Floating-point MicroSIMD Architecture for Fast Geometry Transform

Xiao Yang and Ruby B. Lee
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
{xiaoyang, rblee}@ee.princeton.edu

1. INTRODUCTION

Geometry transform is a key component of a typical 3D graphics pipeline such as OpenGL.\textsuperscript{5} In this stage, objects are transformed from their own coordinate systems to other coordinate systems for further processing such as lighting and clipping. Two major operations used in this stage are $4 \times 4$ matrix-vector multiplication and $4 \times 4$ matrix-matrix multiplication.\textsuperscript{5} They are usually performed in the single-precision floating-point domain. The first operation is used to transform vectors (coordinates and normals of vertices) from one coordinate system into another. The second one is used to concatenate two transform matrices into one. These operations are executed frequently, especially the matrix-vector multiplication, which is done multiple times for each vertex. The speed of performing these operations is critical to the performance of the whole 3D graphics pipeline. Therefore, it is important to optimize the underlying architecture in order to perform these frequently executed operations fast.

Due to the inherent data parallelism in these operations, it is natural to exploit the parallelism using a microSIMD\textsuperscript{4} geometry transform engine. However, because different algorithms expose parallelism differently, and there are different ways to construct microSIMD engines, there exist many solutions to the problem. They are not equally efficient in terms of execution time and resource utilization. Our goal is to find a solution which is efficient in terms of both execution time and resource usage. In order to accomplish this task, we need to solve the following problems: which algorithms to choose, how many FP computing units to use, and how to organize them into an FP microSIMD engine.

First, we introduce a register-based datapath model for the microSIMD geometry transform engine, which is shown in Figure 1. This model consists of a register file, a group of FP microSIMD functional units (FU's), each contains certain numbers of FP computing units, and source data busses and result data busses connecting them. In a typical operation, the microSIMD FU's take the source data from the register file via the source data busses, perform the computations, and put the results onto the result data busses. The results may be written back to the register file, or sent directly to the next processing stage, depending on the context. Data in the register file can come from the system bus, either AGP or PCI, the previous processing stage, or the results of the microSIMD FU's. By having multiple FP microSIMD FU's, we can efficiently exploit the data parallelism, while keeping the bandwidth of the instruction stream and the size of the register file reasonable, and having flexibility of utilizing these FU's.

In this paper, we show how we obtain a solution that can be used to carry out the $4 \times 4$ matrix-vector and matrix-matrix multiplications efficiently. In the rest of the paper, FP refers to single-precision floating point. We assume that the FP microSIMD FU's are pipelined, so that a new FP compute instruction can be started every cycle on each FU. Hence, even with multiple cycle execution latencies of each FP instruction, the effective execution time can be assumed to be one cycle.

2. CHOOSING ALGORITHMS AND MAPPING TO MICROSIMD ARCHITECTURE

The goals of choosing proper algorithms are to perform the desired computation fast and at the same time, make efficient use of the available functional units. In order to achieve the first goal, we need to reduce the dependency height of the instructions for the algorithms. The dependency height of the instructions for an algorithm determines the minimum number of cycles it takes to execute the algorithm, i.e., how fast an algorithm can run. In order to accomplish the second goal, the algorithms need to be able to expose similar level of parallelism in each cycle. To determine this, we schedule instructions for the algorithms with respect to the number of FU's, see if the schedules can use similar numbers of FU's in every cycle. There is an upper bound for the number of FU’s for which the instruction schedules can display scalable parallelism. Beyond this bound, the instruction schedules cannot use all
the FU’s in every cycle and do not scale up in performance. It is desirable that this bound is high, because presumably the algorithms would run faster if they can use more FU’s per cycle. When the number of FU’s is smaller, we can always scale the instruction schedules down.

With these two goals in mind, we select the algorithms for $4 \times 4$ FP matrix-vector multiplication and matrix-matrix multiplication.

2.1. $4 \times 4$ matrix-vector multiplication

The mathematical representation of $4 \times 4$ matrix-vector multiplication is

$$Y = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = AX = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

where

$$y_i = \sum_{j=0}^{3} a_{ij}x_j$$

Each $y_i$ is obtained by taking the dot-product of vector $X$ and one row-vector in matrix $A$. We show the calculation as a group of basic operations:

$$t_0 = a_{00} \times x_0 + a_{11} \times x_1 + a_{22} \times x_2 + a_{33} \times x_3$$

There are several way of organizing and scheduling these operations. Here we present two algorithms corresponding to two different operation schedules. The steps for the first algorithm are

$$t_0 = a_{00} \times x_0 \quad (1)$$
$$t_1 = a_{11} \times x_1 \quad (2)$$
$$t_0 = t_0 + t_1 \quad (3)$$
$$t_2 = a_{22} \times x_2 \quad (4)$$
$$t_3 = a_{33} \times x_3 \quad (5)$$
$$t_2 = t_2 + t_3 \quad (6)$$
$$y_i = t_0 + t_2 \quad (7)$$
Figure 2. (a) The dependency graph of the first algorithm. (b) The dependency graph of the second algorithm.

