Accelerating 3D Geometry Transformation with Intel MMX™ Technology

ECE 734 – Project Report

by

Pei Qi

Yang Wang

Content

Abstract

Three-dimensional (3D) graphics geometry transformation involves very heavy computation load running on micro computer system. The appearance of Intel MMX^{TM} Technology increases the processing power dramatically and thereby speeds up the 3D geometry transformation as well. In this project, the implementation of 3D geometry transformation is implemented under MMX framework and the potential optimization at instruction level is examined to further speed up the execution of MMX instructions code. Reversing (renaming) registers and reordering instructions are two mainly typical methods used to pair instructions such that they could be executed within the same cycle in pipelined processor. Finally, the correctness and performance of our approach is tested through two simulations under Matlab and PLX framework respectively. At last, conclusion can be made that MMX^{TM} Technology effectively accelerates 3D geometry transformation by providing multimedia extensions that are tailored to 2D or 3D graphics processing. In addition to that, through hand optimizing the assembly code, the execution time would be reduced by pairing more instructions and executing them in parallel.

Introduction

2.1 Three-dimensional Geometry Transformation

The three-dimensional objects are usually represented by thousands of polygons. Each vertex of polygons can be represented by a 4-element vector.

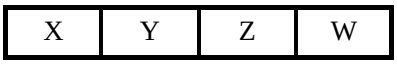

Fig. 1 Representation of 3D Object

X, Y, and Z are coordinates of each polygon in the 3D space. W contains perspective corrective information, which is essentially unused in our project for simplicity.

When a 3D object performs a geometry transformation, such as shift, rotation, expansion and so on, we have to recalculate polygon shape which is actually matrix multiplication.

$$V_T = M \times V_O$$

$$\begin{bmatrix} X_T \\ Y_T \\ Z_T \\ W_T \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \\ c_0 & c_1 & c_2 & c_3 \\ d_0 & d_1 & d_2 & d_3 \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \Rightarrow \begin{aligned} X_T &= a_0 * X + a_1 * Y + a_2 * Z + a_3 * 1 \\ Y_T &= b_0 * X + b_1 * Y + b_2 * Z + b_3 * 1 \\ Z_T &= c_0 * X + c_1 * Y + c_2 * Z + c_3 * 1 \\ W_T &= d_0 * X + d_1 * Y + d_2 * Z + d_3 * 1 \end{aligned}$$

Fig. 2 Matrix-based 3D Geometry Transformation

Where, V_o is the original vertex vector, M is the transformation matrix, and V_T is the transformed vertex vector. Here we do not investigate the details about transformation itself, like how to construct transformation matrix M etc. The only concern here is the computation issue involved with transformation. From the above equations, obviously this operation will definitely incur a computation overhead since each vertex pixel requires 16 multiplies and 12 additions. However, there is also a lot of inherent parallelism which could be exploited to speedup the recalculation process.

2.2 Intel MMX™ Technology

The Intel Architecture MMX™ Technology provides a significant increase in processing power, through the use of its new single-instruction multiple-data (SIMD) extensions. MMX instruction set features 57 SIMD-type instructions and eight 64-bit wide MMX registers. In addition to that, 4 new data types are introduced into MMX instruction set to accelerate the processing by calculating in parallel. Thus, two 32-bit, four 16-bit, or eight 8-bit integer values can be operated on at once.

In our project, we mainly investigate the “PMADDWD” instruction, which is new in MMX instruction and means Packed Multiply and Add, and testify how this instruction speeds up the computation.

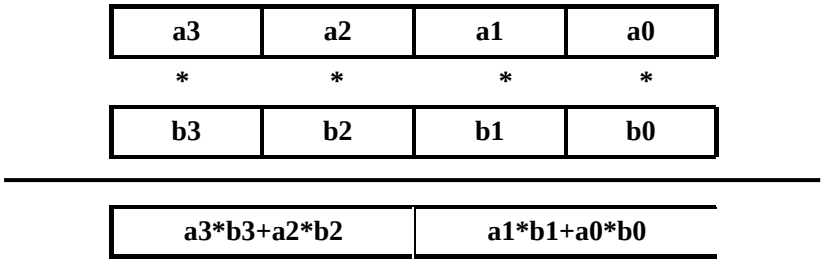

Fig. 3 PMADDWD Instruction

“PMADDWD” instruction can perform both multiply and add on four 16-bit values simultaneously. Therefore, it is an excellent match for the four-element multiply/adds operation, such as 3D geometry transformation. Thus, this instruction reduces the previous workload to 4 multipliers and 2 adders for each of vertex pixel.

