Matrix Operations on the GPU

CIS 665:
GPU Programming and Architecture
TA: Joseph Kider

Overview

- 3 Basic Linear Algebra Operations
  - Vector-Vector Operations
    - \( c = a \cdot b \)
  - Matrix-Matrix Operations
    - \( C = A + B \) - addition
    - \( D = A \cdot B \) - multiplication
    - \( E = E^{-1} \) - inverse
  - Matrix-Vector Operations
    - \( y = Ax \)

Note on Notation:
1) Vectors - lower case, underlined: \( \mathbf{v} \)
2) Matrices - upper case, underlined \( 2x \cdot M \)
3) Scalar - lower case, no lines: \( s \)

Efficiency/Bandwidth Issues

- GPU algorithms are severely bandwidth limited!
- Minimize Texture Fetches
- Effective cache bandwidth…so no algorithm would be able to read data from texture very much faster with texture fetches

Matrix Operations (thanks too…)

- Slide information sources
  - Suresh Venkatasubramanian
    - CIS700 – Matrix Operations Lectures
  - Fast matrix multiplies using graphics hardware by Larsen and McAllister
  - Dense Matrix Multiplication by Ádám Moravánszky
  - Cache and Bandwidth Aware Matrix Multiplication on the GPU, by Hall, Carr and Hart
  - Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication by Fatahalian, Sugerman, and Harahan
  - Linear algebra operators for GPU implementation of numerical algorithms by Krüger and Westermann
Vector-Vector Operations

Inner Product Review

An inner product on a vector space \( V \) over a field \( K \) (which must be either the field \( \mathbb{R} \) of real numbers or the field \( \mathbb{C} \) of complex numbers) is a function \( \langle \cdot, \cdot \rangle : V \times V \to K \) such that, for all \( v, w \) in \( V \) the following properties hold:

1. \( \langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle \) (linearity)
2. \( \langle kw, v \rangle = k \langle w, v \rangle \) (linearity constraints)
3. \( \langle v, w \rangle = \langle w, v \rangle \) (conjugate symmetry)
4. \( \langle v, v \rangle \geq 0 \) (positive definite)

Vector-Vector Operations

Dot Product: Technique 1

- Store each vector as a 1D texture \( a \) and \( b \)
- In the ith rendering pass we render a single point at coordinates \((0,0)\) which has a single texture coordinate \( i \)
- The Fragment program uses \( i \) to index into the 2 textures and return the value \( s + a_i \cdot b_i \) (\( s \) is the running sum maintained over the previous \( i-1 \) passes)
Vector-Vector Operations

- **Dot Product: Technique 1: Problems?**
  - We cannot read and write to the location s is stored in a single pass, we need to use a ping-pong trick to maintain s accurately
  - Takes n-passes
  - Requires only a fixed number of texture locations (1 unit of memory)
  - Does not take advantage of 2D spatial texture caches on the GPU that are optimized by the rasterizer
  - Limited length of 1D textures, especially in older cards

- **Dot Product: Technique 2**
  - (optimized for passes)
    - Wrap a and b as 2D textures

Adding up a texture elements to a scalar value

- Additive blending
- Or parallel reduction algorithm (log n passes)
Matrix-Matrix Operations

- Store matrices as 2D textures

```c
glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, 256, 256, 0, GL_RED, GL_UNSIGNED_BYTE, pData);
```

Matrix-Matrix Operations

- Addition is now a trivial fragment program
  /additive blend

Matrix-Matrix Operations

- Matrix Multiplication Review

\[
\begin{pmatrix}
0.6 & 0.3 \\
0.4 & 0.7
\end{pmatrix}
\begin{pmatrix}
0.6 & 0.3 \\
0.4 & 0.7
\end{pmatrix}
= \begin{pmatrix}
0.6 \times 0.6 + 0.3 \times 0.4 & 0.6 \times 0.3 + 0.3 \times 0.7 \\
0.4 \times 0.6 + 0.7 \times 0.4 & 0.4 \times 0.3 + 0.7 \times 0.7
\end{pmatrix}
\]

So in other words we have:

\[
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix}
\begin{pmatrix}
e & f \\
g & h
\end{pmatrix}
= \begin{pmatrix}
ae + bg & af + bh \\
cf + dh
\end{pmatrix}
\]

In general: 

\[(AB)_{ij} = \sum_{r=0}^{n} a_{ir} \times b_{rj}\]

Naive O(n^3) CPU algorithm

Matrix-Matrix Operations

- GPU Matrix Multiplication: Technique 1

Express multiplication of two matrices as dot product of vector of matrix row and columns

Compute matrix C by:

for each cell of c_{ij} take the dot product of row i of matrix A with column j of matrix B

\[C[i,j] = \sum A[i,k] \times B[k,j]\]
Matrix-Matrix Operations

- **GPU Matrix Multiplication: Technique 1**
  
  **Pass 1**
  
  Output = $a_{x1} \cdot b_{y}$
  
  **Pass 2**
  
  Output = Output$_1$ + $a_{x2} \cdot b_{y}$
  
  ...... 
  