The steps for the second algorithm are

\[
\begin{align*}
  y_i &= a_{i0} \times x_0 \quad (1) \\
  t &= a_{i1} \times x_1 \quad (2) \\
  y_i &= y_i + t \quad (3) \\
  t &= a_{i2} \times x_2 \quad (4) \\
  y_i &= y_i + t \quad (5) \\
  t &= a_{i3} \times x_3 \quad (6) \\
  y_i &= y_i + t \quad (7) \\
  \end{align*}
\]

The dependency graphs for the operations in these two algorithms are shown in Figure 2.

2.1.1. Instruction mapping – non-microSIMD

In order to investigate how we can make efficient use of microSIMD architecture, we first map instructions for the above two algorithms on a conventional, i.e., non-microSIMD, architecture. As can be seen from the dependency graphs for the two algorithms, additions often follow multiplications. Therefore, when mapping instructions to these operations, it is beneficial to use a multiply-accumulate instruction to combine multiplication followed by addition into one operation. This MAC instruction can be implemented using the FP MAC units that exist in many DSP chips. The instruction format of the MAC instruction is

\texttt{fmuladd rd,rs1,rs2,rs3}

where

\texttt{rd=rs1*rs2+rs3}

With the \texttt{fmuladd} instruction, the calculation of each \( y_i \) in the two algorithms would be mapped to 5 instructions and 4 instructions, respectively (see Figure 3). The instructions for the first algorithm is
Figure 3. (a) The dependency graph of the instructions for the first algorithm. (b) The dependency graph of the instructions for the second algorithm.

\[
\text{fmaladd } $(y), $(a0), $(x0), fr0 \quad (1) \quad \text{fmaladd } $(t), $(a2), $(x2), fr0 \quad (2)
\]
\[
\text{fmaladd } $(y), $(a1), $(x1), $(y) \quad (3) \quad \text{fmaladd } $(t), $(a3), $(x3), $(t) \quad (4)
\]
\[
\text{fmaladd } $(y), $(a1), $(x0), fr1, $(t) \quad (5)
\]

where \( fr0 \) and \( fr1 \) contain FP constants 0 and 1, respectively, \( $(v) \) is the register that holds value \( v \). Instructions in a row can be executed in parallel. The instruction sequence for the second algorithm is

\[
\text{fmaladd } $(y), $(a0), $(x0), fr0 \quad (1)
\]
\[
\text{fmaladd } $(y), $(a1), $(x1), $(y) \quad (2)
\]
\[
\text{fmaladd } $(y), $(a2), $(x2), $(y) \quad (3)
\]
\[
\text{fmaladd } $(y), $(a3), $(x3), $(y) \quad (4)
\]

The dependency graphs for these two groups of instructions are shown in Figure 3. We see that the first algorithm has shorter dependency height(3) than the second one(4). Therefore, the fastest instruction schedule for the first algorithm can be executed in three cycles, while the fastest schedule for the second algorithm can be executed in four cycles. However, notice that the first algorithm requires two FP MAC’s to achieve its optimal cycle count, twice the amount required for the second algorithm. For one \( 4 \times 4 \) matrix-vector multiplication that involves the calculation of four \( y_i \)’s, it takes eight FP MAC’s for the first algorithm to achieve its optimal cycle count of three, while it only takes four FP MAC’s for the second algorithm to achieve its optimal four cycles. When only four FP MAC’s are available, the first algorithm takes a minimum of five cycles to do a full \( 4 \times 4 \) matrix-vector multiplication. The minimum number of cycles for the second algorithm in this case is still four. In practice, we want to transform multiple independent vectors at the same time to reduce the effective cycle time per vector transform. Therefore, the performance of the two algorithms are resource bounded. As discussed, when the number of FP MAC’s is not enough for the first algorithm to achieve its optimal cycle time, it is slower than the second algorithm. Another problem of the first algorithm is that it needs to use one temporary register for the calculation of each \( y_i \). Transforming several vectors at the same time requires the use of many temporary registers, which may cause register spills and refills. The second algorithm does not use temporary registers. Due to the more efficient resource usage of the second algorithm, we choose it as the basis for the analysis in the rest of this paper.

The second algorithm can be adapted to microSIMD architecture very easily. Notice that the calculations of the different \( y_i \)’s are independent chains with identical steps. We can group them and execute them in one microSIMD
FU. Because a 4 × 4 matrix-vector multiplication contains four computation chains, it is natural to use 4-way microSIMD.

2.1.2. Instruction mapping – 4-way FP microSIMD FU

In order to use 4-way microSIMD FU’s to do 4 × 4 matrix-vector multiplications, we execute the four computation chains for one matrix-vector multiplication in the same FU. To see how this is done, we put the four chains side by side to inspect what is done in each cycle:

MAC $(y_0), (a_0), (x_0), 0$  MAC $(y_1), (a_10), (x_0), 0$ ...
MAC $(y_0), (a_01), (x_1), (y_0)$  MAC $(y_1), (a_11), (x_1), (y_1)$ ...
MAC $(y_0), (a_02), (x_2), (y_0)$  MAC $(y_1), (a_12), (x_2), (y_1)$ ...
MAC $(y_0), (a_03), (x_3), (y_0)$  MAC $(y_1), (a_13), (x_3), (y_1)$ ...

