

Matrix Operations on the GPU

CIS 665:
GPU Programming and Architecture
TA: Joseph Kider

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, Sugerma, and Harahan
 - Linear algebra operators for GPU implementation of numerical algorithms by Krüger and Westermann

Overview

- **3 Basic Linear Algebra Operations**
 - Vector-Vector Operations
 - $\underline{c} = \underline{a} \underline{b}$
 - Matrix-Matrix Operations
 - $\underline{C} = \underline{A} + \underline{B}$ - addition
 - $\underline{D} = \underline{A} * \underline{B}$ - multiplication
 - $\underline{E} = \underline{E}^{-1}$ - inverse
 - Matrix-Vector Operations
 - $\underline{y} = \underline{A} \underline{x}$

Note on Notation:

- 1) Vectors - lower case, underlined: \underline{v}
- 2) Matrices – upper case, underlined 2x : 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

Vector-Vector Operations

- **Inner Product Review**

An inner product on a vector space (V) over a field (K) (which must be either the field R of real numbers or the field C of complex numbers) is a function $\langle \cdot, \cdot \rangle: V \times V \rightarrow K$ such that, k_1, k_2 in K for all v, w in V the following properties hold:

1. $\langle u+v, w \rangle = \langle u, w \rangle + \langle v, w \rangle$
2. $\langle \alpha v, w \rangle = \alpha \langle v, w \rangle$ (linearity constraints)
3. $\langle v, w \rangle = \overline{\langle w, v \rangle}$ (conjugate symmetry)
4. $\langle v, v \rangle \geq 0$ (positive definite)

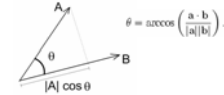
Vector-Vector Operations

- **Inner Product Review**

A vector space together with an inner product on it is called an inner product space. Examples include:

1. The real numbers R where the inner product is given by $\langle x, y \rangle = xy$
2. The Euclidean space R^n where the inner product is given by the dot product:

$$\begin{aligned} \mathbf{c} &= \mathbf{a} \cdot \mathbf{b} \\ \mathbf{c} &= \langle (a_1, a_2, \dots, a_n), (b_1, b_2, \dots, b_n) \rangle \\ \mathbf{c} &= a_1 b_1 + a_2 b_2 + \dots + a_n b_n \\ \mathbf{c} &= \sum \mathbf{a}_i \mathbf{b}_i \end{aligned}$$



3. The vector space of real functions with a closed domain [a,b] $\langle f, g \rangle = \int \mathbf{f} \mathbf{g} \, dx$

Vector-Vector Operations

- **Inner Product Review**

A vector space together with an inner product on it is called an inner product space. Examples include:

1. The real numbers R where the inner product is given by $\langle x, y \rangle = xy$
2. The Euclidean space R^n where the inner product is given by the dot product:

$$\begin{aligned} \mathbf{c} &= \mathbf{a} \cdot \mathbf{b} \\ \mathbf{c} &= \langle (a_1, a_2, \dots, a_n), (b_1, b_2, \dots, b_n) \rangle \\ \mathbf{c} &= a_1 b_1 + a_2 b_2 + \dots + a_n b_n \\ \mathbf{c} &= \sum \mathbf{a}_i \mathbf{b}_i \end{aligned}$$
3. The vector space of real functions with a closed domain [a,b] $\langle f, g \rangle = \int \mathbf{f} \mathbf{g} \, dx$

Vector-Vector Operations

- **Dot Product: Technique 1**

- (Optimized for memory)
- Store each vector as a 1D texture **a** and **b**
- In the *i*th 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 $s + a_i * b_i$ (*s* is the running sum maintained over the previous *i*-1 passes)

Vector-Vector Operations

- **Dot Product: Technique 1: Problems?**

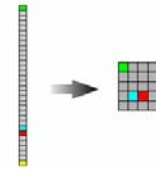
- ⊗ We cannot read and write to the location s is stored in a single pass, we need to use a ping-pong trick to maintain s accurately
- ⊖ Takes n -passes
- ⊖ Requires only a fixed number of texture locations (1 unit of memory)
- ⊖ Does not take advantage of 2D spatial texture caches on the GPU that are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

Vector-Vector Operations

- **Dot Product: Technique 2**

- (optimized for passes)

