GP-GPU 개발을 위한 3차원 그래픽 시뮬레이터 구현

여동영*· 김우영* · 정형기* · 이광엽*

*서경대학교

Implementation of a 3D Graphics Simulator for GP-GPU

Dong-young Yeo* · Woo-young Kim* · Hyung-Ki Jung* · Kwang-Yeob Lee*

*Seokyeong University

E-mail : YeoDong@skuniv.ac.kr

요 약

3차원 그래픽 처리를 위한 가속기 하드웨어가 발표된 이후 GPU(Graphics Processing Unit)의 성능은 끊임없이 개선되어 왔다. 이는 복잡한 그래픽 어플리케이션의 연산을 효율적으로 처리하기 위한 추세이나 실제로 GPU의 리소스를 100% 활용하는 경우는 드물다. 최근 주목받고 있는 GP-GPU(General-Purpose GPU)는 GPU에서 담당하는 연산을 포함하고 CPU가 처리하는 일반적인 연산의 처리도 가능하여 프로세서 자원의 분배에 따라 효율적인 제어가 가능하다.

본 논문에서는 GP-GPU 기반 환경을 가상으로 구현하여 프로그램의 설계 및 디버깅이 가능한 시뮬레이터를 구현하였다. 이를 통해 동시 설계(Co-Design) 환경을 구성하여 동시적 개발 환경을 지원하고 3차원 그래픽 컨텐츠의 디스플레이가 가능한 인터페이스를 구축하여 빠르고 안정적인 검증이 가능하다.

ABSTRACT

Since a hardware accelerator for 3D graphics processing GPU(Graphics Processing Unit)'s performance has been improving constantly. This is the efficient way was introduced for complex graphics application, but it is rarely used to utilize 100% resources on GPU. GP-GPU(general-purpose GPU), including operations on the GPU and supporting common operations can be handled by the processor, is noted by depending on the distribution of resources that can be effectively controlled.

In this paper, the simulator was implemented that supports virtual environment of GP-GPU and available for program design and debugging. Through this, the co-design development environment support simultaneous design fast and reliable verification that are available to build the interface of three-dimensional graphics display.

키워드

GP-GPU, simulator, SystemVerilog, DPI

Ⅰ. 서 론

프로세서의 개발 환경을 구축할 때에 가장 중요한 것은 설계의 동시성과 검증의 효율성을 갖추는 것이다. 개발과정 중에 기본적인 설계 단계에서부터 다양한 H/W의 검증이 수반되고, 단계별로 철저한 검증을 통하여 검증의 신뢰성을 갖추게 된다. 일반적인 HDL 설계의 검증 방법으로 Modelsim, NC-verilog 등의 시뮬레이션 프로그램을 이용하여 파형을 보고 분석하는 방법이 있으나 한눈에 문제를 파악하기가 어렵고 검증 범위에 한계가 존재한다. 특히 GP-GPU의 특성상 픽셀 단위로 출력되는 결과에 대한 디버깅이 요구되기 때문에 수치와 파형을 통한 검증은 효율성의 문제가 뒤따른다. 따라서 별도의 시뮬레이션 툴을 구비할 필요가 있다.

또한 프로젝트를 여러 사람이 모듈을 나누어 동시에 설계를 진행하는 과정에서 설계의 변경이 있거나 인터페이스를 조율하는 과정에서 발생하는 문제들에 대한 대비를 하지 않는다면 설계에

큰 혼란을 가져오게 된다. 때문에 프로세서 구성에 필요한 모듈들의 설계에 있어서 다른 설계에 영향을 최소화하는 독립적인 개발환경이 구축되어야 한다.

이를 위해 전체 프로세서 모듈을 소프트웨어 언어로 빠르게 구현하여 전체 인터페이스를 갖추고, 동시 설계 환경에서 내부 모듈들에 대한 하드웨어 언어 설계 및 검증을 병렬적으로 수행하는 방법이 있다.