Because we have not yet defined the instruction for the microSIMD FU, we use “MAC” for the operation performed in each computational unit in the FU. We can see the operations performed by the microSIMD FU in each cycle by rewriting the above operations:

$pfmuladd, 0 ((y0, y1, y2, y3)), ((x0, -, -, -)), ((a00, a10, a20, a30)), fr0$
pfmuladd, 1 ((y0, y1, y2, y3)), ((-x1, -, -)), ((a01, a11, a21, a31)), ((y0, y1, y2, y3))
pfmuladd, 2 ((y0, y1, y2, y3)), ((-x2, -)), ((a02, a12, a22, a32)), ((y0, y1, y2, y3))
pfmuladd, 3 ((y0, y1, y2, y3)), ((-x3, -)), ((a03, a13, a23, a33)), ((y0, y1, y2, y3))

Here we introduce a new instruction $pfmuladd$ that has the format

$pfmuladd,i$ frd,frs1,frs2,frs3 ;i=0,1,2,3

What the $pfmuladd$ operation does is that for $i = 0$,

frd(0)=frs1(0)*frs2(0)+frs3(0)  frd(1)=frs1(0)*frs2(1)+frs3(1)
frd(2)=frs1(0)*frs2(2)+frs3(2)  frd(3)=frs1(0)*frs2(3)+frs3(3)

for $i = 1$,

frd(0)=frs1(1)*frs2(0)+frs3(0)  frd(1)=frs1(1)*frs2(1)+frs3(1)
frd(2)=frs1(1)*frs2(2)+frs3(2)  frd(3)=frs1(1)*frs2(3)+frs3(3)

and similar for $i = 2, 3$. frd(1) denotes the $i$-th element(subword$^{3,4}$) in the FP register fr. Note that each destination subword uses the operands in corresponding subword tracks, except that they all use the same subword in frs1, given by the sub $i$ of $pfmuladd$.

With the $pfmuladd$ instruction, we can use one 4-way microSIMD FU to perform a 4 × 4 matrix-vector multiplication in four cycles. In practice, we can transform multiple vectors in multiple microSIMD FU’s concurrently to reduce the effective cycle time per vector. For example, when there are four microSIMD FU’s, we can do four 4 × 4 matrix-vector multiplications in four cycles, which gives us an effective rate of one cycle per vector transform.

We notice that in order for the above 4-way microSIMD algorithm to work, the matrix data need to be stored in four FP registers in column-major order, where each FP register holds four 32-bit FP values. Because the transform matrices may not be stored in column-major order initially, we may need to transpose them before executing the vector transforms. How to obtain matrix transpose is further discussed in Section 2.3.

2.1.3. Algorithm scalability

The algorithm we choose are scalable with respect to the number of FP MAC’s used and the microSIMD FU organization. Although we only discuss the implementation of the algorithm on 4-way microSIMD FU’s, we can scale it down in a trivial way to fit into an implementation with 2-way microSIMD FU’s, if we want a lower cost implementation than the one with 4-way microSIMD FU’s. We summarize the performance(cycles/vector transform) of the algorithm with different numbers of FP MAC’s and different FU organizations in table 1. The performance numbers shown for when more than 4 MAC’s are used all consider multiple concurrent vector transforms.
non-microSIMD implementation

<table>
<thead>
<tr>
<th>Number of MAC’s</th>
<th>Cycle count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

2-way microSIMD implementation

<table>
<thead>
<tr>
<th>Number of microSIMD FU’s</th>
<th>Number of MAC’s</th>
<th>Cycle count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

4-way microSIMD implementation

<table>
<thead>
<tr>
<th>Number of microSIMD FU’s</th>
<th>Number of MAC’s</th>
<th>Cycle count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Scalable algorithm performance with respect to different numbers of FP MAC’s used and different FU organizations

2.2. 4 × 4 matrix-matrix multiplication

The mathematical representation of 4 × 4 matrix-matrix multiplication is

\[ C = \begin{pmatrix}
    c_{00} & c_{01} & c_{02} & c_{03} \\
    c_{10} & c_{11} & c_{12} & c_{13} \\
    c_{20} & c_{21} & c_{22} & c_{23} \\
    c_{30} & c_{31} & c_{32} & c_{33}
\end{pmatrix} = AB = \begin{pmatrix}
    a_{00} & a_{01} & a_{02} & a_{03} \\
    a_{10} & a_{11} & a_{12} & a_{13} \\
    a_{20} & a_{21} & a_{22} & a_{23} \\
    a_{30} & a_{31} & a_{32} & a_{33}
\end{pmatrix} \begin{pmatrix}
    b_{00} & b_{01} & b_{02} & b_{03} \\
    b_{10} & b_{11} & b_{12} & b_{13} \\
    b_{20} & b_{21} & b_{22} & b_{23} \\
    b_{30} & b_{31} & b_{32} & b_{33}
\end{pmatrix} \]

where

\[ c_{ij} = \sum_{k=0}^{3} a_{ik} b_{kj} \]

Each element in C is calculated by evaluating the dot-product of a row-vector in A and a column-vector in B. This is the same type of calculation as in matrix-vector multiplication. The only difference is that now we have 16 dot-products to compute rather than four for matrix-vector multiplication. Therefore, we can apply the same analysis and result for 4 × 4 matrix-vector multiplication on 4 × 4 matrix-matrix multiplication. We still use the second algorithm for matrix-vector multiplication in this case and study the instruction mapping on 4-way microSIMD FU’s.

