Lecture 7 - GP-GPU and High Performances Computing

GPU Matrix Multiplication

Previously

- Scan algorithm : inclusive and exclusive
- Dealing with large vector

Reminder on hardware

- Processor hardware
  - Max threads per SM: 2048
  - Max threads per block: 1024
  - Max warps per SM: 64

Example: If 2 blocks are assigned to an SM and each block has 1024 threads, how many warps are there?

- Each block is divided into 1024/32 = 32 warps
- There are 32 × 2 = 64 Warps

Key points:

- At any point in time, only 4 of the 64 warps will be selected for instruction fetch and execution
- One instruction is issued for 1 warp at every cycle (by design)
- 16 cycles are needed to execute 1 instruction on all threads of the block
- SM will interleave warps to optimize execution

Definition

Given two square matrices in $\mathbb{R}^{Width \times Width}$, M and N, we multiply M by N to compute a third square matrix P:

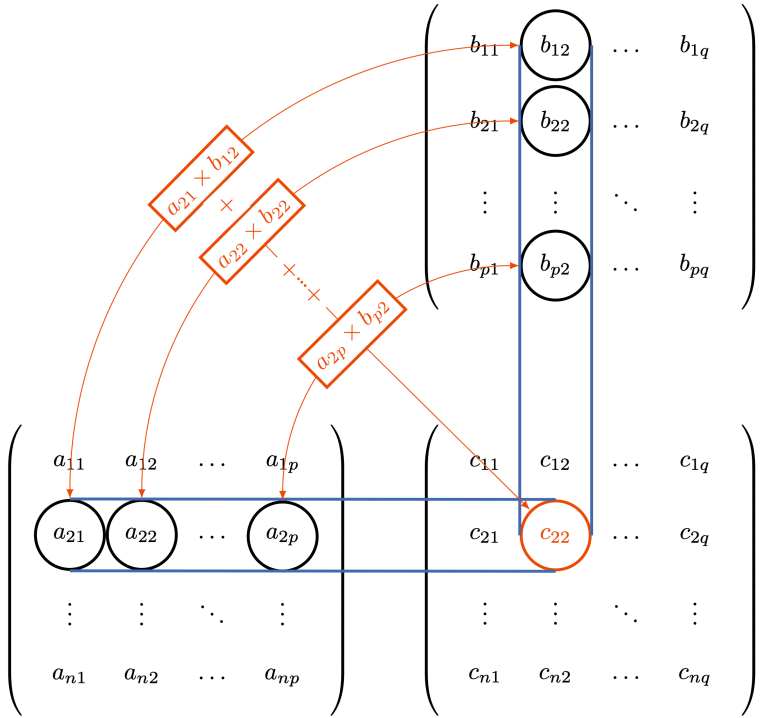
$$P = MN$$

In terms of the elements of P, matrix multiplication implies computing, for all $1 \leq i, j \leq Width$:

$$P_{ij} = \sum_{k=1}^{Width} M_{ik} N_{kj}$$

The complexity for the naïve computation is $\mathcal{O}(Width^3)$.

Sequential design



Sequential Implementation

```
void GMM_CPU(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;
    }
}
```

Define the Design Space

- **3 possible choices:** M, N or P
- The outer loops are all independent computations
- We will focus on the computation of the elements of P
- The inner loop is a scalar product between a row of M and a column of N
  - It can be parallelized using reduction

Workspace Decomposition

P is 2D, one possible choice is to organize threads in 2D as well:

- Split the output P into square tiles of size $TILE\_WIDTH \times TILE\_WIDTH$ (a preprocessor user-defined constant)
- Each thread block produces one tile of $[TILE\_WIDTH]^2$ elements
- Create $[ceil(Width/TILE\_WIDTH)]^2$ thread blocks to cover the output matrix

Example: Tile 2×2

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

Block size: each block has $2 \times 2 = 4$ threads

Number of blocks: $\frac{Width}{TILE\_WIDTH} \Rightarrow 16$ blocks

Matrix divided into 4×4 grid of 2×2 tiles

Example: Tile 4×4

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

Block size: each block has $4 \times 4 = 16$ threads

Number of blocks: $\frac{Width}{TILE\_WIDTH} \Rightarrow 4$ blocks

Matrix divided into 2×2 grid of 4×4 tiles

Invocation of Kernel

```
// TILE_WIDTH is a #define constant
dim3 dimGrid(ceil((1.0*Width)/TILE_WIDTH),
             ceil((1.0*Width)/TILE_WIDTH), 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Kernel Function