Implementation

3.1 Execution before optimization

In the routine executing 3D geometry transformation, the critical section is the multiplication of two matrices: *M* and V_o. The code snippet of the below demonstrates the essence of the execution of transformation using MMX™ instruction set.

```
mov eax,[esp]+ 4
mov ebx,[esp]+ 8
mov ecx,[esp]+12
mov edx,[esp]+16

movq mm0, 0[eax]
NextVect:
movq mm3,[ebx]
movq mm4,mm3
pmaddwd mm4,mm0
movq mm5,mm4
psrlq mm5,32
paddd mm4,mm5
moved [edx],mm4
```

Fig. 4 Code Snippet

16bits	16bits	16bits	16bits	
a0	**a1**	**a2**	**a3**	movq,mm3
X	**Y**	**Z**	**1**	movq,mm0
				pmaddwd
a0*X + a1*Y		**a2*Z + a3*1**		mm4
		a0*X + a1*Y		movq,mm5
				movq,mm4
	(a0*X+a1*Y) + (a2*Z+a3*1)			movq,edx

Fig. 5 Corresponding Register Status

The above code is a demonstration of MMX™ instruction. mm0 ~ mm5 are temporary MMX registers for execution. First, the code pushes in pixel value of a single point x, y, z. and a correction vector 1 together into a 64 bit register mm0. It also takes in a single row of the transformation matrix *M* and stored it in a 64 bit register named mm3. A copy of mm3 is made and called mm4. mm4 was used to do “PMADDWD” instruction with mm0. After the instruction is done, the result is stored back into mm4. A copy of mm4 is made and called mm5. mm5 is shifted 32 bit to the right. mm4 and mm5 is add together and the information is stored in mm4. At the end, mm4, the result is stored in temporary pointer edx for further use.

Optimization

The speed of MMX routine, like other assembly code routines, could benefit greatly from hand optimization. Reversing or renaming registers and reordering instructions are two usually used methods to accelerate the execution of code. The basic idea is to make better use of the super-scalar pipelines, which enables two or more instructions to be executed simultaneously. In other words, we can execute two or more instruction within the same clock cycle and thereby reduce the total execution time eventually.

3.2 Reversing Registers

The purpose of reversing registers is to eliminate the data dependency caused by unintended using temporary registers or using them in inappropriate sequence. A typical problem in this regard is shown as the following:

```
1  movq mm3, [ebx]
2  movq mm4, mm3
3  pmaddwd mm4 mm0
```

Fig. 6 Before Reversing

```
1  movq mm4, [ebx]
2  movq mm3, mm4
3  pmaddwd mm4 mm0
```

Fig. 7 After Reversing

Fig.6 and Fig.7 will yield the completely same result. In Fig.6, instruction 2 reads from mm3, and then writes to mm4, while instruction 3 reads from mm0 and writes to mm4. Obviously it is impossible to execute instruction 2 and 3 simultaneously since both of them write to mm4, which is a typical W-W data dependency. However in Fig.7, after reversing registers mm3 and mm4, instruction 2 read first from mm4 to mm3 and then instruction 3 writes to mm4. The previous data dependency no longer existed. We are able to pair the instruction 2 and 3 and execute them simultaneously. The cycles needed to execute these instructions are also reduced to 2 from early 3 and the execution time is also decreased consequently. This example clearly demonstrates that how to eliminate data dependency by simply reversing involved registers.

3.3 Re-ordering instructions

Reversing registers can only eliminate data dependency locally in assembly code. However when facing the whole segment of code, we need to consider re-ordering the

execution of instructions and thereby break the whole dependency chains affiliated to the present sequence in which instructions are executed. The ultimate goal is to get more instruction paired.

```
NextVect:
1 movq mm4, [ebx]
2 movq mm3, mm4
3 pmaddwd mm4, mm0

4 movq mm5, mm4
5 psrlq mm4, 32
6 paddd mm5, mm4

7 movd [edx], mm5
8 movq mm4, mm3
9 pmaddwd mm4, mm1
10 movq mm5, mm4
11 psrlq mm4, 32
12 paddd mm5, mm4

13 movd [edx+2], mm5
14 movq mm4, mm3
15 pmaddwd mm4, mm2
16 movq mm5, mm4
17 psrlq mm4, 32
18 paddd mm5, mm4

19 movd [edx+4], mm5
20 add ebx, 8
21 add edx, 6

22 dec ecx 1
23 jnz NextVect
```

