쉐이더 구조를 위한 마이크로 아키텍쳐 시뮬레이션 환경

하상원° 이원종 한탁돈
연세대학교 컴퓨터과학과 미디어시스템연구실
bluice°@kurene.yonsei.ac.kr
airtight@yonsei.ac.kr
hantack@kurene.yonsei.ac.kr

KISS Korea Computer Congress 2006

A Cycle-Accurate Simulation Environment for Shader Architecture

Sang-Won Ha°, Won-Jong Lee, Tack-Don Han
Media System Laboratory, Department of Computer Science, Yonsei University

요 약

Shader architecture is one of the fastest growing fields in the ever advancing 3D graphics, and massive amounts of ideas and technologies are being introduced to the market continuously. In this paper, we present a flexible cycle-accurate simulation environment to accelerate and alleviate the process of developing and verifying these ideas and technologies. Combination of 3D graphics API and hardware simulator allows OpenGL applications to be emulated off-the-shelf for a given shader micro-architecture. Easily modified parameters allow the simulation environment to be tailored to specific demands or requirements.

1 Introduction

Shaders in today's GPU (graphics processing unit) play a major role in creating life-like 3D motion images in real-time. They employ programmability to the otherwise fixed-function graphics pipeline and enable numerous effects perviously unthinkable without dedicated hardware. Until shaders were introduced to the general public recently, only hardware engineers who had access to specialized high performance 3D graphics hardware were able to view realistic images on their workstation screens in real-time. Today, as shaders become wide spread, 3D graphics on everyday desktops are nearing motion picture quality. This is due in part to letting the application programmers concentrate on inventing novel algorithms rather than worrying about how to implement and squeeze in those algorithms into the graphics processors. One can truly say that it is indeed the Renaissance in 3D graphics [1].

The paradigm of the problem we are facing today has changed. The ever fast growing 3D graphics market is pumping the evolution of the graphics hardware at a tremendous rate with each successive generations yielding not only more computing power but new functionalities as well. The traditional way of drawing up a new algorithm, implementing in behavioral level, and then to RTL is just too time consuming to simply find out whether the new algorithm will be feasible when new functionalities from major hardware vendors are being poured into the general market every quarter. A simulator which can readily emulate the new algorithm will greatly save time and effort. This, in turn, will free up time to investigate deeper into the algorithm itself and fine-tune technological hurdles that might preside within. Ultimately, new technologies and ideas can be adopted quickly and with ease.

In this paper, we present an easily applied and flexible cycle-accurate simulation environment for shader micro-architecture. The simulation environment is composed of 3D graphics API adopted from the publically available OpenGL clone library called Mesa 3D Graphics Library [2] and the shader architecture simulator itself. Through this combination, real applications based on OpenGL, such as 3D graphics games, can be analyzed off the shelf, cycle by cycle without any modification to the application. Two different levels of architectural abstractions are provided to save time and fit the needs of simulating and verifying the desired architecture or functionality. Parameters for the simulator are easily modified to find the most efficient allocation of hardware resources for specific or arbitrary application. The whole simulation process is automated with minimal user intervention and execution-driven to closely match the workings of a GPU. All in all, the objective of the simulation environment is to provide a platform for quick and hassle-free development and implementation of new architectures or ideas. This characteristic is demonstrated through design space exploration of benchmark applications.

The rest of this paper is divided into four sections. Section 2 contains brief information about replated works. A thorough description and view into the design of the simulator environment is located in section 3 and some case studies done with the simulator environment is shown in section 4. Lastly, we conclude the paper with discussion on future work in section 5.