2.2.1. Instruction mapping – 4-way FP microSIMD FU

From the analysis for 4 × 4 matrix-vector multiplication, it is required that transform matrices are stored in column-major order before the multiplication can take place when using 4-way microSIMD FU’s. Since the purpose of doing matrix-matrix multiplication is to combining transform matrices, it is desirable that the result of it is also in column-major order. This is naturally achieved if the two source matrices for multiplication are in column-major order, as required by the matrix-vector multiplication. This is due to the following fact:

\[ C = AB \Rightarrow C^T = \begin{pmatrix}
    c_{00} & c_{10} & c_{20} & c_{30} \\
    c_{01} & c_{11} & c_{21} & c_{31} \\
    c_{02} & c_{12} & c_{22} & c_{32} \\
    c_{03} & c_{13} & c_{23} & c_{33}
\end{pmatrix} = B^T A^T = \begin{pmatrix}
    b_{00} & b_{10} & b_{20} & b_{30} \\
    b_{01} & b_{11} & b_{21} & b_{31} \\
    b_{02} & b_{12} & b_{22} & b_{32} \\
    b_{03} & b_{13} & b_{23} & b_{33}
\end{pmatrix} \begin{pmatrix}
    a_{00} & a_{10} & a_{20} & a_{30} \\
    a_{01} & a_{11} & a_{21} & a_{31} \\
    a_{02} & a_{12} & a_{22} & a_{32} \\
    a_{03} & a_{13} & a_{23} & a_{33}
\end{pmatrix} \]
Because each column vector \((c_{k1},c_{k2},c_{k3},c_{k4})\) of the result matrix \(C\) (or each row vector of \(C^T\)) is to be stored together in a single FP register, we put the computations of the elements in the same column vector of \(C\) into one 4-way microSIMD FU. Therefore, for each column vector in \(C\), we have

\[
\text{pfmuladd}_0 \quad (((c_{01},c_{11},c_{21},c_{31})), \ldots, ((a_{00},a_{10},a_{20},a_{30})), 0)
\]

\[
\text{pfmuladd}_1 \quad (((c_{01},c_{11},c_{21},c_{31})), \ldots, ((a_{01},a_{11},a_{21},a_{31})), ((c_{01},c_{11},c_{21},c_{31})))
\]

\[
\text{pfmuladd}_2 \quad (((c_{01},c_{11},c_{21},c_{31})), \ldots, ((a_{02},a_{12},a_{22},a_{32})), ((c_{01},c_{11},c_{21},c_{31})))
\]

\[
\text{pfmuladd}_3 \quad (((c_{01},c_{11},c_{21},c_{31})), \ldots, ((a_{03},a_{13},a_{23},a_{33})), ((c_{01},c_{11},c_{21},c_{31})))
\]

In this way, one 4-way microSIMD FU can compute one column-vector in the result matrix in 4 cycles. If there is only one microSIMD FU, the whole \(4 \times 4\) matrix-matrix multiplication can be carried out in 16 cycles. When there are four microSIMD FU’s available, the multiplication can be done in four cycles.

The algorithm for matrix-matrix multiplication has similar scalability as that for matrix-vector multiplication.

2.3. Obtaining matrix transpose

In order to execute the \(4 \times 4\) matrix-vector multiplication algorithm above correctly, we need to transpose the transform matrices to column-major order. There are several ways to do this.

The simplest and most direct way of obtaining transposed transform matrices is to generate the original matrices in column-major order. In OpenGL, for example, the initial transform matrices are generated by functions such as \texttt{glTranslate}, \texttt{glRotate} and \texttt{glScale} or loaded by \texttt{glLoadMatrix}. In this case, it is possible to write these functions to generate and store the matrices in column-major order (\texttt{glLoadMatrix} actually take its parameters which are elements of a \(4 \times 4\) matrix in column-major order). Then all matrices that are constructed using these initial matrices are in column-major order and no subsequent matrix transpose needs to be done.

If for reasons such as compatibility or older conventions, the transform matrices are kept in row-major order, then we have to perform matrix transpose before doing matrix-vector and matrix-matrix multiplications. This can be done in several places.

