GPGPU Programming

Donato D’Ambrosio

Department of Mathematics and Computer Science
Cubo 30B, University of Calabria, Rende 87036, Italy

Lecturer’s email: donato.dambrosio [at] unical.it
Lecturer’s homepage: http://www.mat.unical.it/~donato

Course’s homepage: https://www.mat.unical.it/~donato/gpgpu.html
Course team page: https://bit.ly/3CU9xiR
Code to join the team: **3d98qzr**

Academic Year 2021/22

Table of contents

Memory and Data Locality

Memory and Data Locality

Memory Access Efficiency

- This lecture is about how to efficiently exploit GPU resources to maximize performance.
- Major bottlenecks are global memory's high access latencies (hundreds of clock cycles) and limited bandwidth.
- While numerous threads available for execution can theoretically tolerate long memory access latencies, traffic congestion in the global memory access can prevents all but very few threads from making progress, thus rendering some of the SMs (Streaming Multiprocessors) idle.
- Using CUDA different memory types can circumvent such congestion.
- Re-thinking the serial application is generally necessary to improve the performance.

Importance of Memory Access Efficiency

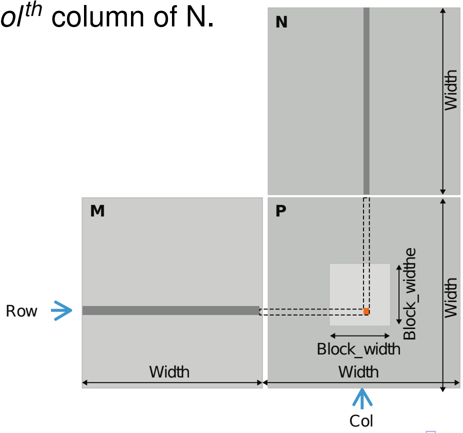
- A metric that can be used to asses the efficiency of a kernel is the compute-to-global-memory-access ratio:

  ```
  r = #(floating point operations)/ #(accesses to the global memory)
  ```

- A modern GPUs can reach 1 TB/s global memory bandwidth and more, with about 12 TFlop peak performance.
- The theoretical floating point (4 bytes) peak load performance is 1000/4 = 250 giga single-precision operands per second.
- With $r = 1$ a kernel will be limited by the rate at which the operands (e.g., the elements of in[] of the image blur kernel) can be delivered. The kernel will achieve no more than 250 Gflops (memory-bound kernel).
- 250 Gflops corresponds to only the 2% 12 Tflops GPU!
  An $r = 48$ would be needed to reach the peak performance!

Matrix Multiplication Example

- Matrix multiplication $P = M \cdot N$ permits to introduce relatively simple techniques for reducing global memory accesses.
- The generic element $P_{Row,Col}$ is the dot product of the Row^{th} row of M and the Col^{th} column of N.



Matrix Multiplication Example

- Let us consider square matrices of dimension `Width`, which is a power of 2. A 2D grid of threads can be defined as:

```
dim3 dimGrid(Width/16.0, Width/16.0, 1);
dim3 dimBlock(16, 16, 1);
MatrixMulKernel<<<dimGrid,dimBlock>>>(M, N, P, Width);
```

- Each thread compute *Row* and *Col* global coordinates as:

```
Row = blockIdx.y*blockDim.y+threadIdx.y;
Col = blockIdx.x*blockDim.x+threadIdx.x;
```

- Given *Row* and *Col*, the dot product can be computed as:

```
Pvalue = 0;
for (int k = 0; k < Width; ++k)
  Pvalue += M[Row*Width+k] * N[k*Width+Col];
```

  - `Row*Width` (in `Row*Width+k`) moves to the beginning of the row `Row`, while `k` moves to the k^{th} element of the same row.
  - Similarly, `k*Width+Col` moves to the beginning of the k^{th} row and then offsets to the element of index *Col*.

