Matrix multiplication with Cuda

Jochen Kreutz (JSC)
Distribution of work

- Each thread computes one element of the result matrix C
- \( n \times n \) threads will be needed
- Indexing of threads corresponds to 2d indexing of the matrices
- Thread\((x, y)\) will calculate element \( C(x, y) \) using row \( y \) of A and column \( x \) of B
Distribution of work

- Block dimensions are limited, hence several thread blocks will be needed
- Use 2d execution grid with \( k \times k \) blocks

Result matrix \( C \) (\( n \times n \) elements)
Distribution of work

- Use 2d execution grid with \( k \times k \) blocks
- Use 2d thread blocks with fixed block size \((m \times m)\)
- \( k = \frac{n}{m} \) (\( n \) divisible by \( m \))
- \( k = \frac{n}{m} + 1 \) (\( n \) not divisible by \( m \))

Result matrix \( C \) (\( n \times n \) elements)
Define dimensions of thread block

```c
dim3 blockDim( size_t blockDimX, size_t blockDimY, size_t blockDimZ );
```

On Jureca (Tesla K80):
- *Max. dim. of a block*: 1024 x 1024 x 64
- *Max. number of threads per block*: 2048

Example:
```c
// Create 3D thread block with 512 threads
dim3 blockDim(16, 16, 2);
```
Define dimensions of grid

On Jureca (Tesla K80):

- Max. dim. of a grid: \((2147483647, 65535, 65535)\)

Example:

```c
// Dimension of problem: \(nx \times ny = 1000 \times 1000\)

dim3 blockDim(16, 16) // Don't need to write \(z = 1\)
int gx = (nx % blockDim.x==0) ? nx / blockDim.x : nx / blockDim.x + 1
int gy = (ny % blockDim.y==0) ? ny / blockDim.y : ny / blockDim.y + 1

dim3 gridDim(gx, gy);
```

Watch out!
# Calling the kernel

<table>
<thead>
<tr>
<th>Define dimensions of thread block</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dim3 blockDim(size_t blockDimX, size_t blockDimY, size_t blockDimZ)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Define dimensions of execution grid</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dim3 gridDim(size_t gridDimX, size_t gridDimY, size_t gridDimZ)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Launch the kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kernel&lt;&lt;&lt;dim3 gridDim, dim3 blockDim&gt;&gt;&gt;([arg]*)</code></td>
</tr>
</tbody>
</table>
Kernel (CUDA)

Kernel function

```c
__global__ void mm_kernel(float* A, float* B, float* C, int n)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < n && col < n) {
        for (int i = 0; i < n; ++i) {
            C[row * n + col] += A[row * n + i] * B[i * n + col];
        }
    }
}

mm_kernel<<<dimGrid, dimBlock>>> (d_a, d_b, d_c, n);
```
Exercise

Simple Cuda MM implementation

.../exercises/tasks/Cuda_MM_simple
Limiting Factor

Kernel function

```c
void mm_kernel ( float* A, float* B, float* C, int n )
{
    for ( int k = 0; k < n; ++k){
        C[i * n + j] += A[i * n + k] * B[k * n + j];
    }
}
```

- One floating point operation per memory access
- One double: 8 bytes
- Limited global memory bandwidth

- Check hints from Visual Profiler for further performance issues
Limiting Factor

- Check hints from Visual Profiler for further performance issues
GPU memory (schematics)
Using shared memory

Allocate shared memory

// allocate vector in shared memory
__shared__ float[size];

// can also define multi-dimensional arrays:
// BLOCK_SIZE is length (and width) of a thread block here
__shared__ float Msub[BLOCK_SIZE][BLOCK_SIZE];

Copy data to shared memory

// fetch data from global to shared memory
Msub[threadIdx.y][threadIdx.x] = M[TidY * width + TidX];

Synchronize threads

// ensure that all threads within a block had time to read / write data
__syncthreads();
Matrix-matrix multiplication with blocks

\[ C_{kl} = \sum_{i=1}^{N} A_{ki} B_{il} \]

\[ C_{kl} = \sum_{i=1}^{N/2} A_{ki} B_{il} + \sum_{i=N/2+1}^{N} A_{ki} B_{il} \]
Matrix-matrix multiplication with blocks

\[ C_{kl} = \sum_{i=1}^{N} A_{ki} B_{il} \]

\[ C_{kl} = \sum_{i=1}^{N/2} A_{ki} B_{il} \]

\[ + \sum_{i=N/2+1}^{N} A_{ki} B_{il} \]

For each element
- Set result to zero
- For each pair of blocks
  - Copy data
  - Do partial sum
  - Add result of partial sum to total
An Example

\[
A = \begin{bmatrix}
1 & 2 & 3 & 4 \\
4 & 1 & 2 & 3 \\
3 & 4 & 1 & 2 \\
2 & 3 & 4 & 1
\end{bmatrix}
\quad B = \frac{1}{40} \begin{bmatrix}
-9 & 11 & 1 & 1 \\
1 & -9 & 11 & 1 \\
1 & 1 & -9 & 11 \\
11 & 1 & 1 & -9
\end{bmatrix}
\quad C = AB
\]

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
4 & 1 \\
3 & 4 \\
2 & 3
\end{bmatrix}
\quad B = \frac{1}{40} \begin{bmatrix}
B_{11} & B_{12} \\
-9 & 11 \\
1 & -9 \\
11 & 1
\end{bmatrix}
\quad C = \begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21}
\]

\[
= \frac{1}{40} \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} -9 & 11 \\ 1 & -9 \end{bmatrix} + \frac{1}{40} \begin{bmatrix} 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 11 & 1 \end{bmatrix}
\]

\[
= \frac{1}{40} \begin{bmatrix} -9+2 & 11-18 \\ -36+1 & 44-9 \end{bmatrix} + \frac{1}{40} \begin{bmatrix} 3+44 & 3+4 \\ 2+33 & 2+3 \end{bmatrix}
\]

\[
= \frac{1}{40} \begin{bmatrix} -7 & -7 \\ -35 & 35 \end{bmatrix} + \frac{1}{40} \begin{bmatrix} 47 & 7 \\ 35 & 5 \end{bmatrix}
\]

\[
= \frac{1}{40} \begin{bmatrix} 40 & 0 \\ 0 & 40 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\]

Do \(C_{12}, C_{13},\) and \(C_{14}\) the same way.
Blockwise Matrix-Matrix Multiplication
Blockwise Matrix-Matrix Multiplication
Blockwise Matrix-Matrix Multiplication
Blockwise Matrix-Matrix Multiplication
Blockwise Matrix-Matrix Multiplication

\[
\begin{align*}
\text{Block 1} & \quad = \quad \text{Block 2} \\
\end{align*}
\]
Blockwise Matrix-Matrix Multiplication

Thread block loops over blocks in blue and yellow matrix:
  Calculate upper left corner
  Load data into shared memory
  Do calculation (one thread is still responsible for an element)
  Add partial sum to result
Exercise

Shared memory Cuda MM implementation

.../exercises/tasks/Cuda_MM_shared