Distributed and Parallel Computing

Lecture 05

Alan P. Sexton

University of Birmingham

Spring 2019

Multi-Dimensional Kernels

Many problems are inherently multi-dimensional:

- 2 Dim: Image analysis, Matrix algebra etc.
- 3 Dim: Spatial sims, Fluid- or Thermo-dynamics, Weather etc.

CUDA provides support in the form of 2 and 3 dimensional kernels:

```
dim3 dimGrid2(GRID_WIDTH, GRID_HEIGHT);
dim3 dimBlock2(BLOCK_WIDTH, BLOCK_HEIGHT);
my2d_kernel<<<dimGrid, dimBlock>>>(...);
...
dim3 dimGrid3(GRID_WIDTH, GRID_HEIGHT, GRID_DEPTH);
dim3 dimBlock3(BLOCK_WIDTH, BLOCK_HEIGHT, BLOCK_DEPTH);
my3d_kernel<<<dimGrid, dimBlock>>>(...);
```

Multi-Dimensional Kernels

Many problems are inherently multi-dimensional:

- 2 Dim: Image analysis, Matrix algebra etc.
- 3 Dim: Spatial sims, Fluid- or Thermo-dynamics, Weather etc.

CUDA provides support in the form of 2 and 3 dimensional kernels:

```
dim3 dimGrid2(GRID_WIDTH, GRID_HEIGHT);
dim3 dimBlock2(BLOCK_WIDTH, BLOCK_HEIGHT);
my2d_kernel<<<dimGrid, dimBlock>>>(...);
...
dim3 dimGrid3(GRID_WIDTH, GRID_HEIGHT, GRID_DEPTH);
dim3 dimBlock3(BLOCK_WIDTH, BLOCK_HEIGHT, BLOCK_DEPTH);
my3d_kernel<<<dimGrid, dimBlock>>>(...);
```

- `dim3` is a `struct` with `x`, `y` and `z` fields and can take 1, 2 or 3 integer parameters in its constructor (missing parameters are initialised to 1).

Multi-Dimensional Kernels

Many problems are inherently multi-dimensional:

- 2 Dim: Image analysis, Matrix algebra etc.
- 3 Dim: Spatial sims, Fluid- or Thermo-dynamics, Weather etc.

CUDA provides support in the form of 2 and 3 dimensional kernels:

```
dim3 dimGrid2(GRID_WIDTH, GRID_HEIGHT);
dim3 dimBlock2(BLOCK_WIDTH, BLOCK_HEIGHT);
my2d_kernel<<<dimGrid, dimBlock>>>(...);
...
dim3 dimGrid3(GRID_WIDTH, GRID_HEIGHT, GRID_DEPTH);
dim3 dimBlock3(BLOCK_WIDTH, BLOCK_HEIGHT, BLOCK_DEPTH);
my3d_kernel<<<dimGrid, dimBlock>>>(...);
```

- `dim3` is a `struct` with `x`, `y` and `z` fields and can take 1, 2 or 3 integer parameters in its constructor (missing parameters are initialised to 1).
- The grid and block can have different dimensionalities

Multi-Dimensional Kernels

Many problems are inherently multi-dimensional:

- 2 Dim: Image analysis, Matrix algebra etc.
- 3 Dim: Spatial sims, Fluid- or Thermo-dynamics, Weather etc.

CUDA provides support in the form of 2 and 3 dimensional kernels:

```
dim3 dimGrid2(GRID_WIDTH, GRID_HEIGHT);
dim3 dimBlock2(BLOCK_WIDTH, BLOCK_HEIGHT);
my2d_kernel<<<dimGrid, dimBlock>>>(...);
...
dim3 dimGrid3(GRID_WIDTH, GRID_HEIGHT, GRID_DEPTH);
dim3 dimBlock3(BLOCK_WIDTH, BLOCK_HEIGHT, BLOCK_DEPTH);
my3d_kernel<<<dimGrid, dimBlock>>>(...);
```

- `dim3` is a `struct` with `x`, `y` and `z` fields and can take 1, 2 or 3 integer parameters in its constructor (missing parameters are initialised to 1).
- The grid and block can have different dimensionalities
- Note that the number of threads per block will be the product of the block dimensionalities and the number of blocks in the kernel will be the product of the grid dimensionalities