```
// Matrix multiplication kernel - per thread code
__global__ void MatrixMulKernel(float* d_M, float* d_N,
                                float* d_P, int Width) {

  // Pvalue is used to store the element of the matrix
  // that is computed by the thread
  float Pvalue = 0;
```

Simple Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N,
                                float* d_P, int Width) {
  // Calculate the row index of the d_P element and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
      Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
    d_P[Row*Width+Col] = Pvalue;
  }
}
```

Memory Bandwidth is Overloaded

That’s a simple implementation:

- GPU kernel is the CPU code with the outer loops replaced with per-thread index calculations!

Unfortunately, performance is quite bad. Why?

With the given approach:

- Global memory bandwidth
- Can’t supply enough data
- To keep the SMs busy!

Each Thread Requires 4B of Data per FLOP

- Each thread accesses global memory for elements of M and N:
  - 4B each, or 8B per pair
  - (And once TOTAL to P per thread—ignore it)
- With each pair of elements, a thread does a single multiply-add
  - 2 FLOP—floating-point operations
- So for every FLOP, a thread needs 4B from memory: **4B / FLOP**

Example Card

One generation of GPUs: 1,000 GFLOP/s of compute power, and 150 GB/s of memory bandwidth.

Dividing bandwidth by memory requirements:

$$\frac{150\ \mathrm{GB/s}}{4\ \mathrm{B/FLOP}} = 37.5\ \mathrm{GFLOP/s}$$

This limits computation!

Reuse Memory Accesses

- 37.5 GFLOPs is a limit
- In an actual execution, memory is not busy all the time, and the code runs at about **25 GFLOPs**
- To get closer to 1,000 GFLOPs, we need to **drastically cut down accesses to global memory**

A Common Programming Strategy

The dilemma:

- Matrices M and N are large
- They fit easily in global memory, but that's slow
- Shared memory is fast, but M and N don't fit

The solution:

- Break M and N into tiles (called blocks in the much older CPU literature)
- Read a tile into shared memory
- Use the tile from shared memory
- Repeat until done

GPU Implementation Strategy

In a GPU, only threads in a block can use shared memory.

Thus, each block operates on separate tiles:

1. Read tile(s) into shared memory using multiple threads to exploit memory-level parallelism
2. Compute based on shared memory tiles
3. Repeat
4. Write results back to global memory

Declaring Shared Memory Arrays

```
__global__ void MatrixMulKernel(float* M, float* N,
                                float* P, int Width)
{
  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
```

Outline of Technique

Identify a tile of global data that are accessed by multiple threads:

1. Load the tile from global memory into on-chip memory
2. Have the multiple threads access their data from the on-chip memory
3. Move on to the next block/tile

Place Global Memory Data into Shared Memory

Key idea:

- Each input element is used to calculate `WIDTH` elements of P
- Load each element into Shared Memory
- Have several threads use the local version to reduce memory bandwidth

Tiled Multiply

Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one subset (tile) of M and N

Loading a Tile

All threads in a block participate:

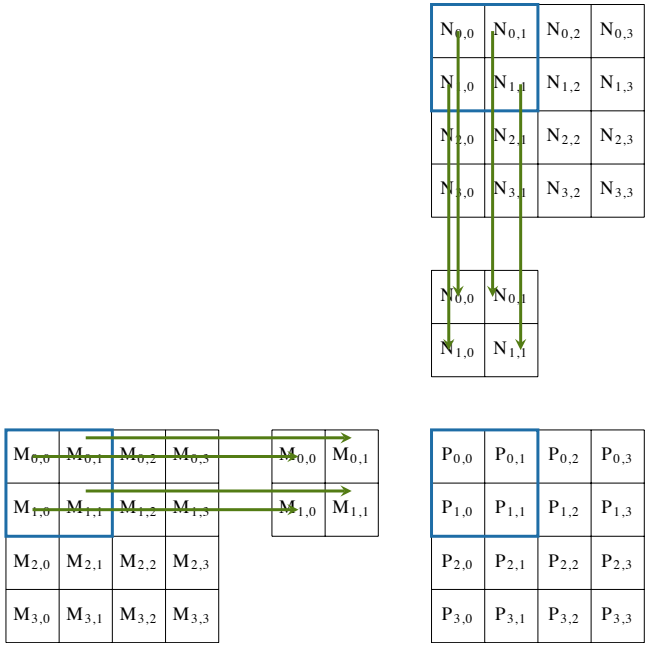
Each thread loads:

