GPU: manycore processors

Intensive Computation
Annalisa Massini
2013/2014
These slides have been prepared by using the book:

- Programming Massively Parallel Processors
  D.B. Kirk, W.W. Hwu
  *Morgan Kaufmann*
Introduction

- Microprocessors based on a single central processing unit (CPU), drove rapid performance increases and cost reductions in computer applications for more than two decades.

- These microprocessors brought giga (billion) floating-point operations per second (GFLOPS) to the desktop and hundreds of GFLOPS to cluster servers.

- With the advances in hardware to the same software simply runs faster as each new generation of processors is introduced.
This drive, however, has slowed since 2003 due to energy consumption and heat-dissipation issues that have limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU.

Microprocessor vendors have switched to models where multiple processing units, referred to as processor cores, are used in each chip to increase the processing power.
Introduction

The introduction of processing cores has exerted a tremendous impact on the software developer community.

The majority of software applications are written as sequential programs running on one of the processor cores, which will not become significantly faster than those in use today.

The performance of applications software will improve with each new generation of microprocessors for parallel programs, in which multiple threads of execution cooperate to complete the work faster.
The practice of parallel programming is no new. The high-performance computing community has been developing parallel programs for decades. These programs run on large-scale, expensive computers. Only few applications can justify the use of expensive computers, limiting the practice of parallel programming to a small number of application developers. Now that all new microprocessors are parallel computers, the number of applications that must be developed as parallel programs has increased dramatically.
Introduction

Since 2003, the semiconductor industry has settled on two main trajectories for designing microprocessor

The **multicore** trajectory maintains the execution speed of sequential programs while moving into multiple cores.

The multicores began as two-core processors.

Intel Core i7 microprocessor (November 2008) has **4 processor cores**, each of which implements the full x86 instruction set; it supports **hyperthreading** with 2 hardware threads and is designed to maximize the execution speed of sequential programs.
In contrast, the many-core trajectory focuses more on the execution throughput of parallel applications.

The many-cores began as a large number of much smaller cores

The NVIDIA GeForce GTX 280 Graphics Processing Unit (June 2008) has 240 cores, each of which is a heavily multithreaded, and shares its control and instruction cache with seven other cores.
Introduction

Many-core processors, especially the GPUs, have led the race of floating-point performance since 2003.

While the performance improvement of general-purpose microprocessors has slowed significantly, the GPUs have continued to improve relentlessly.
Such a large performance gap between many-core GPUs and general-purpose multicore CPUs is due to the differences in the fundamental design philosophies between the two types of processors.
Introduction

- The **design of a CPU** is optimized for **sequential code performance**.

- A sophisticated control logic allows instructions from a **single thread** of execution to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution.

- **Large cache memories** are provided to reduce the instruction and data access latencies.
Introduction

Memory bandwidth is an important issue.

Graphics chips have been operating at approximately 10 times the bandwidth of contemporaneously available CPU chips.

General-purpose processors have to satisfy requirements from legacy operating systems, applications, and I/O devices that limit the memory bandwidth.

In contrast, with simpler memory models and fewer legacy constraints, the GPU designers can more easily achieve higher memory bandwidth.
Introduction

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not

- **CPU**: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic

- **GPU**: maximize throughput of all threads
  - # threads in flight limited by resources → lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency → skip the big caches
  - share control logic across many threads
Introduction

- **GPUs** are designed as *numeric computing engines*.
- **GPUs** will not perform well on some tasks on which **CPUs** are designed to perform well.
- Most applications will use **both CPUs and GPUs**, executing the *sequential parts on the CPU* and *numerically intensive parts on the GPUs*.
- The **CUDA** (Compute Unified Device Architecture) programming model, introduced by NVIDIA in **2007**, is designed to support joint CPU/GPU execution of an application.
Introduction: CUDA

- **Augment C/C++** with minimalist abstractions
  - let programmers focus on parallel algorithms
  - *not* mechanics of a parallel programming language

- Provide straightforward **mapping onto hardware**
  - good fit to GPU architecture
  - maps well to multi-core CPUs too

- **Scale to 100s of cores & 10,000s of parallel threads**
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization
Until 2006, OpenGL or Direct3D techniques were needed to program graphics chips, that is processor cores had to be accessed by the equivalent of graphic application programming interface (API) functions.

Only a few people could master the skills necessary to use these chips to achieve performance for a limited number of applications.

