Design of a Programmable Vertex Processing Unit for Mobile Platforms

Tae-Young Kim[1], Kyoung-Su Oh[2]

[1] Dept. of Computer Engineering, Seokyeong University
136704 Seoul, Korea
tykim@skuniv.ac.kr
[2]Dept. of Media, Soongsil University
156743 Seoul, Korea
oks@ssu.ac.kr

Abstract. Programmable vertex processing unit increases the programmability and enables customizations of transformation and lighting in the graphics pipeline. Furthermore, it offers various effects such as procedural vertex animation and deformation, which were impossible to handle in fixed vertex processing. Since it is hard to find a programmable graphics hardware for the embedded systems such as mobile phones, we've designed and implemented a programmable vertex processing unit based on the OpenGL ES 2.0 specification. In this paper, we explain the architecture, instruction format, implementation and test results of our vertex processing unit.

1 Introduction

In last a few decades, much research has been done to enhance the functionality and efficiency of graphics hardware [1]. Recently, there has been a great deal of advances in functionality of graphics hardware. One of them is the programmable graphics pipeline. It provides a programmer with the full control of the vertex and fragment processes. Various effects such as procedural vertex animation which were impossible with the fixed pipeline can be implemented using the programmable graphics hardware [2-4].

Fig. 1 shows a typical graphics pipeline and also fig. 2 shows the vertex processing operations in detail. The vertex processing in the programmable pipeline does not use the fixed functions T&L (Transformation & Lighting) in fig. 2 but a vertex program written by a programmer. It gives the programmer total control of vertex processing.

Even though the programmable graphics hardware is common in the desktop environments, it is true that mobile phones with such solutions are rare or not yet released. Therfore we have designed and implemented a programmable vertex processing unit for the embedded systems.

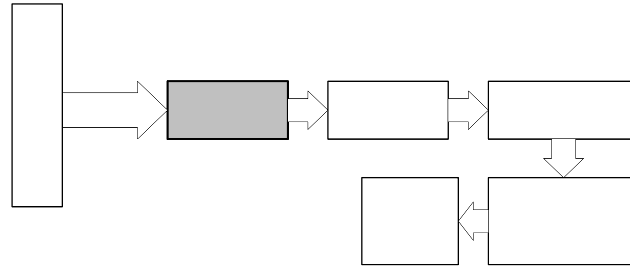
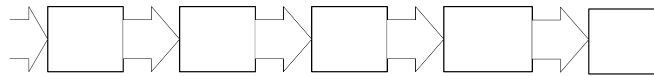
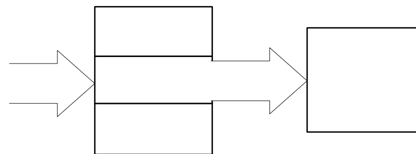


Fig. 1. A typical graphics pipeline



(a)



(b)

Fig. 2. Operation of traditional fixed T&L engine. (a) coordinate transform (b) lighting

Our vertex processing unit is designed based on the OpenGL ES 2.0 [5] and GL_ARB_vertex_program [6]. The GL_ARB_vertex_program is the specification of assembly shading language for programmable graphics processor in the general computing systems. OpenGL ES is a graphics APIs standard for the embedded systems. OpenGL ES 2.0 specifies graphics APIs and high level shading language for the programmable vertex and fragment programs [6-7]. But it does not include low level specification such as assembly language or machine code of the shading language [8]. We tested the feasibility of using assembly language instructions specified in GL_ARB_vertex_program with a DirectX high level shading language compiler. And we found out the fact that most of high level shading language features specified in OpenGL ES 2.0 can be compiled into assembly language instructions specified in GL_ARB_vertex_program. We modified GL_ARB_vertex_program assembly language to fully support OpenGL ES 2.0. In other words, we added some instructions and substituted an instruction with several other primitive instructions to encode/decode an instruction efficiently. These are our main contributions.

