병렬 응용프로그램 실행 시 GPU 구조에 따른 성능 분석

Performance Evaluation of the GPU Architecture Executing Parallel Applications

최홍준, 김철홍
전남대학교 전자컴퓨터공학과

Hongjun Choi(chj6083@gmail.com), Cheolhong Kim(chkim22@chonnam.ac.kr)

요약

통합형셰이더 코어 구조 개발 이후 GPU는 그래픽스 전용 연산장치에서 범용 연산장치로 발달하고 있다. 특히, 병렬 응용 프로그램들은 병렬화된 하드웨어 구조를 효과적으로 활용할 수 있기 때문에, GPU를 활용하여 병렬 응용프로그램들을 실행시키는 기법이 주목을 받고 있다. 하지만, 현재의 GPU 구조는 비그래픽스 응용프로그램을 실행하는데 있어서 병렬성을 충분히 확보하지 못하다는 한계를 가지고 있기 때문에, 이를 해결하기 위해 GPU 구조는 빠르게 변화하고 있다. 본 논문에서는 GPU 구조의 개발 방향을 살펴보기 위해, 비그래픽스 병렬 응용프로그램들을 수행하는 경우에 코어 개수 및 동작 주파수 등의 하드웨어 구조에 따른 GPU의 성능을 상세히 분석하고자 한다. 실험 결과, 코어 개수가 30에서 192로 늘어나고 동작 주파수가 325MHz에서 450MHz로 증가함에 따라 GPU 성능은 28.9%에서 125.8%, 4.4%에서 16.2% 각각 향상되는 반면 성능 향상 효율성은 감소하는 것을 볼 수 있다. 성능 향상 효율성 감소의 주된 원인은 향상된 연산 능력에 맞추어 증가된 데이터 요구를 메모리가 적절하게 처리하지 못하기 때문이다. 결과적으로 GPU의 성능 향상 효율성을 더욱 높이기 위해서는 연산 능력 향상과 더불어 시스템 자원들 또한 GPU 구조에 맞게 변경되어야 함을 구체적인 실험을 통해 알 수 있다.

■ 중심어 : 통합형셰이더 | 그래픽 처리장치 | CUDA | 범용 GPU | 병렬 응용프로그램 |

Abstract

The role of GPU has evolved from graphics-specific processing to general-purpose processing with the development of unified shader core architecture. Especially, execution methods for general-purpose parallel applications using GPU have been researched intensively, since the parallel hardware architecture can be utilized efficiently when the parallel applications are executed. However, current GPU architecture has limitations in executing general-purpose parallel applications, since the GPU is not specialized for general-purpose computing yet. To improve the GPU performance when general-purpose parallel applications are executed, the GPU architecture should be evolved. In this work, we analyze the GPU performance according to the architecture varying the number of cores and clock frequency. Our simulation results show that the GPU performance improves by up to 125.8% and 16.2% as the number of cores increases and the clock frequency increases, respectively. However, note that the improvement of the GPU performance is saturated even though the number of cores increases and the clock frequency increases continuously, since the data cannot be provided to the GPU due to the limit of memory bandwidth. Consequently, to accomplish high performance effectiveness on GPU, computational resources must be more suitably considered.

■ keyword : Unified Shader | GPU | CUDA | GPGPU | Parallel Application |

* 본 연구는 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단 기초연구사업(No. 2011-0003350)의 지원과 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터(NIPA-2012-H0301-12-3005)의 지원을 받아 수행되었음.

접수번호 : #120214-006

심사완료일 : 2012년 04월 20일

접수일자 : 2012년 02월 14일

교신저자 : 김철홍, e-mail : chkim22@chonnam.ac.kr

I. 서론

GPU(Graphics Processing Unit)는 그래픽스 중심의 연산을 주로 수행하기 때문에 OpenGL[1], DirectX[2]와 같은 그래픽스 전용 API들을 지원하기 위하여, 그래픽스 연산 파이프라인을 수행할 수 있는 버텍스셰이더(vertex shader), 지오메트리셰이더(geometry shader)와 같은 코어들이 GPU 내부에 개별적으로 존재한다. 하지만, 최근의 GPU에서는 이러한 연산들을 통합하여 단일 코어에서 모든 그래픽스 연산 파이프라인들을 다룰 수 있는 통합형셰이더(unified shader)를 채택하고 있으며, 이를 통해 성능이 급속도로 향상되고 있다. NVIDIA사에서 출시한 GeForce 8 시리즈는 통합형셰이더로 구성된 대표적인 GPU 제품이다[3]. 특히, NVIDIA사는 통합형셰이더를 기반으로 출시된 GPU를 그래픽스 연산뿐만 아니라, 과학, 경제, 공학과 같은 다양한 분야에 범용적으로 사용할 수 있도록 CUDA(Compute Unified Device Architecture)를 개발하여 보급하고 있다. CUDA를 활용하면 CPU에서 수행되던 범용 응용프로그램들을 GPU에서 처리할 수 있도록 함으로써 시스템의 성능을 크게 개선시킬 수 있다[4][5].

