Segmented Arithmetic Operators for Graphics Processing

Robert Rose, David Zier

***Abstract*— Graphics processing relies on executing similar instructions repeatedly on a large data set. This parallelism in the data gives rise to the Single-Instruction Multiple-Data (SIMD) paradigm which is used in modern processors. This paper explores several techniques that exploit the parallelism in the SIMD execution functional units and proposes several new SIMD methods. The methods discussed in this paper cover SIMD Addition and SIMD Matrix Multiplication.**

I. Introduction to Graphics Processing

GRAPHICS processing and multimedia programs have become the center of electronic entertainment over the recent years. There have been several great encoding schemes created to conserve the amount of space required by multimedia such as JPEG, MPEG, and MP3. Unfortunately, these schemes require a lot of additional computation in order to extract the image or movie. Additionally, 3-D graphics processing relies heavily on vector transformations, vector rasterizations, and pixel shaders–all of which have a great demand for fast matrix multiplication.

One way to improve performance in graphics processing is to take advantage of data and bus size. Most images are created by process pixels consisting of four 8-bit fields; red, green, blue, and alpha. Since recent computers operate on 32-bit buses, this gives us an opportunity to represent the entire pixel in one 32-bit value. Operations on this value are interpreted as four separate operations, also known as Single Instruction Multiple Data (SIMD).

Segmented arithmetic operators are the main source for SIMD operations. This idea gives rise to a number of optimizations in hardware that can speed up the operation over the non-SIMD counterpart. This paper will explore packed arithmetic operations for addition and multiplication, such as PAPA and a Fast Matrix Multiplication technique that we propose.

II. Single-Instruction Multiple-Data

Single-Instruction Multiple-Data (SIMD) refers to a class of operations in a CPU that perform the same operation simultaneously over a given data set. Prior to SIMD, performing the same operation over a data set involved operating on the data set in series. Although pipelining can aid this situation, SIMD now allows a compiler or clever assembly level programmer to instruct the CPU to perform the given operation simultaneously and not in series, decreasing the amount of time it would have previously taken to perform the same operations. To effectively implement SIMD in an application the we must: have additional logic in the CPU to handle the SIMD operators, the format of the data to be used in SIMD operations, and inform the compiler how to properly make use of the SIMD operations.

A. SIMD Formats

In order to perform an SIMD operation, the data set that is to be operated upon must be loaded into an SIMD register. Most SIMD architectures, if not all, specify several formats that a data set can be loaded into an SIMD register with. The formats vary depending upon the architecture, but generally an SIMD register holds 1-16 members of the data set at a time. The format also specifies the type of data, for example, signed or unsigned, integer or floating point, etc. Typically the format does not need to be specified when loading the register, the format is determined when the SIMD operator is invoked, so that the same register can be used for many different kinds of operations. An example 64-bit SIMD register is given in Figure 1.

<table>
<tr><td colspan="4">SFloat64</td></tr>
<tr><td colspan="2">SFloat32</td><td colspan="2">SFloat32</td></tr>
<tr><td colspan="2">SInt32</td><td colspan="2">SInt32</td></tr>
<tr><td>SInt16</td><td>SInt16</td><td>SInt16</td><td>SInt16</td></tr>
<tr><td>UInt16</td><td>UInt16</td><td>UInt16</td><td>UInt16</td></tr>
</table>

Fig. 1. Multiple formats may be represented in the same SIMD register

To take full advantage of SIMD operations a program's memory must be organized so that formatting the data set for SIMD registers does not incur too great of a memory access penalty. For most data sets, two different memory formats may be used: Structure of Arrays (SoA) or Array of Structures (AoS)[1]. Examples of these memory formats in C code are given in Figures 2 and 3. From the example, if the operation you wish to perform on your data set given in the examples is $A[i] + B[i] = C[i]$, you can see that formatting the data set as SoA would incur large memory access penalties as each operand is located 1000 units apart. Typically SIMD processors also have store and load operations that are the same width as the SIMD register, so formatting the data in such a way that all operators for the SIMD operation are adjacent allows them all to be loaded simultaneously, reducing memory access penalties and cache misses. Data set formatting issues such as this must be taken into consideration by the programmer if he wishes to get the most out of SIMD operations.

```
struct {
    float A[1000], B[1000], C[1000];
} SoA_data;
```

Fig. 2. Structure of Arrays formatted data

```
struct {
    float A, B, C;
} AoS_data[1000];
```

Fig. 3. Array of Structures formatted data

B. SIMD Operations

Once members of the data set are stored in an SIMD register, an SIMD operator may be applied on them. SIMD operations vary from processor to processor, but generally all include basic arithmetic operations: logical operations (and, or, xor, etc.), shifts, signed/unsigned addition, subtraction, and multiplication. Many processors include floating-point counterparts. Generally these basic operations are augmented with helper operations for conditional branching based on the SIMD registers and load/store operations. Some processors also include specialized operators for computer graphics and digital signal processing such as saturated multiplication, saturated addition and reciprocal square root. SIMD division is uncommon.

