Programmable Vertex Processing Unit for Mobile Game Development

Tae-Young Kim[1], Kyoung-Su Oh[2], Byeong-Seok Shin[3], CheolSu Lim[1]

[1] Dept. of Computer Engineering, Seokyeong University 136-704 Seoul, Korea
[2] Dept. of Media, Soongsil University 156-743 Seoul, Korea
[3] Dept. of Computer Engineering, Inha University 402-751 Inchon, Korea
tykim@skuniv.ac.kr, oks@ssu.ac.kr, bsshin@inha.ac.kr, cslim@skuniv.ac.kr

Abstract. Programmable vertex processing unit increases flexibility and enables customizations of transformation and lighting in the graphics pipeline. Because most embedded systems such as mobile phones and PDA's have only the fixed-function pipeline, various special effects essential in development of realistic 3D games are not provided. We designed and implemented a programmable vertex processing unit for mobile devices based on the OpenGL ES 2.0 specification. It can be used as a development platform for 3D mobile games. Also, assembly instruction set and encoding scheme are examples of standard interface to high-level shading languages.

1 Introduction

In last a few decades, much research has been done to enhance the functionality and efficiency of graphics hardware [1]. One of them is the programmable graphics pipeline, which provides a programmer with the full control of the vertex and fragment processes. Various special effects which were impossible with the fixed pipeline can be implemented [2][3]. The vertex processing in the programmable pipeline does not use the fixed-function T&L (Transformation & Lighting) but a vertex program written by a programmer. As a result, this enables us to make realistic 3D games.

Unfortunately, most embedded systems such as mobile phones and PDA's only have fixed-function pipeline. Although some mobile 3D game consoles equip with specially designed programmable units, they require a lot of computing resource. Since they are subset of GPU's for desktop PC, they cannot be applied to generic mobile phones or PDA's. Therefore, we have designed and implemented a programmable vertex processing unit for the mobile devices.

Our vertex processing unit is designed based on the OpenGL ES 2.0 [4] and GL_ARB_vertex_program [5]. 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, which specifies graphics APIs and high level shading language for the programmable vertex and fragment programs [6]. But it does not include low-level specification of the shading language [7]. We modified GL_ARB_vertex_program assembly language to fully support OpenGL ES 2.0. We defined some instructions and substituted an instruction with several other primitive instructions to encode/decode an instruction efficiently. Since it provides high-order flexibility to simple mobile devices, we can use them as mobile 3D game consoles. Also, our instruction design and operand encoding scheme can be used as an interface standard between low-level and high-level shader language.

In Sect. 2 we present the structure of our vertex processing unit. Instruction set design and encoding schemes are explained in Sect. 3. Implementation and results are in the next section. Lastly, we summarize and conclude our work.

2 Architecture of Vertex Processing Unit

A vertex program is a sequence of vector operations that determines how a set of program parameters and per-vertex input parameters are transformed to a set of per-vertex result parameters. Fig. 1 shows the architecture of our vertex processing unit.

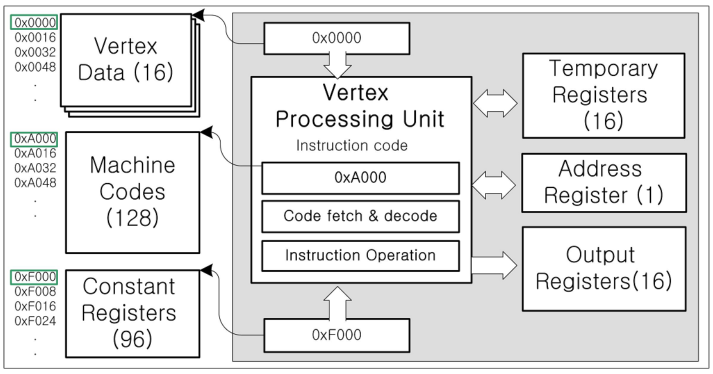

Fig. 1. Architecture of our vertex processing unit. It consists of seven components.