우선적으로 C와 HDL 언어 사이에 인터페이스를 갖춤으로써 프로세서의 일부가 완성된 HDL 소스와 다른 부분을 C로 빠르게 구현하여 검증 가능한 상태까지 이끌어야 한다. 이 상태에 이르면 C로 작성된 모듈을 다시 HDL로 작성하여 기존 C 모듈을 대체하면서 동시에 전체 검증을 통해 결과에 따라 작성된 HDL 소스가 프로세서에 적합한지 여부를 검증할 수 있다. 또한 각 모듈을 여러 사람이 동시에 설계하고 후에 이를 합하여 발생할 수 있는 오류 또한 적어지게 된다.

본 논문에서는 [그림 1] 과 같이 동시 설계 환경을 갖춘 GUI interface를 지원하는 GP-GPU 프로세서 개발 시뮬레이터 'TestDrive' 를 구현하였다. 이를 통해 프로젝트 개발 보드가 제작되기 이전에 프로세서의 각 컴포넌트들을 통합시뮬레이션 함으로써 시스템의 오류를 정확하게 검출하고, 수행성능을 예측해볼 수 있다.

분문의 각 장에서는 동시 설계 환경을 지원하기 위해 적용된 프로그래밍 기법과, GP-GPU 환경에 최적화하기 위해 시뮬레이터에 구현한 기능들을 소개한다.

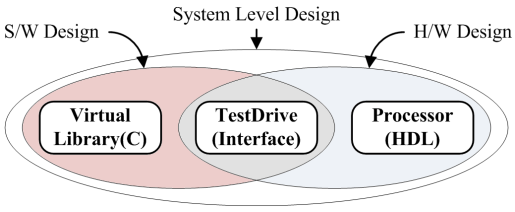

그림 1. System Level Design 시뮬레이션 환경

II. DPI를 활용한 동시설계 기법

동시 설계 환경을 갖추기 위해 C와 HDL 간에 인터페이스를 구축하여 H/W, S/W 디자인을 함께 할 수 있는 System level design이 이루어져야 한다. 이를 위해 현재까지 국내/외에서 주로 사용된 Verilog PLI(Programming Language Interface)는 일종의 시뮬레이션 API로서 Verilog design과 C 언어 사이의 인터페이스 루틴을 제공한다. 높은 안정성과 실행속도가 빠른 장점이 있으나, 함수가 너무 많아서 reference가 없으면 적절한 함수를 찾아 쓰기 어렵고 시뮬레이터와의 연결이 어려운 단점이 존재한다.

'Testdrive'에서는 C와 HDL의 연동을 위한 방법으로 SystemVerilog DPI(Direct Programming Interface)를 사용한다.

이 기술은 2005년도에 등장한 동시 시뮬레이션(Co-Simulation) 인터페이스로 사용자가 C/C++로 컴파일한 DLL(Dynamic Link Library) 파일의 함수를 SystemVerilog에서 사용하거나 반대로 SystemVerilog의 Task 함수를 C/C++에서 사용하게 하는 기술이다.

DPI는 아직 최신 기술이어서 국내에는 잘 알려지지 않은 편이지만 SystemC보다 융통성이 높고, 별도의 문법을 익힐 필요가 없으며, HDL과 C 문법 사이에 상호 접근이 가능케 하고 신뢰성이 높다. 때문에 소프트웨어로 작성된 전체 프로세서 모듈에 대해 부분적인 HDL 변환 및 검증과정의 간소화가 가능하고, 추후에 전체 프로세서의 설계 과정에서 C 모델의 추가나 수정이 쉬운 이점이 있다.