C. Popular SIMD Architectures

Six popular SIMD architectures are discussed below. A summary table comparing the architectures is given in Figure 4.

	Registers	2x32 FP	4x32 FP	2x64 FP
MMX	8			
VIS	32	✓		
3DNow!	8	✓		
SSE	16	✓	✓	
SSE2	16	✓	✓	✓
AltiVec	32	✓	✓	

Fig. 4. Brief comparison of popular SIMD architectures

1) MMX: Intel's MultiMedia eXtensions (MMX) was Intel's first foray into SIMD for consumer processors. First appearing in the i860 processor, it was later popularized in the Pentium processor where it became a regular member of the 32-bit Intel Architecture (IA32)[2]. MMX provides eight 64-bit SIMD registers (this was accomplished through sharing with the existing IA32 registers), with four integer formats for each: eight 8-bit, four 16-bit, two 32-bit and one 64-bit. The MMX operators included add, subtract, multiply, and logical operators. MMX did not support division or floating-point operations.

MMX's distinct disadvantage was that the MMX registers overlapped the regular IA32 floating-point registers, requiring programmers to separate their (integer) SIMD operations from their floating-point code in order to achieve any performance benefit[3]. MMX also required the processor to be in "MMX mode" in order to perform SIMD operations, which made context-switching between processes unnecessarily expensive.

2) VIS: The Visual Instruction Set (VIS) was introduced by Sun Microsystem in the UltraSPARC-I processor[4]. In many ways VIS was Sun's answer to Intel's MMX. Like MMX, VIS used the CPU's existing registers to perform SIMD operations. However, unlike MMX, VIS had 32 registers (vs. MMX's eight), and supported floating-point SIMD operations. VIS provides four integer and four floating-point formats: 8-bit, 16-bit, 32-bit and 64-bit. VIS operators included add, subtract, multiply, and logical operators for both integer and floating-point data sets. VIS does not support division.

VIS used the same registers for SIMD operations as the regular SPARC registers, so it also required programmers to separate their SIMD operations from their regular operations. Having many more registers (VIS added an addition 16 registers on top of the regular SPARC 16 for a total of 32) helped mitigate this downfall.

3) 3DNow!: 3DNow! is an extension to the MMX instruction set developed by AMD. 3DNow! implemented the full MMX instruction set, but added to it floating-point support. 3DNow! was first introduced in the AMD K6-2 processor family[5].

4) SSE and SSE2: Streaming SIMD Extensions (SSE), developed by Intel for the Pentium III processor, is an extension to MMX intended to compete with AMD's 3DNow! extensions. SSE adds eight new 128-bit SIMD registers that can support four 32-bit and eight 16-bit floating-point formats[6]. The Pentium 4 processor introduced SSE2, which added the capability for the eight registers in SSE to be formatted as two 64-bit floating-point values and new operators such as saturated addition and multiplication and reciprocal square root.

Although SSE does not overlap the floating-point registers as MMX did, it still shares the same circuitry, so SIMD and regular floating-point can not overlap in the processor's pipeline.

5) AltiVec: AltiVec is an extension to the PowerPC architecture first introduced in the G4 processor by Motorola. AltiVec is a complete extension–the SIMD registers are completely separate from the regular PowerPC registers. AltiVec consists of 32 128-bit SIMD registers that can be formatted as sixteen 8-bit, eight 16-bit, four 32-bit, two 64-bit and one 128-bit integer, or four 32-bit IEEE-754 floating-point values. AltiVec features all of the usual SIMD operators plus floating-point division, saturated addition and multiplication, square root and half-precision reciprocal square root, absolute value, average, and a variety highly customized SIMD load/store operators to maximize memory efficiency[7].

AltiVec has proven to be very useful as an SIMD extension set to an existing consumer processor. IBM has recently added AltiVec support to their PowerPC 970 processor.