GPU를 활용하여 범용 응용프로그램들을 효과적으로 처리할 수 있는 시스템을 개발하기 위해서는, 그래픽스 연산처리 능력 향상만을 위해 개발된 GPU가 다양한 특성을 지닌 범용 분야의 연산처리에서도 효율성을 나타낼 수 있는지에 대한 분석이 필요하다[6][7]. 그래픽스 처리 측면에서의 GPU 성능 향상은 동작 주파수 향상과 내부 코어 개수의 증가에 기인한 바가 크지만, GPU에서 범용 응용프로그램들을 수행하는 환경에서는 기존과 같은 방법으로 GPU의 연산 능력을 높일 수 있을지는 미지수이기 때문이다. 그러므로 본 논문에서는 범용 목적 GPU의 성능 개선 기법을 개발하기 위한 연구의 일환으로, NVIDIA사의 GPU에서 비그래픽스 병렬 응용프로그램들을 수행할 때, GPU의 동작 주파수 증가 및 내부 코어 개수 증가에 따른 성능 변화 양상과 추가적으로 발생하는 오버헤드에 대하여 상세하게 분석하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 분석 대상

인 GPU의 구조적 특성에 대하여 기술하고, 3장에서는 실험 환경 및 GPU의 내부 연결망 구성에 대하여 간단하게 설명한다. 4장에서는 내부 코어 개수 증가 및 동작 주파수 증가에 따른 GPU의 성능 변화와 추가적으로 발생하는 오버헤드에 대하여 실험 결과를 통해 상세하게 분석하고, 마지막으로 5장에서 결론을 맺는다.

II. 관련 연구

1. 통합형셰이더 코어를 기반으로 하는 GPU 구조

GPU는 일반적으로 매우 많은 수의 내부 코어를 이용하여 데이터를 병렬적으로 처리함으로써 빠른 연산을 제공한다. 높은 병렬 연산처리 능력을 가진 GPU는 3차원 그래픽스 연산과 같은 멀티미디어 데이터 처리에 적합한 구조를 가지고 있다. GPU 설계 기술이 발달함에 따라, 최근의 GPU는 그래픽스 파이프라인 연산을 고정적으로 수행하던 기존 구조에서 파이프라인 연산을 모두 통합하여 수행할 수 있는 구조로 변화하고 있다.

그림 1. 통합형셰이더 코어

[그림 1]은 분리형셰이더 코어 구조와 통합형셰이더 코어 구조에서의 연산 처리 흐름을 보여주고 있다[8]. 그림에서 보이는 바와 같이, 분리형셰이더 코어 구조에서는 각 파이프라인 별로 독립적인 셰이더 코어(shader A ~ shader D)가 존재하여 순차적으로 연산을 수행하지만, 통합형셰이더 코어 구조에서는 하나의 셰이더 코어를 통해 각 파이프라인 별로 반복 수행하는 형태로 연산을 수행한다. 본 논문에서는 최근 주로 사용되고

있는 통합형셰이더 코어 구조를 적용한 GPU를 분석 대상으로 한다.

그림 2. GPU 코어의 내부 구조

통합형셰이더 코어 구조를 사용하는 GPU의 내부 구조는 [그림 2]와 같이 구성된다. 일반적으로 통합형셰이더 코어는 순차적 멀티코어 스트리밍 프로세서(SP, Streaming Processor) 또는 CUDA 코어(CUDA core)로 명명하는데, 본 논문에서는 통합형셰이더 코어를 간략하게 표현하고자 하는 경우 SP라고 표현하고자 한다. GPU에서는 SP들을 효율적으로 활용하기 위해 복수 개의 SP를 묶어서 스트리밍 멀티프로세서(SM, Streaming Multiprocessor)라 부르는 보다 큰 단위의 연산 블록을 구성한다. [그림 2]에 보이는 바와 같이, SM 내부에는 여러 개의 SP들을 효과적으로 활용하여 연산을 수행하기 위해 SP에 스레드를 분배하여 병렬 연산을 수행할 수 있도록 도와주는 스레드 스케줄러(thread scheduler)가 존재한다. 이외에도 SM 내부에는 디코더(decoder)와 메모리 등이 존재한다. 이와 같은 여러 구성요소들로 이루어진 GPU에서 실질적으로 작업을 수행하는 기본 단위인 SM에서 작업이 처리되는 과정은 다음과 같다. 스레드 스케줄러에서 인출된 스레드 데이터는 디코더를 통해 레지스터 파일의 데이터에 접근하게 되고, 이러한 데이터는 SP를 통해 연산이 처리된다. SP가 연산을 수행하는 과정에서 전역/지역(global/local) 변수 캐쉬, 공유(shared) 캐쉬 및 레지스터 파일에서는 데이터를 읽거나 저장할 수 있고, 텍스처(texture), 컨스턴트(constant) 캐쉬의 경우는 읽기 전용이므로 데이터를 읽어오는 경우에만 사용된다. 위와 같은 과정을 통하여 SM은 다수의 SP들을 효율적으로

제어함으로써 GPU의 병렬 연산을 수행할 수 있다.