Host processor Some microprocessor architectures such as HP PA-RISC\textsuperscript{3} and IA-64\textsuperscript{2} have subword rearrangement instructions. In this case, it is efficient to transpose matrices on the host processor because in general it has much higher clock rate than the graphics engine. The matrix transpose can be done using instructions like the \texttt{mix} instructions defined in both PA-RISC and IA-64. If each FP register in the host processor can hold four single-precision FP data, then the following eight \texttt{mix} operations can be used to transpose a \(4 \times 4\) matrix.

\[
\begin{align*}
\text{mixL,32}((a_{00},a_{01},a_{02},a_{03}), (a_{10},a_{11},a_{12},a_{13})) &= (a_{00},a_{10},a_{02},a_{12}) \\
\text{mixR,32}((a_{00},a_{01},a_{02},a_{03}), (a_{10},a_{11},a_{12},a_{13})) &= (a_{01},a_{11},a_{03},a_{13}) \\
\text{mixL,32}((a_{20},a_{21},a_{22},a_{23}), (a_{30},a_{31},a_{32},a_{33})) &= (a_{20},a_{30},a_{22},a_{32}) \\
\text{mixR,32}((a_{20},a_{21},a_{22},a_{23}), (a_{30},a_{31},a_{32},a_{33})) &= (a_{21},a_{31},a_{23},a_{33}) \\
\text{mixL,64}((a_{00},a_{10},a_{02},a_{12}), (a_{20},a_{30},a_{22},a_{32})) &= (a_{00},a_{10},a_{20},a_{30}) \\
\text{mixR,64}((a_{00},a_{10},a_{02},a_{12}), (a_{20},a_{30},a_{22},a_{32})) &= (a_{02},a_{12},a_{22},a_{32}) \\
\text{mixL,64}((a_{01},a_{03},a_{13}), (a_{21},a_{31},a_{23},a_{33})) &= (a_{01},a_{11},a_{21},a_{31}) \\
\text{mixR,64}((a_{01},a_{03},a_{13}), (a_{21},a_{31},a_{23},a_{33})) &= (a_{03},a_{13},a_{23},a_{33})
\end{align*}
\]

where 32 and 64 are the width of the data elements (subwords) that get mixed. In reality, PA-RISC 2.0 only has \texttt{mix} instructions defined on 16-bit and 32-bit subwords in the integer datapath, rather than the \texttt{mix} instructions for 32-bit and 64-bit subwords in the floating-point datapath that we show. Furthermore, both PA-RISC and IA-64 have floating-point registers that can only hold two single-precision FP data. For such 2-way microSIMD FP functional units, we can use eight different \texttt{mix} operations (defined as \texttt{fmix} in IA-64 floating point instructions\textsuperscript{2}) to transpose a
4 \times 4 \text{ matrix.}

\begin{align*}
mixL, 32((a_{00}, a_{01}), (a_{10}, a_{11})) &= (a_{00}, a_{10}) \\
mixR, 32((a_{00}, a_{01}), (a_{10}, a_{11})) &= (a_{01}, a_{11}) \\
mixL, 32((a_{02}, a_{03}), (a_{12}, a_{13})) &= (a_{02}, a_{12}) \\
mixR, 32((a_{02}, a_{03}), (a_{12}, a_{13})) &= (a_{03}, a_{13}) \\
mixL, 32((a_{20}, a_{21}), (a_{30}, a_{31})) &= (a_{20}, a_{30}) \\
mixR, 32((a_{20}, a_{21}), (a_{30}, a_{31})) &= (a_{21}, a_{31}) \\
mixL, 32((a_{22}, a_{23}), (a_{32}, a_{33})) &= (a_{22}, a_{32}) \\
mixR, 32((a_{22}, a_{23}), (a_{32}, a_{33})) &= (a_{23}, a_{33})
\end{align*}

**Graphics processor** If the host processor does not support subword rearrangement instructions, we may perform the matrix transpose in the graphics processor by adding subword rearrangement instructions such as *mix* to the graphics processor. The area of the additional hardware for supporting *mix* is comparable to that of a shifter of the same width. Suppose we adopt the 4-way microSIMD design, we can use the same eight 4-way *mix* operations as shown above to do the matrix transpose. We see that among the eight *mix* operations, the first four can be executed in the same cycle and so can the second four. Therefore, by adding four *mix* units, we have a two-cycle overhead for the matrix transpose.

**System bus interface** Another alternative is to perform the matrix transpose at the system bus interface. This can be done on either the sending side or the receiving side. We show the hardware used for doing it on the receiving side in Figure 4. The 32-bit data from the bus are put into the input buffer. We can get one row of a 4 \times 4 transform matrix every four bus transfers. Notice that because the bus may operate at a much higher clock speed than the graphics chip, we need to have proper buffering to ensure that the data are not corrupted when they are transferred to the transpose unit. a and b are write-enable signals for double buffering. Four cycles are needed to fill the matrix data into the shift registers and another four cycles are needed to move the transposed matrix to the registers of the graphics processor. This adds up to a latency of eight cycles. On input to the shift registers, the four elements of one row are dispersed to four shift registers. After four cycles, a whole transposed row can be output from one shift register. The two sets of shift registers are used for double buffering so that we can pipeline this process. This is beneficial when we send multiple matrices together, in which case we can achieve the throughput of one matrix transpose per four cycles. This matrix transpose during host to geometry chip bus transfers can be done on the sending end too.

Since we just need to transpose once for each of the initial transform matrices, no matter which method we use, it only incurs very small overhead if each matrix is used to transform many vertices, which is normally the case. Therefore, the actual choice of method depends mostly on the implementation cost.

