

CIS 700/010: Matrix Operations I

Suresh Venkatasubramanian
Scribed by Kennedy Behrman

March 3, 2005

1 Matrix operations

There are three basic matrix operations that would be part of any GPU matrix toolkit:

1. The inner product of two vectors $c = a \cdot b$.
2. Matrix-vector operations: $y = Ax$.
3. Matrix-Matrix operations: $C = A + B, D = AB, E = A^{-1}$.

A number of problems can be solved once one has these basic operations (especially in physical simulations). This is one of the most studied problems on the GPU.

2 The Inner Product

Consider the inner product $c = a \cdot b$, which we rewrite as $c = \sum_{i=1}^n a_i b_i$.

2.1 Technique 1: Small memory

Each vector is stored in a 1D texture. In the i^{th} rendering pass, we render a single point at coordinates (0,0), having a single texture coordinate i . The fragment program uses i to index into the two textures and returns the value $s + a_i + b_i$, where s is the running sum maintained over the previous $i - 1$

passes. Note that since we cannot read and write the location where s is stored in a single pass, we use a *ping-pong* trick to maintain s .

This procedure takes n passes, and requires only a fixed number of texture locations (excluding the storage for a and b).

2.2 Technique 2: Fewer passes

The second technique uses more working memory (n units), but requires fewer passes. We write a and b as 2D textures (2D textures allow for more storage, since the dimension of a texture is typically bounded, and are better optimized by the rasterizer).

We now multiply the contents of the textures, storing the result in a third texture c . This can be done with a simple fragment program that takes the fragment coordinates and looks up the a and b textures, returning their product. We render a single quad in order to activate the fragment program.

$$c = \begin{bmatrix} a_0b_0 & a_1b_1 & \dots \\ \dots & a_ib_i & \dots \\ \dots & \dots & a_nb_n \end{bmatrix}$$

Finally, all the numbers in c must be summed together. This can be done in $\log n$ passes, using a standard `reduce` operation.

This procedure takes only $\log n$ passes, but requires $3n$ units of texture memory.

3 Matrix-Matrix operations

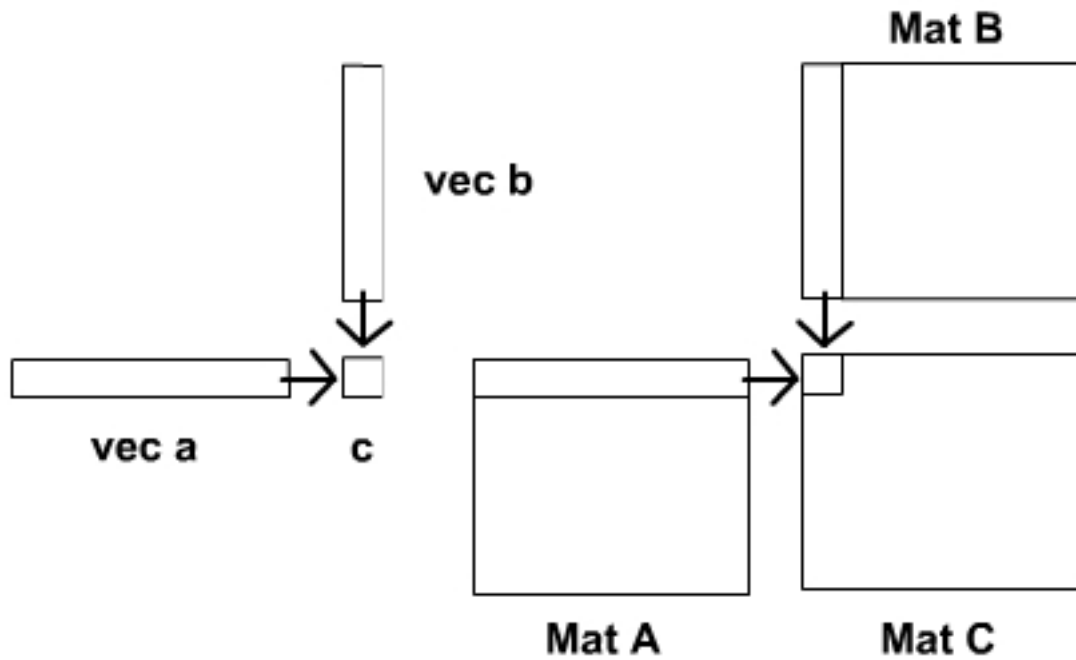
We can store matrices as 2D textures. Addition is trivial.