GPU 내부에 적재된 여러 개의 SM들을 묶어서 하나의 TPC(Texture/Processor Cluster)로 부르기도 한다. TPC는 독립적으로 연산을 수행할 수 있도록 구성되어 있기 때문에 작은 크기의 GPU라고도 할 수 있으며, 최근의 NVIDIA GPU에서는 TPC를 GPC(Graphics Processor Cluster)로 바꾸어 부르는 추세이다. GPU에서 TPC 또는 GPC와 같은 프로세서 클러스터를 활용하고자 하는 경우, 필요한 데이터들은 내부 연결망(interconnection network)을 통해 L2 캐쉬에 접근하여 가져온다. 대용량 데이터 연산이 많은 GPU는 많은 메모리 인터페이스를 가지고 있는데 이는 그래픽스 전용 주 메모리와 SM간의 데이터 전송을 위해 사용된다. 그래픽스 전용 메모리는 매우 높은 대역폭을 제공하지만, GPU에서 메인 메모리를 접근하는 경우에는 최대 600 사이클 정도의 매우 긴 지연시간이 요구되므로, 이러한 오버헤드를 극복하기 위하여 다수의 스레드를 동시에 처리하는 다중 스레드(multi-thread) 연산을 지원한다. 다중 스레드 연산은 여러 개의 스레드를 단일 워프(warp)로 묶어 이를 동시에 수행시킴으로써, 스레드 수준의 병렬 연산을 수행하여 연산 속도를 높여줌과 동시에 추가적인 메모리 접근을 줄이기 때문에 메모리 접근으로 인해 발생하는 접근 지연시간을 숨길 수 있다. 하지만, 이 과정에서는 GPU 내부의 코어에 스레드를 분배하는 스레드 스케줄링 부분에서 높은 오버헤드가 발생하기 때문에 성능이 저하되는 단점이 존재하기도 한다. 이와 같이 GPU의 성능 향상을 위해서 많은 설계 기법들이 지속적으로 연구되어 왔으며, 앞으로도 더욱 많은 연구들이 진행될 것으로 예상된다[9-11].

2. CUDA를 이용한 범용 GPU 연산

범용 GPU 연산은 주로 DirectX 또는 OpenGL과 같은 셰이딩 언어(shading language)를 이용하여 수행된다. 셰이딩 언어를 이용하여 범용 GPU 연산을 수행하고자 하는 경우에는, GPU 내부 구조와 셰이딩 언어에 대한 높은 이해도를 필요로 하기 때문에, 이에 익숙하지 않은 소프트웨어 개발자들은 GPU를 이용한 범용 GPU 연산을 수행하는 프로그램을 개발하는데 있어서

많은 어려움을 겪고 있다. 하지만, 최근에는 GPU의 높은 병렬 연산 성능이 더욱 큰 관심을 받게 되면서 GPU를 단순히 그래픽스 연산이 아닌, 범용 목적 용도로 사용하기 위한 노력들이 계속되고 있다. 이로 인해, GPU를 이용한 고성능 컴퓨터 시스템을 구성하는 방안 또한 꾸준히 연구가 진행되고 있다. 특히, CUDA 기술은 개발자들에게 친숙한 C 언어를 기반으로 개발되었기 때문에 범용 GPU 연산을 쉽게 활용할 수 있는 대표적인 기술로 사용되고 있다. 수학 또는 과학 분야와 같이 많은 연산을 요구하며 병렬처리에 적합한 프로그램의 경우, CUDA를 통해 GPU를 범용 목적으로 활용한다면 다수의 코어에서 다수의 스레드를 동시에 실행시킬 수 있기 때문에 더욱 뛰어난 수행 성능을 얻을 수 있다.

그림 3. CUDA를 이용한 프로그램의 실행 패턴

CUDA로 작성된 프로그램이 실행되는 패턴을 살펴보면 [그림 3]에서 보이는 바와 같다. 그림에서 디바이스(device)로 표현된 GPU를 이용하여 CUDA로 작성된 프로그램을 수행하기 위해서는 호스트(host)로 표현된 CPU의 도움이 필요하다. CUDA를 통해 GPU를 활용하고자 하는 경우에는 호스트에서 동작하는 호스트 전용 코드와 함께 디바이스에서 동작하는 디바이스 전용 코드를 작성하여 GPU에서 수행되는 프로그램 함수들이 생성될 수 있도록 코드를 작성해야 한다. 각각의 호스트 코드들은 순차문으로써 단일 스레드만을 처리하지만, 디바이스 코드들은 GPU에서 동작하므로 병렬 처리를 통해 매우 많은 스레드들을 동시에 처리할 수 있다.