Built-in Variables available to Threads

Every thread has access to a number of variables: either `dim3` or `uint3` structs (`uint3` is like `dim3` except without constructor support)

- `gridDim`: `dim3`: the dimensions of the grid
- `blockDim`: `dim3`: the dimensions of the block
- `blockIdx`: `uint3`: the block index within the grid
- `threadIdx`: `uint3`: the thread index within the block

Thread Order

Because of issues of Global Memory Coalescing, Cache lines and Shared Memory Bank Conflicts, the ordering of threads, and the layout of vectors and matrices in the C language is important:

Threads are ordered in a block first by their `z` index, then by their `y`, then by their `x` index. Thus, in a $2 \times 2 \times 2$ dimensional block:

thread 0 has `threadIdx.z` $= 0$, `threadIdx.y` $= 0$, `threadIdx.x` $= 0$
thread 1 has `threadIdx.z` $= 0$, `threadIdx.y` $= 0$, `threadIdx.x` $= 1$
thread 2 has `threadIdx.z` $= 0$, `threadIdx.y` $= 1$, `threadIdx.x` $= 0$
thread 3 has `threadIdx.z` $= 0$, `threadIdx.y` $= 1$, `threadIdx.x` $= 1$
thread 4 has `threadIdx.z` $= 1$, `threadIdx.y` $= 0$, `threadIdx.x` $= 0$
thread 5 has `threadIdx.z` $= 1$, `threadIdx.y` $= 0$, `threadIdx.x` $= 1$
thread 6 has `threadIdx.z` $= 1$, `threadIdx.y` $= 1$, `threadIdx.x` $= 0$
thread 7 has `threadIdx.z` $= 1$, `threadIdx.y` $= 1$, `threadIdx.x` $= 1$

Memory Layout

Multi-dimensional arrays, `A[k][j][i]`, are ordered by their inner index (`k`) first, then their middle index (`j`), then by their outer index (`i`)

Thus if `A` is a $2 \times 2 \times 2$ matrix, it would be layed out in consecutive memory locations as:

```
0:  A[0][0][0]
1:  A[0][0][1]
2:  A[0][1][0]
3:  A[0][1][1]
4:  A[1][0][0]
5:  A[1][0][1]
6:  A[1][1][0]
7:  A[1][1][1]
```

Multi-Dimensional Indexing

Even when working multidimensionally, we often have to explicitly apply threads, which have their grid and block dimensionality, to the 2- or 3-dimensional structures of the dimensionality of our problem domain

For a problem domain data structure D of dimension $N \times N$, and the block dimensionality of size $K \times K$, we may, in our kernel access it as follows:

```
int i = blockIdx.x * K + threadIdx.x;
int j = blockIdx.y * K + threadIdx.y;

// assuming D is just a pointer to a block of memory:
... D[i + j*N] ...

// assuming D has been declared as a 2-dimensional C array:
... D[j][i] ...
```

Aside: global thread number in a 3-D model

Sometimes you need to know the thread number of a thread in the whole grid. The most general case is in a 3-d grid of 3-d blocks:

```
__device__ int getGlobalIdx_3D_3D()
{
  int bId = blockIdx.x
            + gridDim.x * blockIdx.y
            + gridDim.x * gridDim.y * blockIdx.z;
  int tId = bId * blockDim.x * blockDim.y * blockDim.z
            + threadIdx.x
            + blockDim.x * threadIdx.y
            + blockDim.x * blockDim.y * threadIdx.z;
  return tId;
}
```

Note that in a 1 or 2 dimensional block or grid configuration, all the `*Dim.*` values for the unused dimensions will have value 1 and all the `*Idx.*` values for those dimensions will have value 0

Coalesced Global Memory Accesses

```
int i = blockIdx.x * K + threadIdx.x;
int j = blockIdx.y * K + threadIdx.y;

... D[i + j*N] ...
```

- thread 0 of the block has `threadIdx.x` $= 0$ and `threadIdx.y` $= 0$
- thread 1 of the block has `threadIdx.x` $= 1$ and `threadIdx.y` $= 0$
- ...