- One M element (corresponding to the global index of the thread)
- One N element (corresponding to the global index of the thread)

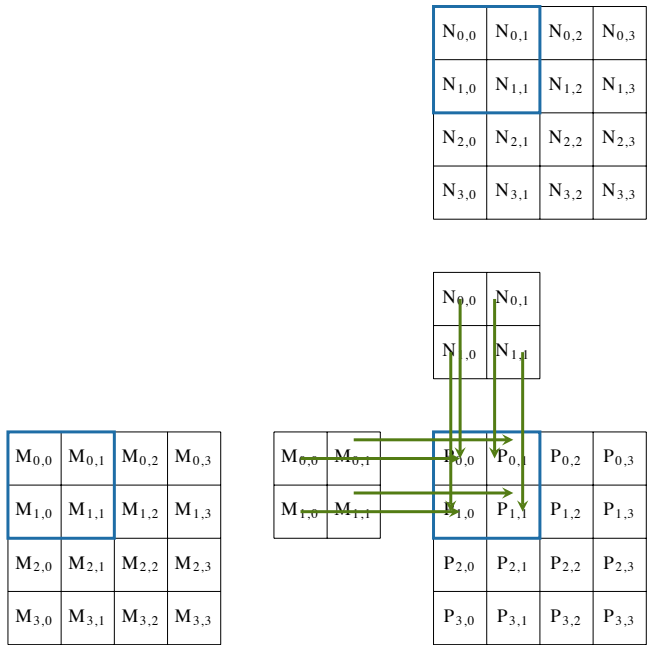
In basic tiling code:

- Assign the loaded element to each thread such that the accesses within each warp are coalesced (more later)

Work for block (0,0): load



Work for block (0,0): use



We Are Not There Yet

But…

- How can a thread know that another thread has finished its part of the tile?
- Or that another thread has finished using the previous tile?

There is a need to synchronize

Leveraging Parallel Strategies

Bulk synchronous execution: threads execute roughly in unison

- Do some work
- Wait for others to catch up
- Repeat

Much easier programming model:

- Threads only parallel within a section
- Debug lots of little programs
- Instead of one large one

Dominates high-performance applications

Bulk Synchronization Barrier

How does it work?

Use a barrier to wait for threads to 'catch up.'

A barrier is a synchronization point:

- Each thread calls a function to enter barrier
- Threads block (sleep) in barrier function until all threads have called
- After last thread calls function, all threads continue past the barrier

In CUDA

Use `__syncthreads` for CUDA Blocks

- Only within thread blocks!
- The function: `void __syncthreads(void);`
- All threads in block must enter (no subsets)
- All threads must enter the SAME static call (not the same as all threads calling function!)

Barrier Guarantees

What exactly is guaranteed to have finished?

Are shared memory operations before a barrier (e.g., stores) guaranteed to have completed?

- What about global memory ops?
- What about atomic ops with no return values?
- What about I/O operations?

CUDA manual: all global and shared memory ops (which presumably includes atomic variants) have completed.

Avoid assumptions about I/O (such as `printf`)

Tiled Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N,
                                float* P, int Width)
{
  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
  __shared__ float subTileN[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 Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

  // Loop over the M and N tiles required to compute P element
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of M and N tiles into shared memory
    subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
    subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
    __syncthreads();

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

Use of Large Tiles Shift Bottlenecks

Recall our example GPU: 1,000 GFLOP/s, 150 GB/s

16×16 tiles use each operand for 16 operations

- Reduce global memory accesses by a factor of 16
- 150GB/s bandwidth supports $(150/4) \times 16 = 600$ GFLOPs!