  **Pass K**
  
  Output = Output$_{k-1}$ + $a_{xk} \cdot b_{ky}$
  
  Uses: $n$ passes
  
  Uses: $N=n^2$ space

- **GPU Matrix Multiplication: Technique 2**
  
  Blocking
  
  Instead of making one computation per pass, compute multiple additions per pass in the fragment program.
  
  **Pass 1**
  
  Output = $a_{x1} \cdot b_{y1} + a_{x2} \cdot b_{y2} + ... + a_{xk} \cdot b_{yk}$
  
  ...... 
  
  **Passes = $n$/Blocksize**
  
  Now there is a tradeoff between passes and program size/fetches

- **GPU Matrix Multiplication: Technique 3**
  
  Modern fragment shaders allow up to 4 instructions to be executed simultaneously
  
  (1) output = v1.abgr*v2.ggab
  
  This is issued as a single GPU instruction and numerically equivalent to the following 4 instructions being executed in parallel
  
  (2) output.r = v1.a * v2.g
  
  output.g = v1.b * v2.g
  
  output.b = v1.g * v2.a
  
  output.a = v1.r * v2.b
  
  In v1.abgr the color channels are referenced in arbitrary order.
  
  This is referred to as **swizzling**.
  
  In v2.ggab the color channel (g) is referenced multiple times.
  
  This is referred to as **smearing**.

- **GPU Matrix Multiplication: Technique 3**
  
  Smearing/Swizzling
  
  Up until now we have been using 1 channel, the red component to store the data, why now store data across all the channels (RGBA) and compute instructions 4 at a time

  Suppose we have 2 large matrices A, B, we whose dimensions are power of 2 by $A_{11}, A_{12}, ...$ are sub matrices of 2 by 2 rows/columns.
Matrix-Matrix Operations

Note on Notation:
\[ C(r) = A(r)^*B(r) \] used to denote the channels

Example:
\[
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} & B_{13} \\
B_{21} & B_{22} & B_{23} \\
B_{31} & B_{32} & B_{33}
\end{bmatrix}
\]

So now the final matrix multiplication can be expressed recursively by:

\[
A_{rgb}(A_{rgb}) = A_{rgb}(B_{rgb}) + A_{rgb}(C_{rgb})
\]

Efficiency/Bandwidth Issues

- Problem with matrix multiplication is each input contributes to multiple outputs \( O(n) \)
- Arithmetic performance is limited by cache bandwidth
- Multipass algorithms tend to be more cache friendly

2 Types of Bandwidth

- External Bandwidth: Data from the CPU \( \rightarrow \) GPU transfers limited by the AGP or PCI express bus
- Internal Bandwidth (Blackbox): read from textures/write to textures tend to be expensive

Back of the envelope calculation:

\([2 \text{ texture read/write lookups} \times \text{blocksize} + 2\times \text{previous pass lookup} \times \text{precision/(n^2)}\]

\((2 \times 32 + 2 \times 32)(1024) = 4GB \text{ of Data being thrown around}\)

GPU Benchmarks

<table>
<thead>
<tr>
<th>Peak Arithmetic Rate</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pent IV</td>
<td>22</td>
</tr>
<tr>
<td>5900</td>
<td>54</td>
</tr>
<tr>
<td>6800</td>
<td>164</td>
</tr>
<tr>
<td>7800</td>
<td>520</td>
</tr>
<tr>
<td>8800</td>
<td>330</td>
</tr>
<tr>
<td>AT19800</td>
<td></td>
</tr>
<tr>
<td>AT1X800</td>
<td></td>
</tr>
<tr>
<td>AT1X1900</td>
<td></td>
</tr>
</tbody>
</table>

Previous Generation GPUs

<table>
<thead>
<tr>
<th>Multiplication of 1024x1024 Matrices</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPS</td>
<td></td>
</tr>
</tbody>
</table>
### Next Generation GPUs

**Multiplication of 1024x1024 Matrices**

<table>
<thead>
<tr>
<th></th>
<th>GFLOPS</th>
<th>GB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4 3Ghz</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>6800 Ultra</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>X800 XT PE</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

- **GFLOPS**
- **Bandwidth**

---

### Matrix-Vector Operations

- **Matrix Vector Operation Review**

  **Example 1:**
  
  \[
  \begin{bmatrix}
  A & B & C \\
  D & E & F \\
  G & H & I \\
  \end{bmatrix}
  \begin{bmatrix}
  P \\
  Q \\
  R \\
  \end{bmatrix}
  =
  \begin{bmatrix}
  AF + BQ + CR \\
  DP + EQ + FR \\
  GF + HQ + IR \\
  \end{bmatrix}
  
