http://dx.doi.org/10.5392/JKCA.2014.14.03.022

계층적 메모리 구성에 따른 GPU 성능 분석

Analysis on the GPU Performance according to Hierarchical Memory Organization

최홍준*, 김종면**, 김철홍*
전남대학교 전자컴퓨터공학과*, 울산대학교 전기공학부**

Hongjun Choi(chj6083@gmail.com)*, Jongmyon Kim(jmkim07@ulsan.ac.kr)**,
Cheolhong Kim(chkim22@chonnam.ac.kr)*

요약

병렬 연산에 최적화된 하드웨어를 가진 GPU를 그래픽스 작업 이외에 범용 작업에 활용하고자, 최근에 GPGPU 기술이 큰 관심을 받고 있다. GPU와 같은 대용량 병렬처리 장치에서는 메모리 시스템이 성능에 큰 영향을 미치게 된다. GPU에서는 메모리 시스템의 효율성을 향상시키기 위하여, 메모리 대역폭 사용률을 감소시켜주는 계층적 메모리 구조와 메모리를 요청하는 트랜잭션을 줄여주는 메모리 주소 접합과 메모리 요청 합병 등의 기술들을 사용한다. 본 논문에서는 메모리 시스템 효율성 향상을 위해 활용되는 기법들이 GPU 성능에 미치는 영향을 정량적으로 평가하고 분석하기 위해, 다양한 메모리 구조에 대한 실험을 수행한다. 실험 결과에 따르면, 캐쉬를 사용하지 않는 경우에 비해 8KB, 16KB, 32KB, 64KB의 L1 캐쉬를 추가하면 평균적으로 15.5%, 21.5%, 25.5%, 30.9%의 성능이 각각 향상된다. 하지만, 일부 벤치마크 프로그램에서는 데이터 일관성을 유지하기 위하여 메모리 트랜잭션이 증가함에 따라 오히려 성능이 감소하는 결과를 보이기도 한다. 그리고 메모리 요청에 대한 미스가 많이 발생하는 경우에는 캐쉬 레벨이 증가함에 따라 평균 메모리 접근 지연 시간이 증가하기도 한다.

■ 중심어 : | 그래픽 처리장치 | 메모리 시스템 | 계층적 메모리 구조 | 메모리 요청 병합 |

Abstract

Recently, GPGPU has been widely used for general-purpose processing as well as graphics processing by providing optimized hardware for parallel processing. Memory system has big effects on the performance of parallel processing units such as GPU. In the GPU, hierarchical memory architecture is implemented for high memory bandwidth. Moreover, both memory address coalescing and memory request merging techniques are widely used. This paper analyzes the GPU performance according to various memory organizations. According to our simulation results, GPU performance improves by 15.5%, 21.5%, 25.5%, 30.9% as adding 8KB L1, 16KB L1, 32KB L1, 64KB L1 cache, respectively, compared to case without L1 cache. However, experimental results show that some benchmarks decrease performance since memory transaction increases due to data dependency. Moreover, average memory access latency is increased as the depth of hierarchical cache level increases when cache miss occurs significantly.

■ keyword : | GPU | Memory System | Hierarchical Memory Architecture | Memory Request Merging |

* 본 연구는 2013년도 정부(미래창조과학부)의 재원으로 한국연구재단 기초연구사업의 지원(2012R1A1B4003492)과 미래창조과학부 및 정보통신산업진흥원의 대학IT연구센터육성 지원사업의 연구결과로 수행되었음(NIPA-2013-H0301-13-3005)
접수일자 : 2013년 11월 14일
수정일자 : 2013년 12월 24일
심사완료일 : 2013년 12월 26일
교신저자 : 김철홍, e-mail : chkim22@chonnam.ac.kr

I. 서 론

싱글코어 프로세서의 주파수 증가가 물리적으로 한계점에 도달함에 따라, 최근에는 코어의 개수를 증가시켜 마이크로프로세서의 성능을 향상시키는 멀티코어 프로세서 설계 기술이 새로운 패러다임으로 정착되었다[1-3]. 두 개 이상의 코어를 가지는 멀티코어 프로세서는 병렬성을 활용하여 싱글코어 프로세서에 비해 우수한 성능과 전력효율성을 제공하기 때문에, 최신에 출시되는 마이크로프로세서들은 대부분 멀티코어로 설계된다[4]. 이와 같이, 컴퓨팅 시스템의 성능 향상에 있어서 마이크로프로세서의 병렬성이 중요해짐에 따라 개발자들은 강력한 병렬 처리 하드웨어 자원을 가진 GPU에 주목하기 시작하였다[5].