In this paper, we present the architecture of our vertex processing. We define 28 instructions and 3 macro instructions, and design 64bit machine code to encode the assembly language instructions. Our instruction design and operand encoding scheme can be used as an interface standard between low-level and high-level shader language.

In section 2 we present the structure of our vertex processing unit. And in section 3 we explain the instruction format. Our implementation and results are in the next section. We conclude in section 5.

2 The Vertex Processing Unit Architecture

A vertex program is a sequence of floating-point 4-component vector operations that determines how a set of program parameters and a set of per-vertex input parameters are transformed to a set of per-vertex result parameters. Fig. 3 shows the architecture of our vertex processing unit, based on the GL_ARB_vertex_program.

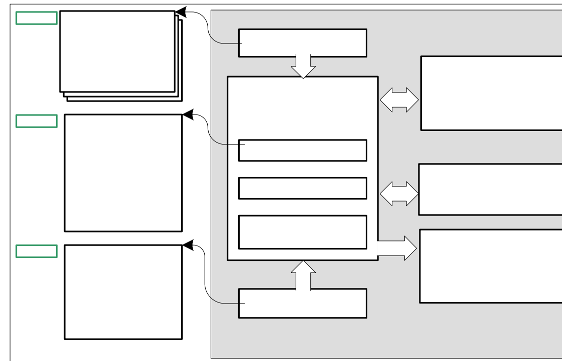


Fig. 3. The Vertex Processing Unit Architecture

- Machine code: Binary codes to be executed in vertex processing unit. Up to 128 machine codes can be handled.
- Vertex Processing Unit: A processing engine that fetches, decodes and operates each machine code, respectively. It reads vertex data and writes the processing results such as position, color, and texture coordinates to vertex output registers.
- Vertex data: A set of 16 read-only registers containing 4-component floating point vector. Each register are for position, colors, normal of vertex and user-defined vertex parameters. The data is passed through normal OpenGL mechanisms (per-vertex API or vertex arrays).
- Constant Registers: A set of 96 read-only registers containing 4-component floating point vector. It stores parameters such as matrices, lighting parameters and constants required by vertex programs. They are modifiable only outside of glBegin/glEnd pair.
- Temporary Registers: A set of 16 readable and writable registers containing 4-component floating point vector to hold temporary results that can be read or written during the execution of a vertex program.

- Address Register: A register containing an integer that can be used as an index to perform indirect accesses to constant data during the execution of a vertex program.
- Output Registers: A set of 16 write-only registers containing 4-component floating point vectors to hold the final results of a vertex program. Vertex program outputs are passed to the remaining graphics pipelines. These outputs correspond to the set of vertex attributes, i.e. homogeneous clip space position, primary, secondary colors, fog coordinates, point size, texture coordinates, used during primitive assembly and rasterization.

3 Instruction format

A vertex program is programmed using low-level assembly language and assembled into machine codes which can be executable by the vertex processing unit. In the GL_ARB_vertex_program specification, 27 assembly language instructions are presented [6].

In this paper, we define 28 primitive instructions and 3 macro instructions based on the operation processing method, as shown tables 1 and 2. Since macro instruction means an instruction which can be replaced by a series of primitive instructions, each one is translated into multiple primitive instructions in assembling time. Therefore our vertex processing unit handles only the primitive instructions. In table 1, the instructions in shadowed parts are additional instructions which are not included in the GL_ARB_vertex_program instruction set. We added them in order to implement the macro instructions in table 2.

The instruction is composed of an opcode, a destination operand, and up to 3 source operands, respectively as shown in fig. 4:

Fig. 4. Instruction format.

Fig. 5 shows the 64bit machine code structure. The low bit fields [4th ~ 18th bit] can be used as a source operand (Src_2) field or an extended swizzle field. They are recognized as a source operand field in MAD instruction, and as an extended swizzle field in other cases. MAD is the only instruction which have three source operands.