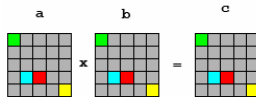
- Wrap a and b as 2D textures



Vector-Vector Operations

- **Dot Product: Technique 2**

- Multiply the two 2D textures by rendering a single quad with the answer

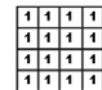


- Add the elements in (c) the result 2D texture together

Vector-Vector Operations

- Adding up a texture elements to a scalar value

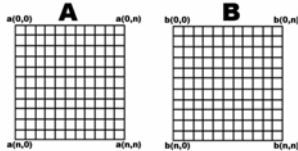
- Additive blending
- Or parallel reduction algorithm (log n passes)



```
//example Fragment program for performing a reduction
float main (float2 texcoord: TEXCOORD0, uniform
sampler2D img): COLOR
{
    float a, b, c, d;
    a=tex2D(img, texcoord);
    b=tex2D(img, texcoord + float2(0,1));
    c=tex2D(img, texcoord + float2(1,0));
    d=tex2D(img, texcoord + float2(1,1));
    return (a+b+c+d);
}
```

Matrix-Matrix Operations

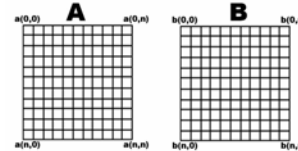
- Store matrices as 2D textures



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

Matrix-Matrix Operations

- Store matrices as 2D textures



- 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:

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

In general:

$$(AB)_{ij} = \sum_{r=0} a_{ir} b_{rj}$$

Naïve $O(n^3)$ CPU algorithm

for $i = 1$ to n
for $j = 1$ to n

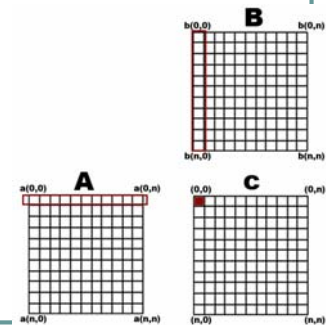
$$C[i,j] = \sum A[i,k] * B[k,j]$$

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



Matrix-Matrix Operations

GPU Matrix Multiplication: Technique 1

Pass1

$$\text{Output} = a_{x1} * b_{1y}$$

Pass2

$$\text{Output} = \text{Output}_1 + a_{x2} * b_{2y}$$

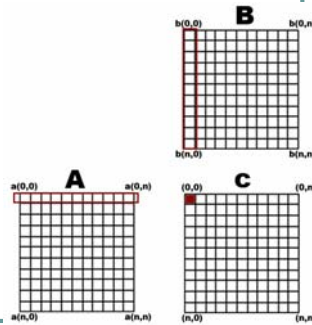
.....

PassK

$$\text{Output} = \text{Output}_{k-1} + a_{xk} * b_{ky}$$

Uses: n passes

Uses: N=n² space



Matrix-Matrix Operations

GPU Matrix Multiplication: Technique 2

Blocking

Instead of making one computation per pass. Compute multiple additions per pass in the fragment program.

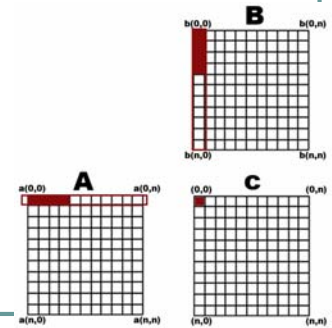
Pass1

$$\text{Output} = a_{x1} * b_{1y} + a_{x2} * b_{2y} + \dots + a_{xb} * b_{by}$$

.....

Passes: = n/Blocksize

Now there is a tradeoff between passes and program size/fetches



Matrix-Matrix Operations

GPU Matrix Multiplication: Technique 3

- Modern fragment shaders allow up to 4 instructions to be executed simultaneously

$$(1) \text{ output} = v1.abgr * v2.ggab$$

This is issued as a single GPU instruction and numerically equivalent to the following 4 instructions being executed in parallel

$$(2) \begin{aligned} \text{output.r} &= v1.a * v2.g \\ \text{output.g} &= v1.b * v2.g \\ \text{output.b} &= v1.g * v2.a \\ \text{output.a} &= v1.r * v2.b \end{aligned}$$

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**.