GPU는 높은 데이터 처리 능력을 요구하는 그래픽스 관련 연산을 처리하기 위하여 개발된 대용량 병렬 데이터 처리장치이다. GPU는 강력한 연산 자원을 효과적으로 사용하기 위하여, 수 천개의 스레드들을 동시에 수행시킴으로써 응용프로그램의 병렬성을 최대한 활용한다. 최신 GPU들은 동시에 수 천개의 스레드들을 동시에 수행하기 때문에, 수 Gflop/s의 높은 데이터 처리량을 보여준다. 이와 같이 뛰어난 연산 능력을 보여주는 GPU를 그래픽스 작업 이외에 높은 연산량을 요구하는 범용 작업에 활용하고자 다양한 연구가 진행되고 있다[6][7]. 대표적인 기술이 그래픽 처리장치 병렬컴퓨팅(GPGPU: General Purpose computation on the Graphics Processing Unit)이다[8][9].

기존의 GPGPU는 그래픽 파이프라인과 같이 GPU의 하드웨어 기술에 대한 높은 이해도를 필요로 하였기 때문에 개발자들이 쉽게 접근할 수 없었던 반면에, 최근의 GPGPU는 GPU 회사들이 다양하고 친숙한 API(Application Program Interface)들을 제공하기 때문에 개발자들이 쉽게 범용 응용프로그램들을 GPU를 활용하여 수행할 수 있다. 대표적인 GPGPU 기술로는 Khronos Group의 OpenCL, ATI(AMD)의 Stream Technology, 그리고 NVIDIA 의 CUDA(Compute Unified Device Architecture) 등이 존재한다[10-12].

GPU는 스트리밍 멀티프로세서(SMs, Streaming Multiprocessors)로 명명되는 멀티스레드 프로세서를 다수 포함한다. 스트리밍 멀티프로세서는 실제 연산을 담당하는 다수의 GPU 코어(GPU core)[1]를 활용하여 수많은 스레드들을 통해 단일명령어 다중데이터(SIMD, Single Instruction Multiple Data) 형식으로 연산을 수행한다. SIMD 형식의 연산을 지원하기 위해서, GPU는 같은 명령어를 수행하는 스레드들을 그룹화한다. 그룹화된 스레드들 각각은 그들 자신의 피연산자들을 수행한다[12].

그룹화된 스레드들을 NVIDIA에서는 워프(Warp)라고 칭하는 반면에, AMD(ATI)에서는 웨이브프론트(wavefront)라고 부른다. 본 논문에서는 NVIDIA사의 GPU를 기본 GPU로 설정하기 때문에 그룹화된 스레드들을 워프(Warp)라는 용어로 표현한다. 뿐만 아니라, 본 논문에서는 GPU 관련 용어에 대한 기술에 있어서 특별한 언급이 없다면 NVIDIA에서 사용하는 용어를 사용한다[11][12].

일반적으로 각 스트리밍 멀티프로세서가 외부 메모리로 접근을 요청하는 경우 수 백 사이클의 접근 시간을 요구한다. 외부 메모리 접근으로 인해 발생하는 수백 사이클의 지연시간으로 인한 성능 저하를 극복하기 위하여, 스트리밍 멀티프로세서는 동시에 다수의 워프를 지원한다. 다시 말하면, 스트리밍 멀티프로세서는 다수의 스레드를 갖는 워프를 여러 개 할당받아 연산을 처리하기 때문에 수많은 스레드를 동시에 처리한다. 이와 같은 대규모 멀티스레드 기법은 GPU의 처리량을 크게 향상시키는 반면에, 다수의 메모리 요청들이 동시에 발생할 가능성이 높기 때문에 심각한 메모리 병목현상을 종종 유발한다. 그러므로 GPU에서는 메모리 시스템의 효율성을 향상시키기 위하여, 다양한 기법들이 활용된다[13][14].