2 Related work

Due to the increase in interest and demand for 3D graphics, there have been many simulation related works. A co-design framework utilizing both hardware and software for graphics hardware accelerator was presented by Ewins et al [3]. In this framework, a number of software tools were developed in the C++ language to work either standalone or alongside hardware models written in a high level hardware description language (VHDL) to aid in algorithm research and hardware design. Lee et al [4] developed a testing environment for general graphics architectures. They supported OpenGL applications and used a hardware description macro to support hardware modeling and architecture modification. Sheaffer et al [5] developed a configurable micro-architectural simulator for GPUs. Their simulator could evaluate time-dependent behaviors of various functional units. They demonstrated the use of the simulator on a simple hypothetical architecture to analyze performance bottlenecks and to explore new GPU micro-architectures. Crisu et al [6] presented a hardware/software co-simulation framework for embedded systems. In combination with SystemC RTL models of graphics pipeline, their framework supports tools to assist in the visual debugging of graphics algorithms implemented in hardware, and to estimate the performance in terms of throughput, power consumption, and area.

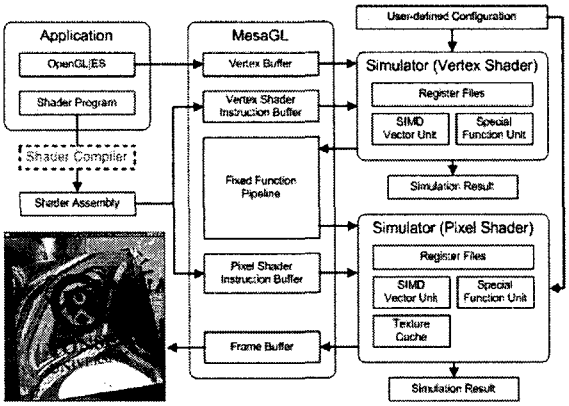

Fig. 1 Internal block diagram of the simulation environment

Previous researches were mainly focused on the general graphics hardware with no or minimal consideration for shader architecture. Although Moya et al [7] constructed a simulator with shader architecture in mind, it states a proprietary model unfit for immediate use with the current graphics applications. In this paper, we concentrate on simulating the shader micro-architecture embedded within the graphics pipeline. Our simulation environment enhances productivity in shader architecture design with straightforward layout and user configurable parameters. As a result, more thought and time can be invested in more critical tasks.

3 A cycle-accurate shader architecture simulator

One of the numerous benefits of our simulator environment is that real 3D graphics applications based on OpenGL will run without any modification to the application itself, therefore not needing anything other than the binary file and files required for execution. In addition, simulation results of processed cycles are shown simultaneously as the application is being executed normally. The following subsections describe how this is accomplished and deal more about the design and tracking mechanism of the simulator environment.

3.1 Integrating 3D graphics API and H/W simulator

As a whole, the simulator environment takes the form of the hardware simulator grafted onto the 3D graphics API named MesaGL derived from Mesa 3D Graphics Library [2]. MesaGL is in charge of communicating with the 3D graphics application and feeding instructions and data to the simulators. It also performs fixed-function portion of the 3D graphics pipeline. Fig. 1 shows the internal block diagram of the simulator environment. The simulation environment is initiated by calls to OpenGL. Therefore, in order for the simulation environment to be used, the host machine's default OpenGL library file must be replaced with the simulation environment's MesaGL library file. This one time chore is all that is needed to run simulations. These calls are intercepted and sent to MesaGL where fixed function pipeline processing is done. At the same time, vertex and pixel shader simulators are loaded with instructions and necessary register file entries from shader assembly codes compiled during run-time. MesaGL forwards vertices to the vertex shader simulator and reads back the processed vertices. Afterwards, these processed vertices are rasterized to pixel fragments which are then sent to pixel shader simulator for further processing and the resulting pixel fragments are returned to MesaGL to be finally shown on screen. Internally, vertex and pixel shader simulators execute instructions in their instruction buffer cycle by cycle in a pipeline manner. The state of the fetch & decode unit, reservation stations, functional units, and writeback actions are recorded in the simulation result files every cycle.

Because MesaGL and vertex and pixel shader simulators work seamlessly masquerading as a normal OpenGL, the 3D graphics application has no way of knowing that it is being run by a simulator and the 3D graphics hardware. This characteristic works in our favor as real, unmodified 3D graphics applications run on host computers indiscriminative of whether the applications were built with shader programs specially designed for NVidia GPUs (NV) or written in Cg [8]. Of course, shader programs written for arbitrary GPUs (ARB) are also supported.