III. SIMD Addition

SIMD addition involves adding two sets of numbers together simultaneously such that $A_0 + B_0 = C_0, A_1 + B_1 = C_1, ..., A_n + B_n = C_n$. Generally this involves taking the values stored in two registers, adding them together, and storing the result in a new register or one of the source registers. Figure 5 shows an example SIMD add operation over four values stored in two registers A and B, storing the result in register C. To reduce processor area it is often

desirable to have SIMD operations share circuitry with non-SIMD operations. This paper discusses three possible SIMD addition implementations that are designed to share circuitry with other non-SIMD addition operations: carry ripple adder, carry skip adder and packed arithmetic prefix adder (PAPA).

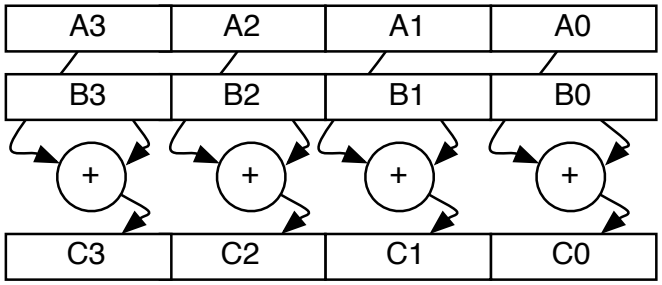

Fig. 5. Example SIMD addition operation

A. Carry Ripple Adder

Modifying a carry ripple adder (CRA) to perform SIMD is straightforward: simply stop the carry propagation at select points. Figure 6 demonstrates a possible modification to a carry ripple adder to support SIMD.

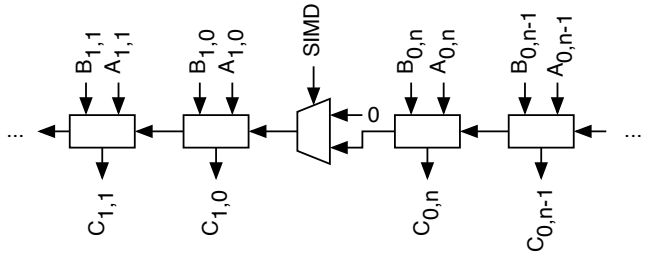

Fig. 6. Example SIMD Carry Ripple Adder

B. Carry Skip Adder

Modifying a carry skip adder (CSK) to perform SIMD is also straightforward if the CSK uses a fixed group size, but becomes more complicated if the CSK implementation uses variable group size. Effective integration of SIMD into a variable group size CSK may render the benefits of using variable group size CSK ineffective for non-SIMD operations. Figure 7 shows an example modification to a fixed group size CSK for SIMD addition.

C. Packed Arithmetic Prefix Adder (PAPA)

Neil Burgess proposed a method of utilizing the don't care states in a Carry Skip Prefix Adder to speed up the SIMD addition [8]. This proposed method is called Packed Arithmetic Prefix Adder or PAPA for short. Figure 8 illustrates the block diagram used by PAPA.

The main component is the carry skip prefix adder. It is based off a Knowles' [2,2,1,1,1] prefix adder[9] as illustrated

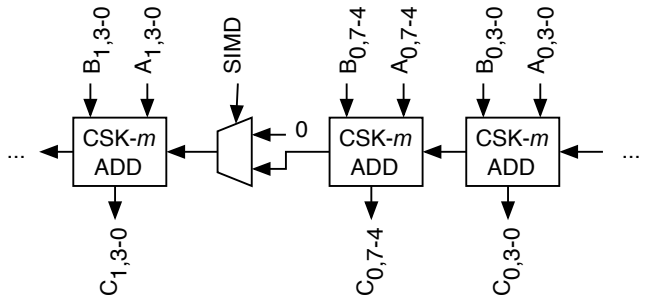

Fig. 7. Example SIMD Carry Skip Adder

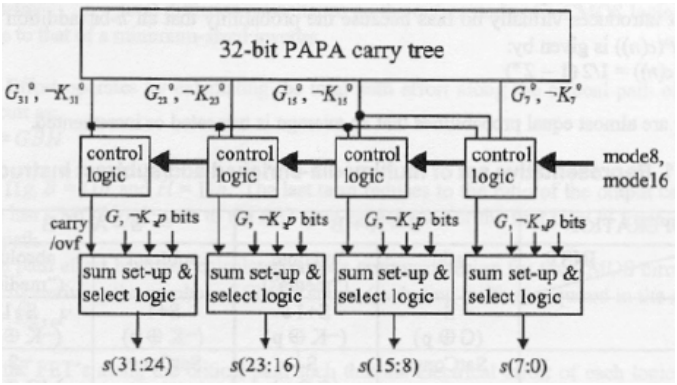

Fig. 8. PAPA Diagram

in Figure 9. The black squares are prefix cells which implement the equation pair:

$$G_z^w = G_z^y \vee \neg K_z^y \wedge G_x^w \quad (1)$$

$$\neg K_z^w = \neg K_z^y \wedge \neg K_x^w \quad (2)$$

and the grey squares are cut-down prefix cells implementing (1) only. The output sum signals are derived according to:

$$A + B : s(i) = G_{i-1}^0 \oplus p(i) \quad (3)$$

It has been proven that the final group generate and group not kill signals can be combined with control signals, denoting "*inc*" and "*abs*", to derive alternative carry signals, $c_{inc}(i)$ and $c_{abs}(i)$ respectively, that yield results related to the original sum. PAPA then introduces a fourth symbol into tree, denoted CB, that prevents carry information from traversing a column in the adder. The CB condition can be represented by the "don't care" combination $(G_z^w, \neg K_z^w) = (1, 0)$, implying that the CG (Carry Generate) condition must be represented by $(G_z^w, \neg K_z^w) = (1, 1)$ and not $(G_z^w, \neg K_z^w) = (1, X)$. These allow the creation of new cells which block the carry propagation which utilize the following equations.

$$G_z^w = G_z^y \vee \neg K_z^y \wedge G_x^w \quad (4)$$

$$\neg K_z^w = \neg K_z^y \wedge (\neg K_x^w \vee G_z^y) \quad (5)$$

With these new conditions for generate and not kill signals, PAPA is able to dynamically control SIMD addition of various sizes with little impact to the original prefix adder. These signals are used in the control logic boxes which determines which carries propagate depending on the data size denoted by the "*mode8*" and "*mode16*" signals. Finally the sum set-up and select logic module are used to finalize the carry propagation based on passed signals. This additional logic adds very minimal delay to the entire circuit.

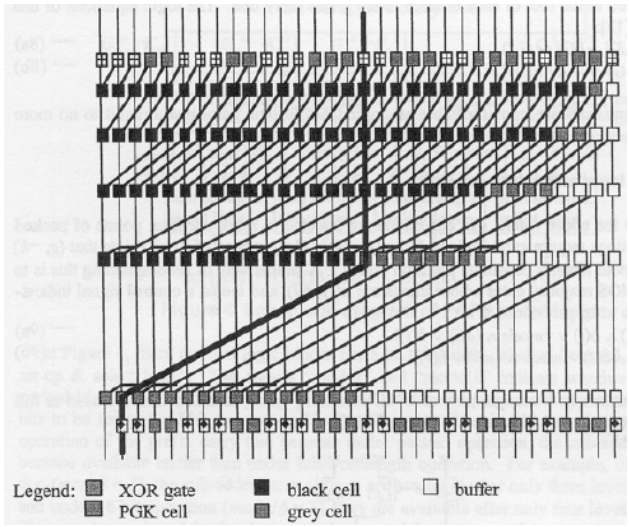

Fig. 9. Knowles' [2,2,1,1,1] Prefix Adder

Burgess also points out the modifications to the prefix adder do not increase the size of the prefix, since each cell can be represented as a single CMOS gate. The entire PAPA circuit can be represented in an area equivalent to 255 INVs with a worst case delay of $10t_{INV}$ [8].

IV. Matrix Multiplication

In the realm of computer graphics, matrix multiplication reigns supreme. Whether we are performing vector transformations or color conversions, matrix multiplication is often the fastest and most efficient way to multiply large sets of related numbers. The reason for this efficiency is the ability to represents several related equations into a single matrix multiplication. This allows for simpler algorithms and coding techniques from a programmers point of view.

But from a hardware perspective, the program will still perform countless number of individual multiplications and additions that can be very time consuming. This section will explore the inherent parallelism involved in the matrix multiplication and propose a new hardware technique for Fast Matrix Multiplication (FMM) through SIMD. Currently, there are techniques for SIMD Matrix Multiplications that use Multiply-Add (MADD) instuctions, but this paper will show that the performance can be further improved upon by adding additional hardware.

A. Saturated Multiply

When multiplying packed data, overflow is generally not an issue. Instead, saturation is used to set a minimum and maximum value for operation. For example if two RGB colors are added together, having a single component overflow and set to a different value is not the value we expect. Instead the color would be saturated to the minimum or maximum value of the color component.