Here i is the fastest changing index of the threads in the block, and an increment of 1 in i contributes a jump of 1 word location in D, or consecutive threads are accessing consecutive words. Hence the access is coalesced.

Coalesced Global Memory Accesses

```
int i = blockIdx.x * K + threadIdx.x;
int j = blockIdx.y * K + threadIdx.y;

... D[j + i*N] ...
```

- thread 0 of the block has `threadIdx.x` $= 0$ and `threadIdx.y` $= 0$
- thread 1 of the block has `threadIdx.x` $= 1$ and `threadIdx.y` $= 0$
- ...

Here i is the fastest changing index of the threads in the block, and an increment of 1 in i contributes a jump of N word locations in D, that is thread 0, 1, 2... is accessing $D[0], D[N], D[2N], \ldots$. Hence the access is **NOT** coalesced.

Shared Memory Bank Conflicts

Here, to avoid shared memory bank conflicts, each thread in a warp should, as far as possible, access a different memory bank in the same memory access instruction.

```
// Block dimension is K times K
    int x = threadIdx.x;    int y = threadIdx.y;
    __shared__ int tile[K][K];

    ... tile[y][x] ...
    ... tile[x][y] ...
```

- If K is 32, then the address that `tile[y][x]` accesses is `x+K*y` words from the start of `tile`.
- If `tile` starts on a 32 word boundary, this accesses shared memory bank $(x + Ky) \mod 32$
- `tile[y][x]`: T 0, 1, . . . : addr 0, 1, 2, . . . ≡ bank 0, 1, 2, . . .
- `tile[x][y]`: T 0, 1, . . . : addr 0, 32, 64, . . . ≡ bank 0, 0, 0, . . .

Shared Memory Bank Conflicts

Here, to avoid shared memory bank conflicts, each thread in a warp should, as far as possible, access a different memory bank in the same memory access instruction.

```
// Block dimension is K times K
    int x = threadIdx.x;    int y = threadIdx.y;
    __shared__ int tile[K][K];

    ... tile[y][x] ...
    ... tile[x][y] ...
```

- If K is 32, then the address that `tile[y][x]` accesses is `x+K*y` words from the start of `tile`.
- If `tile` starts on a 32 word boundary, this accesses shared memory bank $(x + Ky) \mod 32$
- `tile[y][x]`: T 0, 1, . . . : addr 0, 1, 2, . . . $\equiv$ bank 0, 1, 2, . . .
- `tile[x][y]`: T 0, 1, . . . : addr 0, 32, 64, . . . $\equiv$ bank 0, 0, 0, . . .
- What if K is $32k$ for some positive integer k?

Shared Memory Bank Conflicts

Here, to avoid shared memory bank conflicts, each thread in a warp should, as far as possible, access a different memory bank in the same memory access instruction.

```
// Block dimension is K times K
    int x = threadIdx.x;    int y = threadIdx.y;
    __shared__ int tile[K][K];

    ... tile[y][x] ...
    ... tile[x][y] ...
```

- If K is 32, then the address that `tile[y][x]` accesses is `x+K*y` words from the start of `tile`.
- If `tile` starts on a 32 word boundary, this accesses shared memory bank $(x + Ky) \mod 32$
- `tile[y][x]`: T 0, 1, ...: addr 0, 1, 2, ... $\equiv$ bank 0, 1, 2, ...
- `tile[x][y]`: T 0, 1, ...: addr 0, 32, 64, ... $\equiv$ bank 0, 0, 0, ...
- What if K is $32k$ for some positive integer k?
- What if K is 16?

Shared Memory Bank Conflicts

Here, to avoid shared memory bank conflicts, each thread in a warp should, as far as possible, access a different memory bank in the same memory access instruction.

```
// Block dimension is K times K
    int x = threadIdx.x;    int y = threadIdx.y;
    __shared__ int tile[K][K];

    ... tile[y][x] ...
    ... tile[x][y] ...
```

- If K is 32, then the address that `tile[y][x]` accesses is `x+K*y` words from the start of `tile`.
- If `tile` starts on a 32 word boundary, this accesses shared memory bank $(x + Ky) \mod 32$
- `tile[y][x]`: T 0, 1, ...: addr 0, 1, 2, ... $\equiv$ bank 0, 1, 2, ...
- `tile[x][y]`: T 0, 1, ...: addr 0, 32, 64, ... $\equiv$ bank 0, 0, 0, ...
- What if K is $32k$ for some positive integer k?
- What if K is 16?
- What if K is 32 but declaration is: `tile[K][K+1]`?

