OpenGL ESSL Optimizing Compiler for Embedded 3D Graphic Processor

Soojun Im
Sungkyunkwan University
Suwon Korea
lang33@skku.edu

Dongkun Shin
Sungkyunkwan University
Suwon Korea
dongkun@skku.edu

***Abstract*—Recently, graphic processing unit (GPU) becomes a mandatory component in mobile consumer devices such mobile phones. The vertex and fragment shader programs in embedded GPU are programmed with embedded system shading language(ESSL). The shader compiler for ESSL should be designed considering several distinct features of ESSL and GPU. In this paper, we present ESSL compiler techniques for embedded GPU. The compiler can optimize the code and data memory size as well as improve the performance of shader code by fully exploiting the special architecture of target GPU. Experiments show that the proposed optimization techniques can reduce the code size by up to 10.3% and the execution cycles by up to 16.8%.**

Keywords-GPU;Compiler;ESSL; Embedded Systems

I. INTRODUCTION

As recent mobile consumer devices such as mobile phones and mobile pads are supporting rich 3D user interfaces, 3D games, and WebGL, graphic processor units (GPUs) are widely adopted by mobile consumer devices. In order to support more complex graphic processing, recent GPUs use programmable shader pipeline. Users are able to program the desired graphic processing operations with shading language. OpenGL ESSL [1] is a standard language for vertex and fragment shader programs in embedded systems. Currently, OpenGL ES 2.0 standard has been published.

ESSL is similar to the conventional C language, but there are many differences on data types such as vector typed variables, matrix and sampler data. Furthermore, each variable must be specified with the allocated location in the physical memory by qualifier. To execute a shader program at a target GPU, it should be compiled by ESSL compiler [2]. Since both shader program and GPU have unique characteristics, ESSL compiler should be designed considering such features. For example, GPUs generally use the multi-way VLIW architecture or multithreaded architecture to execute multiple independent instructions simultaneously. ESSL compiler should find independent instructions to generate compact VLIW instructions. In addition, GPUs have distinct memory components such as global buffer, stream buffer, texture/sampler memory, and constant memory. Since each memory component can be used only for specified data, there is no memory space for general purpose such as stack memory. Therefore, the traditional register allocation techniques such as register spilling cannot be used for GPU. However, there is no intensive academic study on optimizing ESSL compiler while there are many commercial ESSL compiler products.

In this paper, we present a shader compiler for OpenGL ES 2.0 standard. It is carefully designed considering the special architecture of the target GPU. The compiler optimization techniques include the inter-procedural register allocation for stack-less GPU, the uniform/constant packing for global memory size optimization, the sub-bank register allocation for code memory size optimization, and the instruction reordering for two-way VLIW architecture.

II. TARGET GPU ARCHITECTURE

Our target GPU is a two-way VLIW architecture supporting vector-type instructions [3]. Fig. 1 shows the architecture of our target GPU. Since the GPU is designed for embedded systems, it has several features for low power consumption, small code size, and low hardware cost as follows: The GPU has only two Arithmetic-Logical Units (ALUs), which can execute up to two vector instructions (Instruction #0 and Instruction #1) in parallel, and one Special Functional Unit (SFU) for special functions such as log, exponent, etc. Each vector data can have up to four components, x, y, z, and w.

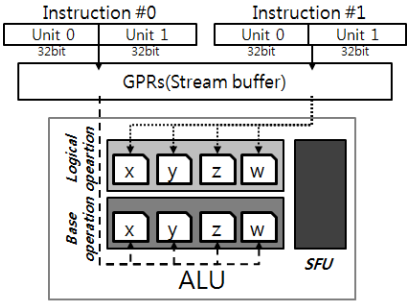

Figure 1. Target GPU Architecture

The GPU can execute two instructions simultaneously only when they have different operation types. Therefore, it is important to find two instructions which are independent and different operation types in order to maximize the utilization of GPU. The size of an instruction is variable, either one unit (32 bits) or two units (64 bits). A simple instruction can be encoded with one unit while a complex instruction may require two units. Such two modes of instruction formats are suitable for code size optimization. ESSL compiler should generate one-unit instructions if possible to minimize the code size.

Fig. 2 shows the overall memory architecture of the target GPU. There are instruction bank, global buffer (GLB), stream buffers (STBs), and texture/sampler memory. The compiled code of shader program is stored in the instruction bank. The GLB has uniform and constant variables declared in the shader program. There are twelve STBs, each of which is allocated for each thread. The target GPU executes 12 threads in a pipelined

978-1-4673-1501-2/12/$31.00 ©2012 IEEE

manner. The STB for each thread has 128 vector registers for temporal variables and input/output variables such as attribute and varying data. The texture/sampler memory is used to store the texture or sampler images for fragment shader.