3.2 Level of architectural abstraction

There are two levels of abstractions supported on the simulator environment and they are called functional simulation mode and performance simulation mode. The functional simulation mode incorporates methods used in SimpleScalar [9]. This simple and quick mode provides the ability to test the instruction set architecture and to visually verify the algorithms for executing each instruction. In addition, it fast-forwards to a point where a more elaborate testing is desired. The performance simulation mode is based on Tomasulo's algorithm [10][11] to execute and complete instructions out-of-order. A detailed view of the idea behind performance simulation mode is illustrated in Fig. 2. All instructions are register to register, just as GPUs are built like, and the simulator emulates 4 stage pipeline of fetch, decode, execute, and writeback. Reservations stations are appended to each functional unit. The temporary register file which is the only read-write register file in the simulator, has ready bit and tag augmented to each register. The fetch & decode width, size of the reservation station, types and numbers of functional units, latency and throughput of functional units, and size of each register file are all modifiable through the user-defined configuration file to suit the requirements.

3.3 Simulation results

Tracing the pipeline status of the simulator is one of the most important task in using the simulation environment. During application's run-time, vertex and pixel shader simulators continuously output simulation information of every vertex or pixel fragment for every cycle. This information is stored separately for vertex and pixel shader simulators in two files for

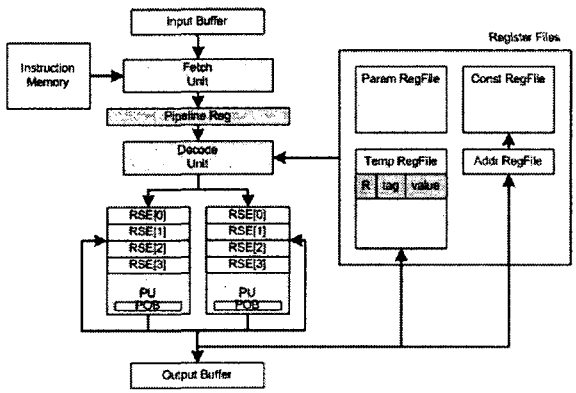

Fig.2 Abstracted architecture in performance simulation mode

later viewing. Inside each file, information about where the instruction is located in the pipeline is given for every cycle. For simplicity, instructions of a given shader program is enumerated from 0 to n-1 where n is the total number of instructions. When viewing a particular cycle, enumerations of instructions are found spread across four sections. The four sections correspond to the four pipeline stages of the simulator. Unlike other stages, the execute stage is sub-divided into three subsections. The reason is to show which instruction is in the reservation station, which has begun processing, and which is in the functional unit's output buffer waiting for a free slot on the result bus. At the end of completion of a vertex or pixel program, a summary report of when each instruction completed each pipeline stage is presented.

4 Case study

In this section, we tested the simulator for processing arbitrary instruction set architecture with proprietory algorithms for instruction execution. Four benchmark applications were picked from demo applications in NVidia's SDK [12]. They were selected based on the diversity of shader hardware utilization. For example, bumpy shiny patch application was programmed for ARB supporting GPUs while the rest are dedicated to NV compatible NVidia GPUs.

4.1 Functional verification

To test the flexibility of functional aspect of the simulation environment, we applied a modified vertex shader version 1.1 and pixel shader version 1.4 instruction set architecture targeted at mobile devices to our simulation environment. Fig. 3 displays the captured screen of the benchmark applications. Even when the images were magnified to limit of the screen, there were no visible differences between our simulator and the images rendered by the GPU.

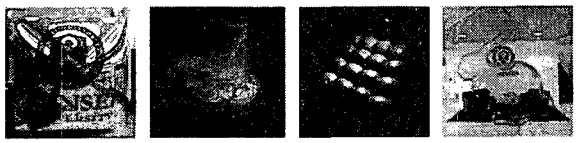

Fig. 3 Benchmark programs for functional verification. (a) bumpy shiny patch (BSP), (b) bump mapping (BM), (c) dot product (DP), and (d) refraction (REF).