Transpose

Transpose is a simple, but important operation on 2-dimensional data types:

- For all `i, j`: swap `M[i][j]` with `M[j][i]`

```
// Matrix size (N x N)
#define N 1024

void transpose_HOST(int in[], int out[])
{
    for (int j = 0; j < N; ++j) // Loop over Rows
        for (int i = 0; i < N; ++i) // Loop over Columns
            out[j + i*N] = in[i+j*N];
}
```

Serial: 1 thread

```
#define N 1024

__global__ void transpose_serial
        (int in[], int out[])
{
    for (int j = 0; j < N; ++j) // Loop over Rows
        for (int i = 0; i < N; ++i) // Loop over Columns
            out[j + i * N] = in[i + j * N];
}
...
    transpose_serial<<<1, 1>>>(d_in, d_out);
```

Run the profiler

Run the profiler from within nsight, or from the command line with:

```
nvvp ./Transpose
```

- Make sure you are building a **Release** and not a **Debug** build
- If in nsight, make sure you have chosen the "Profile" perspective (top right of window)
- In the analysis tab at bottom, click on "`Examine Individual Kernels`", select a Kernel in the "`Results`" window, then click on "`Perform Additional Analysis`"
- At the bottom of the Results window, you can select a kernel and see the detailed results in the "`Properties`" window

Results

Some fields only appear when relevant.

Duration:	Time this kernel took
Register/Thread:	# registers allocated to each thread
Shared Memory/Block:	How much shared memory allocated to each block
Global Load Efficiency:	% of loads used (c.f. caches)
Global Store Efficiency:	% of stores used (c.f. caches)
Shared Efficiency:	% of shared accesses used (c.f. 32-word bus)
Warp Execution Efficiency:	100% = no divergence
Non-Predicated Warp Execution Efficiency:	100% = no divergence and no predication
Occupancy Achieved:	% of maximum # of warps active each active cycle on average
Occupancy Theoretical:	Peak achievable with this config
Occupancy Limiter:	Any config issue that limits occupancy

Result: Serial

Factor	GTX 1080 Ti	GTX 960
Duration:	160ms	192ms
Register/Thread:	20	15
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	12.5%	12.5%
Global Store Efficiency:	12.5%	12.5%
Shared Efficiency:	n/a	n/a
Warp Execution Efficiency:	3.1%	3.1%
Non-Predicated Warp Execution Efficiency:	3.1%	3.1%
Occupancy Achieved:	1.6%	1.6%
Occupancy Theoretical:	50%	50%
Occupancy Limiter:	Grid Size	Grid Size

1 thread per row

```
__global__ void transpose_thread_per_row
        (int in[], int out[])
{
    int i = threadIdx.x;
    for (int j = 0; j < N; ++j) // Loop over Rows
        out[j + i * N] = in[i + j * N];
}
...
    transpose_thread_per_row<<<1, N>>>(d_in, d_out);
```

Result: Thread per Row

Factor	GTX 1080 Ti	GTX 960
Duration:	1.18ms	1.59ms
Register/Thread:	17	15
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	100%	100%
Global Store Efficiency:	12.5%	12.5%
Shared Efficiency:	n/a	n/a
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	49.9%	49.9%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

1 thread per element

```
#define N 1024
#define K 32

__global__ void transpose_thread_per_element
        (int in[], int out[])
{
    int i = blockIdx.x * K + threadIdx.x;
    int j = blockIdx.y * K + threadIdx.y;

    out[j + i * N] = in[i + j * N];
}
...
    dim3 blocks(N / K, N / K);
    dim3 threads(K, K);
    transpose_thread_per_element<<<blocks, threads>>>(d_in,
        d_out);
```