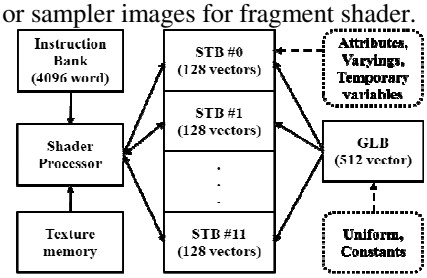

Figure 2. Memory architecture of target GPU

III. ESSL COMPILER IMPLEMENTATION

We used the ESSL compiler front-end modules (preprocessor, scanner, and parser) presented by 3DLabs and implemented the compiler back-end modules including code generator and optimizer for the target GPU.

A. Inter-Procedural Register Allocation

Each variable in the target shader program must be allocated to a proper physical memory during the code generation. Especially, the STBs should store effectively temporal variables with lifetime analysis. We used the graph coloring algorithm for the temporal variable register allocation. The conventional register allocation technique analyzes the lifetime of a variable only within procedure since the temporal variables can be saved in and restored from the stack memory at procedure call and return. However, since the target GPU has no memory space for stack, the temporal variables are alive across procedures.

Therefore, we used an inter-procedural lifetime analysis for register allocation. The register allocation gives a higher priority to the temporal variable in deeper and more frequently called function. If two variables have non-overlapped lifetimes, they can share single register in STBs.

B. Memory Size Optimization

To encode a STB register number in an instruction, seven bits are required since a STB has 128 registers. In the target GPU, the lower five bits are stored in unit 0 and the higher two bits are stored in unit 1. If the instruction uses the operand in the low-bank registers, i.e., registers 0~31, it can be encoded with only one unit. To optimize the program code size, more instructions should use the low-bank registers. For the purpose, we divided the register allocation into two steps: First, each variable is assigned with a virtual register number. Second, each virtual register is allocated to a physical STB register number. In the second step, we use the reference count of a virtual register as priority, and the higher priority virtual registers are assigned with the low-bank registers.

In addition, in order to store space-efficiently the variable-sized uniform and constant variables into GLBs, our compiler packs them with a knapsack-like algorithm.

C. Instruction Reordering

To utilize the two-way VLIW architecture of the target GPU, compiler checks whether two sequential instructions are independent and use different types of operations. If the condition is not satisfied, compiler reorders instructions in a basic block to make more chances.

IV. EXPERIMENTS

To evaluate our compiler implementation, we used 3D IP test bench for target GPU and OpenGL ES Conformance Testing set [4]. Figure 3 shows the effect of code size optimization technique. The average code size is reduced by 4.7%. Especially, for the fragment shader in the *texture* program, the code size is reduced by 10.3% since many one-unit instructions are generated. Since the fragment shader in the *disable* program has few variables in the source code, it is not significantly affected by the code size optimization techniques.

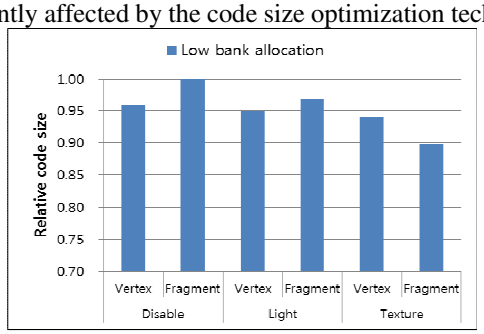

Figure 3. The effect of code size optimization

In Figure 4, we compared the performance of two-way instruction code with that of one-way instruction code. The average execution cycle is decreased by 10.7%. Especially, there is about 16.8% of performance improvement for the *texture* program.

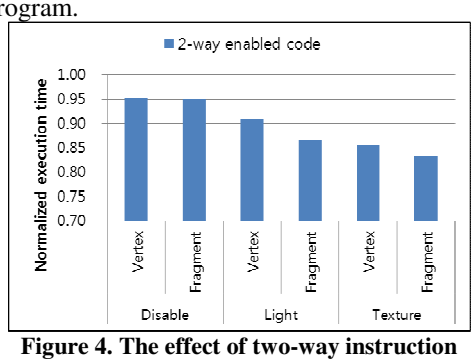

Figure 4. The effect of two-way instruction

V. CONCLUSION

In this paper, we implemented ESSL compiler supporting OpenGL ES 2.0 standard. The compiler is highly optimized considering several features of embedded GPU such as stack-less architecture, variable-length instruction and data, and multi-way instructions.

ACKNOWLEDGMENT

This work was supported by the IT R&D program of MKE/KEIT. [KI0018-10041244, SmartTV 2.0 Software Platform]

REFERENCES

[1] Khronos Group, OpenGL ES 2.0 http://www.khronos.org/opengles/

[2] Robert, M. Hill, S., "ESSL compiler for embedded 3D graphics architecture," Proc. Of International Conference on Consumer Electronics (ICCE), pp. 10-14, Jan. 2009.

[3] Woo-Young Kim, Bo-Haeng Lee, Kwang-Yeob Lee, and Jae-Chang Kwak, "Design of a fully programmable shader processor for low power mobile devices", TENCON `09, pp. 23-26, Jan. 2009.

[4] OpenGL ES Adoption and Conformance Testing, http://www.khronos.org/opengles/adopters/