CUDA에서는 GPU 메모리를 칩 내부 메모리

(on-chip)과 칩 외부 메모리(off-chip)으로 나누어 관리한다. 칩 내부 메모리는 레지스터(register)와 지역(local) 메모리, 공유(shared) 메모리가 있고, 칩 외부 메모리에는 전역(global) 메모리, 컨스턴트(constant) 메모리, 텍스처(texture) 메모리가 있다.

그림 4. CUDA 프로그램 메모리 구조

[그림 4]는 CUDA 프로그램의 메모리 구조와 각각의 메모리 종류에 따른 연관 관계를 나타낸다. 칩 내부 메모리인 레지스터와 공유 메모리는 GPU 코어 내부에 있는 읽기/쓰기 메모리로 사용되고, 코어 내부에 있으므로 SP와 같은 연산 코어에서 1사이클에 접근 가능하므로, 레지스터와 공유 메모리를 사용하는 경우에는 매우 빠른 속도로 데이터를 처리할 수 있다. 읽기 전용으로 사용되는 칩 외부 메모리인 텍스처 메모리와 컨스턴트 메모리는 메인 메모리에 할당되고 캐시 메모리가 존재하므로 캐시 히트가 발생하는 경우에는 매우 빠른 속도로 데이터를 처리할 수 있지만, 캐시 미스가 발생하는 경우에는 메인 메모리로부터 데이터를 읽어오기 때문에 이에 따른 접근 오버헤드가 발생하여 데이터 처리 속도가 증가할 가능성이 있다. 전역/지역 메모리는 읽기와 쓰기가 모두 가능한 칩 외부 메모리이다. CUDA를 통해 GPU를 활용하는 경우 메모리 접근으로 인한 지연시간을 줄이기 위해서는 내부 칩 메모리를 주로 사용하도록 하는 것이 보다 효율적이지만, 내부 칩 메모리의 용량은 매우 작기 때문에 메모리 간의 특성을 고려하여 효율적으로 프로그램을 작성해야 한다. 즉, GPU 메모리 구조의 특성들을 충분히 고려해서 범용 GPU 연산을 위한 CUDA 프로그램을 작성한다면 GPU

의 연산 능력을 보다 효과적으로 활용하는 컴퓨터 시스템을 구현할 수 있다.

III. 실험 환경

1. 모의실험 환경

본 논문에서는 GPU의 코어 개수 및 동작 주파수 변화에 따른 성능 변화를 측정하기 위해 GPGPU-SIM을 사용한다[12]. GPGPU-SIM은 GPU와 같이 다수의 코어를 사용하는 멀티 코어 프로세서 구조의 성능을 사이클 단위의 정확성으로 평가하는 신뢰성이 검증된 시뮬레이터이다. GPGPU-SIM은 CUDA와 OpenCL 응용 프로그램을 입력으로 받아 들여, 응용 프로그램의 수행 결과와 더불어 프로세서 구조의 성능을 IPC와 메모리 접근 지연 시간 및 적중률 등 다양한 측면에서의 측정 결과를 출력해 준다.

벤치마크로는 NVIDIA SDK[13] 프로그램들과 Rodinia[14] 벤치마크 프로그램들 중에서 병렬성이 높은 응용 프로그램들을 선별하여 총 6개의 프로그램들을 이용한다. 선택된 벤치마크 프로그램에 대한 상세한 설명은 [표 1]에 기술한다.

표 1. 벤치마크 프로그램

벤치마크 종류	벤치마크 명	설명
CUDA SDK[13]	SimpleTexture	텍스처와 맵핑 연산을 수행하는 프로그램
	Transpose	전치 연산 행렬을 빠르게 처리하는 프로그램
	BFS(Breadth First Search)	그래프 넓이 우선 평가 알고리즘 프로그램
Rodinia [14]	HotSpot	프로세서의 온도를 평가하는 열 차등 방정식을 재구현하는 프로그램
	Back Propagation	계층 신경 네트워크의 노드를 연결하는 가중치를 연산하는 알고리즘이 구현된 프로그램
	LU Decomposition	선형 방정식의 집합 솔루션을 연산하는 알고리즘 프로그램

실험에서 GPU의 각 구성요소는 QuadroFX5800[15] 및 Samsung GDDR5 메모리[16]를 참고하여 설정한다.

GPU 기본 구조는 [표 2]에서 보이는 바와 같은 구성요소를 기반으로 설정하고, 내부 동작 주파수는 코어 325MHz, 내부 연결망 650MHz 그리고 메모리 1000MHz를 사용한다.