Fig. 8 Before Re-ordering

```
NextVect:
1 movq mm3, [ebx]
2 movq mm4, mm3
3 movq mm5, mm4
4 pmaddwd mm3, mm0

5 pmaddwd mm4, mm1
6 pmaddwd mm5, mm2
7 movq mm6, mm3
8 psrlq mm3, 32

9 paddd mm3, mm6
10 movq mm6, mm4
11 psrlq mm4, 32

12 paddd mm4, mm6
13 movq mm6, mm5
14 psrlq mm5, 32

15 paddd mm5, mm6
16 movd [edx], mm3
17 movd [edx+2], mm4
18 movd [edx+4], mm5
19 add edx, 6
20 add ebx, 8 1 - 1

21 dec ecx 1
22 jnz NextVect 1 - 1
```

Fig.9 After Re-ordering

Fig.9 demonstrates that how re-order the sequence of execution shown in Fig.8 and thereby get more instructions paired. A very important issue must be presented at this point that all the optimizations cannot affect or change the correctness of original code. This is the prerequisite for any optimization and we develop a Matlab program to do the same operation and then compare the results from two executions. This concern will be more discussed in next section.

Simulation

4.1 Correctness

We just mentioned in last section, correctness is our first concern when doing optimization. In our project, we compare the two results. One was simulating the execution of optimized code in instruction form. The other was directly multiplying transformation matrix M and original vertex vector V_o. Our optimized code yielded the same result with later and therefore testified the correctness of the optimized code. The only exception is when a very large value is used for coordinates X, Y or Z, the optimized code showed different result due to the truncation applied to keep the 16-bit integer representation.

4.2 Performance Improvement

In our simulation, we testified the performance of optimized code measured with execution time and cycle consumption using Matlab and PLX framework respectively.

Matlab

Test Case	Execution Time (sec)	
	Un-Optimized	Optimized
Vertices number: 320	0.644	0.6118
Vertices number: 3200	5.6943	5.601
Vertices number: 32000	78.0654	77.0027

Fig. 10 Execution Time Comparison Using Matlab

From the above table, we found that the execution time was reduced after doing optimization. Especially this trend became more obvious when the vertices number was increased.

PLX

Test Case	Cycles	
	Un-Optimized	**Optimized**
Single set instruction iteration	60	52

Fig. 11 Cycle Count Using PLX

Figure 11 shows cycle count comparison between un-optimized result and optimized one. The result shows a 8 cycle reduction in a single instruction iteration. This number will become more significant when multiple iterations executed. Since this number increase linearly, the result of multiple iterations is provided here.

Conclusion

The new instructions in MMX™ Technology can effectively accelerate 3D geometry transformation. In our project, we first illustrated how “PMADDWD” instruction can handle and save computation in hardware level. We then simulated our implementation with Matlab and PLX simulators.

By rearranging instruction execution time, we can further reduce the total number of clock cycle, which makes our program more efficient. Both simulation results show clock cycle reduction. As a result, these results proves that we can improve the performance of execution through some optimizations at instruction level, such as reversing register, re-ordering instruction without sacrificing the correctness of code.

References

[1] W. Ma, C. Yang, "*Using Intel SIMD Extension for 3D Geometry Processing*", 2000

L. Gwennap, "*Intel's MMX Speeds Multimedia*", Vol 10(3), 1996

[2] C. Yang, B. Sano, A. R. Lebeck, "*Expliting Parallelism in Geometry Processing with General Purpose Processors and Floating-Point SIMD Instructions*", Vol 49(9), September 2000

[3] MMX™ Technology Application Notes, Intel, "*Using MMX™ Instructions to Perform 3D Geometry Transformation*"

[4] H. Nguyan, L. K. John, "*Exploiting SIMD parallelism in DSP and Multimedia Algorithm Using AltiVec Technology*", In ICS99, 1999

[5] K. Akelay, T. Jermoluk, "*Hight-Performance Polygon Rendering*", In *Computer Graphics,* pages 239-246, 1988

Percentage Contributions

Tasks performed	Pei Qi	Yang Wang
Literature search	50%	50%
Method discussion	50%	50%
Power point presentation	50%	50%
Matlab programs	50%	50%
Testing case preparation	50%	50%
Report writing	50%	50%
Overall	50%	50%

Team Member#1: Pei Qi _____

Signature

Team Member#2: Yang Wang _____

Signature