### 3. MICROSIMD FU CONSTRUCTION

#### 3.1. Number of FP MAC’s to use

A full 4 \times 4 matrix-vector multiplication can be done on a 4-way microSIMD FU, which contains four FP MAC’s, in four cycles. This is as fast as we can get with the algorithm chosen. We can also transform multiple vectors at the same time to effectively bring the cycle time per vector down. For example, with four 4-way microSIMD FUs (equivalent of 16 FP MAC’s), we can effectively achieve one cycle per vector performance. For 4 \times 4 matrix-matrix multiplication, our algorithm is also scalable with respect to the number of FP MAC’s. The algorithm can finish in 16 cycles with four FP MAC’s and in four cycles with 16 FP MAC’s. Theoretically, we can achieve higher performance with more FP MAC’s. In practice, there are several factors that limit the number of FP MAC’s that we can use.

#### 3.1.1. Hardware cost

Having more FP MAC’s means higher data bandwidth going out of and into the register file. Because a multiply-accumulate instruction has three source operands and one result operand, a 32-bit single-precision FP MAC needs three 32-bit sources to produce one 32-bit result. 16 FP MAC’s would need $3 \times 32 \times 16 = 1336$ bits of source data and produce $32 \times 16 = 512$ bits of result, which is a huge number of bit lines. Having more FP MAC’s would mean more
Figure 4. Hardware for doing matrix transpose on the receiving side of the system bus.
bit lines. Although we can reduce the number of source bits by sharing data among different FU’s, we still have a large number of bit lines to cope with. We need to have either many register ports or a wide register file when there are many MAC’s. Both of these two options incur design complexity when the number of MAC’s becomes large. We also have to consider the width of the data buses and the cost of the FP MAC’s themselves.

### 3.1.2. System bus data rate

The transform matrices and vectors (vertex coordinates and normals) are transferred to the graphics chip (transform engine) via the system bus, either AGP or PCI. It is not necessary to process data at a much faster rate than what the system bus can supply, although we need the geometry transform engine to consume data faster than the peak rate of the system data bus for the highest performance geometry processing. Currently, the fastest system data bus in a PC system is a quad-speed AGP bus, whose data rate is $66 \text{MHz} \times 4 \times 4B \approx 1.066 \text{GB/s}$. For simplicity, we assume that the AGP bus is used exclusively for transferring vertices to the graphics chip and the data is not in compressed form. For the extreme case, we assume the entire bus bandwidth is used for sending vectors. This is not practically possible because a vertex has many attributes, such as coordinates, normal, material properties and texture coordinates, all of which consume bandwidth. Furthermore, each vertex may need to be moved back and forth a few times on the system bus and this also uses a lot of bandwidth. So the actual rate of sending vectors to the graphics chip is only a fraction of the 1.066GB/s. For simplicity, we only consider the extreme case here. Each vector has four components, which need 16 bytes. Therefore, roughly a maximum of 66M vectors can be transferred in one second. Assume the clock speed of the graphics chip is 100MHz, which is a safe assumption because a lot of contemporary consumer graphics chips run at speeds a lot faster than this. If we can process one vector per cycle, we can process 100M vectors per second, which is faster than the maximum bus transfer rate. If we only process one vector per two cycles, we may not meet the maximum bus transfer rate. To obtain one vector per cycle performance, we need 16 FP MAC’s. This gives a performance of 50M vertices per second when lighting is enabled (vertex coordinates and normal) and 100M vertices per second when lighting is disabled (vertex coordinates only). In practice, since AGP bus is used for many purposes other than transferring vertex data, such as sending transform matrices, properties of light sources, texture maps, commands for the graphics chip and bus communication overhead, the AGP bus can not send vertex data at the rate we assume. Considering this fact, 16 FP MAC’s are much more than enough for doing geometry transform. However, notice that geometry transform is not the only component in a 3D pipeline. We can use the FU’s for other operations as well. Operations such as lighting calculations are very complex and slow. They need to consume a lot of processing bandwidth and resources. We need to reserve resources for these operations too.

### 3.2. MicroSIMD FU organization

The width of the microSIMD FU’s needs to be carefully chosen. On one hand, we want the microSIMD FU’s to be wide. Having wide microSIMD FU’s can reduce the number of register ports and the bandwidth of the instruction stream. On the other hand, we have to have many pieces of data undergoing the same operation in order to use wide microSIMD FU’s. We also need to avoid having to rearrange the packed subwords when increasing the FU width. Because data rearrangement takes extra cycles and extra hardware, it would negate the benefit of wide microSIMD FU’s. According to our analysis of the algorithms, 4-way microSIMD is well-suited for our application. Lower way microSIMD would not fully utilize the available data parallelism in graphics transform. Higher way microSIMD will incur data rearrangement and more complexity in the design of the FU’s and the SIMD parallelism will not be fully utilized every cycle. The configuration with four 4-way microSIMD FU’s requires a 128-bit wide register file, which has 12 read ports (3 for each FU) and 5 write ports (1 for each FU and 1 for load). As a result, there will be 1536 bits lines for the source data buses and 640 bit lines for the result data buses. The number of register ports and bit lines are very large and will present a hard design problem. In the next section, we discuss optimizations based on data sharing to reduce the number of register ports and data busses.