32×32 tiles use each operand for 32 operations

- Reduce global memory accesses by a factor of 32
- 150 GB/s bandwidth supports $(150/4) \times 32 = 1,200$ GFLOPs!
- **Memory bandwidth is no longer the bottleneck!**

Requires Parallel Access to Memory

Shared memory size:

- Implementation dependent
- 164kB per SM in Ampere (163kB max per block)

Given `TILE_WIDTH` of 16 (256 threads/block):

- Each thread block uses $2 \times 256 \times 4B = 2$ kB of shared memory, which limits active blocks to 82
- Maximum of 2048 threads per SM, which limits blocks to 8
- Thus up to $8 \times 512 = 4,096$ pending loads (2 per thread, 256 threads per block)

Choice of Tile Size

Given `TILE_WIDTH` of 32 (1,024 threads/block):

- Each thread block uses $2 \times 1024 \times 4B = 8$ kB of shared memory, which limits active blocks to 20
- Maximum of 2,048 threads per SM, which limits blocks to 2
- Thus up to $2 \times 2,048 = 4,096$ pending loads (2 per thread, 1,024 threads per block)
  - Same memory parallelism exposed

Get Up to Date with Current GPU

Number of devices in the system:

```
int dev_count;
cudaGetDeviceCount(&dev_count);
```

Capability of devices:

```
cudaDeviceProp dev_prop;
for (i = 0; i < dev_count; i++) {
  cudaGetDeviceProperties(&dev_prop, i);
  // decide if device has sufficient resources and capabilities
}
```

`cudaDeviceProp` is a built-in C structure type

- `dev_prop.maxThreadsPerBlock`
- `dev_prop.sharedMemoryPerBlock`

Handle Non-Square Matrices

How to Handle Matrices of Other Sizes?

Use tiles: assumed integral number of tiles (thread blocks) in all matrix dimensions.

How can we avoid this assumption?

One answer: add padding, but not easy to reformat data, and adds transfer time.

Other ideas?

Major Cases in Boundary Handling

Threads that calculate valid P elements but can step outside valid input:

- Second tile of Block(0,0), all threads when k is 1

Threads that do not calculate valid P elements:

- Block(1,1), Thread(1,0): non-existent row
- Block(1,1), Thread(0,1): non-existing column
- Block(1,1), Thread(1,1): non-existing row/column

Write 0s Strategy

Test during tile load: is target within input matrix?

- If yes, proceed to load
- Otherwise, just write 0 to shared memory

The benefit:

- No specialization during tile use!
- Multiplying by 0 guarantees that unwanted terms do not contribute to the inner product

What About Threads Outside of P?

If a thread is not within P:

- All terms in sum are 0
- No harm in performing FLOPs
- No harm in writing to registers
- **Must not be allowed to write to global memory!**

Solution: Threads outside of P calculate 0, but store nothing.

Modify Tile Count

Original code:

```
for (int m = 0; m < Width/TILE_WIDTH; ++m)
```

The bound for m implicitly assumes that Width is a multiple of `TILE_WIDTH`. We need to round up.

Modified code:

```
for (int m = 0; m < (Width - 1)/TILE_WIDTH + 1; ++m)
```

- **For non-multiples:** quotient is unchanged; add one to round up
- **For multiples:** quotient is now one smaller, but we add 1

Modifying the Tile Loading Code

We had:

```
// Collaborative loading of M and N tiles into shared memory
subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
```

Note: the tests for M and N tiles are NOT the same.

```
if (Row < Width && m*TILE_WIDTH+tx < Width) {
  subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
} else {
  subTileM[ty][tx] = 0;
}
```

Modifying the Tile Use Code

We had:

```
for (int k = 0; k < TILE_WIDTH; ++k)
  Pvalue += subTileM[ty][k] * subTileN[k][tx];
```

Note: no changes are needed, but we might save a little energy (fewer floating-point ops)?

```
if (Row < Width && Col < Width) {
  for (int k = 0; k < TILE_WIDTH; ++k)
    Pvalue += subTileM[ty][k] * subTileN[k][tx];
}
```

Modifying the Write to P

We had:

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

We must test for threads outside of P:

```
if (Row < Width && Col < Width) {
  P[Row*Width+Col] = Pvalue;
}
```

Important Notes

For each thread, conditions are different for:

- Loading M element
- Loading N element
- Calculation/storing output elements

Branch divergence:

- Affects only blocks on boundaries
- Should be small for large matrices

What about rectangular matrices?

Conclusion

Conclusion

Themes of this class:

- Organization of a computation with respect to data and architecture
- Usage of shared memory

Key takeaway:

Computations should carefully adapt to the architecture and maximize the usage of the resources