This is why it did not become a widespread programming phenomenon.

Everything changed in 2007 with the release of CUDA.
Introduction

- More recently, several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA, have jointly developed a standardized programming model called **OpenCL**

- The **OpenCL** programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery in massively parallel processors.

- Applications developed in OpenCL can run without modification on all processors that support the OpenCL language extensions and API.
Introduction

- The GPU is viewed as a compute **device** that:
  - Is a coprocessor to the CPU or **host**
  - Has its own DRAM (**device memory**)
  - Runs many **threads in parallel**

- Data-parallel portions are executed on the device as **kernels** which run in parallel on many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few
A typical CUDA-capable GPU is organized as follows:

- An array of streaming multiprocessor SMs
- 2 SMs form a building block
- The number of SMs in a building block can vary
Architecture of a modern GPU

- Each SM has a number of streaming processors (SPs) that share control logic and instruction cache.
- Each SP has a multiply–add (MAD) unit and an additional multiply unit.
Architecture of a modern GPU

- The G80 that introduced the CUDA architecture has 128 SPs (16 SMs, each with 8 SPs).
- With 128 SPs, that’s a total of over 500 gigaflops.
Architecture of a modern GPU

- The G80 chip supports up to 768 threads per SM, which sums up to about 12,000 threads for this chip.
- The more recent GT200 supports 1,024 threads per SM and up to about 30,000 threads.
Data parallelism

- In many applications, program sections exhibit data parallelism, allowing many arithmetic operations to be performed on program data structures simultaneously.

- Let us consider a simple example:

  matrix-matrix multiplication - \( P = M \times N \)

- Each element of matrix \( P \):
  - is computed by executing the multiplication of a row of matrix \( M \) and a column of matrix \( N \) (dot product)
  - do not influence the computation of other matrix elements

- More elements of matrix \( P \) can be computed in parallel
Data parallelism

- For a 1000x1000 matrix multiplication has 1,000,000 independent elements (independent dot product) each requiring 1,000 multiply and 1,000 additions.

- A GPU can significantly accelerate the execution of the matrix multiplication over a traditional host CPU.

- The data parallelism is not always as simple as this.

- A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU.
Data parallelism

- The phases that exhibit little or **no data parallelism** are implemented in **host code**.
- The phases that exhibit **rich amount of data parallelism** are implemented in the **device code**.
- A CUDA program is a unified source code encompassing both host and device code.
- The NVIDIA C compiler (**nvcc**) separates the two during the compilation process.
Data parallelism

- The **host code** is straight ANSI C code
- It is further compiled with the host’s standard C compilers and runs as an ordinary CPU process.
- The **device code** is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures.
- The device code is typically further compiled by the `nvcc` and executed on a GPU device.
- The **kernel** functions (or, simply, kernels) typically generate a large number of threads to exploit data parallelism.
Data parallelism

- In matrix multiplication, the entire matrix multiplication computation can be implemented as a kernel where each thread is used to compute one element of output matrix P.

- For a 1000 x 1000 matrix multiplication, the kernel would generate 1,000,000 threads when it is invoked.

- CUDA threads are of much lighter weight than the CPU threads.

- CUDA threads take very few clock cycles to generate and schedule due to efficient hardware support.
Kernel, grid, thread

- The execution of a typical CUDA program
  - starts with host (CPU) execution
  - when a **kernel** function is invoked, or launched, the execution is moved to a device (GPU)
  - a large number of **threads** is generated to take advantage of big data parallelism
- All the **threads** that are generated by a **kernel** during an invocation are collectively called a **grid**.
When all **threads** of a **kernel** complete their execution, the corresponding **grid** terminates, and the execution continues on the host until another **kernel** is invoked.
Matrix-matrix multiplication

- For C programs, the placement of a 2-dimensional matrix into this linear addressed memory is done according to the row-major convention:
  - All elements of a row are placed into consecutive memory locations.
  - The rows are then placed one after another.
Matrix-matrix multiplication

```c
void MatrixMulOnHost(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;
        }
}
```
Matrix-matrix multiplication

- Modify the program to port the matrix multiplication function into CUDA.

```c
void MatrixMulOnDevice(float* M, float* N, float* P, int Width) {
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;
    ...
    1. // Allocate device memory for M, N, P and
       // load M, N to device memory
    2. // Kernel invocation code to have the device to perform
       // the actual matrix multiplication
    3. // copy P from the device
       // Free device matrices
}
```
Execution