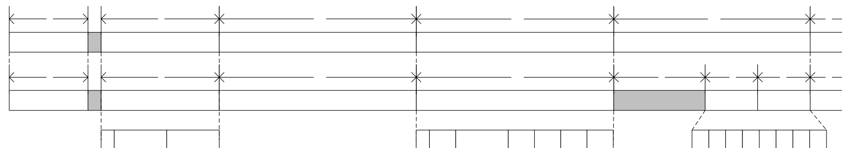


Fig. 5. Machine code fomat (: unused bit)

Table 1. Primitive instructions.

Table 2. Macro instructions.

Macro Instruction	description
LIT	: Light coefficients LIT f, a, b *clamp tmp, a.0, b* *rLG2 tmp.w, tmp.w* *MUL tmp.w, tmp.w, tmp.y* *rEX2 tmp.w, tmp.w* *mulz f, tmp.1xz1, tmp.w*
POW	: Power POW f, a, b ($f = a^b$) *LG2 tmp, a* *MUL tmp, tmp, b* *EX2 f, tmp*
SUB	: Subtraction SUB f, a, b (f = a - b) *ADD f, a, -b*

Each field of the machine code has the following bit format:

- Opcode field: opcode (Fig. 6)

The opcode has 6 bits, so it is possible up to 64 instructions.

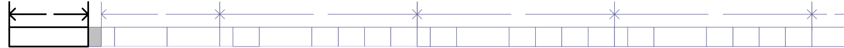


Fig. 6. Machine code format (opcode field)

- Destination operand field: Dest (Fig. 7)

The destination field (register type, index, and mask information) has 9 bits. Each bit is translated as follows:

T (1bit) : type / 0 (Temporary register)
/ 1 (Output register)
index (4bits) : register index (0~15)
mask (4bits) : mask flag for each component

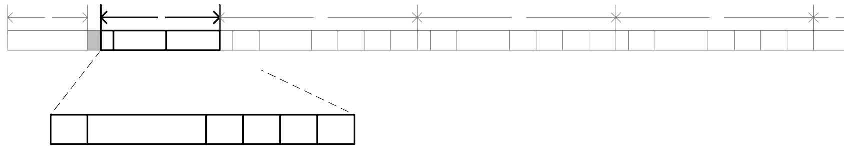


Fig. 7. Machine code format (Destination operand field)

- Source operand(n) field: Src(n), $0 \leq n \leq 2$ (Fig. 8)

The source field (register type, index, and swizzle information) has 15 bits. Each bit is translated as follows:

neg(1bit): negation flag
type(2bits): type / 00(Temporary register)
/ 01(Vertex data)
/ 10(Constant register, absolute addressing)
/ 11(Constant register, relative addressing)
index (4bits): register index (0~15)
Src_?(2bits): component swizzle / 00(x component) 01(y component)
/ 10(z component) 11(w component)

6
opcode
T index
1 4
Destinatio

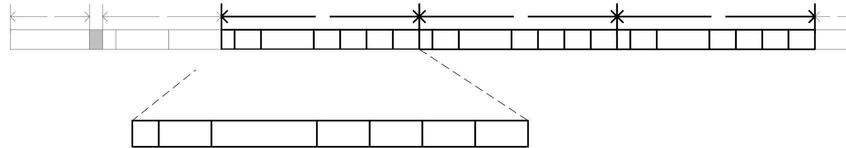


Fig. 8. Machine code format (Source operand field)

- Exteneded swizzle field (Fig. 9)

The extended swizzle field has additional swizzle information of source operand 0. With swizzle information, four components of source operand 0 can be negated or changed with other components value, zero or one. For example, if the swizzle suffix is ".yzzx" and the specified source register value is contains {2,8,9,0}, the swizzled operand used by the instruction is {8,9,9,2}.

```
PARAM Colr = {5, 6, 7, 8};
TEMP Tmp1, Tmp2;
SWZ Tmp1, Colr.xy01;    // Tmp1 = { 5, 6, 0, 1};
SWZ Tmp2, Colr.-x-yz1; // Tmp2 = {-5,-6,7, 1};
```

In this field, $N_?$ and $S_?$ mean negation and zero or one value flags for each component.