### 3.3. Optimize to reduce register ports and data busses

When we use multiple 4-way microSIMD FU’s for $4 \times 4$ matrix-vector and matrix-matrix multiplications, it is possible to share source data among different FU’s. To see this, we look at the instruction sequences executed in each FU. For matrix-vector multiplication, we have

- `pfmuladd, 0 $Y0, X0, At0, fr0`
- `pfmuladd, 1 $Y0, X0, At1, Y0`
- `pfmuladd, 2 $Y0, X0, At2, Y0`
- `pfmuladd, 3 $Y0, X0, At3, Y0`

This reduces the number of required register ports and data busses.
where $x_i$ is the $i$-th source vector and $y_i$ is the transformed $i$-th vector, $a_{ti}$ is the $i$-th row of the matrix $A^T$. For matrix-matrix multiplication, the sequences are

\[
\text{pfmuladd}, 0 \, Ct0,\text{fr}0,\text{At0},\text{fr}0 \\
\text{pfmuladd}, 1 \, Ct0,\text{fr}0,\text{At1},\text{fr}0 \\
\text{pfmuladd}, 2 \, Ct0,\text{fr}0,\text{At2},\text{fr}0 \\
\text{pfmuladd}, 3 \, Ct0,\text{fr}0,\text{At3},\text{fr}0 \\
\text{pfmuladd}, 0 \, Ct1,\text{fr}0,\text{At0},\text{fr}0 \\
\text{pfmuladd}, 1 \, Ct1,\text{fr}0,\text{At1},\text{fr}0 \\
\text{pfmuladd}, 2 \, Ct1,\text{fr}0,\text{At2},\text{fr}0 \\
\text{pfmuladd}, 3 \, Ct1,\text{fr}0,\text{At3},\text{fr}0
\]

We see that for both matrix-vector multiplication and matrix-matrix multiplication, in every cycle, all \text{pfmuladd} instructions share the same $a_{ti}$. When we have four 4-way microSIMD FU’s, we can eliminate three register read ports and the corresponding data busses without affecting the result. Observation also reveals that the third operands of all the \text{pfmuladd} instructions, which are the operands used for accumulation, are either 0 or the results produced by the current FU in the last cycle. Therefore, we can add a latch for each FU to hold the result for that FU and feed the value back to the FU in the next cycle. In this way, we can save four more register read ports and the corresponding data busses.

3.4. The reduced datapath

The outcome of the above analysis – the reduced datapath for geometry transform is shown in Figure 5. It contains a 128-bit wide register file, which has only 5 read ports and 5 write ports, with the corresponding data busses. The number of read ports has been dramatically reduced from the theoretical value of 12 to 5 by the optimization in the previous section. Previously, 1536 bit lines for the 12 source data busses and 640 bit lines for the 5 result data busses were needed. Now only 640 bit lines are needed for the 5 source data busses and 640 bit lines for the 5 result data busses. The datapath has four 4-way FP microSIMD FU’s, each having 4 single-precision FP MAC’s. There is an accumulator(latch) for each FU to buffer the result of that FU. The values in the register file come from either the results of the microSIMD FU’s or the system data bus. The results of the microSIMD FU’s can either be written back to the register file, go to the accumulator associated with the FU, or go to the next processing stage.

3.5. The resulting new instruction

The new instruction we introduce is

\[
\text{pfmuladd}, i \, \text{frd,frs1,frs2,frs3}
\]
as defined earlier for \( i = 0, 1, 2, 3 \). If multiple microSIMD FU’s are available to perform multiple \texttt{pfmuladd} instructions simultaneously (as in Figure 5), then the same \texttt{frs1} is sent to all the FU’s, while different \texttt{frs2} are sent over separate data busses to each FU. The reduced datapath also requires the need to specify whether the third source comes from the accumulator or is the constant 0, with the result going back to the accumulator or a register. Hence, the following instruction sequence is typical for calculating a transformed vector.

\[
\text{pfmuladd}, \text{i} \quad \text{Acc,frs1,frs2}, \#0 \\
\text{pfmuladd}, \text{i} \quad \text{Acc,frs1,frs2,Acc} \\
\text{pfmuladd}, \text{i} \quad \text{Acc,frs1,frs2,Acc} \\
\text{pfmuladd}, \text{i} \quad \text{frd,frs1,frs2,Acc}
\]

We suggest the rather than introducing new instruction variants to encode this, certain register numbers are reserved to denote the accumulator, and the constant 0 source.

