CUDA Coding Example: Matrix Multiply

Mark Greenstreet and Ian M. Mitchell

CpSc 418 – March 2, 2018

Table of Contents

Outline

Matrix Indexing

Consider matrix A with m rows and n columns.

- We say $A \in \mathbb{R}^{m \times n}$.
- Elements are written as $a_{i,j}$ where i is the row and j the column.
- Whole matrix can be written as

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

- What would matrix $B \in \mathbb{R}^{n \times p}$ look like?

Remember: “rows comma columns”.

Matrix Multiply

Consider matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$

- Then the product $C = AB$ is such that $C \in \mathbb{R}^{m \times p}$
- The elements of C are given by

$$c_{i,k} = \sum_{j=0}^{n-1} a_{i,j} b_{j,k}$$

- How many summations are needed?
- What is the total computational cost (including constants)?
- How much input data?
- How much output data?

Outline

Matrix Indexing: Code Version

Two obvious options to store a 2D array of data in 1D memory.
Option 1: Row major (standard in C)

$$\begin{bmatrix} \texttt{a[0]} & \texttt{a[1]} & \texttt{a[2]} & \cdots & \texttt{a[} \quad \texttt{]} \\ \texttt{a[} \quad \texttt{]} & \texttt{a[} \quad \texttt{]} & \texttt{a[} \quad \texttt{]} & \cdots & \texttt{a[} \quad \texttt{]} \\ \texttt{a[} \quad \texttt{]} & \texttt{a[} \quad \texttt{]} & \texttt{a[} \quad \texttt{]} & \cdots & \texttt{a[} \quad \texttt{]} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \texttt{a[} \quad \texttt{]} & \texttt{a[} \quad \texttt{]} & \texttt{a[} \quad \texttt{]} & \cdots & \texttt{a[} \quad \texttt{]} \end{bmatrix}$$

- What is the indexing formula? $a_{i,j}$ is in element:

Matrix Indexing: Code Version

Two obvious options to store a 2D array of data in 1D memory.
Option 2: Column major (standard in Fortran and many scientific codes)

$$\begin{bmatrix} \texttt{a[0]} & \texttt{a[\ \ ]} & \texttt{a[\ \ ]} & \cdots & \texttt{a[\ \ ]} \\ \texttt{a[1]} & \texttt{a[\ \ ]} & \texttt{a[\ \ ]} & \cdots & \texttt{a[\ \ ]} \\ \texttt{a[2]} & \texttt{a[\ \ ]} & \texttt{a[\ \ ]} & \cdots & \texttt{a[\ \ ]} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \texttt{a[\ \ ]} & \texttt{a[\ \ ]} & \texttt{a[\ \ ]} & \cdots & \texttt{a[\ \ ]} \end{bmatrix}$$

- What is the indexing formula? $a_{i,j}$ is in element:

Indexing implemented by macros `IDX2C(i,j,m,n)` and `IDX2F(i,j,m,n)` in example and template code.

- For example, see `cpu-helper.h` in HW4 template code.

Matrix Multiply: CPU Version

Assume

- Matrices A, B and C are stored in arrays pointed to by `a`, `b` and `c` respectively.
- Size parameters are stored in integers `m`, `n` and `p`.
- Row-major ordering (or macro `IDX2C(i,j,m,n)`).

Remember

$$c_{i,k} = \sum_{j=0}^{n-1} a_{i,j} b_{j,k}$$

Write the serial CPU C code to compute $C = AB$.

Is this problem data parallel?

Matrix Multiply: GPU Version

Assume

- Matrices A, B and C are stored in arrays pointed to by `a`, `b` and `c` respectively.
- Size parameters are stored in integers `m`, `n` and `p`.
- Row-major ordering (or macro `IDX2C(i,j,m,n)`).

Remember

$$c_{i,k} = \sum_{j=0}^{n-1} a_{i,j} b_{j,k}$$

Write the parallel CUDA GPU C code to compute $C = AB$.

Do we achieve significant speedup?

Reminder: Typical GPU Workflow

- Host: Gather data from disk or network.
- Host: Allocate memory on GPU.
- Host: Copy data to GPU.
- Repeat:
  - Host: Launch kernel on GPU.
  - GPU: Run many many threads until kernel completes. (Host may continue work or wait for kernel completion.)
  - Host: If necessary, copy data to or from GPU.
- Host: Visualize, transmit or save results.

Is It Fast?

Depends on what you mean by “fast”.

- Much better than CPU: Speedup ~ 110 for $m = n = p = 3000$ on `lin13` (0.271 sec vs 29.6 sec).
- Nowhere near peak compute power
  - Floating point operations $(2)(3000)^3 = 54(10^9)$ in 0.27 sec $= 199(10^9)$ FLOPS.
  - Compare to GTX 1060 3GB peak of $3470(10^9)$ FLOPS.

Why so slow?