표 2. GPU 기본 구성 변수

GPU Parameters	
Number of Shader Core	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	512
Number of CTAs / Core	8
Number of Registers / Core	16,384(Byte)
Shared Memory / Core	32(KB)
Number of Memory Channels	8
Bandwidth per Memory Module	8(Bytes/Cycle)
DRAM Request Queue Capacity	32
GDDR5 Memory Timing	
tCL=9, tRP=13, tRC=34, tRAS=21, tRCD=12, tRRD=8	

표 3. 코어 개수와 동작 주파수 변수

Number of Cores				
	BASE	Level-1	Level-2	Level-3
Number of Cores	30	48	144	192
GPU Clock Frequency(MHz)				
	BASE	Level-1	Level-2	Level-3
Core	325	350	435	450
Interconnection	650	700	870	900
Memory	1000	1000	1000	1000

GPU의 동작 주파수 증가 및 코어 개수 증가에 따른 성능 변화 양상을 분석하는 연구를 위하여 [표 3]에서 보이는 바와 같이 변수들을 변화시키며 실험을 수행한다.

2. 내부 연결망 구성

GPU의 내부 연결망 구성은 GPGPU-SIM 시뮬레이터에 연동된 Booksim[17] 시뮬레이터를 활용한다. 실험에 사용한 내부 연결망의 형태 및 내부 연결망을 구성하는 라우터의 특성은 [표 4]에서 보이는 바와 같다.

표 4. 내부 연결망 구성 변수

Parameter	Value
Topology	Crossbar
Routing Mechanism	Dimension Order
Routing Delay	1
Virtual Channels	2
Virtual Channel Buffers	4
Virtual Channel Allocator	iSLIP
VC Allocator Delay	1
Input Speedup	2
Flit Size(Bytes)	16

그림 5. GPU의 내부 연결망 구조

[그림 5]에서 보여주듯이, 본 실험에서 사용한 GPU의 내부 연결망인 크로스바 구조에서 각 코어들은 크로스바 내부 연결망을 통하여 메모리 인터페이스를 거쳐 주 메모리로 데이터의 전송 및 인출을 수행한다.

IV. 실험 결과

본 논문에서는 각 벤치마크 별로 고정된 동작 주파수에서 코어 개수의 증가에 따른 GPU의 성능 변화와, 고정된 코어 개수에서 동작 주파수를 증가하는데 따른 GPU의 성능 변화를 상세하게 분석하고자 한다. 실험에 사용된 벤치마크들을 Back Propagation은 B, Hotspot은 H, SimpleTexture는 Tex, LU Decomposition는 Lud, Transpose는 Trans로 표기하며, 나머지 BFS는 그대로 표기한다[13][14]. [그림 6-10]의 가로축에서 볼 수 있는 벤치마크 이름 뒤에 레벨 단계(BASE, LV1,

LV2, LV3)와 붙은 숫자(30, 48, 144, 192)는 [표 3]에 나타난 동작 주파수와 코어 개수를 나타낸다.

[그림 6-10]에서 코어 개수에 따른 분석은 각 벤치마크 프로그램과 동작 주파수에 따라 진행된다. 예를 들면, 동작 주파수 H_LV3는 [표 3]의 동작 주파수 레벨 3단계의 값을 가지는 GPU에서 Hotspot 프로그램을 수행하는 경우를 의미한다. 그리고 H_LV3은 총 3가지 색상(파란색, 빨간색, 녹색)의 막대를 가지고 있다. 막대의 색상은 그림 상단의 범례에서 보이듯이, 코어의 수인 48, 144, 192를 나타낸다. 특히, 코어의 개수가 30개인 경우를 기준으로 성능 향상 폭을 나타내고 있기 때문에 코어의 수가 30개인 경우는 그래프에서 보여주지 않는다. 다시 말하면, H_LV3의 막대 그래프 3개는 벤치마크 프로그램과 동작 주파수의 조합 중 Hotspot 벤치마크 프로그램이 동작 주파수 LV3에 수행되는 경우에 대해서 코어의 개수 변화에 따른 분석 결과를 보여준다.

이와 유사하게, [그림 6-10]에서 동작 주파수 변화에 따른 분석은 각 벤치마크 프로그램과 코어의 개수에 따라 진행된다. 예를 들면, BFS_48은 48개의 코어를 가진 GPU에서 BFS 프로그램을 수행하는 경우를 의미한다. 각 “벤치마크_코어 수”는 BASE 단계를 기준으로 동작 주파수 변화에 따른 성능 향상을 보여준다. 그림 상단의 범례에서 보이듯이, 기준인 BASE를 제외한 1단계(LV1), 2단계(LV2) 그리고 3단계(LV3)의 값에서 수행한 결과 값을 나타낸다. 다시 말하면, BFS_48의 막대 그래프 3개는 벤치마크 프로그램과 코어 개수의 조합 중 BFS 벤치마크 프로그램이 48개의 코어를 가진 GPU에서 수행되는 경우 코어의 수가 변화함에 따른 분석 결과를 보여준다. 모든 실험 결과는 3.1에서 설명한 기본구조를 기준으로 하는 상대적인 값을 사용한다. 그리고 벤치마크 프로그램에 따른 변화량의 폭이 너무 큰 경우에는, 실험 결과를 보다 자세히 보여주기 위하여 그림을 분할하여 Y축의 값을 다르게 사용하여 보여준다.