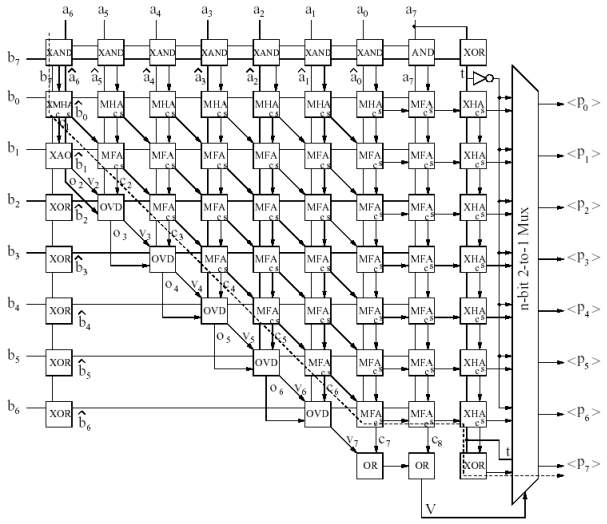

Fig. 10. Two's Complement 8-bit Saturating Multiplier

This is true for multiplication as well. Schulte, Balzola, et.al. suggested a fast and efficient way to perform saturated multiplication [10]. There method was to detect a carry out to upper bits of the result early in order to reduce the logic needed to perform the saturated multiply. Figure 10 illustrates the suggested block diagram for a Two's Complement 8-bit Saturating Multiplier.

Although Figure 10 illustrates the 8-bit signed saturated multiplier, this method can be extended to fit any n-bit signed saturated multiplier. Therefore, the multiplier requires $(n^2+5n-6)/2$ AND gates, n 2-input OR gates, $n-3$ 3-input OR gates, $(3n-1)$ XOR gates, $(2n-2)$ HAs, $(n^2-n)/2$ FAs, and an n-bit 2-to-1 MUX. The worst case delay through this multiplier is equal to the delay through one AND gate, three XOR gates, two HAs, $(n-1)$ FAs, and an n-bit 2-to-1 multiplexor.

For the examples in this section, we will be concentrating on an 8-bit signed saturated multiplier. For simplicity in this section, we will be comparing size and timing requires based on the estimated size and delay of the gates in terms of a simple inverter gate. Figure 11 describes [1] these estimations.

Taking the estimations into consideration, we get an estimated area of 295 INVs with a worst case delay of $41t_{INV}$. This delay can be further reduced by adding flip-flops and pipelining the processes. The pipeline of course adds additional gate area.

B. SIMD Matrix Multiplication

The idea behind matrix multiplication is to generate an $n \times n$ transformation matrix and multiply it against several $n \times$

[1] Estimated values for an n-bit 2-to-1 Multiplexor

Gate	Area (A_{INV})	Delay (t_{INV})
AND	1	1
OR-2	1	1
OR-3	2	1
XOR	2	2
HA	2	2
FA	5	4
MUX-n	$2n$	2

Fig. 11. Gate Size and Delay Estimations

1 vectors. Equation 6 defines how the matrix multiplication works.

$$\begin{pmatrix} y_{1,1} & \cdots & y_{n,1} \\ \vdots & \ddots & \vdots \\ y_{1,n} & \cdots & y_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} z_1 \\ \vdots \\ z_n \end{pmatrix} \quad (6)$$

Each element in the resulting matrix Z is computed by

$$z_i = x_1 \cdot y_{i,1} + \cdots + x_n \cdot y_{i,n}. \quad (7)$$

It is also important to note that this operation takes $O(n^2)$ to compute. This makes it a very computationally heavy algorithm to run in software. Fortunately, there is enough parallelism in matrix multiplication to utilize SIMD operations. Each $n \times 1$ vector can be thought of as an n-segmented SIMD register. Each $n \times n$ matrix can interpreted as n n-segmented SIMD registers. With this in mind, we see that Equation 7 can be computed in parallel with all the other resulting elements in n operation calls.