- The revised MatrixMultiplication() function is essentially an outsourcing agent that
  - ships input data to a device
  - activates the calculation on the device
  - collects the results from the device

- The memory organization is very important
  - the host and devices have separate memory spaces
  - in fact devices are typically hardware cards that come with their own dynamic random access memory (DRAM)
Kernel Execution

- To execute a **kernel** on a device, the programmer needs:
  - to allocate memory on the device
  - to transfer pertinent data from the host memory to the allocated device memory

- After device execution, the programmer needs:
  - to transfer result data from the device memory back to the host memory
  - to free up the device memory that is no longer needed
CUDA supports several types of memory that can be used by programmers to achieve high execution speeds in their kernels:

- registers
- shared memory
- global memory
- constant memory

Registers and shared memory are on-chip memories.
CUDA device memory model

**Host code** can
- transfer data to and from the device by using **global** and **constant memory**

**Device code** can
- R/W per-thread **registers**
- R/W per-thread **local memory**
- R/W per-block **shared memory**
- R/W per-grid **global memory**
- Read-only per-grid **constant memory**
CUDA device memory model

Note that the **host memory** is not explicitly shown, but is assumed to be contained in the host.

The CUDA memory model is supported by **API functions** that help programmers to manage data in these memories.
CUDA device memory model: API

- `cudaMalloc()`
  - allocates a piece of **global memory** for an object
  - requires two parameters
    - the **address of a pointer variable** that must point to the allocated object
    - the **size of the object** to be allocated, in bytes
CUDA device memory model: API

- `cudaFree()`
  - frees the storage space from the device **global memory**
  - is called with **pointer to the object** as input
CUDA device memory model: API

- **cudaMemcpy()**
  - memory data transfer
  - requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host - Host to Device - Device to Host - Device to Device
  - Transfer to device is asynchronous
CUDA device memory model: API

Code example:

- Allocate a 64 * 64 single precision float array
- Attach the allocated storage to Md
- “d” is often used to indicate a device data structure

TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);
cudaFree(Md);
CUDA device memory model: API

Code example:

- Transfer a float array
- M is in host memory and Md is in device memory
- cudaMemcpyHostToDevice and cudaMemcpyDeviceToDeviceToHost are symbolic constants

```c
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToDeviceToHost);
```

Note that the two symbolic constants, cudaMemcpyHostToDevice and cudaMemcpyDeviceToDeviceToHost are recognized, predefined constants of the CUDA programming environment.
void MatrixMulOnDevice(float* M, float* N, float* P, int Width) {
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;
    ...
    1. // Allocate and Load M, N to device memory
       cudaMemcpy(&Md, size);
       cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
       cudaMemcpy(&Nd, size);
       cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    Note that the two symbolic constants, cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are recognized, predefined constants of the CUDA programming environment.
CUDA device memory model: example

// Allocate P on the device
cudaMalloc(&Pd, size);

2. // Kernel invocation code – to be shown later

3. // Read P from the device
cudaMemcpy(P, Pd, size, CudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
Keyword __global__

In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase.

The keyword __global__ indicates that the function is a kernel and that it can be called from a host function to generate a grid of thread on a device.

The function will be executed on the device and can only be called from the host.
Keyword __device__ and __host__

__device__ indicates the function is a CUDA device function
- executes on a CUDA device
- can only be called from a kernel function or another device function
- can have neither recursive function calls nor indirect function calls through pointers in them

__host__ indicates the function is a CUDA host function
- is simply a traditional C function that executes on the host
- can only be called from another host function.
- by default, all functions in a CUDA program are host functions if they do not have any of the CUDA keywords in their declaration
CUDA Function Declarations

Then C function declarations are extended by CUDA with three qualifier keywords:

__global__ __device__ __host__

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>
Keyword threadIdx.x and threadIdx.y

- Other extensions of C are the keywords `threadIdx.x`, `threadIdx.y` and `threadIdx.z` which refer to the thread indices.
- All threads execute the same kernel code → a mechanism to allow them to distinguish themselves / direct themselves toward the parts of the data structure they work on.
- These keywords identify predefined variables
- Different threads will see different values in their variables `threadIdx.x`, `threadIdx.y` and `threadIdx.z`.
- Coordinates reflect the threads multidimensional organization.
Matrix multiplication kernel

// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;
  for (int k = 0; k < Width; ++k)
  {
    float Mdelement = Md[ty * Md.width + k];
    float Ndelement = Nd[k * Nd.width + tx];
    Pvalue += Mdelement * Ndelement;
  }
  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
Threads and blocks

- When a **kernel** is invoked, or launched, it is executed as **grid** of parallel **threads**.
- Each CUDA **thread grid** typically is comprised of thousands to millions of lightweight GPU **threads** per kernel invocation.
- Creating enough **threads** to fully utilize the hardware often requires a large amount of data parallelism.
- For example, each element of a large array might be computed in a separate thread.

```
0 | 1 | 2 | 254 | 255
```
```
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

---

50
Threads and blocks

- Threads in a grid are organized into a **two-level hierarchy**
- At the top level, each **grid** consists of one or more **thread blocks**
- All blocks in a grid have the same number of threads
- In the figure: Grid 1 is organized as a 2x2 array of 4 blocks
- Each block has a unique three-dimensional coordinate given by the CUDA keywords `blockIdx.x` `blockIdx.y` and `blockIdx.z`. 

![Diagram of CUDA thread organization](image-url)
Threads and blocks

- Each thread block is, in turn, organized as a three-dimensional array of threads with a total size of up to 512 threads.
- The coordinates of threads in a block are uniquely defined by three thread indices: threadIdx.x, threadIdx.y and threadIdx.z.
Threads and blocks

- Not all applications will use all three dimensions of a thread block.
- In figure, each thread block is organized into:
  - a 4x2x2 three-dimensional array of threads.
- This gives Grid 1 a total of $4 \times 16 = 64$ threads.

This is obviously a simplified example.
Threads and blocks

Summing up

- Parallel **kernels** are composed of many threads
  - all threads execute the same sequential program

- Threads are grouped into **thread blocks**
  - threads in the same block can cooperate

- Threads/blocks have unique IDs
Execution configuration

- When the host code invokes a kernel, it sets the grid and thread block dimensions via `execution configuration` parameters.

- Two struct variables of type `dim3` are declared.

Example

```c
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);

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

Example

```c
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);
// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

- **dimBlock** describes the configuration of blocks, defined as 16x16 groups of threads.
- **dimGrid** describes the configuration of the grid (in this example, we only have one (1x1) block in each grid.)
Execution configuration

Example

```c
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);
// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

The final line of code invokes the kernel.

It provides

- the dimensions of the grid in terms of number of blocks
- the dimensions of the blocks in terms of number of threads.
Execution configuration

In general

- a **grid** is organized as a 3D array of **blocks**.
- each **block** is organized into a 3D array of **threads**.

- The total size of a block is limited to 512 threads, with flexibility in distributing these elements into the three dimensions if the total number of threads $\leq 512$
  
  For example $(512, 1, 1)$, $(8, 16, 2)$, and $(16, 16, 2)$

The exact organization of a **grid** is determined by the execution configuration provided at **kernel** launch.
Syncthreads

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function \texttt{__syncthreads()}

When a kernel function calls \texttt{__syncthreads()}, the thread that executes the function call will be held at the calling location until every thread in the block reaches the location.

This ensures that all threads in a block have completed a phase of their execution of the kernel before any moves on to the next phase.
Execution configuration

- **Thread** scheduling is strictly an implementation concept.
- Once a block is assigned to a **streaming multiprocessor**, it is further divided into 32-thread units called **warps**.
- **Warps** are not part of the CUDA specification and their size is implementation specific.
- The **warp** is the unit of **thread** scheduling in SMs.
- Each **warp** consists of 32 **threads** of consecutive **threadIdx** values.
- With enough **warps** around, the hardware will likely find a **warp** to execute at any point in time, thus filling the latency of expensive operations (**latency hiding**).
CUDA supports several types of memory that can be used by programmers to achieve high execution speeds in their kernels:

- registers
- shared memory
- global memory
- constant memory

Registers and shared memory are on-chip memories.
Cuda device memory types

- R/W per-grid **global memory**
- R only per-grid **constant memory**
- **Shared memory** is an efficient means for threads to cooperate by sharing their input data and the intermediate results
Cuda device memory types

- **Shared memory** is allocated to thread blocks; all threads in a block can access variables in the shared memory locations allocated to the block.

- **Registers** are allocated to individual threads; each thread can only access its own registers.
By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.