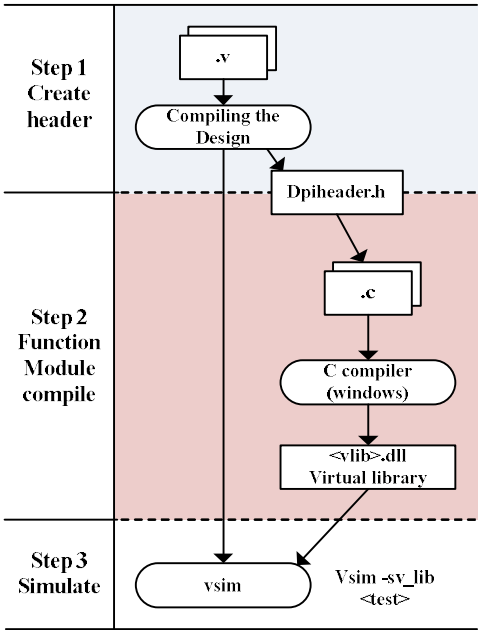

그림 2. DPI flow diagram

[그림 2] 는 Modelsim 5.8이상에서 제공하는 DPI의 compile 과정을 나타낸다. 먼저 HDL을 시뮬레이션 하기 위해 Top Testbench를 작성한다. 이때 HDL로 진행되지 않은 설계를 C 코드로 구현하여 DLL 형태로 컴파일한 Virtual library가 준비되어야 한다. Testbench는 SystemVerilog 형태로 작성되며 컴파일 시에 C 코드로 작성된 함수를 필요에 따라 간단한 import 명령을 통해 추가할 수 있다.

Modelsim과 Virtual library 사이에서 인터페이스 역할을 하는 dpiheader.h 파일이 존재한다. 헤

더파일의 내부에는 HDL 코드에서 import 하고 있는 C 코드 함수들 또는 C에서 HDL을 통해 export하고 있는 함수들이 선언되게 되며 Modelsim에서 vlog 명령을 통해 자동으로 생성된다. 컴파일 후에 시뮬레이션 단계에서 virtual library를 추가하면 별도의 인터페이스의 필요로 하지 않고 통합 시뮬레이션 환경의 구현이 가능하다.

III. 시뮬레이터의 동작 과정

위에서 언급한 DPI 기술을 활용하여 [그림 3]과 같은 'TestDrive' 라는 시뮬레이터를 구현하였다. 이 프로그램은 향후 리눅스 전환이 용이하도록 WTL(Window Template Library)로 제작된 윈도우용 응용프로그램이다. 때문에 시뮬레이터를 구동하기 위해 윈도우 기반에서 Modelsim 프로그램이 실행될 수 있는 환경이 구축되어야 한다.

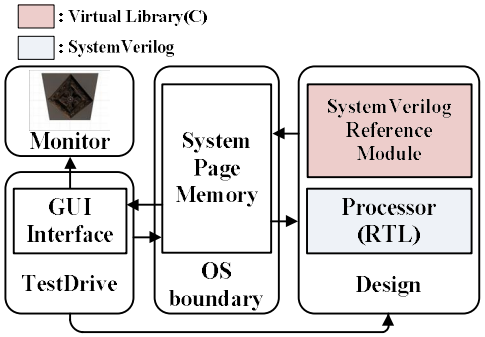

그림 3. DPI를 이용한 시뮬레이션 환경

'TestDrive'에서 시뮬레이션을 수행하게 되면 프로젝트에서 지정한 메모리 크기만큼 OS에서 Page memory로 할당한다. 이 메모리는 Virtual library와 데이터를 공유하기 위해 사용되어 시뮬레이션 초기 과정에서 GP-GPU 의 동작에 필요한 instruction, global, framebuffer 등의 메모리를 할당하여 테스트보드와 동일한 가상 메모리 환경을 구성하도록 구현하였다.

그 후 시뮬레이터에서 별도로 지정한 batch 명령에 의해 Modelsim에서 시뮬레이션이 실행되고 DLL에서 작성된 함수들을 Virtual library로부터 import하여 C 함수의 호출을 요청한다. 그러면 Virtual library는 Modelsim에 제공할 함수들을 export하고 'TestDrive' 에서 제공하는 System page memory를 공유하여 Testbench에서 다루는 모든 데이터를 'TestDrive'의 Page memory에 저장되도록 한다. Modelsim과 Virtual library의 준비가 끝나면 Testbench는 import된 함수들을 사용하여 시뮬레이션을 시작하게 되고 결과 데이터를 'TestDrive'의 System page memory에 저장한 후에 Modelsim은 자동 종료된다. 시뮬레이션 파형을 살펴보기 위해 testbench의 종료시점에 stop 명령을 추가하여 파형과 log를 동시에 확인하며 디버깅을 수행할 수 있다.