메모리 요청으로 인한 메모리 대역폭을 줄이기 위한 기법 중 가장 대표적인 기법은 데이터를 캐싱(caching)하는 것이다. 그러므로 최신 GPU들에서는 메모리 대역폭 사용을 줄이기 위하여 다양한 캐쉬를 사용하여 데이터를 캐싱한다. 그리고 GPU에서 수행되는 병렬 응용프

1) GPU 구조에 따라, NVIDIA에서 셰이더 코어(shader core), 쿠다 코어(CUDA core) 등으로도 불린운다.

로그램은 대부분 높은 지역성을 가지고 있기 때문에, 다수의 메모리 접근들을 병합하는 메모리 주소 접합과 메모리 요청 합병 등과 같은 기술들이 사용된다. 예를 들면, CUDA 프로그래밍은 하나의 워프 내의 스레드들이 연속된 메모리 지역으로 순차적으로 접근하고자 하는 병렬 메모리 접근들을 접합(coalescing)하여 메모리 요청을 위한 트랜잭션을 보다 적게 생성한다. 이와 유사하게, GPU에서는 동일한 메모리 블록에 접근하는 다수의 접근들을 줄이기 위하여 메모리 요청들을 합병(merging)하여 메모리 접근 횟수를 감소시킨다. 메모리 접합과 메모리 합병이 스레드들 사이의 공간적 지역성을 활용하여 메모리 시스템의 효율성을 향상시키는 반면, 데이터 캐싱은 각각의 스레드들이 가지고 있는 접근 패턴에 따른 공간적, 지역적 지역성을 활용한다[15][16].

GPU와 같은 대용량 병렬처리 장치의 높은 연산 능력을 효과적으로 활용하기 위해서는 효율적인 메모리 시스템이 필요하다는 것은 자명하다. 그러므로 본 논문에서는 대용량 병렬처리에 효과적인 메모리 시스템을 개발하기 위한 선행 연구로써 GPU의 메모리 효율성을 향상시키기 위하여 활용되는 메모리 요청 병합 기법들과 데이터 캐싱에 따른 영향을 상세하게 분석하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로써 GPU의 기본 구조와 메모리 시스템에 대해 설명한다. 3장은 메모리 구조에 따른 메모리 시스템의 효율성을 평가하고자 구축한 실험 환경에 대하여 기술하고 4장은 실험을 통해 얻은 결과를 상세하게 분석한다. 마지막으로 5장에서 본 논문의 결론을 맺는다.

Ⅱ. GPU

1. GPU 구조

본 장에서 GPU 구조에 대해 간략하게 설명하고자 한다. 앞서 언급한 바와 같이, 본 논문에서 대상으로 한 GPU는 NVIDIA의 GPU이다. 그러므로 본 논문에서는 NVIDIA의 대표 GPU인 Quadro 제품을 기준으로 GPU의 구조에 대해서 기술한다[17].

[그림 1]은 GPU의 기본 구조를 보여주고 있다. 그림에서 보이듯이, GPU는 스트리밍 멀티프로세서들과 내부연결망 그리고 메모리 컨트롤러를 통해 접근되는 DRAM으로 구성되어 있다. DRAM은 오프-칩(off-chip)에 위치해 있으며, DRAM을 제외한 나머지 부분은 온-칩(on-chip)에 위치해 있다.