  **Example 2:**
  
  \[
  \begin{bmatrix}
  1 & 1 & 1 \\
  1 & 1 & 1 \\
  1 & 1 & 1 \\
  \end{bmatrix}
  \begin{bmatrix}
  2 \\
  3 \\
  4 \\
  \end{bmatrix}
  =
  \begin{bmatrix}
  9 \\
  9 \\
  9 \\
  \end{bmatrix}
  
- **Technique 1:** Just use a Dense Matrix Multiply

  **Pass 1**
  
  Output = \(a_{x1} \times b_{y1} + a_{x2} \times b_{y2} + \ldots + a_{xn} \times b_{yn}\)

  **Passes:** \(\frac{n}{\text{Blocksize}}\)

- **Technique 2:** Sparse Banded Matrices \(A \times x = y\)

  A band matrix is a sparse matrix whose nonzero elements are confined to diagonal bands.

  **Algorithm:**
  - Convert Diagonal Bands to vectors
  - Convert (N) vectors to 2D-textures, pad with 0 if they do not fill the texture completely

---

### Matrix-Vector Operations

- **Technique 1:** Just use a Dense Matrix Multiply

  **Pass 1**
  
  Output = \(a_{x1} \times b_{y1} + a_{x2} \times b_{y2} + \ldots + a_{xn} \times b_{yn}\)

  **Passes:** \(\frac{n}{\text{Blocksize}}\)

- **Technique 2:** Sparse Banded Matrices \(A \times x = y\)

  A band matrix is a sparse matrix whose nonzero elements are confined to diagonal bands.

  **Algorithm:**
  - Convert Diagonal Bands to vectors
  - Convert (N) vectors to 2D-textures, pad with 0 if they do not fill the texture completely
Matrix-Vector Operations

- Technique 2: Sparse Banded Matrices
  - Convert the multiplication vector \((x)\) to a 2D texture
  - Pointwise multiply \((N)\) Diagonal textures with \((x)\) texture
  - Add the \((N)\) resulting matrices to form a 2D texture
  - Unwrap the 2D texture for the final answer

- Technique 3: Sparse Matrices
  Create a texture lookup scheme

\[
\mathbf{y}^* = \mathbf{A}^* \mathbf{x} + \sum_{i=0}^{k-1} \mathbf{A}^* [i + d + x^* (c^* [i + d])]
\]

Matrix Operations in CUDA

Take 2.

CUDA

- Thread: concurrent code and associated state executed on the CUDA device in parallel with other threads
  - The unit of parallelism in CUDA
- Warp: a group of threads executed physically in parallel in G80
- Block: a group of threads that are executed together and form the unit of resource assignment
- Grid: a group of thread blocks that must all complete before the next phase of the program can begin
A quick review

- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = group of SIMD threads that execute a kernel and can communicate via shared memory

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>NA</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>

CUDA: Matrix – Matrix Operations

- $P=M \times N$ of size $WIDTH \times WIDTH$
- Without blocking:
  - One thread handles one element of $P$
  - $M$ and $N$ are loaded $WIDTH$ times from global memory

CUDA: Matrix – Matrix Operations

- One Block of threads compute matrix $P$
  - Each thread computes one element of $P$
- Each thread:
  - Loads a row of matrix $M$
  - Loads a column of matrix $N$
  - Perform one multiply and addition for each pair of $M$ and $N$ elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block

Is this method any good?

- Product of two $N \times N$ matrices
- Streaming approach
  - Each thread computes single value of the output
  - Is it any good??? No!
    - Arithmetic Intensity = $(2N-1)/(2N+1) \Rightarrow$ Max performance: $22$ GFLOPs (instead of $343$!!!)
  - Why? $O(N)$ data reuse is NOT utilized
    - Optimally: Arithmetic intensity $= (2N-1)/(2N/N + 1) = O(N) \Rightarrow$ CPU bound!!!!!
CUDA: Matrix – Matrix Operations

- \( P = M \times N \) of size \( WIDTH \times WIDTH \)
- With blocking:
  - One thread block handles one \( BLOCK\_SIZE \times BLOCK\_Size \) sub matrix \( P_{sub} \) of \( P \)
  - \( M \) and \( N \) are only loaded \( WIDTH / BLOCK\_SIZE \) times from global memory
  - Great savings of memory bandwidth
  - Better balance of work to bandwidth

Generalized Approach to Shared Memory

- Think of it as a distributed user-managed cache
- When regular access pattern - better to have implicit cache management
  - In matrix product we know “implicitly” that the access is sequential
- Less trivial for irregular access pattern -> implement REAL cache logic interleaved into the kernel
  - devise cache tag, handle misses, tag collisions, etc,
  - analyze it just like regular cache