Result: Thread per Element

Factor	GTX 1080 Ti	GTX 960
Duration:	$58\mu s$	$241\mu s$
Register/Thread:	8	8
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	100%	100%
Global Store Efficiency:	12.5%	12.5%
Shared Efficiency:	n/a	n/a
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	67.5%	68.3%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

Copy thread per element, coalesced

```
__global__ void copy_thread_per_element_coalesced
        (int in[], int out[])
{
    int i = blockIdx.x * K + threadIdx.x;
    int j = blockIdx.y * K + threadIdx.y;

    out[i + j * N] = in[i + j * N];
}
...
    copy_thread_per_element_coalesced<<<blocks, threads>>>(
        d_in, d_out);
```

Result: Copy thread per element, coalesced

Factor	GTX 1080 Ti	GTX 960
Duration:	23μs	98μs
Register/Thread:	8	8
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	100%	100%
Global Store Efficiency:	100%	100%
Shared Efficiency:	n/a	n/a
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	72.4%	78.8%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

Copy — thread per element, non-coalesced

```
__global__ void copy_thread_per_element_non_coalesced
        (int in[], int out[])
{
    int i = blockIdx.x * K + threadIdx.x;
    int j = blockIdx.y * K + threadIdx.y;

    out[j + i * N] = in[j + i * N];
}
...
    copy_thread_per_element_non_coalesced<<<blocks, threads
        >>>(d_in, d_out);
```

Result: Copy thread per element, non-coalesced

Factor	GTX 1080 Ti	GTX 960
Duration:	89μs	341μs
Register/Thread:	8	8
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	12.5%	12.5%
Global Store Efficiency:	12.5%	12.5%
Shared Efficiency:	n/a:	n/a
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	72.4%	70.3%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

1 thread per element tiled

```
__global__ void transpose_thread_per_element_tiled
        (int in[], int out[])
{
    int in_corner_i = blockIdx.x * K;
    int in_corner_j = blockIdx.y * K;

    int out_corner_i = blockIdx.y * K;
    int out_corner_j = blockIdx.x * K;

    int x = threadIdx.x;
    int y = threadIdx.y;

    __shared__ int tile[K][K];

    tile[y][x] = in[(in_corner_i+x) + (in_corner_j+y)*N];
    __syncthreads();
    out[(out_corner_i+x) + (out_corner_j+y)*N] = tile[x][y];
}
...
    transpose_thread_per_element_tiled<<<blocks, threads>>>(
        d_in, d_out);
```

Result: 1 thread per element tiled

Factor	GTX 1080 Ti	GTX 960
Duration:	38μs	161μs
Register/Thread:	11	12
Shared Memory/Block:	4 KiB	4 KiB
Global Load Efficiency:	100%	100%
Global Store Efficiency:	100%	100%
Shared Efficiency:	6.1%	6.1%
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	84.8%	82.4%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

1 thread per element tiled and padded

```
__global__ void transpose_thread_per_element_tiled_padded
        (int in[], int out[])
{
    int in_corner_i = blockIdx.x * K;
    int in_corner_j = blockIdx.y * K;

    int out_corner_i = blockIdx.y * K;
    int out_corner_j = blockIdx.x * K;

    int x = threadIdx.x;
    int y = threadIdx.y;

    __shared__ int tile[K][K + 1];

    tile[y][x] = in[(in_corner_i+x) + (in_corner_j+y)*N];
    __syncthreads();
    out[(out_corner_i+x) + (out_corner_j+y)*N] = tile[x][y];
}
...
    transpose_thread_per_element_tiled_padded<<<blocks,
        threads>>>(d_in, d_out);
```

Result: 1 thread per element tiled and padded

Factor	GTX 1080 Ti	GTX 960
Duration:	23μs	100μs
Register/Thread:	11	10
Shared Memory/Block:	4.125 KiB	4.125 KiB
Global Load Efficiency:	100%	100%
Global Store Efficiency:	100%	100%
Shared Efficiency:	100%	100%
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	?%	90.4%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

Waits on barrier syncs?

Lots of warps in block, maybe delays while they have to wait for all warps to sync?