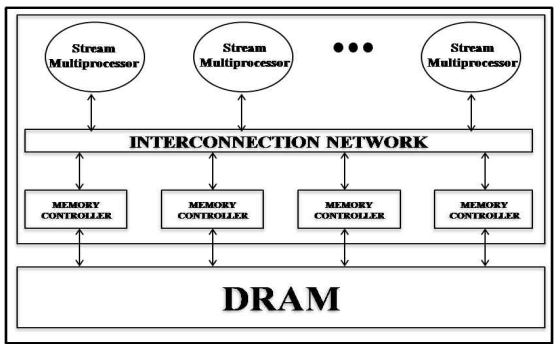

그림 1. 기본 GPU 구조

스트리밍 멀티프로세서는 다수의 GPU 코어들을 활용하여 하나의 워프 내의 스레드들을 동시에 수행한다. 하나의 워프 내의 스레드들은 동일한 명령어를 수행하며, 수행되는 명령어들을 순차적 실행 순서(in-order) 방식으로 실행된다. 만약 연산을 수행하는 과정에서 발생되는 메모리 요청을 스트리밍 멀티프로세서 내의 온-칩 메모리에서 제공하지 못하는 경우, 해당 메모리 요청은 메모리 포트를 통해 외부 메모리로 접근한다. 메모리 요청이 오프-칩 메모리(즉, DRAM)로 접근해야 하는 경우에는 내부 연결망을 통해야 하며, 해당 메모리 요청은 메모리 컨트롤러에 의해 처리된다. 스트리밍 멀티프로세서 내부의 메모리 구조는 다음 2.2장에서 보다 상세히 설명하도록 한다[6][7].

2. GPU 메모리 구조

GPU 내부에 존재하는 각 스트리밍 멀티프로세서는 수행 중인 응용프로그램 내의 병렬 코드를 다수의 스레드로 동시에 수행할 수 있도록 많은 수의 GPU 코어를 가지고 있다. 수많은 스레드들은 GPU의 병렬성을 증가시키는 요소인 반면에, 메모리 시스템에 큰 부담을 주는 요소이기도 하다. 그러므로 우리는 CPU 시스템에

비해 월등히 뛰어난 GPU의 병렬 처리 연산을 효율적으로 활용하기 위해서 메모리 시스템이 매우 중요하다는 것을 알 수 있다. 우리는 본 장에서 GPU 메모리 구조에 대해 상세히 기술하고자 한다.

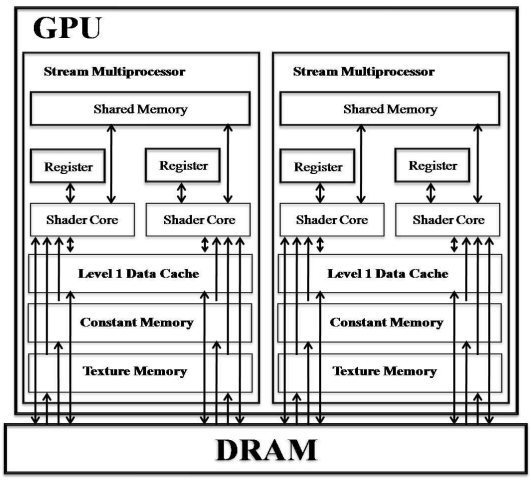

그림 2. GPU의 계층적 메모리 구조

메모리 시스템의 효율성을 향상시키기 위하여, 최신 GPU에서 각 스트리밍 멀티프로세서의 메모리 구조는 계층적으로 구성된다. [그림 2]에서 보이듯이, 크게 온-칩 메모리와 오프-칩 메모리로 분류된 GPU의 메모리 구조에서 온-칩 메모리는 스트리밍 멀티프로세서 내부에 존재하는 공유 메모리, L1 데이터 캐쉬, 상수 캐쉬, 그리고 텍스쳐 메모리이다. 그리고 오프-칩 메모리는 DRAM이다[18].

온-칩 메모리 중 텍스쳐 캐쉬와 상수 캐쉬는 읽기 전용(read-only)이며, 공유 캐쉬와 L1 데이터 캐쉬는 읽기와 쓰기 모두를 지원한다. 공유 캐쉬는 하나의 스트리밍 프로세서 내에서 수행되는 스레드들 사이의 데이터를 주고받기 위해 사용되는 캐쉬이다. 텍스쳐 캐쉬는 높은 공간 지역성 특징을 가진 텍스쳐 필터링 연산 능력을 증가시켜 주는 캐쉬이며, 상수 캐쉬는 응용프로그램 수행 동안 변하지 않는 상수 값을 제공하는 캐쉬이다. 두 캐쉬 모두 메모리 요청으로 인한 대역폭 소비와 응답 시간을 크게 감소시킴으로써 메모리 시스템 효율성을 향상시킨다. 그리고 L1 데이터 캐쉬는 많은 접근 시간을 요구하는 DRAM으로의 접근 빈도를 줄이고자 DRAM의 데이터 중 일부를 캐싱하고 있다.