SIMD matrix multiplication is possible by transposing the matrix so that each row in the matrix has the appropriate $y_{i,j}$ component for the corresponding x_j component. Equation 8 explains the transposition. Now we can use use several multiply and add SIMD operations to perform the matrix multiply. The algorithm in Figure 12 illustrates how the multiplication takes place. Note that what looks like *several* multiplications and additions is actually a *single* SIMD instruction.

$$\begin{pmatrix} y_{1,1} & \cdots & y_{n,1} \\ \vdots & \ddots & \vdots \\ y_{1,n} & \cdots & y_{n,n} \end{pmatrix}^T = \begin{pmatrix} y_{1,1} & \cdots & y_{1,n} \\ \vdots & \ddots & \vdots \\ y_{1,n} & \cdots & y_{n,n} \end{pmatrix} \quad (8)$$

Ideally, the algorithm in Figure 12 can be further reduced by the use of a Multiply and Add (MADD) instruction. The MADD instruction is a widely used instruction in most SIMD architectures and allows us to combine the two instructions used in the SIMD Matrix Multiplication. Using this method and algorithm, Intel came up with a very fast and efficient matrix multiply routine using the Pentium's Streaming SIMD Extensions (SSE) [11][12].

Although the exact architectural implementation of Intel's SSE remains a corporate secret, the results of the algorithm is not. Figure 13 illustrates the cycle requirements of matrix multiplication of various sizes. It is important to point out that the SIMD algorithm almost doubles the speed in which the algorithm could be done without SIMD.

```
MatMul4 ( Vector a, Matrix m )
   m <= Transpose(m)
   for i = 0 to n - 1 do
       // SIMD Multiply Instruction
       t[0] <= a[0] * m[i][0]
       t[1] <= a[1] * m[i][1]
       t[2] <= a[2] * m[i][2]
       t[3] <= a[3] * m[i][3]
       // SIMD Addition Instruction
       z[0] <= z[0] + t[0]
       z[1] <= z[1] + t[1]
       z[2] <= z[2] + t[2]
       z[3] <= z[3] + t[3]
   return Vector z
```

Fig. 12. Algorithm for SIMD Matrix Multiplication

Matrix Operation	FPU (cycles)	SIMD (cycles)
$B_{3x3} \times C_{3x1}$	31	29
$B^T_{3x3} \times C_{3x1}$	31	23
$B_{3x3} \times C_{3x3}$	79	59
$B_{4x4} \times C_{4x1}$	53	31
$B^T_{4x4} \times C_{4x1}$	53	27
$B_{4x4} \times C_{4x4}$	172	90
$B_{6x6} \times C_{6x1}$	113	60
$B_{6x6} \times C_{6x6}$	652	307

Fig. 13. Results of SSE Matrix Multiplications

C. Fast Matrix Multiplication in Hardware

Although the SIMD matrix multiplication algorithm improves matrix multiplication, there is still more parallelism with the operation. In this section, we propose a Fast Matrix Multiplication (FMM) that is completely implemented in hardware. This allows for a single instruction to execute all of the required steps for Matrix Multiplication.

For the example algorithm and implementation, n was chosen to be 4. Multiplying a 4×4 matrix by a 4×1 array is a common spectacle in computer graphics. Most images are represented by Red, Green, Blue, and Alpha (RGBA) components, while 3-D transformations require X, Y, Z, and W components. We propose a new technique that uses $n = 4$ to make a significant impact on graphical processes.

1) FMM Algorithm: The FMM algorithm improves on the SIMD Matrix Multiplication algorithm from Section IV-B by removing the need to transpose the initial matrix and allowing to operate on the whole matrix at once, instead of one row at a time. The entire matrix is represented by n n-segmented registers and the multiplying array is represented in a separate n-segmented register. Additionally there is an the output result register and a temporary multiplication result register for a total of $n+3$ n-segmented registers. Because of the overhead, this algorithm might not be feasible for multiplying matrices of large n. Therefore we are concentrating on $n = 4$ and the

data set to be signed 8-bit numbers[2].

FMM operates on each of the components in parallel, one component at a time. In each cycle all the input registers are rotated 8-bits to the left out of the LSB. The bits that were shifted out are then shifted back in on the MSB, thus preserving the matrix registers for the next vector multiplication. This way we can easily chain multiple matrix multiplication back to back if they use the same base matrix. This technique is important in 3-D graphics where every vertex in the image in transformed against the same matrix or in color conversion.