시뮬레이션을 마치게 되면 'TestDrive' 는 Page memory로부터 수행 결과를 얻어 디버깅에 필요한 정보를 출력하고 프로세서의 성능을 산출하여 Log를 통해 출력한다.

또한 'TestDrive'는 GUI 환경을 구축하여 최종 시뮬레이션 실행 결과에 따른 framebuffer 메모리의 픽셀값을 모니터를 통해 출력하여 검증의 효율을 높일 수 있다.

IV. 명령어 검증 환경 구축

본 개발에 사용된 GP-GPU는 그래픽 처리 연산 이외에 일반적인 목적으로 사용이 가능하도록 프로세서 내부에 별도의 dedicate 하드웨어를 두지 않고 모든 프로세스를 명령어로 구현하도록 설계되었다. 그렇기 때문에 명령어를 통해 shader 드라이버를 작성하게 되고 성능 향상을 위해 개발 이후에도 끊임없는 알고리즘 개선이 요구된다. 'TestDrive'는 사용자가 만든 Assembly 소스를 컴파일 하도록 지원하며, S/W, H/W 컴파일 결과와 최종적으로 통합 시뮬레이션을 마친 후에 출력되는 Log의 파일 출력기능을 구현하였다.

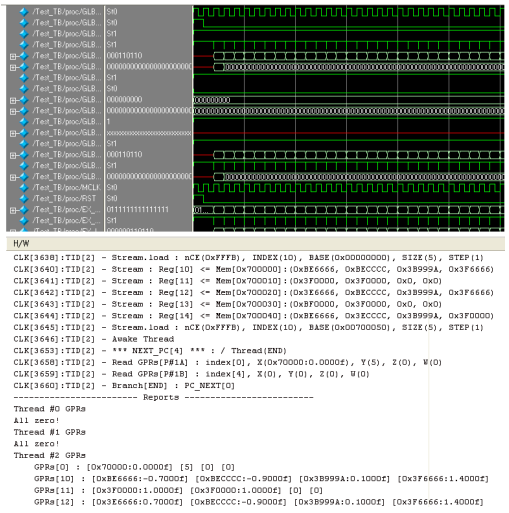

그림 4. 시뮬레이션 수행 결과

개발에 사용된 GP-GPU는 한 개의 프로세서에 여러 개의 스레드가 존재하여 각 스레드가 동시에 명령어를 처리하는 Multi-thread 구조로 설계되어 있다. 또한 각각의 스레드가 고유의 GPRs(General Purpose Registers)를 갖는 GP-GPU의 특성 때문에 오류 발생시, 분석에 어려움이 존재하였다.

이 때문에 명령어의 검증을 위해 [그림 4]와 같이 시뮬레이션 결과에 대한 output log에서 명령어의 수행 시점, 수행에 소요된 클럭, 사용 레지스터 등을 한눈에 확인할 수 있도록 구현하였다.

각 스레드에 중복되는 GPR 번호를 구분하기 위해 스레드 ID를 출력하고 수행 과정에서의 레지스터의 변화와 최종 report를 통해 사용된 모든 레지스터 영역에 대한 결과값을 표시한다.

명령어의 검증에 있어 어셈블리 코드 자체에서 발생하는 오류에 대해서도 처리를 해 줄 필요가 있다. 그래서 HDL에 의한 output log 외에 어셈블리 명령어의 컴파일과 그 결과를 S/W log로 출력하도록 지원한다. 어셈블리 컴파일러는 콘솔 기반의 별도로 지원되는 실행파일로 어셈블리 소스를 프로세서가 디코드할 수 있는 바이너리 파일로 변환한다.