3.1 Multiplication

3.2 Technique 1: The Basic Approach

”Fast matrix multiplies using graphics hardware” by Larsen and McAllister”[2]

Express multiplication of two matrices as dot product of vectors of matrix rows and columns. That is to compute some cell c_{ij} of matrix C , we take the dot product of row i of matrix A with column j of matrix B :



1st program used multitexturing and blending, each plane would compute each place in the answer. In 1st pass: AB :

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$*$$

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} a_{11}b_{11} & \dots & \dots \\ a_{21}b_{21} & \dots & \dots \\ a_{31}b_{31} & \dots & \dots \end{bmatrix}$$

We can use inner quad idea to do this:

pass 1

if at location (x,y)

$output \leftarrow a_{x1} * b_{1y}$

pass 2

$$output \leftarrow c + a_{x2} * b_{2y}$$

pass k

$$output \leftarrow c + a_{xk} * b_{ky}$$

$$C_{xy} = \sum_{k=1}^n a_{xk} b_{ky}$$

1) uses n passes

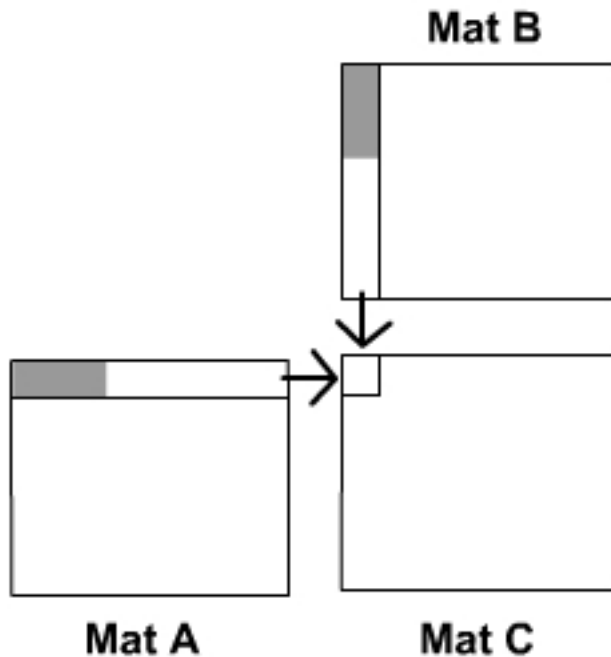
2) space $N = n^2$

3.3 Technique 2: A Speedup

”Dense Matrix Multiplication” by \acute{A} ad \acute{a} m Morav \acute{a} nszky

To make it faster:

Instead of making one computation per pass, compute multiple additions per pass in fragment program:



Pass 1 becomes: $output \leftarrow a_{x1}b_{1y} + a_{x2}b_{2y} + a_{x3}b_{3y} + a_{x4}b_{4y}$

Must consider that there is a tradeoff between the length of fragment program vs. the number of passes.

3.4 Technique 3: Using All Channels

”Cache and Bandwidth Aware Matrix Multiplication on the GPU”, by Hall, Carr and Hart”[1].

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

We have been using only the red component, propose storing across all colors:

$$\begin{bmatrix} A_{11R} & A_{12G} \\ A_{21B} & A_{22A} \end{bmatrix} \begin{bmatrix} B_{11R} & B_{12G} \\ B_{21B} & B_{22A} \end{bmatrix}$$
$$\begin{bmatrix} A_{11}B_{11} + A_{12}B_{12} & A_{11}B_{21} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{12} & A_{22}B_{21} + A_{22}B_{22} \end{bmatrix}$$

divide into 2:

$$\begin{bmatrix} A_{11}B_{11} + A_{12}B_{12} \\ A_{21}B_{11} + A_{22}B_{12} \end{bmatrix}$$
$$\begin{bmatrix} A_{22}B_{21} + A_{22}B_{22} \\ A_{11}B_{21} + A_{12}B_{22} \end{bmatrix}$$

$A_{rrbb}B_{rgrg} + A_{ggaa}B_{baba}$ Basically a swizzle operation to speed things up.
Closing thought: The basic idea here is using the inner product calculation in parallel.

References

- [1] HALL, J., CARR, N., AND HART, J. C. Cache and bandwidth aware matrix multiplication on the gpu. Tech. Rep. UIUCDCS-R-2003-2328-1, UIUC, 2003.
- [2] LARSEN, E., AND MCALLISTER, D. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)* (2001), ACM Press, pp. 55–55.