The table presents the CUDA syntax for declaring program variables into the various types of device memory.

Such declaration gives a scope and lifetime to the variable.

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic Variable</td>
<td>register</td>
<td>thread</td>
<td>kernel</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong></td>
<td>int</td>
<td>shared</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>int</td>
<td>global</td>
<td>grid</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong></td>
<td>int</td>
<td>constant</td>
<td>grid</td>
</tr>
</tbody>
</table>
Use of device memories

- **Global memory** is large but slow, whereas the **shared memory** is small but fast.

- A common strategy is to partition the data into subsets called **tiles** such that each **tile** fits into the shared memory.
  - Size of **tiles** is chosen so they can fit into the **shared memory**.
  - Each **tile** corresponds to a block of **threads**:
    - Load the **tile** from global memory to shared memory.
    - Use multiple **threads** to exploit parallelism in memory access.
    - Execute the computation on the **tile** in the shared memory; each **thread** can use a memory element more than once.
    - Copy results from share memory to global memory.
Threads and blocks

Summing up

A CUDA kernel is executed by a grid of threads

- All threads in a grid run the same kernel code (SPMD)
- Each thread has an index that it uses to compute memory addresses and make control decisions

```c
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```

...
Threads and blocks

Summing up

- Divide thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate

```
Thread Block 0
0 1 2 254 255
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];

Thread Block 1
0 1 2 254 255
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];

Thread Block N-1
0 1 2 254 255
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
```
Matrix Multiplication Example

The matrix multiplication example illustrates the basic features of memory and thread management in CUDA programs:

- Leave shared memory usage until later
- Local, register usage
- Thread ID usage
- Memory data transfer API between host and device
- Assume square matrix for simplicity
Matrix Multiplication Example

- Let us consider a matrix 4x4.
- Let us consider 4 blocks 2x2.
- The four thread of block (0,0) compute $P_{d_{0,0}}$, $P_{d_{1,0}}$, $P_{d_{0,1}}$ and $P_{d_{1,1}}$.
- Arrows show which elements of $M_d$ and $N_d$ are used by thread (0,0) and (1,0) of block (0,0).
Matrix Multiplication Example

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

    Pd[Row*Width+Col] = Pvalue;
}
```
Matrix Multiplication Example

- The table shows which elements are accessed in the global memory by all threads of block (0,0).
- Each thread accesses 4 elements of Md and 4 elements of Nd, with a clear superimposition.
- Example: thread (0,0) and thread (1,0) access the whole row 0 of M.

<table>
<thead>
<tr>
<th>Access order</th>
<th>( P_{0,0} ) thread ( _{0,0} )</th>
<th>( P_{1,0} ) thread ( _{1,0} )</th>
<th>( P_{0,1} ) thread ( _{0,1} )</th>
<th>( P_{1,1} ) thread ( _{1,1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>M(<em>{0,0}) * N(</em>{0,0})</td>
<td>M(<em>{0,0}) * N(</em>{1,0})</td>
<td>M(<em>{0,1}) * N(</em>{0,0})</td>
<td>M(<em>{0,1}) * N(</em>{1,0})</td>
<td></td>
</tr>
<tr>
<td>M(<em>{1,0}) * N(</em>{0,1})</td>
<td>M(<em>{1,0}) * N(</em>{1,1})</td>
<td>M(<em>{1,1}) * N(</em>{0,1})</td>
<td>M(<em>{1,1}) * N(</em>{1,1})</td>
<td></td>
</tr>
<tr>
<td>M(<em>{2,0}) * N(</em>{0,2})</td>
<td>M(<em>{2,0}) * N(</em>{1,2})</td>
<td>M(<em>{2,1}) * N(</em>{0,2})</td>
<td>M(<em>{2,1}) * N(</em>{1,2})</td>
<td></td>
</tr>
<tr>
<td>M(<em>{3,0}) * N(</em>{0,3})</td>
<td>M(<em>{3,0}) * N(</em>{1,3})</td>
<td>M(<em>{3,1}) * N(</em>{0,3})</td>
<td>M(<em>{3,1}) * N(</em>{1,3})</td>
<td></td>
</tr>
</tbody>
</table>
Performance

- Each **thread** accesses to global memory to load matrix elements

- Each element is accessed twice, then if **threads** collaborate in accesses to the global memory we can reduce the traffic to the global memory
Performance

- The potential reduction in global memory traffic in the matrix multiplication example is proportional to the dimension of the blocks used.