기존의 GPU 메모리 구조에서는 L1 캐쉬를 사용하지 않기도 하였으나, 최근의 GPU 메모리 구조에서는 메모리 시스템 효율성을 향상시키기 위하여 다중 레벨 캐쉬 구조를 사용하기도 한다. 예를 들어, 최신 GPU 구조인 Tesla를 채택한 제품에서는 L1 캐쉬뿐만 아니라, L2 캐쉬까지 탑재하기도 한다[19].

메모리 요청을 위한 트랜잭션을 감소시켜 메모리 시스템의 효율성을 향상시키는 메모리 주소 접합 기술과 메모리 요청 합병 기술에 대하여 상세히 설명하고자, 우리는 다음 하부 절에서 두 기술을 기술한다.

2.1 메모리 주소 접합 기술[15]

GPU를 활용하고자 하는 대부분의 응용프로그램들은 높은 지역성을 갖도록 작성된다. 이것은 워프내의 스레드들에서 발생된 다수의 메모리 요청의 주소는 유사 또는 동일하다는 것을 의미한다. GPU의 인출 및 저장 장치(LSU, load/store unit)는 메모리 대역폭 사용을 줄이기 위하여, 하나의 워프에서 발생된 다수의 순차적 메모리 주소들을 접합(coalescing)하는 동작을 수행하는 메모리 접합 로직(MCL: Memory Coalescing Logic)을 포함한다.

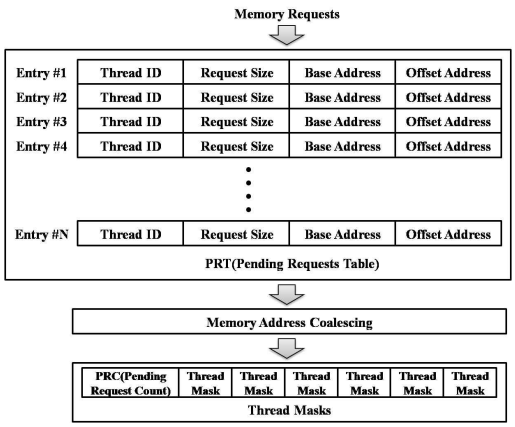

그림 3. 메모리 접합 로직

설명을 명확하고 간략하게 하기 위하여, 본 하부 절에서는 모든 메모리 요청들이 하나의 워프에서 발생된다고 가정한다. [그림 3]는 메모리 접합 로직의 주요 구

성요소를 보여주고 있다. 각각의 메모리 접합 로직은 다수의 엔트리로 구성된 PRT(Pending Requests Table)를 가지고 있으며, PRT 엔트리는 스레드 색인, 메모리 접근의 기본 주소와 오프셋 주소, 그리고 하나의 워프에 의해 동시에 요청된 모든 메모리 요구들의 크기에 대한 정보를 저장한다. 메모리 요청이 발생될 때마다 PRT 엔트리들의 정보는 작성된다.

접합된 메모리 요청들의 수를 결정하기 위해서 메모리 접합 로직은 첫 번째 스레드의 메모리 요청의 기본 주소와 PRT 엔트리에 남아있는 모든 메모리 요청들의 기본 주소들을 비교한다. 비교 결과, 첫 번째 스레드의 메모리 요청의 기본 주소와 일치하는 엔트리 내의 메모리 요청은 첫 번째 스레드의 메모리 요청과 접합되며 이 스레드의 마스크는 "0"으로 설정된다. 스레드의 마스크 정보는 별도의 분류된 스레드 마스크 배열에 저장된다. 이 과정은 PRT 엔트리에 남아있는 메모리 요청이 없을 때까지 반복하여 수행된다. 그 이후, 스레드 마스크가 설정된 스레드들이 요구하는 메모리 요청들을 하나씩 생성한다. 달리 말하면, 동일한 기본 주소를 가진 메모리 요청들을 접한 한 다음 하나의 메모리 요청을 생성한다. PRC는 메모리 요청을 메모리로 보내는 경우 하나씩 증가하고 메모리로부터의 해당 데이터를 받는 경우 하나씩 감소한다. 그러므로 PRC가 0이라는 것은 워프가 요구한 메모리 요청들에 대한 응답을 모두 받았다는 것을 의미한다.