Matrix-Matrix Operations

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

$$\left[\begin{array}{c|c} A & B \end{array} \right] \Rightarrow \left[\begin{array}{c|c} R & G \\ B & A \end{array} \right]$$

The matrix multiplication can be expressed as follows:

$$\left[\begin{array}{c|c} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right] \left[\begin{array}{c|c} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array} \right] \Rightarrow \left[\begin{array}{c|c} a_{11} * a_{12} & a_{11} * a_{22} \\ a_{21} * a_{12} & a_{21} * a_{22} \end{array} \right]$$

Suppose we have 2 large matrices A B, wog whose dimensions are power of 2s
A₁₁, a₁₂ ... are sub matrices of 2ⁿ rows/columns

Matrix-Matrix Operations

Note on Notation:

$C(r)=A(r)*B(r)$ used to denote the channels

Example:

$$A(rrbb)A(rgrg) = \begin{bmatrix} A(R)^*B(R) & A(R)^*B(G) \\ A(B)^*B(R) & A(B)^*B(G) \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} \\ a_{21}b_{11} & a_{21}b_{12} \end{bmatrix}$$

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

$$A(rgba)A(rgba) = A(rrbb)B(rgrg) + A(ggaa)B(baba)$$

$$A(rgba)A(rgba) = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} \\ a_{21}b_{11} & a_{21}b_{12} \end{bmatrix} + \begin{bmatrix} a_{12}b_{21} & a_{12}b_{22} \\ a_{22}b_{21} & a_{22}b_{22} \end{bmatrix} = C(rgba)$$

Matrix-Matrix Operations

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

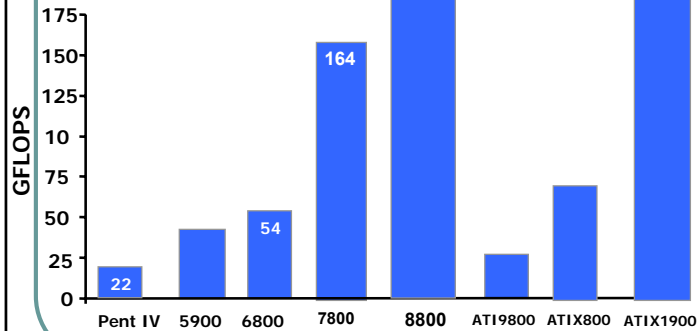
- External Bandwidth: Data from the CPU \leftrightarrow GPU transfers limited by the AGP or PCI express bus
- Internal Bandwidth (Blackbox): read from textures/write to textures tend to be expensive

Back of the envelope calculation:

- $((2 \text{ texture read/write lookups}) * \text{blocksize} + 2(\text{previous pass lookup}) * (\text{precision})(n^2))$
- $(2 * 32 + 2)(32)(1024) = 4\text{GB of Data being thrown around}$

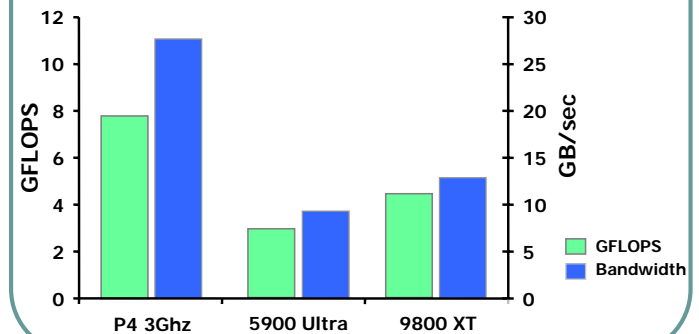
GPU Benchmarks

Peak Arithmetic Rate



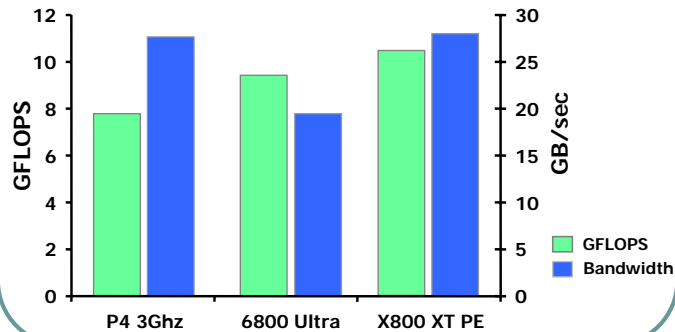
Previous Generation GPUs

Multiplication of 1024x1024 Matrices



Next Generation GPUs

Multiplication of 1024x1024 Matrices



Matrix-Vector Operations

- Matrix Vector Operation Review

Example 1:

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} P \\ Q \\ R \end{bmatrix} = \begin{bmatrix} AP + BQ + CR \\ DP + EQ + FR \\ GP + HQ + IR \end{bmatrix}$$

Example 2:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 9 \\ 9 \\ 9 \end{bmatrix}$$

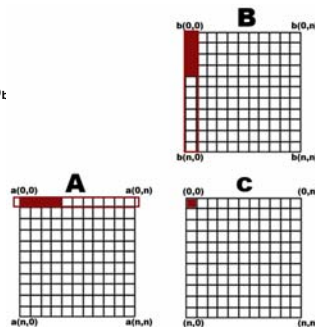
Matrix-Vector Operations

- Technique 1: Just use a Dense Matrix Multiply

Pass1

$$\text{Output} = a_{x1} * b_{11} + a_{x2} * b_{21} + \dots + a_{xb} * b_{t}$$

Passes: = n/Blocksize



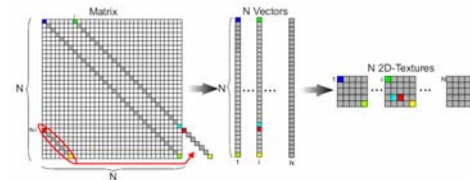
Matrix-Vector Operations

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

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

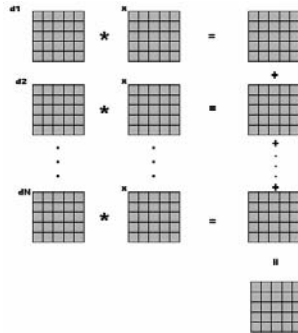
- **Technique 2: Sparse Banded Matrices**

- 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

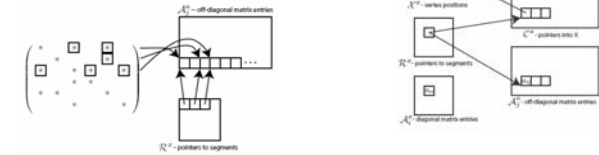
- unwrap the 2D texture for the final answer



Matrix-Vector Operations

- **Technique 3: Sparse Matrices**

Create a texture lookup scheme



inner product kernel between a sparse matrix row and the vector o unknowns becomes

$$j = \mathcal{R}^x[i]$$

$$\mathcal{Y}^x[i] = \mathcal{A}_i^x[j] * \mathcal{X}^x[j] + \sum_{c=0}^{k_i-1} \mathcal{A}_i^x[j+c] * \mathcal{X}^x[j+c],$$

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



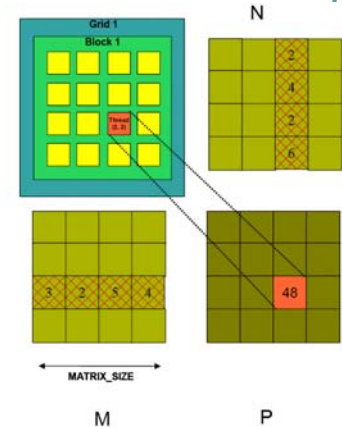
- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = group of SIMD threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

© NVIDIA Corporation 2009

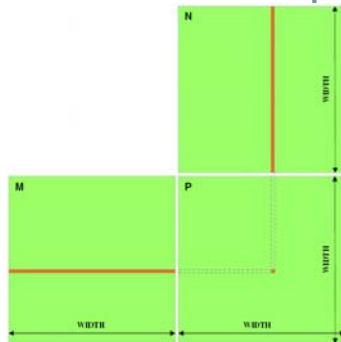
CUDA: Matrix – Matrix Operations

- 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



CUDA: Matrix – Matrix Operations

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



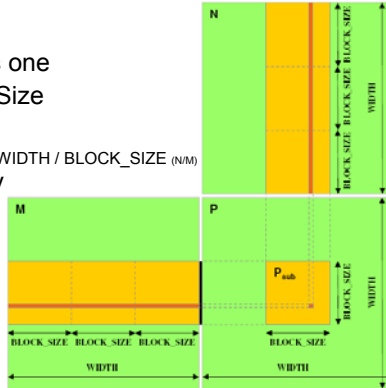
CUDA: Matrix – Matrix Operations

- Is this method any good?
 - Product of two NxN 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: 22GFLOP/s (instead of 345!!!!)
 - 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

- $P=M*N$ 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 $WIDTH / BLOCK_SIZE$ (N/M) 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