Matrix Multiplication Example

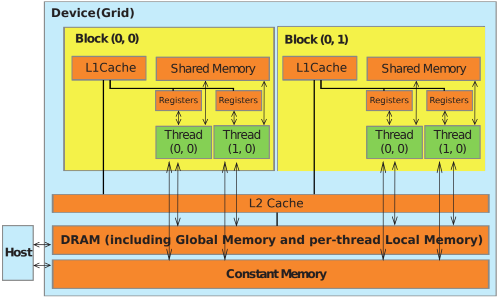
- A straightforward matrix multiplication kernel can be the following:

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
int Width) {
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k] * N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }
}
```

- In every iteration of the for loop (which is computationally predominant), we find 2 global memory accesses for two floating-point operation (one multiplication and one addition), resulting in the compute-to-global-memory-access ratio $r = 1$. The kernel is memory-bound in almost each current GPU!

CUDA Memory Types

- Exploiting CUDA memory types can considerably improve the performance.



Device code can:

- R/W per-thread **Registers** (physical)
- R/W per-thread **Local Memory** (physical)
  (either in Registers or in DRAM)
- R/W per-block **Shared Memory** (physical)
- R/W per-grid **Global Memory** (in **DRAM** - physical)
- Read-only per-grid **Constant Memory** (physical)

Device code CAN NOT:

- Explicitly use **L1** and **L2 Caches** (physical)

Host code can:

- Tranfer data to/from per-grid
  global and constant memory

- Constant memory is a kind of global read-only cache.
- There are also L1 and L2 caches we will exploit later.

CUDA Memory Types

- Global/constant memory can be written and read by the host.
  - Global memory can be written and read by the device.
  - Constant memory is a device read-only short-latency, high-bandwidth memory.
  - L1 and L2 caches are also device short-latency, high-bandwidth memories that are managed by the run-time (they are transparent to the user).
- Registers and shared memory are SM on-chip high-speed memories (can not be accessed by the host).
  - Registers are allocated to individual threads; each thread can only access its own registers. Registers can be thought like CPU registers: few KBs, very fast access.
  - Shared memory locations are allocated to thread blocks; all threads in a block can access shared memory variables allocated to the block. Shared memory is a *scratchpad memory*, i.e., a non-transparent cache-like memory.

CUDA Memory Types

- When the processor accesses data in the shared memory, it needs to perform a memory load operation, similar (though faster) to accessing data in the global memory.
- Shared memory has longer latency and lower bandwidth than registers (also because of the need to perform a load operation).
- *The variables in the shared memory are accessible by all threads in a block, whereas register data are private to a thread*.
- Shared memory is designed to allow multiple processing units to simultaneously access its contents to support efficient data sharing among threads in a block.

CUDA Memory Types

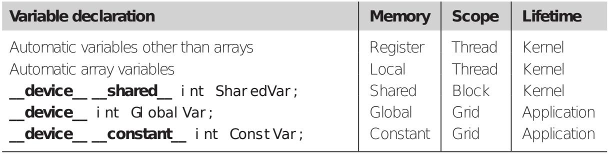
- Automatic scalar variables declared in kernel and device functions are placed into registers. Their scopes are within individual threads, i.e., a private copy is generated for each thread.
- WARNING: Exceeding the capacity of the registers will reduce the number of active threads assigned to each SM. *It is often a good idea to break out long kernels in more smaller kernels to reduce the per-kernel registers data*.
- Automatic arrays are stored in local memory. Small, constant-indexed arrays can be physically stored in Registers (loop unrolling permits to have constant indices), in DRAM otherwise.
- The `__shared__` and `__constant__` keywords[1] must precede objects declaration to use shared and constant memory, respectively.

[1] Optionally, `__device__` can be added at the beginning of the declaration.

CUDA Memory Types

- A variable whose declaration is preceded only by `__device__` is a global variable and is placed in the slow global memory[2].
- Global variables are visible to all threads of all kernels. Their contents also persist throughout the entire program execution.
- Global memory can not use to synchronize threads of different blocks. However, this can be easily obtained by terminating the current kernel execution. The kernel execution temrination works like a barrier.

CUDA Variable Type Qualifyers



Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
__device__ __shared__ int SharedVar;	Shared	Block	Kernel
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application

[2] Latency and throughput of accessing global variables have been improved with caches in relatively recent devices.

CUDA Memory Types

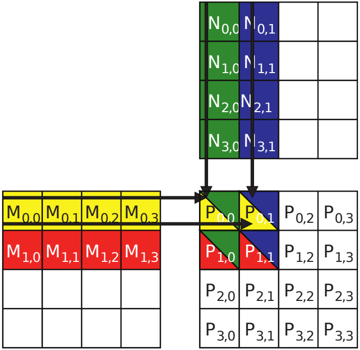
- In CUDA, pointers are used to point to data objects in the global memory. Pointer usage arises in kernel and device functions in two ways:
    - If an object is allocated by a host function, the pointer to the object is initialized by `cudaMalloc` and can be passed to the kernel function as a parameter (e.g., the parameters M, N, and P)
    - The address of a variable declared in the global memory is assigned to a pointer variable. For instance, the statement

      ```
      float* ptr= &GlobalVar;
      ```

      in a kernel function assigns the address of `GlobalVar` into an automatic pointer variable `ptr`.

A Faster Matrix Multiplication Example

- A common strategy for improving the matrix multiplication kernel is to partition the problem into tiles that can be computed independently of each other, so that each tile fits into the shared memory. Let us consider 4 blocks and tile dimensions equal to those of the blocks.



A Faster Matrix Multiplication Example

- Here the global memory accesses performed by all threads in block(0,0).

Access order →

$\text{thread}_{0,0}$	$\boxed{M_{0,0}} * N_{0,0}$	$M_{0,1} * \boxed{N_{1,0}}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$\text{thread}_{0,1}$	$\boxed{M_{0,0}} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$\text{thread}_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * \boxed{N_{1,0}}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$\text{thread}_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

- Each thread accesses 4 elements of M and 4 elements of N during execution. A significant overlap occurs. For instance:
  - Thread(0,0) and thread(0,1) access the whole row 0 of M.
  - Thread(0,1) and thread(1,1) access the whole column 1 of N.
  - ...
- The original kernel is written so that both thread(0,0) and thread(0,1) access row 0 elements of M from the global memory.

A Faster Matrix Multiplication Example

- Every M and N element is accessed exactly twice during the execution of block(0,0). Therefore, if all four threads can be made to collaborate in their accesses to global memory, traffic to the global memory can be reduced by half.

Access order →

$\text{thread}_{0,0}$	$\boxed{M_{0,0}} * N_{0,0}$	$M_{0,1} * \boxed{N_{1,0}}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$\text{thread}_{0,1}$	$\boxed{M_{0,0}} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$\text{thread}_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * \boxed{N_{1,0}}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$\text{thread}_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

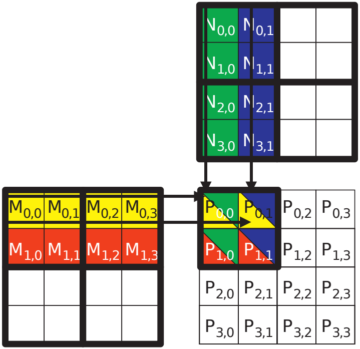
- With Width × Width blocks, the potential reduction of global memory traffic would be Width. Thus, if we use 16 × 16 blocks, the global memory traffic can be potentially reduced to 1/16 through collaboration between threads.

A Faster Matrix Multiplication Example

- In the context of parallel computing, tiling is a program transformation technique that localizes the memory locations accessed among threads.
- It divides the long access sequences of each thread into phases and uses barrier synchronization to keep the timing of accesses to each section at close intervals.
- The basic idea is for the threads to collaboratively load subsets of the M and N elements into the shared memory before they individually use these elements in their dot product calculation.
    - WARNING: The size of the shared memory is quite small (96 KiB only in the GTX 980), and the capacity of the shared memory should not be exceeded when these M and N elements are loaded into the shared memory.
    - This condition can be satisfied by dividing the M and N matrices into smaller tiles so that they can fit into the shared memory.

A Faster Matrix Multiplication Example

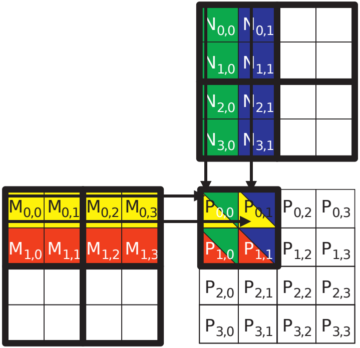
- The dot products performed are divided into phases.
- In each phase, all threads in a block collaborate to load a tile of M and a tile of N into the shared memory.
- Every thread in a block loads one element of M and one element of N into the shared memory.



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

time →

A Faster Matrix Multiplication Example



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

time →

- In the the first phase:
  - The upper-left tiles of M and N are loaded into the shared memory;
  - Threads sincronize;
  - Partial sum are computed.
- The second phase is similar to the first one: The raimainig tiles are loaded and the final value computed.

A Faster Matrix Multiplication Example

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_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;
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
    Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
    Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
  }
  d_P[Row*Width + Col] = Pvalue;
}
```