- Try reducing block size ($K = 16$)
- More blocks, fewer warps per block: different blocks can run on the same SM without waiting for each other

But: beware of limiting factors:

Factor	GTX 1080 Ti	GTX 960
# Threads/Block	64	32
# Thread/SM	2048	2048
Registers/Block	65536	65536
Shared Mem/Block	49142 bytes	49152 bytes

Result: 1 thread per element tiled and padded, K=16

Factor	GTX 1080 Ti	GTX 960
Duration:	23μs	103μs
Register/Thread:	11	10
Shared Memory/Block:	1.062 KiB	1.062 KiB
Global Load Efficiency:	100%	100%
Global Store Efficiency:	100%	100%
Shared Efficiency:	50%	50%
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	88.7%	92.3%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

Reduced waiting on `__syncthreads()` balanced out by reduced shared efficiency

Matrix-Matrix Multiplication

$$\begin{bmatrix} A_{11} & \dots & A_{1n} \\ \vdots & & \vdots \\ A_{m1} & \dots & A_{mn} \end{bmatrix} \begin{bmatrix} B_{11} & \dots & B_{1p} \\ \vdots & & \vdots \\ B_{n1} & \dots & B_{np} \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{n} A_{1k} B_{k1} & \dots & \displaystyle\sum_{k=1}^{n} A_{1k} B_{kp} \\ \vdots & & \vdots \\ \sum_{k=1}^{n} A_{mk} B_{k1} & \dots & \displaystyle\sum_{k=1}^{n} A_{mk} B_{kp} \end{bmatrix}$$

- For simplicity we will restrict ourselves to square matrices ($m = n = p$).

Kernel Invocation

- We use a 2-dimensional layout to match the matrix structure.
- Each thread will calculate a single component of the result.

```
#define BLOCK_WIDTH 16
...
int numBlocks = (n - 1) / BLOCK_WIDTH + 1 ;
dim3 dimGrid(numBlocks, numBlocks) ;
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH) ;
simpleMMM<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, n) ;
...
```

Simple Matrix-Matrix Multiplication Kernel

For a single component of the result matrix:

$$(AB)_{\mathrm{row,col}} = \sum_{k=1}^{n} A_{\mathrm{row},k} B_{k,\mathrm{col}}$$

$$= \begin{bmatrix} A_{\mathtt{row},1} & \cdots & A_{\mathrm{row},n} \end{bmatrix} \begin{bmatrix} B_{1,\mathrm{col}} \\ \vdots \\ B_{n,\mathrm{col}} \end{bmatrix}$$

```
__global__ void simpleMMM(float *d_A, float *d_B,
                          float *d_C, int n)
{
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    if (row < n && col < n)
    {
        float val = 0 ;
        for (int k = 0 ; k < n ; k++)
            val += d_A[row*n + k] * d_B[k*n + col] ;
        d_C[row*n + col] = val ;
    }
}
```

Memory Access Efficiency

- Note: multiple threads reading the same global addresses:
  - thread 0,0 combines row 0 of A with column 0 of B
  - thread 0,1 combines row 0 of A with column 1 of B
  - ...
- Recall the discussion previously (lecture 04) of the *Compute to Global Memory Access (CGMA)* ratio.
- What is the CGMA ratio of the inner loop of `simpleMMM`?

Memory Access Efficiency

- Note: multiple threads reading the same global addresses:
  - thread 0,0 combines row 0 of A with column 0 of B
  - thread 0,1 combines row 0 of A with column 1 of B
  - ...
- Recall the discussion previously (lecture 04) of the *Compute to Global Memory Access (CGMA)* ratio.
- What is the CGMA ratio of the inner loop of `simpleMMM`?
- Each iteration, 2 global word memory accesses, one floating mult and one floating add: hence a CGMA ratio of 1.
- GTX1080 Ti: 5505MHz, 352bits bus $\Rightarrow$ 242.22GB/s
- GTX960: 3600MHz, 128 bits bus $\Rightarrow$ 57.6 GB/s