1. 코어 개수 증가에 따른 성능 변화

GPU의 코어 개수 증가에 따른 성능 변화를 분석하기 위해, 동작 주파수가 고정된 상태에서 코어의 개수를 변경하여 각 벤치마크 별로 성능 변화 양상을 측정한다.

결과는 [그림 6]에 보이는 바와 같다. 본 논문에서는 성능을 평가하기 위하여 사이클 당 명령어 수를 나타내는 IPC(Instructions per Cycle)을 사용한다. IPC는 동일 주파수를 사용하는 프로세서의 성능을 평가하는 경우, 널리 사용되는 성능 평가 척도 중 하나이다. 실험 결과, 가장 높은 성능 향상은 동작 주파수가 BASE이고 코어 개수가 144인 GPU 구조에서 SimpleTexture를 수행한 결과인 320.6%이고, 동작 주파수가 LV3이고 코어 개수가 48개일 때 BFS를 수행하는 경우에는 오히려 0.8%의 성능이 감소한다. 코어 개수 증가에 따른 GPU의 성능 향상은 코어 개수가 48, 144, 192인 경우에는, 평균적으로 28.9%, 122.3%, 125.8%로 코어의 수가 192개인 경우 가장 높게 나타난다. 코어 개수가 144, 192인 경우, 평균 성능은 192개인 경우가 더 높지만 그렇지 않는 경우도 많이 존재하고 성능 향상의 폭 또한 대단히 제한적임을 알 수 있다. 그러므로 GPU의 코어 개수를 증가시키는 것이 항상 성능 향상을 가져온다고는 단언할 수 없으며, 오히려 설계비용 측면을 고려한다면 오히려

못하다고 할 수 있다.

2. 동작 주파수 증가에 따른 성능 변화

코어 개수를 고정된 상태에서 GPU의 동작 주파수를 증가시키며 각 벤치마크 별로 성능 변화 양상을 측정하는 실험을 수행한 결과는 [그림 7]에 보이는 바와 같다. 실험 결과, GPU의 동작 주파수를 증가시키면 모든 벤치마크에서 성능이 향상되는 것을 확인할 수 있다. 동작 주파수 증가에 따른 성능을 평가하는 경우, 한 사이클에 소모되는 물리적 시간이 상이하기 때문에 IPC 값을 사용할 수 없으므로 본 논문에서는 프로그램 수행시간을 사용하여 동작 주파수 증가에 따른 성능을 평가하고자 한다.

가장 높은 성능 향상은 코어 개수 144, 동작 주파수 LV3의 GPU 구조에서 Transpose를 수행한 결과로써 28%이다. 반면에, 가장 낮은 성능 향상은 코어 개수 192, 동작 주파수가 LV1의 GPU 구조에서 Backprop을 수행하는 경우에 얻을 수 있는 0.2%이다. 동작 주파수

그림 6. 코어 수 증가에 따른 IPC 증가율

그림 7. 동작 주파수 증가에 따른 수행시간 감소율

증가에 따른 GPU의 성능 향상은 동작 주파수가 LV1, LV2, LV3인 경우, 평균적으로 4.4%, 14.9%, 16.2%로 동작 주파수가 높아질수록 크게 나타난다. 하지만, 성능 향상 폭 측면으로 본다면 LV2, LV1, LV3 순으로 성능 향상의 효과를 얻을 수 있다. 그러므로 동작 주파수를 증가하는 경우 발생하는 전력과 온도 문제들을 함께 고려한다면 무조건 높은 주파수로 동작시키는 것보다는 최대한의 효과를 얻을 수 있는 동작 주파수를 선택하는 것이 적절하다고 판단된다.

[그림 6][그림 7]을 살펴보면, 코어 개수가 증가할수록 동작 주파수 증가에 따른 성능 향상의 효율성은 감소하는 것을 볼 수 있다. 뿐만 아니라, 동작 주파수가 증가할수록 코어 개수 증가에 따른 성능 향상의 효율성 또한 감소하는 것을 볼 수 있다. 다음 절에서는 코어 개수 및 동작 주파수 증가에 따른 성능 향상 효율성 저하 요소들에 대해 상세히 분석하고자 한다.