A Faster Matrix Multiplication Example

- The `ph` variable indicates the current phases. In each phase:
  - `Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];`
    loads the appropriate M element into the shared memory, where

    `[Row*Width + ph*TILE_WIDTH + tx] ~ [Row][ph*TILE_WIDTH + tx]`

    is the relation between linearized and two-dimensional index.
  - `Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];`
    loads the appropriate N element into the shared memory, where

    `[(ph*TILE_WIDTH + ty)*Width + Col] ~ [ph*TILE_WIDTH + ty][Col]`

    is the relation between linearized and two-dimensional index.
  - The barrier `__syncthreads()` ensures that all threads have finished loading the tiles of M and N into Mds and Nds before any of them can move forward.

A Faster Matrix Multiplication Example

- The tiled algorithm reduces the global memory accesses by a factor of `TILE_WIDTH`.
- If one uses 16 × 16 tiles, we can reduce the global memory accesses by a factor of 16.
- This increases the compute-to-global-memory-access ratio from 1 to 16.
- This improvement allows the memory bandwidth of a CUDA device to support a computation rate close to its peak performance; e.g. a device with 150 GB/s global memory bandwidth can approach ((150/4)*16) = 600 GFLOPS!

Memory as a Limiting Factor to Parallelism: Registers

- Any smart CUDA programmer must pay attention to not exceed the memory resources to avoid reducing the level of parallelism.
- To illustrate register usage impact of a kernel on the level of parallelism, let us consider a device D where each SM can accommodate up to 1536 threads and 16,384 registers.
  - To support 1536 threads, each thread can use only 16,384/1536 = 10 registers!
  - If each thread uses 11 registers, the number of threads that can be executed concurrently in each SM will be reduced at the block granularity! For instance, if each block contains 512 threads, the reduction of threads will be accomplished by reducing 512 threads at a time.
  - This procedure can substantially reduce the number of warps available for scheduling, thereby decreasing the ability of the processor to find useful work in the presence of long-latency operations.

Memory as a Limiting Factor to Parallelism: Shared Memory

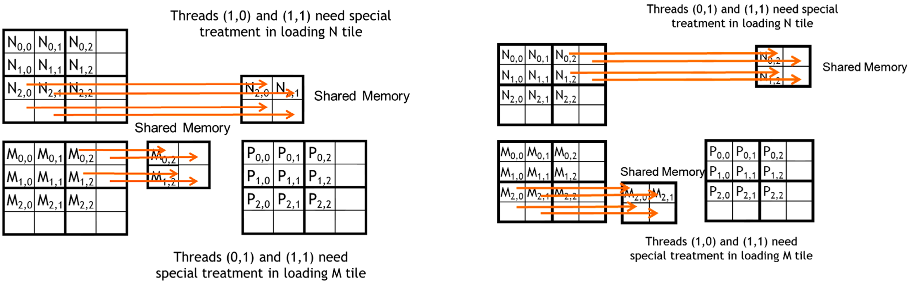
- Shared memory usage can also limit the number of threads. We can assume that the same device D has 16,384 (16K) bytes of shared memory and can accommodate up to 8 blocks.
    - For the matrix multiplication kernel, if `TILE_SIZE` is 16, each block needs 16×16×4=1K bytes for Mds and 1K byte for Nds, resulting in 2K bytes of shared memory.
    - The 16K-byte shared memory allows 8 blocks to simultaneously reside in an SM. Since this is the same as the maximum allowed by the threading hardware, shared memory is not a limiting factor for this tile size.
    - In this case, the real limitation is the threading hardware limitation that only allows 1536 threads in each SM. This constraint limits the number of blocks in each SM to six. Consequently, only 6*2KB= 12KB of the shared memory will be used.