Figure 14 presents pseudo code that describes the algorithm. Note that this code gives an estimate of what is performed in hardware where each loop represents a single cycle. This algorithm implements the matrix multiplication in Equation 6 with $n = 4$. Additionally, since this instruction is geared towards manipulating pixel color data, the values are saturated to 8-bit signed data on the multiplication.

```
FMM4 ( Vector a, Matrix m )
   m <= Transpose(m)
   for i = 0 to 3 do
       t[0] <= a[0] * m[0][0]
       t[1] <= a[0] * m[1][0]
       t[2] <= a[0] * m[2][0]
       t[3] <= a[0] * m[3][0]
       z[0] <= z[0] + t[0]
       z[1] <= z[1] + t[1]
       z[2] <= z[2] + t[2]
       z[3] <= z[3] + t[3]
       a <= (a >> 8) & (a[0] << 24)
       m[0] <= (m[0]>>8) & (m[0][0]<<24)
       m[1] <= (m[1]>>8) & (m[1][0]<<24)
       m[2] <= (m[2]>>8) & (m[2][0]<<24)
       m[3] <= (m[3]>>8) & (m[3][0]<<24)
   return Vector z
```

Fig. 14. Algorithm for Fast Matrix Multiplication

To further illustrate how the algorithm works, Figure 15 shows an example of multiplying the matrix illustrated in Equation 9. There is also room for more parallelization–for example–you will note that each of the row multiplications are independent of one another, thus all of the multiplications can be done simultaneously.

$$\begin{pmatrix} 16 & -2 & 4 & 0 \\ 3 & 8 & 5 & -6 \\ 5 & 9 & -4 & -9 \\ 10 & 0 & 3 & -8 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 7 \\ 4 \end{pmatrix} = \begin{pmatrix} 54 \\ 41 \\ -27 \\ 9 \end{pmatrix} \quad (9)$$

2) FMM Implementation: The FMM algorithm can be implemented with $(n + 1)$ Rotate-m (mn)-bit Registers, one (mn)-bit register, n m-bit Saturating Multipliers, and an SIMD CRA. The algorithm is generic and can be used on any format of any size, both floating-point and fixed-point. The only difference is changing the Saturating Multipliers and CRA.

[2]32-bit Color Bitmaps represent each pixel as a 32-integer that contains Red, Green, Blue, and Alpha components, each of 8-bits.

Cycle	M				A	T	Z
0	16	−2	4	**0**	2	0	0
	3	8	5	**−6**	3	0	0
	5	9	−4	**−9**	7	0	0
	10	0	3	**−8**	**4**	0	0
1	0	16	−2	**4**	4	0	0
	−6	3	8	**5**	2	−24	−24
	−9	5	9	**−4**	3	−36	−36
	−8	10	0	**3**	**7**	−32	−32
2	4	0	16	**−2**	7	28	28
	5	−6	3	**8**	4	35	11
	−4	−9	5	**9**	2	−28	−64
	3	−8	10	**0**	**3**	21	−11
3	−2	4	0	**16**	3	−6	22
	8	5	−6	**3**	7	24	35
	9	−4	−9	**5**	4	27	−37
	0	3	−8	**10**	**2**	0	−11
4	16	−2	4	0	2	32	54
	3	8	5	−6	3	6	41
	5	9	−4	−9	7	10	−27
	10	0	3	−8	4	20	9

Fig. 15. FFM Algorithm Demonstration

Additionally, if you choose to perform all the multiplications at the same time, this would require a total of n^2 Saturating Multipliers and an additional Carry-Save Adder to add all n intermediate values together simultaneously.

For the purposes of this paper we are concentrating on a FMM operation with $m = 8$ and $n = 4$ using signed integers. Figure 16 illustrates the block diagram layout for the multiplier. The saturated multipliers used in this example are described in Section IV-A. The CPA used in this example is the PAPA adder that was described in Section III-C. Note that the diagram in Figure 16 does not include timing or control logic.

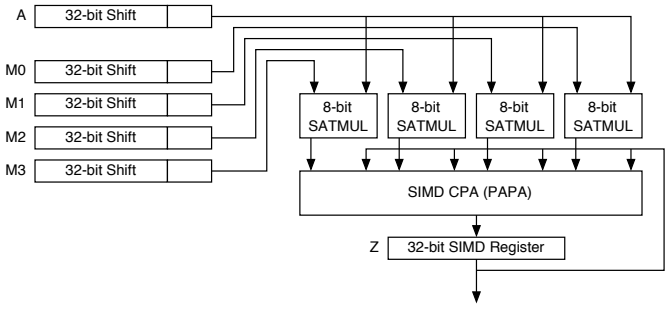

Fig. 16. Implementation of FMM

3) FMM Results: In order to get comparable results in the area and delay, we will use the gate size and delay estimates from Table 11. For accurate size, we will also make some assumptions about the register sizes. A standard m-bit register will require m flip-flops. Each flip-flop can be thought of as having the same area as 4 INVs. Thus for our example, the 32-bit register has an area of 128 INVs. The delay in the register is negligible since it does not directly affect the path delay. The m-bit Rotate-n register is very similar to the m-bit

register, except that it has additional circuitry for the rotation. Thus the area of a 32-bit Rotate-8 register is approximately 140 INVs.