- **Machine code**: (Up to 128) binary codes to be executed in vertex processing unit.
- **Vertex Processing Unit**: A processing engine that fetches, decodes and operates each machine code.
- **Vertex data**: A set of 16 read-only registers containing 4-component floating point vector. Each register represents position, colors, normal of vertex.
- **Constant Registers**: A set of 96 read-only registers. It stores parameters such as matrices, lighting parameters and constants required by vertex programs.
- **Temporary Registers**: A set of 16 readable and writable registers to hold temporary results that can be read or written during the execution of a vertex program.
- **Address Register**: A register containing an integer 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 to hold the final results of a vertex program. They are passed to the remaining graphics pipelines.

3 Instruction Set and Encoding Scheme

We define 28 primitive instructions and 3 macro instructions based on the operation processing method, as shown tables 1. Since macro instruction means an instruction that can be replaced by a series of primitive instructions, each one is translated into multiple primitive instructions in assembling time. In table 1, the instructions in shadowed entries 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 as shown in table 1 (below).

Table 1. Primitive instructions and macro instructions used in our implementation

Instruction	Description	Instruction	Description
ARL	Address register load	DPH	Homogenous dot product
MOV	Move	DP3	3-component dot product
ABS	Absolute	DP4	4-component dot product
FLR	Floor	clamp	Clamp
FRC	Fraction	Mulz	Multiply on z
SWZ	Extended swizzle	MAD	Multiply and add
ADD	Addition	EXP	Exponential base 2(approximate)
MUL	Multiply	LOG	Logarithm base 2(approximate)
DST	Distance vector	EX2	Exponential base 2
XPD	Cross product	LG2	Logarithm base 2
MAX	Maximum	RCP	Reciprocal
MIN	Minimum	RSQ	Reciprocal square root
SGE	Set on greater or equal than	rEX2	Exponential base 2(rough)
SLT	Set on less than	rLG2	Logarithm base 2(rougn)

macro	description	macro	description	macro	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, temp.w* *Mulz f, tmp.1xz1, tmp.w*	POW	: Power ($f = a^b$) POW f, a, b *LG2 tmp, a* *MUL tmp, tmp, b* *EX2 f, tmp*	SUB	: Subtraction : (f = a – b) SUB f, a, b *ADD f, a, -b*

Fig. 2 shows the 64bit machine code structure, which is composed of an opcode, a destination operand, and up to 3 source operands. 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 having three source operands.

The opcode has 6 bits, so it is possible up to 64 instructions. The destination operand field (register type, index, and mask) 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

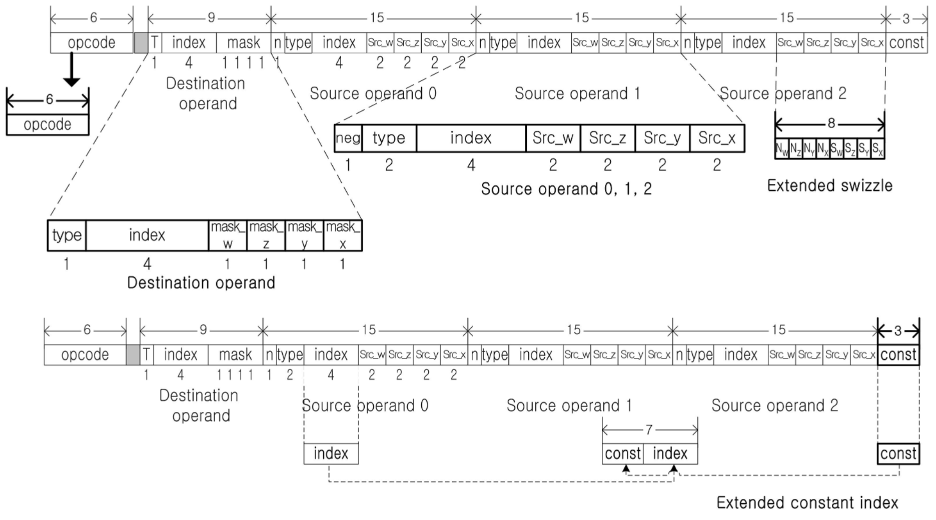

Fig. 2. Machine instruction format : an opcode field, a destination operand field, source operand fields, extended swizzle field, and extended constant index field

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)
```

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}.