2.2 메모리 요청 합병[16]

메모리 시스템은 메모리 주소 접합 외에도 메모리 요청의 수를 줄이고자 다양한 하드웨어 레벨에서의 메모리 요청 합병을 시도한다. 메모리 요청 합병에는 내부 코어 합병(intra-core merging)과 상호 코어 합병(inter-core merging)이 있다. 내부 코어 합병은 스트리밍 멀티프로세서 내의 GPU 코어에서 발생된 메모리 요청들을 합병하며, 상호 코어 합병은 GPU 내의 스트리밍 멀티프로세서에서 발생된 메모리 요청들을 합병한다.

[그림 4]는 내부 코어 합병과 상호 코어 합병의 예를 보여주고 있다. [그림 4](A)에서 보이듯이, 내부 코어 합병을 위해 각각의 스트리밍 멀티프로세서는 메모리 요구 큐(memory request queue)를 가지고 있다. 메모리 요구 큐에 첫 번째 메모리 A(ADDR. A)가 대기 한 상태에서 GPU 코어1(GPU Core#1)가 요청한 메모리 B(ADDR. B)와 GPU 코어2(GPU Core#2)가 요청한 메모리 C(ADDR. C)가 순차적으로 메모리 요구 큐에 대기한다. 그 이후 GPU 코어3(GPU Core#3)가 메모리 B(ADDR. B)를 요청하는 경우, 메모리 큐에 메모리 B(ADDR. B)가 존재하기 때문에, GPU 코어3(GPU Core#3)이 요청한 메모리 B(ADDR. B)는 메모리 요구 큐 내의 메모리 B(ADDR. B)에 합병된다. 결과적으로 메모리 요청은 총 4번 발생하지만, 내부 코어 합병을 통해 메모리 트랜잭션은 3번만 요청된다.

[그림 4](B)에서 보이듯이, 메모리 컨트롤러는 메모리 요구 버퍼(memory request buffer)를 포함한다. 메모리 컨트롤러 내부의 메모리 요구 버퍼들은 각 스트리밍 멀티프로세서에서 발생된 메모리 요청들을 DRAM 스케쥴링 기법에 따라 응답이 완료되기까지 대기한다. 상호 코어 합병에서는 스트리밍 멀티프로세서에서 발생한 메모리 요청이 메모리 컨트롤러 내부의 메모리 요청 버퍼에 존재한다면, 그 요청을 합병한다. 이와 같은 동작 방식은 내부 코어 합병과 동일하다.