Thus the total area for the FMM ciruit is given by the following equation:

$$AREA = 5A_{rotate} + A_{reg} + 4A_{SatMul} + A_{PAPA}$$

This makes the total area of the circuit equivalent to 2179 INVs. And the total delay can be thought of:

$$t_{Delay} = t_{SatMul} + t_{PAPA}$$

Therefore the total delay has an estimated delay of $65t_{INV}$. This delay is very comparable to a standard Multiply and Add function since it requires the same circuitry. Although the exact area and delay aren't known for Pentium's SSE implementation, we can get a comparable estimation. Table I gives an overview over of all the mentioned matrix multiplication techniques. The FMM2 circuit mentioned in the result table is just the normal FMM circuit with all the multiplications done at once and an additional CSA to add four values together.

TABLE I
MATRIX MULTIPLICATION RESULTS

	MADD	FMM	FMM2
Area (INV)	1607	2179	5699
Delay (t_{INV})	65	65	72
Cycles	27	5	2

Using FMM over the currently used traditional method gives us a 540% improvement on the clock cycles and FMM2 gives us a 1350% improvement! But there are some sever draw backs in choosing this design. The foremost drawback is the area requirement. FMM requires an additional 135% area whereas FMM2 requires an additional 355% area. These values are not exact, they are estimations on the potential significants of the FMM algorithm. A potential drawback to using FMM is that it is not dynamic in relation to the format of the SIMD register–in order to generated a FMM module for another SIMD format, a completely new and seperate module must be created. Additionally, the delay for everything can be reduced by inserting a register in between the multipliers and CPA.

V. CONCLUSION

The realm of graphics processing is vast and would require several books to come close to covering all aspects. This paper covered some of the basics involved in graphics processing and showed examples of several specific areas. More importantly, graphics processing involves many computations that are inherently parallel in nature which creates room for improved both from a software perspective and hardware perspective.

This paper explored graphics processing from a computer arithmetic point of view and explored SIMD operations. By using SIMD operations existing methods for performing operations serially were shown to be improved, as in PAPA. Additionally, we proposed a new technique which could be used to significantly improve matrix multiplications. Further investigation is warranted to develop an FMM circuit that can be used with different SIMD formats and matrix sizes, and to discover if alternate encoding formats, such as Carry-Save, can aid FMM.

In conclusion, the area of computer graphics processing is always changing and reinventing itself which leaves a wide room open for improvements and enhancements.

REFERENCES

[1] W.-C. Ma and C.-L. Yang, "Using intel streaming simd extensions for 3d geometry processing," 2002. [Online]. Available: citeseer.nj.nec.com/ma02using.html
[2] M. Mittal, A. Peleg, and U. Weiser, "Mmx technology overview," *Intel Technology Journal*, no. Q3, 1997.
[3] Internet, "http://en2.wikipedia.org/wiki/MMX," 2003.
[4] S. Microsystems, *VIS Instruction Set User's Manual*, 1997.
[5] A. M. Devices, *AMD Technology Manual*, 2000.
[6] M. Bargeron, T. Craver, and M. Phlipot, "Applications tuning for streaming simd extensions," *Intel Technology Journal*, no. Q2, 1999.
[7] I. Oallmann, "Altivec," CalTech, Tech. Rep., 2003.
[8] N. Burgess, "PAPA - Packed Arithmetic on a Prefix Adder for Multimedia Applications," in *The IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'02)*. IEEE, 2002, pp. 197 – 207.
[9] S. C. Knowles, "A Family of Adders," in *14th IEEE Symp. Computer Arithematic*, Adelaide, April 1999, pp. 30–34.
[10] M. J. Schulte, P. I. Balzola, A. Akkas, and R. W. Brocato, "Integer Multiplication with Overflow Detection or Saturation," in *IEEE Transactions on Computers*, vol. 49, no. 6, June 2000, pp. 100 – 110.
[11] R. Zohar and Z. Devir, "Optimized Matrix Library for use with the Intel Pentium 4 Processor's Streaming SIMD Extensions," Intel, Tech. Rep., 2001.
[12] Intel, "Streaming SIMD Extensions - Matrix Multiplication," Intel, Tech. Rep. AP-930, June 1999.