S/W 컴파일 수행 중 어셈블리 문법의 오류 발생시, 에러 내용을 출력하도록 구현하였다. 이외에도 어셈블리 코드의 디코딩 과정을 단계별로 출력하도록 설정이 가능하여 별도의 컴파일러 없이 명령어의 디버깅이 가능하도록 지원하여 명령어의 개발 및 검증 환경을 개선하였다.

Ⅴ. TestDrive

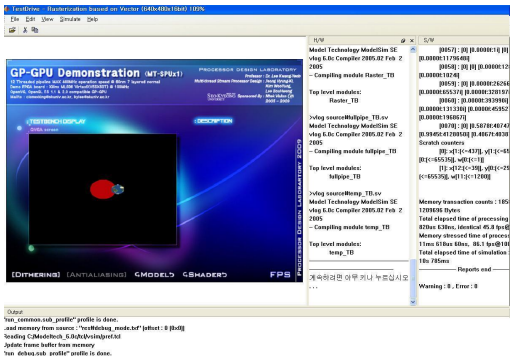

그림 5. 시뮬레이터 실행화면

[그림 5]는 'TestDrive'의 실행화면이다. 모든 동작은 메뉴 바를 통해 지원하며 키보드 시뮬레이션 플랫폼은 폴더로 나누어 project 파일 형태로 정의할 수 있다. 실행 결과가 디스플레이 되는 framebuffer 영역의 크기 조정이 가능하며 이와는 별도로 모니터에 출력되는 화면의 크기를 조정할 수 있도록 지원한다. 실행에 필요한 데이터들은 텍스트 형태의 파일로 저장되며 정해진 문법에 의해 정의된다. 컴파일 후에 시뮬레이션 수행시, memory load 명령을 통해 page memory 영역으로 지정한 텍스트를 불러올 수 있도록 설계자가 플랫폼의 수정을 통해 변경이 가능하다.

시뮬레이터 화면의 오른쪽에 출력되는 H/W, S/W log에 대해서는 프로젝트 설정이나 Testbench의 수정을 통해 출력하고자 하는 메모리 영역을 지정할 수 있도록 지원하여 효율성을 높일 수 있다.

Ⅵ. 결 론

DPI 기술을 응용한 'TestDrive'는 다중 IP들의 복합체를 설계하고 검증하는 프로세서 설계 전반에 있어 매우 큰 이점을 준다. 상위 개발자는 최종 결과물을 미리 C로 구현하여 이를 여러 하위 개발자들에 제공하면 각각의 IP의 HDL 설계, 검증을 병렬적으로 수행할 수 있게 된다. 다시 말하면 하위 개발자는 자신이 담당한 IP의 설계 및 검증만 담당하고 'TestDrive' 시뮬레이션 시스템에서의 해당 IP 적합성을 눈으로 확인하면 된다. 이는 여러 IP가 함께 구성되어 실행될 때의 잠재적인 문제점을 방지할 수 있으며, 이미 모든 IP의 연계성이 정의된 상태이기 때문에 하위 개발자들이 동시에 설계하더라도 최종단계에서 각 모듈의 인터페이스를 조율하는 단계에서 오는 마찰이 없어진다.

Ⅶ. ACKNOWLEDGEMENT

* 본 논문은 중소기업청 중소기업 기술 혁신 개발 사업, ETRI SoC 산업 진흥 센터, IDEC의 지원을 받아 제작되었습니다.

참고문헌

[1] Bergeron, Writing Testbenches Using Systemverilog, Springer-Verlag New York Inc, 2005

[2] SystemVerilog 3.1a Language Reference Manual, accellera, 2004

[3] H.K. Jeong, Design of 3D Graphics Geometry Accelerator using the Programmable Vertex Shader ITC-CSCC 2006

[4] Mauricio Breternitz, Jr., Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU Proceedings of the 12th international conference on parallel architectures and compilation techniques

[5] James C. Lelterman, Learn Vertex and Pixel Shader Programming with DirectX9 Wordware Publishing, Inc. 2004

[6] Sutherland, Systemverilog for Design - A Guild to Using Systemverilog for Hardware Design And Modeling, Springer-Verlag New York Inc, 2007