB per Compute Unit (CU) in nVidia Maxwell (48 kB max per work-group)
Given TILE_WIDTH of 16 (256 work-items/work-group),
each work-group uses 2*256*4B = 2kB of shared memory
Limits active work-groups to 32
max. of 2048 work-items per CU
Limits work-groups to 8 (= 2048/256)
Thus, up to 8*512 = 4,096 pending loads per CU (2 per work-item, 256 work-items per work-group)

4.12 Handling Arbitrary Matrix Sizes

- Avoid accessing out-of-bounds memory

  if (valid index) normal load
  else write 0 to subtile

- Why? Summing extra 0s does not affect matrix multiplication results.

4.13 Memory Hierarchy & DRAM Optimization

- DRAM is slow due to large capacitance.
- Solution: DRAM bursting:
  - Transfers multiple bits per memory access.
  - Banking helps parallel memory accesses.

4.14 Work-item Scheduling & Thread Mapping

- Each work-group executed as 32 work- item warps
- Warps are divided based on their linearized thread index
  – Work-items 0-31: warp 0
  – Work-items 32-63: warp 1, etc.
- If 3 blocks are assigned to an CU and each work-group has 256 work-items, how many work-items are there in an CU?
  – Each work-group is divided into 256/32 = 8 work- items
  – 8 items/group * 3 groups = 24 items
- Coalescing memory accesses improves efficiency.
  - threads in a warp access contiguous memory locations

- fewer memory transactions and higher throughput.
- If adjacent threads access adjacent memory locations, fewer memory transactions are needed.

4.15 Branch Divergence

- Threads in a warp take different paths in the program
- GPUs can use **predicated execution**
  - Each thread computes a yes/no answer for each path
  - Multiple paths taken by threads in a warp are executed serially
- Try to make branch granularity a multiple of warp size (remember, it may not always be 32)

```
if ((threadIdx.x / WARP_SIZE)  2) {
// THEN path (lots of lines)
} else {
// ELSE path (lots of lines)
}
```

Block Granularities Considerations

- For colorToGreyscaleConversion, should one use 8x8, 16x16 or 32x32 blocks? Assume that in the GPU used, each SM can take up to 1,536 threads and up to 8 blocks.
  - **8x8:** 64 threads per block. A SM has a maximum of 1,536 threads (24 blocks). But each SM can only take up to 8 Blocks, only 512 threads (16 warps) will go into each SM!
  - **16x16:** 256 threads per block. A SM has max of 1,536 threads (6 blocks). All 6 blocks fit in one SM, thus we use the full thread capacity of an SM.
  - **32x32:** 1,024 threads per block. Only one block can fit into an SM, using only 2/3 of the thread capacity of an SM.

65