Memory as a Limiting Factor to Parallelism: Shared Memory

- These limits change from one device to another but can be determined at run-time with device queries.
- A further parameter could be provided to the kernel, that could allocate the proper amount of shared memory.
- Device-side, in the kernel, the buffer can be dynamically allocated by declaring it as `extern __shared__ Mds[];` and can be allocated by simply using `malloc` (as we usually do in C).
- Note that, however, the buffer must be one-dimensional due to the current CUDA C limitations.
- Here you can find a useful reference: http://campynet.com/?p=454.

Generalizing the Tiled Matrix Multiplication Algorithm

- The tiled multiplication algorithm discussed so far is valid for squared matrices with width equal to a multiple of the tile width.
- Let us consider 3x3 matrices and 2x2 tiles (and blocks).


$N_{0,0}$ $N_{0,1}$ $N_{0,2}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$
Shared Memory
Shared Memory
Shared Memory
Shared Memory

Threads (1,0) and (1,1) need special treatment in loading N tile

Threads (0,1) and (1,1) need special treatment in loading M tile

Threads (0,1) and (1,1) need special treatment in loading N tile

Threads (1,0) and (1,1) need special treatment in loading M tile

Figure: Phase 1 of block(0,0) and phase 0 of block(1,0).

Generalizing the Tiled Matrix Multiplication Algorithm

- When a thread intends to load an input tile element, it should test that input element for validity, which is easily done by examining the y and x indexes.
- The boundary condition test would be that both indexes are smaller than Width: `(Row<Width) && (ph*TILE_WIDTH+tx)<Width` for M, and `(ph*TILE_WIDTH+ty)<Width && Col<Width` for N.
- If the condition is not met, the thread can put 0.0 in the shared memory location (the value does not change the result of the matrix multiplication).
- Finally, a thread should only store its final inner product value if it is responsible for calculating a valid P element. The test for this condition is `(Row < Width) && (Col < Width)`.

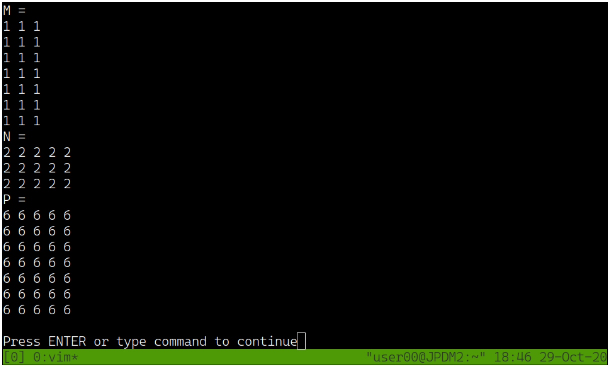
Generalizing the Tiled Matrix Multiplication Algorithm

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
   unsigned int j, unsigned int k, unsigned int l) {
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
  int tx = threadIdx.x; int ty = threadIdx.y;
  int Row = blockIdx.y*TILE_WIDTH+ty, Col = blockIdx.x*TILE_WIDTH+tx;

  float Pvalue = 0
  for (int ph = 0; ph < k/TILE_WIDTH; ++ph) {
    if ((Row < j) && (ph*TILE_WIDTH+tx) < k)                 //<--
      Mds[ty][tx] = d_M[Row*k + ph*TILE_WIDTH + tx];
      else Mds[ty][tx] = 0;
    if ((ph*TILE_WIDTH+ty) < k && Col < l)                   //<--
      Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*l + Col];
      else Nds[ty][tx] = 0;
    __syncthreads();
    for (int i = 0; i < TILE_WIDTH; ++i)
      Pvalue += Mds[ty][i] * Nds[i][tx];
    __syncthreads();
  }
  if ((Row<j) && (Col<l)) d_P[Row*l + Col] = Pvalue;          //<--
}
```

Generalizing the Tiled Matrix Multiplication Algorithm

- The above kernel can still only handle square matrices.
- Extending the kernel to non-square matrices is left as a mandatory exercise. You can start working on it right now!



```
M =
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
N =
2 2 2 2 2
2 2 2 2 2
2 2 2 2 2
P =
6 6 6 6 6
6 6 6 6 6
6 6 6 6 6
6 6 6 6 6
6 6 6 6 6
6 6 6 6 6
6 6 6 6 6

Press ENTER or type command to continue
[0] 0:vim*                                   "user00@JPDM2:~" 18:46 29-Oct-20
```