3. 코어 개수 및 동작 주파수 증가에 따른 성능 향상 효율성 저하 요소

성능 향상 효율성 감소에는 많은 원인들이 존재하지만, 주 메모리 내부 전송 큐 대기 시간 증가, 코어에서 주 메모리로의 데이터 전송 시간 증가, 파이프라인에서의 데이터 대기 시간 증가 등이 가장 크게 영향을 미친 요소들로 분석된다. 주 메모리 내부 전송 큐 대기 시간 증가는 메모리 인터페이스 대기 큐의 포화상태에 따른 오버헤드를 의미하고, 코어에서 주 메모리로의 데이터 전송 시간 증가는 세이더 코어에서 메모리 인터페이스로 데이터를 전송할 때 발생하는 지연 시간의 증가에 의해 발생하는 오버헤드이며, 파이프라인에서의 데이터 대기 시간 증가는 GPU에서 파이프라인 수행 중에 워프 스케줄러에서 스레드를 기다리는데 발생하는 오버헤드이다. 우리는 성능 향상 효율성 감소에 대한 원인 분석 또한 코어 개수 증가와 동작 주파수 증가의 두 가지 경우로 나누어서 수행하고자 한다. [그림 8-10]은 코어 개수 및 동작 주파수 증가에 따른 대표적인 성능 향상 효율성 저하 요소들로 인해 발생하는 오버헤드의 원인을 정량적인 데이터를 통해 보여주고 있다. 주 메모리 내부 전송 큐 대기 시간은 코어 개수가 48, 144,

(A) 코어 개수 증가에 따른 변화율

(B) 주파수 증가에 따른 변화율

그림 8. 성능 향상 효율성 감소 원인 분석 - 주 메모리 내부 전송 큐 대기 시간

192로 증가하는 경우 평균 50%, 331%, 375%(최대 165%, 1637%, 1764%) 증가하고, 동작 주파수가 LV1, LV2, LV3로 증가하는 경우에는 평균 25%, 109%, 127%(최대 140%, 533%, 597%)로 증가한다. 코어에서 주 메모리로의 데이터 전송 시간은 코어 개수가 48, 144, 192로 증가하는 경우 평균 62%, 318%, 485%(최대 82%, 634%, 1010%) 증가하고, 동작 주파수가 LV1, LV2, LV3로 증가하는 경우에는 평균 7%, 32%, 38%(최대 20%, 69%, 81%)로 지속적으로 증가한다.

마지막으로 파이프라인에서의 데이터 대기 시간은 코어 개수가 48, 144, 192로 증가하는 경우 평균 55%, 1129%, 2379%(최대 84%, 4114%, 8463%) 증가하고, 동작 주파수가 LV1, LV2, LV3로 증가하는 경우에는 평균 10%, 50%, 59%(최대 31%, 176%, 219%)로 급격히 증가한다. 실험 결과에서 볼 수 있듯이, 주 메모리 내부 전송 큐 대기 시간, 코어에서 주 메모리로의 데이터 전송 시간 그리고 파이프라인에서의 데이터 대기 시간은 매우 높은 증가율을 보이고 있다. 그러므로 코어 개수 및 동작 주파수를 증가시키는 경우 동시에 급격하게 증

가하는 3가지의 오버헤드들로 인해 성능 향상의 효율성을 저해되는 것으로 분석된다. 특히, 동작 주파수를 증가시킬 때 발생하는 오버헤드 보다 GPU 내부의 코어 개수를 증가시킬 때 발생하는 오버헤드의 비율이 매우 높게 나타난다.

위에서 기술한 현상이 나타나는 이유는 코어 개수 및 동작 주파수 증가로 인해 향상된 연산 능력을 충분히 활용할 수 있도록 메모리 능력이 부족하기 때문이다. 다시 말하면, GPU 코어 내부에 적재된 메모리에서 데이터를 효과적으로 코어에 전달하지 못해서 생기는 문제라고 할 수 있다.

본 논문에서는 분석 결과를 증명하기 위해서 메모리가 코어로 데이터를 전달하는 메모리 인터페이스 수를 증가시켜 실험을 수행한 결과를 보여주고자 한다. [그림 11]은 메모리 인터페이스의 수를 8개에서 10개로 증가시킨 뒤 기존의 실험과 동일한 방법으로 BFS 벤치마크 프로그램을 수행한 결과이다. 앞서 설명한 바와 같이, 성능은 코어 개수 증가의 경우는 IPC로 평가하며 동작 주파수 증가의 경우에는 실행 시간으로 평가한다.

(A) 코어 개수 증가에 따른 변화율

(B) 주파수 증가에 따른 변화율

그림 9. 성능 향상 효율성 감소 원인 분석 - 코어에서 주 메모리로의 데이터 전송 시간

(A) 코어 개수 증가에 따른 변화율

(B) 주파수 증가에 따른 변화율

그림 10. 성능 향상 효율성 감소 원인 분석 - 파이프라인에서의 데이터 대기 시간

인터페이스 수를 증가한 이후의 성능은 이전과 비교하여 최대 6.8배가 향상됨을 [그림 11]에서 확인할 수 있다. 그러므로 코어로 데이터를 전달하는 메모리 인터페이스 수를 증가시키는 것과 같이 메모리의 능력을 향상시킨다면 보다 뛰어난 GPU 성능 향상 효과를 가져올 수 있을 것으로 기대된다.