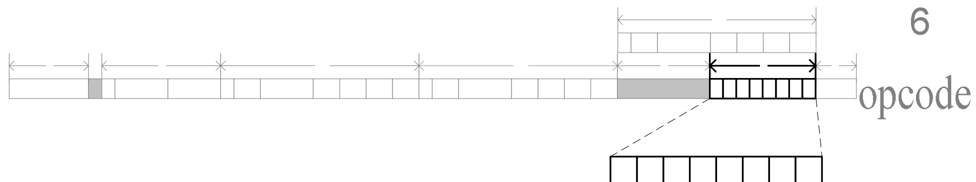


Fig. 9. Machine code format (Extended swizzle field)

- Extended constant index field (Fig. 10)

The extended index field has 3 bits, which is used for indexing the location of constant register. Totally, 7 bits indexing is possible with the 4 bits in the source operand field and the 3 bits in the extended constant index field.

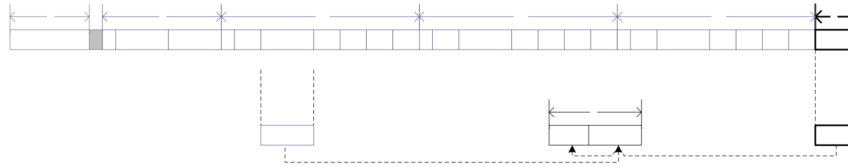


Fig. 10. Machine code format (Extended constant index field)

4. Implementation and Test Results

We implemented our programmable vertex processing unit in software emulation. Our implementation can be used to emulate mobile applications including vertex programs on PC. We tested the performance of our work on a desktop PC with 4.3 GHz Pentium processor and ATI Radeon 9800 XT graphics card. We implemented only vertex processing unit. Vertices that are transformed and lighted by our vertex processing unit are sent to OpenGL system installed on desktop system.

To test our vertex processing unit, we implemented the OpenGL ES 2.0 APIs related with vertex processing. Using the OpenGL ES 2.0 APIs, vertex data are stored and passed to our vertex processing unit. A vertex program written by programmers is assembled into machine codes and they are passed to the vertex processing unit by our APIs. The vertex processing unit calculates the position and the color of each vertex by fetching, decoding, and executing the machine codes. The outputs of our vertex processing unit are collected and are sent to the OpenGL graphics pipeline installed in our computer via the original OpenGL APIs. Therefore, the vertex processing is performed by our vertex processing unit and the remaining operations, after vertex processing such as triangle setup, rasterization and fragment processing, are performed by the original OpenGL pipeline. (Fig. 11)

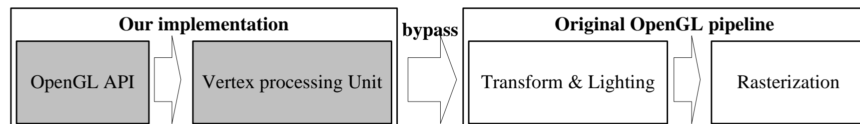


Fig. 11. Overview of our implementation

The arithmetic unit in our vertex processing unit supports 24 bit floating point format which satisfies the requirement of the OpenGL ES 2.0. We tested 3 vertex programs listed in fig. 12.

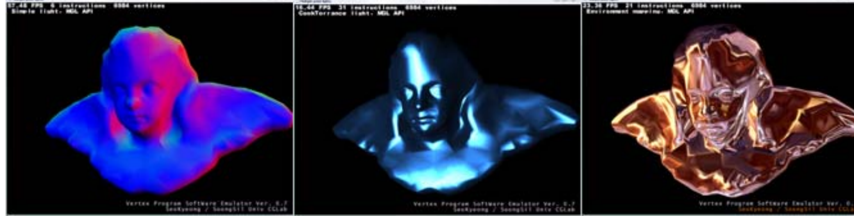


Fig. 12. Test vertex programs, left: Normal value, middle: Cook-Torrance illumination [9], right: Environment map. All programs use same model whose vertex count is 6,984.