```
Colr.{-}[01xyzw] {-}[01xyzw] {-}[01xyzw] {-}[01xyzw]
```

```
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. 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.

4 Implementation and Results

We implemented our programmable vertex processing unit in software emulation. Our implementation can be used to emulate mobile game applications including vertex programs. We tested the performance of our work on a desktop PC with 4.3 GHz Pentium processor and ATI Radeon 9800 XT graphics card.

To test our vertex processing unit, we implemented the OpenGL ES 2.0 APIs related with vertex processing. Using the APIs, vertex data are stored and passed to our vertex processing unit. A vertex program is assembled into machine codes and they are passed to the vertex processing unit through 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 sent to the OpenGL graphics pipeline installed in our computer via the original OpenGL APIs.

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 three vertex programs as shown in Fig.3.

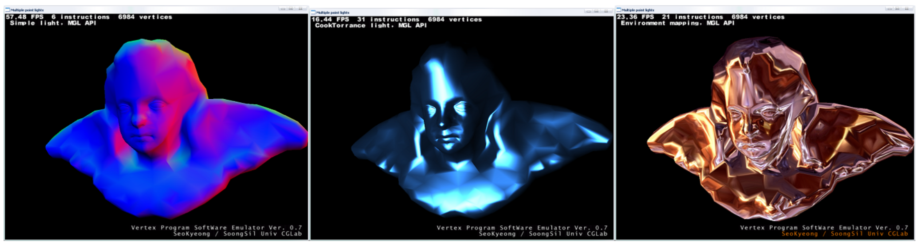
Multiple point lights
57.48 FPS 6 instructions 6984 vertices
Simple light. MGL API
Vertex Program Software Emulator Ver. 0.7
SeoKyeong / SoongSil Univ CGLab
Multiple point lights
16.44 FPS 31 instructions 6984 vertices
CookTorrance light. MGL API
Vertex Program Software Emulator Ver. 0.7
SeoKyeong / SoongSil Univ CGLab
Multiple point lights
23.36 FPS 21 instructions 6984 vertices
Environment mapping. MGL API
Vertex Program Software Emulator Ver. 0.7
SeoKyeong / SoongSil Univ CGLab

Fig. 3. Test vertex programs, left: Normal value, middle: Cook-Torrance illumination, 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 on PC. We found little differences that cannot be recognized with naked eye. 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 2. Frame rate accoding to the number of instructions

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

5 Conclusion

We design and implement a programmable vertex processing unit for the mobile environments based on the OpenGL ES 2.0 specification. Since the final draft of

OpenGL ES 2.0 came out about September 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. 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. At present we have only implemented the vertex processing unit. However, the fragment processing unit is also under development and the both units will be implemented as H/W chip.

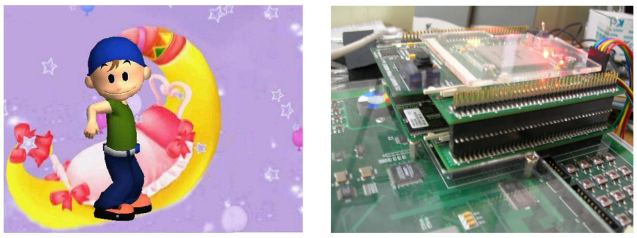

Fig. 4. A screen shot of mobile game implemented with our vertex processing unit (left) and a hardware prototype of target system using FPGA (right).

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. Michael M., Stefenus D. T., Tiberiu P., Bryan C., Kevin M.:Shader algebra, Transaction on Graphics, Vol 23, ACM Press, Newyork (2004)
4. OpenGL ES 2.0 specification. Available at http://www.khronos.org/opengles/2_X/
5. OpenGL ARB Vertex program specification. Available at http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt
6. OpenGL ARB Fragment program specification. Available at http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt
7. Randi J. R.: OpenGL Shading Language, Addison Wesley, Boston (2004)