- With NxN blocks, the potential reduction of global memory traffic would be $N$. 
Traffic to the global memory

- Threads have to collaborate to reduce the traffic to the global memory.
- The threads collaboratively load \( \textbf{Md} \) and \( \textbf{Nd} \) elements into the shared memory.
- The shared memory is small and the capacity can not be exceeded when loading \( \textbf{Md} \) and \( \textbf{Nd} \).
- This can be obtained by dividing the \( \textbf{Md} \) and \( \textbf{Nd} \) matrices into smaller tiles that can fit into the shared memory.
- In the simplest form, the tile dimensions equal those of the block.
Divide Md and Nd in tile

- Divide Md and Nd in 2x2 tile as shown
- the calculation of each dot product is performed in **two phases**
- In each phase the **threads** can collaboratively load the subset (one element of Md and one element of Nd) into the **shared memory**
Execution phases of threads

The shared memory array for the Md and Nd elements are called Mds and Nds.

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T₀₀</strong></td>
<td><strong>Md₀₀</strong></td>
</tr>
<tr>
<td></td>
<td>↓ Mds₀₀</td>
</tr>
<tr>
<td><strong>Md₂₀</strong></td>
<td><strong>Nd₀₂</strong></td>
</tr>
<tr>
<td></td>
<td>↓ Mds₀₀</td>
</tr>
<tr>
<td><strong>T₁₀</strong></td>
<td><strong>Md₁₀</strong></td>
</tr>
<tr>
<td></td>
<td>↓ Mds₁₀</td>
</tr>
<tr>
<td><strong>Md₃₀</strong></td>
<td><strong>Nd₁₂</strong></td>
</tr>
<tr>
<td></td>
<td>↓ Mds₁₀</td>
</tr>
<tr>
<td><strong>T₀₁</strong></td>
<td><strong>Md₀₁</strong></td>
</tr>
<tr>
<td></td>
<td>↓ Mds₀₁</td>
</tr>
<tr>
<td><strong>Md₂₁</strong></td>
<td><strong>Nd₀₃</strong></td>
</tr>
<tr>
<td></td>
<td>↓ Mds₀₁</td>
</tr>
<tr>
<td><strong>T₁₁</strong></td>
<td><strong>Md₁₁</strong></td>
</tr>
<tr>
<td></td>
<td>↓ Mds₁₁</td>
</tr>
<tr>
<td><strong>Md₃₁</strong></td>
<td><strong>Nd₁₃</strong></td>
</tr>
<tr>
<td></td>
<td>↓ Mds₁₁</td>
</tr>
</tbody>
</table>
Execution phases of threads

- At the beginning of **Phase 1**, the four threads of block(0,0) collaboratively load a tile of \( \text{Md} \) into shared memory.
- Similarly a tile of \( \text{Nd} \)
- Loaded values are used in the calculation of the dot product.
- **Remark** Each value in the shared memory is used **twice**.
- **Note** how \( \text{Md}_{0,1} \) and \( \text{Nd}_{1,0} \) are used
Execution phases of threads

In Phase 2 the tile for Md and Nd are loaded to complete the calculation of the dot product.

Note that in each phase, the same Mds and Nds are used.
kernel tiled function

kernel tiled function for the use of the shared memory

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, 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 Pd element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
}
```
Kernel tiled function

7. float Pvalue = 0;
   // Loop over the Md and Nd tiles required to compute the Pd element
8.   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
   // Collaborative loading of Md and Nd tiles into shared memory
9.       Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.      Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
11.     __syncthreads();
12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.       Pvalue += Mds[ty][k] * Nds[k][tx];
14.     synchthreads();
15. }
16. Pd[Row*Width+Col] = Pvalue;
Tiled Multiply

- Each **block** computes a submatrix $P_{d_{sub}}$ which dimensions are TILE_WIDTH
- Each **thread** computes an element of $P_{d_{sub}}$
Granularity Considerations

For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?

- For **8X8**, we have 64 threads per Block.
  Since each SM can take up to 768 threads, it can take up to 12 Blocks. However, each SM can only take up to 8 Blocks, only **512 threads** will go into each SM!

- For **16X16**, we have 256 threads per Block.
  Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve **full capacity** unless other resource considerations overrule.

- For **32X32**, we have 1024 threads per Block.
  Not even one can fit into an SM!