4.2 Design space exploration of performance simulation mode

After verifying that the instruction set architecture was feasible in terms of visual quality, we switched to performance simulation mode to find out which combination of functional unit quantity and reservation station count will give us the best performance, cycle-wise. Fig. 4 exhibits the IPC (instructions per cycle) of the benchmark applications with varying number of reservation stations and functional units. As you can see, most of the applications do not benefit from the increase in number of functional units or reservation stations. The reason for this phenomenon is due to the fact that typical shader programs' instructions are tightly dependent on one another. In other words, read-after-write, also called true dependency, is too high to be overcome by method using reservations stations.

5. Conclusion

In this paper we introduced a simulation environment that emulates a shader micro-architecture producing cycle accurate simulation results. By combining 3D graphics API and a custom built hardware simulator, detailed information about how the instructions are being processed out-of-order inside the given shader architecture can be extracted from real 3D graphics applications without any modifications. Two levels of architectural abstractions are provided to speed-up and ease the verification process. Configurable performance simulation mode allows custom tuning of the simulator to fit the confronting requirements.

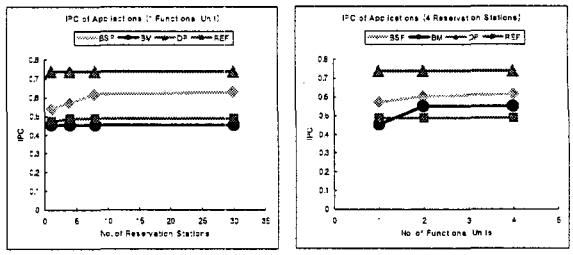

Fig. 4 IPC of benchmark applications with different number of reservations stations and functional units

The current simulation environment only supports vertex and pixel shader version 1.3 and below. In the near future, we are planning to raise the version up to the current market standard and support branch prediction and texture access in vertex shader. Also, incorporating multithreading techniques, power estimation, and support for multiprocessors will be implemented.

References

[1] Matt Pharr, editor., "GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation," Addison Wesley, 2005, ppxxix

[2] Mesa 3D Grahpics Library, http://www.mesa3d.org

[3] Ewins, J.P., Watten, P.L., White, M., McNeill, M.D.J., and Lister, P.F., "Codesign of graphics hardware accelerators," Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware, 1997, pp103-110

[4] Lee, I.H., Kim, J.Y., Im, Y.H., Choi, Y., Shin, H., Han, C., Kim, D., Park, H., Seo, Y.I, Chung, K., Yu, C.H., Chun, K., and Kim, L.S., "A hardware-like high-level language based environment for 3D graphics architecture exploration," Proceedings of IEEE International Symposium on Circuits and Systems, May 2003, pp512-515

[5] Sheaffer, J.W., Luebke, D., and Skadron, K., "A flexible simulation framework for graphics architectures," Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware, Aug. 2004, pp85-94

[6] Crisu, D., Cotofana, S.D., Vassiliadis, S., Liuha, P., "GRAAL - a development framework for embedded graphics accelerators," Proceedings of Design, Automation and Test in Europe, Feb. 2004, pp1366 - 1367

[7] Moya, V., Gonzalez, C., Roca, J., Fernandez, A., Espasa, R., "Shader Performance Analysis on a Modern GPU Architecture," Micro-38, Nov. 16, 2005

[8] Cg Homepage, http://www.nvidia.com/cg

[9] SimpleScalar, http://www.simplescalar.com

[10] Weiss, S., Smith, J.E., "Instruction Issue Logic for Pipelined Supercomputers," In Proceedings of International Symposium on Computer Architecture, pp. 110-118.

[11] Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal, Jan. 11, 1967

[12] NVidia SDK, http://developer.nvidia.com/object/sdk_home.html