We compared an image rendered by our system with an image rendered by pure OpenGL system on PC. In all test scenes, we could not find differences with naked eye. Therefore, we computed color difference images and turned all non black pixel values into white to exaggerate the differences. The result is fig. 13. There were some errors near the boundary of the model due to numerical resolution differences between 32 bit floating point arithmetic unit and 24 bit unit. As you can see, sample 3 shows more errors compared to samples 1 and 2 because in environment mapping, small changes in normal values result in changes in texture coordinates. These errors cause the difference of colors fetched from texture.

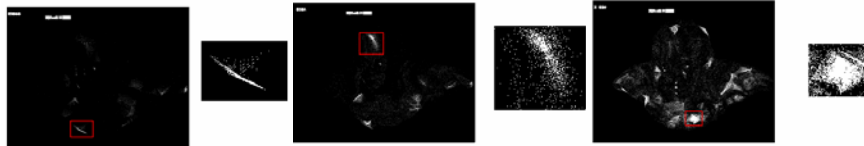


Fig. 13. Difference images between our system and pure PC OpenGL system

Comparison of frame rates among test programs is shown in table 3. We can see that the frame rate is inversely proportional to the number of assembly commands.

Table 3. Frame rate.

	sample 1	sample 2	sample 3
Number of assembly commands	6	31	21
FPS	61.79	17.54	26.2

5. Conclusion

Programmable vertex processing offers user defined vertex processing and provides the programmer with an opportunity to take full control of vertex processing. In this paper, we design and implement a programmable vertex processing unit for the mobile environments based on the OpenGL ES 2.0 specification. It specifies graphics APIs and high level shading language for the embedded systems. Since the final draft

of OpenGL ES 2.0 came out about September of 2005 it is hard to find software or hardware implementation based on the specification.

We present the architecture and instruction format of vertex processing unit. And we define 28 primitive instructions and 3 macro instructions based on the operation processing method. The vertex processing is performed by our vertex processing unit and the remaining operations, after vertex processing such as triangle setup, rasterization and fragment processing, are performed by the original OpenGL pipeline. Our implementation and test results show that error is negligible and the performance is inversely proportional to the number of vertices and the number of instructions in the vertex program as we expected.

We are in the process of developing a fragment processing unit for the next part of our work on vertex processing unit.

Acknowledgement

This work was supported by the Ministry of Culture & Tourism and KOCCA under the Culture and Content Technology Research Center (CTRC) Support Program.

References

[1] James D. F., Andries van D., Steven K. F., John F. H.:Computer Graphics: Principles and Practice in C Addison-Wesley Professional, Boston (2005)
[2] Matt, P., Randima, F.:GPU GEMS 2. Addison-Wesley Professional, Boston (2005)
[3] Kyoungsu O., Keechul J.: GPU implementation of neural networks, International Journal of Pattern Recognition, Vol. 37, Elsevier(2004).
[4] Michael M., Stefenus D. T., Tiberiu P., Bryan C., Kevin M.:Shader algebra, Transaction on Graphics, Vol 23, ACM Press, Newyork (2004)
[5] OpenGL ES 2.0 specification. Available at http://www.khronos.org/opengles/2_X/
[6] OpenGL ARB Vertex program specification. Available at http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_ program.txt
[7] OpenGL ARB Fragment program specification. Available at http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_ program.txt
[8] Randi J. R.: OpenGL Shading Language, Addison Wesley, Boston (2004)
[9]] R. L. Cook., K. E. Torrance.: A Reflectance model for Computer Graphics, Transaction on Graphics, Vol 1, ACM Press, Newyork (1982)