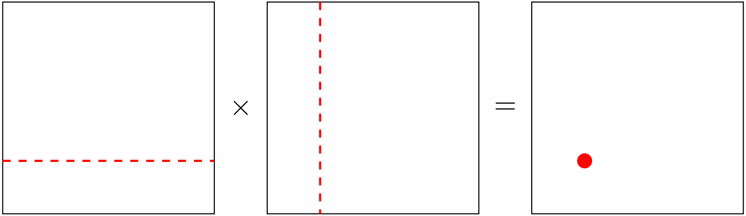
Memory Access Efficiency

- Note: multiple threads reading the same global addresses:
  - thread 0,0 combines row 0 of A with column 0 of B
  - thread 0,1 combines row 0 of A with column 1 of B
  - ...
- Recall the discussion previously (lecture 04) of the *Compute to Global Memory Access (CGMA)* ratio.
- What is the CGMA ratio of the inner loop of `simpleMMM`?
- Each iteration, 2 global word memory accesses, one floating mult and one floating add: hence a CGMA ratio of 1.
- GTX1080 Ti: 5505MHz, 352bits bus $\Rightarrow$ 242.22GB/s
- GTX960: 3600MHz, 128 bits bus $\Rightarrow$ 57.6 GB/s
- If global memory bandwidth is 200GB/s, and 4 bytes/word, then we are limited to 50 Gflops, when the hardware could support maybe 1500 Gflops.

Memory Access Efficiency

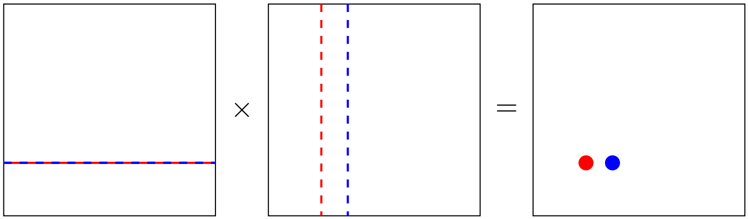
- Note: multiple threads reading the same global addresses:
  - thread 0,0 combines row 0 of A with column 0 of B
  - thread 0,1 combines row 0 of A with column 1 of B
  - ...
- Recall the discussion previously (lecture 04) of the *Compute to Global Memory Access (CGMA)* ratio.
- What is the CGMA ratio of the inner loop of `simpleMMM`?
- Each iteration, 2 global word memory accesses, one floating mult and one floating add: hence a CGMA ratio of 1.
- GTX1080 Ti: 5505MHz, 352bits bus $\Rightarrow$ 242.22GB/s
- GTX960: 3600MHz, 128 bits bus $\Rightarrow$ 57.6 GB/s
- If global memory bandwidth is 200GB/s, and 4 bytes/word, then we are limited to 50 Gflops, when the hardware could support maybe 1500 Gflops.
- Need to work in shared memory.

Matrix Multiplication Graphically



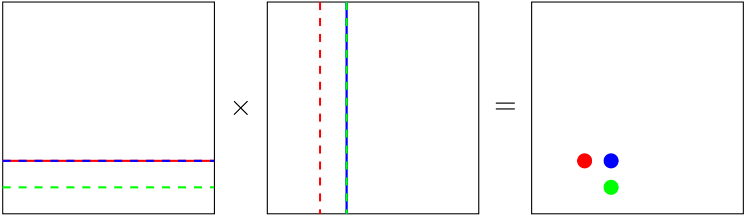
- Each element of C is made up of the dot-product of one row of A and one column of B

Matrix Multiplication Graphically



- Each element of C is made up of the dot-product of one row of A and one column of B
- Each row of A is read once for every column of B, i.e. n times

Matrix Multiplication Graphically



- Each element of C is made up of the dot-product of one row of A and one column of B
- Each row of A is read once for every column of B, i.e. n times
- Each column of B is read once for every row of A, i.e. n times

Tiled Matrix-Matrix Multiplication Idea

Work in tiles:

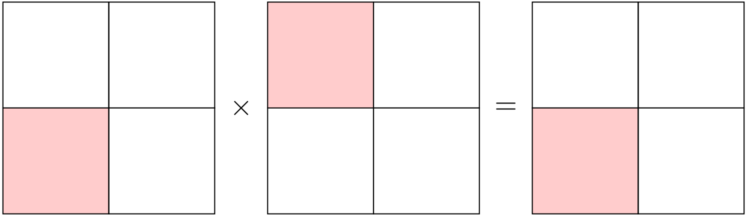
- Each thread calculates same value as before
- But reorganise into nested loop over tiles. Instead of:

```
for each dot product pair in matrix for this location
    accumulate dot product result
```

do the following:

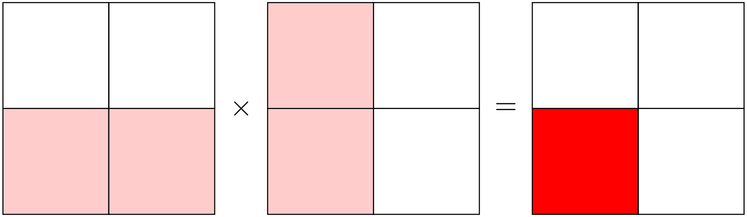
```
for each tile
    copy global tile to shared tile
    for each dot product pair in tile for this location
        accumulate dot product result
```

Tiled Matrix Multiplication Graphically



- Example: assume tile size is 32x32 and matrices are 64x64, i.e. matrices are 2x2 tiles
- When calculating the (row=1, col=0) tile of C, first copy to shared memory the (1,0) tile of A and the (0,0) tile of B and calculate the **PARTIAL** dot products for all cells of the (1,0) tile of C ...

Tiled Matrix Multiplication Graphically



- Example: assume tile size is 32x32 and matrices are 64x64, i.e. matrices are 2x2 tiles
- When calculating the (row=1, col=0) tile of C, first copy to shared memory the (1,0) tile of A and the (0,0) tile of B and calculate the **PARTIAL** dot products for all cells of the (1,0) tile of C ...
- ...then copy the (1,1) tile of A and the (1,0) tile of B and complete calculating the dot products for all cells of the (1,0) tile of C

Tiled Matrix-Matrix Multiplication

```
__global__ void simpleMMM(float *d_A, float *d_B,
                          float *d_C, int n)
{
    __shared__ float As[TILE_DIM][TILE_DIM];
    __shared__ float Bs[TILE_DIM][TILE_DIM];
    int bx = blockIdx.x;     int by = blockIdx.y;
    int tx = threadIdx.x;    int ty = threadIdx.y;
    int row = by * TILE_DIM + ty;
    int col = bx * TILE_DIM + tx;
//  can't do test because of sync: pad matrices instead
//  if (row < n && col < n)
    {   float val = 0 ;
        for (int m = 0 ; m < n/TILE_DIM ; m++)
        {
            As[ty][tx] = d_A[row * n + m * TILE_DIM + tx];
            Bs[ty][tx] = d_B[(m * TILE_DIM + ty) * n + col];
            __syncthreads();
            for (int k = 0 ; k < TILE_DIM ; k++)
                val += As[ty][k] * Bs[k][tx] ;
            __syncthreads();
        }
        d_C[row*n + col] = val ;
    }
}
```

Results

- Each thread is still calculating the same value as before.
- Calculation is now divided into phases: 1 phase per tile
- Each tile copied from global to shared memory allows `TILE_DIM` threads to run without further global memory accesses
- Hence number of global memory accesses is divided by `TILE_DIM`

Results

- Each thread is still calculating the same value as before.
- Calculation is now divided into phases: 1 phase per tile
- Each tile copied from global to shared memory allows `TILE_DIM` threads to run without further global memory accesses
- Hence number of global memory accesses is divided by `TILE_DIM`
- `TILE_DIM` $= 16 \Rightarrow$ 200Gb memory bandwidth can support $(200/4) \times 16 = 800 \mathit{GFlops}$

Results

- Each thread is still calculating the same value as before.
- Calculation is now divided into phases: 1 phase per tile
- Each tile copied from global to shared memory allows `TILE_DIM` threads to run without further global memory accesses
- Hence number of global memory accesses is divided by `TILE_DIM`
- $\texttt{TILE\_DIM} = 16 \Rightarrow$ 200Gb memory bandwidth can support $(200/4) \times 16 = 800\,GFlops$
- Modern GPUs can support square tile dimensions of size 32, i.e. 1600GFlops.