3.6. Scalability

The datapath shown in Figure 5 is quite expensive to implement because of the 16 FP MAC’s. In practice, in order to reduce cost, we can scale the above implementation down to get a simpler one. Because of the scalability of the algorithm, we can get proportionally scaled down performance. Figure 6 shows a reduced implementation with two 2-way microSIMD FU’s. In this case, a 4-vector \( X \) is stored in two FP registers as \( (x_0, x_1) \) and \( (x_2, x_3) \). We call these two halves \( X_L \) and \( X_H \), respectively. Similarly, a \( 4 \times 4 \) matrix \( A \) is stored in eight FP registers as \( (a_{00}, a_{01}), (a_{02}, a_{03}), (a_{10}, a_{11}), (a_{12}, a_{13}), (a_{20}, a_{21}), (a_{22}, a_{23}), (a_{30}, a_{31}), (a_{32}, a_{33}) \). We name them \( A_{0L}, A_{0H}, A_{1L}, A_{1H}, A_{2L}, A_{2H}, A_{3L}, A_{3H} \), respectively. The instructions executed on the two FU’s for matrix-vector multiplication are

\[
\begin{align*}
\text{pfmuladd}, \text{0} \quad &\text{\$Y0L,\$X0L,\$At0L,fr0} &\quad \text{pfmuladd}, \text{0} \quad &\text{\$Y1L,\$X1L,\$At0L,fr0} \\
\text{pfmuladd}, \text{1} \quad &\text{\$Y0L,\$X0L,\$At1L,$Y0L} &\quad \text{pfmuladd}, \text{1} \quad &\text{\$Y1L,\$X1L,\$At1L,$Y1L} \\
\text{pfmuladd}, \text{0} \quad &\text{\$Y0L,\$X0H,\$At2L,$Y0L} &\quad \text{pfmuladd}, \text{0} \quad &\text{\$Y1L,\$X1H,\$At2L,$Y1L} \\
\text{pfmuladd}, \text{1} \quad &\text{\$Y0L,\$X0H,\$At3L,$Y0L} &\quad \text{pfmuladd}, \text{1} \quad &\text{\$Y1L,\$X1H,\$At3L,$Y1L}
\end{align*}
\]

where \( X_0 \) and \( X_1 \) are two source vectors, \( Y_0 \) and \( Y_1 \) are two result vectors. The instructions for matrix-matrix multiplication are similar. As can be seen, the second operands of the \texttt{pfmuladd} instructions executed together are still shared among the two FU’s. Therefore, the register file of the reduced implementation only has three read ports and three write ports rather than four read ports and three write ports. With this datapath, we can perform two vector transforms per eight cycles. The effective performance is one vector transform per four cycles, which is a quarter of the performance of the implementation shown in Figure 5 when the two implementations have the same clock rate. The computing resources(FP MAC’s) are also one quarter of those for the expensive implementation. This shows the good scalability of the algorithm we choose. With the simpler datapath, in order to achieve the same performance as the complex one, we need to increase the clock speed proportionally. At 400MHz, the simpler datapath have the same performance(100M vector transforms/sec) as the complex one operating at 100Mhz.

3.7. Impact of multi-cycle execution

During the whole course of our analysis, we implicitly make the assumption that the \texttt{pfmuladd} instruction has only one execution cycle, to simplify the architecture. However, the MAC operation is slow and may become the limiting factor of the system clock rate and in turn, the system performance. This suggests that we look at multi-cycle pipelined MAC alternatives. Suppose the \texttt{pfmuladd} instruction has a 3-cycle execution latency, then because of the data dependency between the adjacent \texttt{pfmuladd} instructions for one 4-vector, we cannot issue the next \texttt{pfmuladd} instruction before the previous one finishes. This introduces a 3-cycle gap between two dependent \texttt{pfmuladd} instructions. However, this gap can be filled by interleaving the execution of the calculation for multiple independent vectors. Namely, we can scheduling the \texttt{pfmuladd} instructions as
Figure 6. A simpler datapath of the microSIMD engine for doing geometry transform.

In this way, we can still get one output per cycle, and the effective throughput is still one vector per four cycles, which remains unchanged. The difference in hardware is that the MAC unit is changed from a 1-cycle unit to 3-cycle pipelined unit. Another change is that in order to avoid the values in the accumulators shown in Figure 5 to be overwritten before they are used, we may need to replace each latch with three levels of clocked latches.

4. CONCLUSION

The geometry transform is a key component of a 3D graphics pipeline. $4 \times 4$ FP matrix-vector multiplication and $4 \times 4$ matrix-matrix multiplication are the two most commonly used operations in this stage. Doing these two operations fast is important to the performance of the whole 3D graphics pipeline. In this paper, we propose a new pfmuladd instruction for the geometry transform engine that can be used to perform these two operations efficiently. We also do an analysis on the hardware requirement and system bus bandwidth to derive the microSIMD FU organization. Finally, we propose an optimized datapath for the geometry transform engine that has four 4-way FP microSIMD
FU’s, each containing four FP MAC’s, a 128-bit wide register file that has 5 read ports and 5 write ports and the corresponding data busses. This design can effectively achieve one vector transform per cycle performance and it can also do a full $4 \times 4$ matrix-matrix multiplication in four cycles, provided each \texttt{pfmuladd} instruction is finished in one cycle. At 100MHz, this design can handle 100M vectors per second, which is faster than the data rate of the currently fastest AGP 4x bus, even if the bus is exclusively used for sending vector data. In practice, this is more than enough for just geometry transform operations. The extra resources can be used for other operations such as lighting. The necessity of these resources for fast geometry transform processing will be tempered by future studies of other computation intensive geometry stages such as lighting and clipping. The architecture we propose is scalable, and a scaled-down version using one fourth the FP MAC’s is also shown.

REFERENCES