GP-GPU and High Performances Computing

Lecture 07 – Matrix Multiplication

January 21, 2024

Previously

- Example of scan algorithm
- Processor hardware
  - Max threads per SM : 2048
  - Max threads per block : 1024
  - Max warps per SM : 64
- If 2 blocks are assigned to an SM and each block has 1024 threads, how many warps are there in an SM?
  - Each block is divided into 1024/32 = 32 warps
  - There are 32 * 2 = 64 Warps
- 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 interleaved warps to optimize execution.

Definition

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

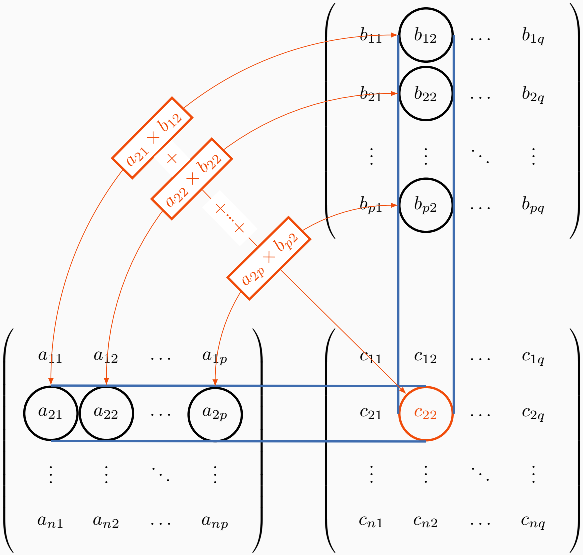
$$P = MN$$

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 $\mathcal{O}(Width^3)$.

Matrix multiply illustrated



b_{11} b_{12} $\dots$ b_{1q}
b_{21} b_{22} $\dots$ b_{2q}
b_{p1} b_{p2} $\dots$ b_{pq}
$a_{21} \times b_{12}$
$+$
$a_{22} \times b_{22}$
$+ \dots +$
$a_{2p} \times b_{p2}$
a_{11} a_{12} $\dots$ a_{1p}
a_{21} a_{22} $\dots$ a_{2p}
a_{n1} a_{n2} $\dots$ a_{np}
c_{11} c_{12} $\dots$ c_{1q}
c_{21} c_{22} $\dots$ c_{2q}
c_{n1} c_{n2} $\dots$ c_{nq}

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 possibles 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 parallelize 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 as $2 \times 2 = 4$ threads.

Number of blocks : $\dfrac{Width}{TILE\_WIDTH} \implies$ 16 blocks.

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 as $4 \times 4 = 16$ threads.

Number of blocks : $\dfrac{n}{TILE\_WIDTH} \implies 4$ blocks.

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 Widt
  // 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!

Global memory access

```
__global__ void MatrixMulKernel(float* d_M, float* d_N,
                                float* d_P, int Width) {
  // Calculate the row index of d_P  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;
  }
}
```

Each Thread Requires 4B of Data per FLOP

- Each threads access global memory
  - 4B each, or 8B per pair.
  - (And once TOTAL to P per thread—ignore it.)

  –for elements of M and N:
- 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: 150 GB/S,
  Host 4B.Flop = 37.5 GFLOP/S which 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.

A common programming strategy

- In a GPU, only threads in a block can use shared memory.
- Thus, each block operates on separate tiles: -
  - Read tile(s) into shared memory using multiple threads to exploit memory-level parallelism.
  - Compute based on shared memory tiles.
  - Repeat.
  - 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];
```

Outiline of technique

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

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

Idea: Place global memory data into Shared Memory for reuse

- 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.

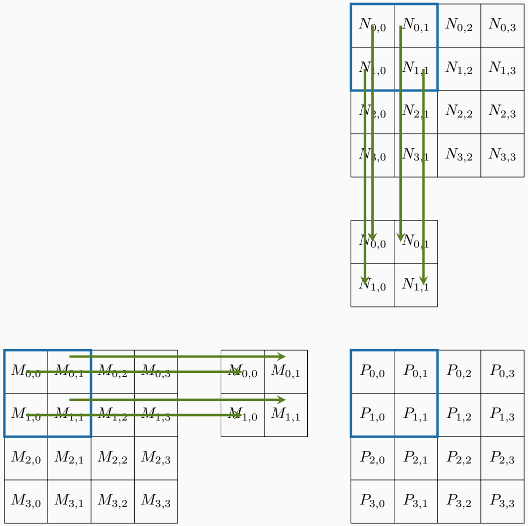
Tiled multiply

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

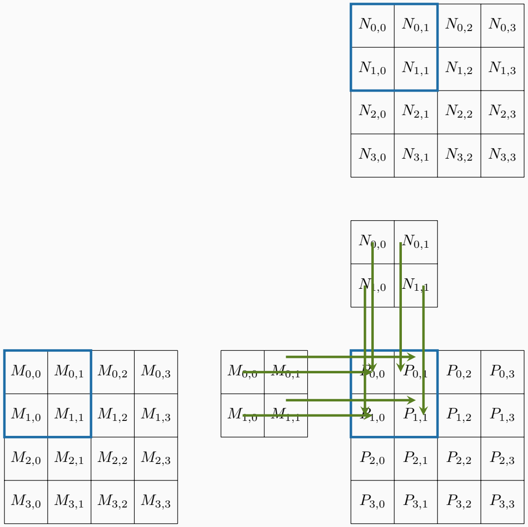
- All threads in a block participate
- Each thread loads
    - one M element (corresponding to the global index of the thread)
    - one N element (corresponding to the global index of the thread)
    - in basic tiling code: Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later)

Work for block (0,0): load



$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Work for block (0,0): use



$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

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 thread 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
- How does it work in CUDA?
- 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 trauma: what is actually done

- 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; // note: blockDim.x == TILE_WIDTH
   int Col = bx * TILE_WIDTH + tx; // blockDim.y == TILE_WIDTH
   float Pvalue = 0;
  // Loop over the M and N tiles required to compute the P element
  // The code assumes that the Width is a multiple of TILE_WIDTH!
  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();
   }
   P[Row*Width+Col] = Pvalue;
}
```

Classical matrix multiply

```
{
  // Calculate the row index of the P element and M
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  // Calculate the column index of P and 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 += M[Row*Width+k] * N[k*Width+Col];
    P[Row*Width+Col] = Pvalue;
  }
}
```

Use of large tiles shift bottlenecks

- Recall our example GPU: 1,000 GFLOP/s, 150 GB/s
- 16x16 tiles use each operand for 16 operations
    - reduce global memory accesses by a factor of 16
    - 150GB/s bandwidth supports (150/4)*16 = 600 GFLOPs!
- 32x32 tiles use each operand for 32 operations
    - reduce global memory accesses by a factor of 32
    - 150 GB/s bandwidth supports (150/4)*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 $2 \times 256 \times 4B = 2kB$ 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 $2 \times 1024 \times 4B = 8kB$ 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.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 case in toy example

- 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

- 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!

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

Modify tile count

- `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.
- `for (int m = 0; m < (Width \item 1)/TILE_WIDTH + 1; ++m)`
  - For non-multiples of `TILE_WIDTH`:
    - quotient is unchanged;
    - add one to round up.
  - For multiples of `TILE_WIDTH`:
    - 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) {
    // as before
      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) {
  // as before
  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) {
  // as before
  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
- Computations should carefully adapt to the architecture and maximize the usage of the ressources