그림 11. 메모리 인터페이스 수에 따른 성능 향상

V. 결론

본 논문에서는 병렬 응용프로그램 수행 시 GPU의 동작 주파수 증가 및 코어 개수 증가에 따른 성능 변화 양상과 추가적으로 발생하는 오버헤드에 대하여 상세하게 분석하였다. 실험 결과, 코어 개수가 48, 144, 192개로 증가하는 경우, GPU의 성능은 평균 28.9%, 122.3%, 125.8%로 지속적으로 향상한다. 그리고 동작 주파수가 LV1, LV2, LV3로 높아지는 경우에도 평균 4.4%, 14.9%, 16.2%로 성능이 향상된다. 하지만, 코어 개수 및 동작 주파수 증가에 따른 GPU의 성능은 지속적으로 향상되는 반면, 성능 향상의 효율성은 점차 감소하는 것을 확인할 수 있었다. 분석 결과, 성능 향상 효율성 감소는 주 메모리 내부 전송 큐 대기 시간 증가, 코어에서 주 메모리로의 데이터 전송 시간 증가, 그리고 파이프라인에서의 데이터 대기 시간 증가 등의 원인 때문으로 분석된다. 즉, 향상된 GPU의 연산 능력을 메모리가 충분히 뒷받침 해주지 못함으로써 성능 향상이 저해됨을 실험을 통해 확인하였다. 이는 본 논문에서 수행한 메

모리 인터페이스 개수를 증가시킨 실험에서 성능이 이전과 비교하여 월등히 향상되는 것을 통해 확인할 수 있었다.

참 고 문 헌

- [1] <http://www.khronos.org/opencv/>
- [2] K. Gray, *The Microsoft DirectX 9 Programmable Graphics Pipeline*, Microsoft Press, 2003.
- [3] <http://www.nvidia.co.kr/page/geforce8.html>
- [4] NVIDIA CUDA™, *Programming Guide Version 2.3.1*, NVIDIA Corporation, 2009.
- [5] <http://www.gpgpu.org>
- [6] Y. H. Jang, C. Park, J. H. Park, N. Kim, and K. H. Yoo, "Parallel Processing for Integral Imaging Pickup using Multiple Threads," *International Journal of Korea Contents*, Vol.5, No.4, pp.30-34, 2009.
- [7] Y. H. Jang, C. Park, J. S. Jung, J. H. Park, N. Kim, J. S. Ha, and K. H. Yoo, "Integral Imaging Pickup Method of Bio-Medical Data using GPU and Octree," *International Journal of Korea Contents*, Vol.10, No.9, pp.1-9, 2009.
- [8] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE MICRO*, Vol.28, No.2, pp.39-55, 2008.
- [9] Yao Zhang and John D. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," In *Proceedings of International Symposium on High Performance Computer Architecture*, pp.382-393, 2011.
- [10] Veynu Narasiman, Chang Joo Lee, Michael Shebanow, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," In *Proceedings of international*

symposium on Microarchitecture, 2011.

- [11] Wilson W. L. Fung, Inderpreet Singh, A. Brownsword, and Tor M. Aamodt, "Hardware Transactional Memory for GPU Architectures," In *Proceedings of international symposium on Microarchitecture*, 2011.
- [12] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, pp.163-174, 2009.
- [13] <http://developer.download.NVIDIA.com/compute/cuda/sdk/website/samples.html>
- [14] <http://lava.cs.virginia.edu/Rodinia/>
- [15] http://www.NVIDIA.com/object/product_quadro_fx_5800_us.html
- [16] <http://www.datasheetarchive.com/samsung%20gddr5-datasheet.html>
- [17] <http://nocs.stanford.edu/booksim.html>

저 자 소 개

최 홍 준(Hongjun Choi)

정회원

- 2009년 : 전남대학교 전자컴퓨터 공학부 공학사
- 2011년 : 전남대학교 전자컴퓨터 공학과 공학석사
- 2011년 ~ 현재 : 전남대학교 전자컴퓨터공학과 박사과정

<관심분야> : 컴퓨팅 시스템, 병렬처리, 컴퓨터구조

김 철 홍(Cheolhong Kim)

정회원

- 1998년 : 서울대학교 컴퓨터공학사
- 2000년 : 서울대학교 컴퓨터공학부 공학석사
- 2006년 : 서울대학교 전기컴퓨터공학부 공학박사
- 2005년 ~ 2007년 : 삼성전자 반

도체총괄 SYS.LSI 사업부 책임 연구원

- 2007년 ~ 현재 : 전남대학교 전자컴퓨터공학부 교수
<관심분야> : 임베디드시스템, 컴퓨터구조, SoC 설계, 저전력 설계