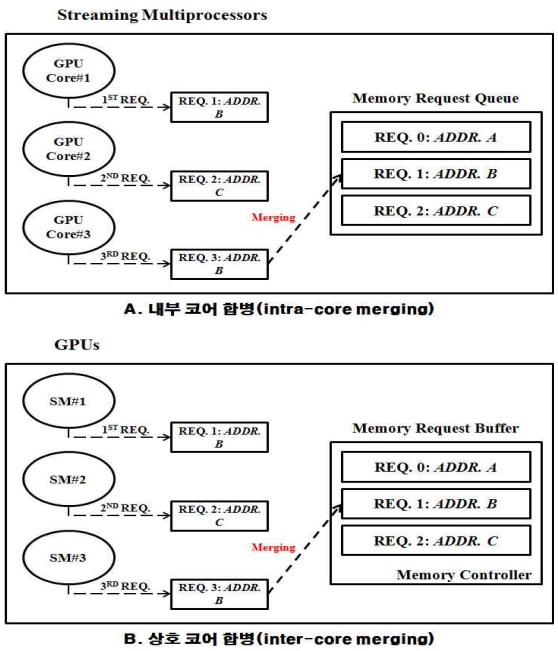

그림 4. 메모리 요청 합병

Ⅲ. 실험 환경

본 논문에서는 메모리 구조에 따른 메모리 시스템의 성능을 평가하기 위하여 GPGPU-SIM을 사용한다[20]. GPGPU-SIM은 Simplescalar[21]를 기반으로 개발된 GPU와 같은 매니코어 프로세서의 성능을 사이클 단위로 정확하게 평가하는 시뮬레이터이다.

표 1. 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
GDDR3 Memory Timing	
tCL=10, tRP =10, tRC=35, tRAS=25, tRCD=12, tRRD=8	

표 2. 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

GPU의 성능 평가에 있어서 GPU 하드웨어 구성 요소와 내부 연결망 구성 요소는 매우 중요하다. 그러므로 본 실험에서는 GPGPU-SIM과 더불어 내부 연결망 구성을 지원할 수 있는 Booksim[22]을 사용한다. Booksim은 다양한 구조의 내부 연결망 네트워크 구조를 지원하고 다른 시뮬레이터와 협동이 가능하다. 본 논문에서는 NVIDIA사의 Quadro FX5800를 참고하여 기본 GPU를 모델링하였으며, 이와 관련된 상세한 GPU 구성 변수들을 [표 1]에서 보여준다. [표 2]는 본 논문에서 사용한 크로스 바(cross-bar) 구조의 상세한 내부 연결망 구성 변수들을 보여준다.

우리는 벤치마크 프로그램으로 현재 가장 널리 사용되는 CUDA 프로그래밍의 SDK들을 사용한다. 하지만 지면상의 한계로 인하여, 메모리 구조에 따른 메모리 시스템의 효율성을 잘 보여주는 7개의 프로그램들(Breadth-First Search, Black‐Scholes, 3D Laplace Solver, MUMmerGPU, Neural Network, N-Queens Solver, Ray Tracing)을 선택하여 결과를 보여준다. 본 논문에서 수행한 벤치마크 프로그램에 대한 설명은 아래와 같다[23].

Breadth First Search: Harish와 Narayanan에 의해 개발된 그래픽 알고리즘으로써, 이 벤치마크 프로그램은 하나의 그래프를 너비 우선 알고리즘으로 검색을 수행한다.

BlackScholes: BlackScholes 벤치마크 프로그램은 Black‐Scholes‐Merton 모델이라고도 불리우며, 파생상품 투자를 포함한 금융 시장의 수학적 모델이다.

3D Laplace Solver: Laplace는 높은 병렬성의 재정 응용 프로그램으로, 메모리 주소 접합을 하기위해서 프로그램 작성의 높은 주의가 필요하다.

MUMmerGPU: MUMmerGPU는 표준 DNA 뉴클레오티드들(A,C,T,G)의 쌍을 병렬적으로 검색하는 프로그램이다.

Neural Network: 이 벤치마크 프로그램은 손으로 쓴 숫자를 인식하기 위한 나선형의 컴퓨터 신경 네트워크를 사용한다.

N-Queens Solver: N-Queens Solver는 NxN 체스 판에서 퀸이 다른 유닛에 잡히지 않도록 퀸을 위치시키는 고전적인 퀴즈 형태를 구현한 프로그램이다.

Ray Tracing: 실사와 근접한 렌더링 그래픽스의 한 방법인 ray tracing을 구현해 놓은 벤치마크 프로그램으로, 각 렌더링 픽셀은 하나의 스레드에 할당되어 수행된다.

Ⅳ. 실험 결과

본 논문에서는 데이터 캐싱에 대한 영향을 평가하기

위해서, 캐쉬를 사용하지 않는 경우와 L1 캐쉬만을 사용하는 경우, 그리고 L2 캐쉬까지 사용하는 경우에 대한 실험을 수행한다. 뿐만 아니라 계층적 캐쉬 구조에 따른 특징을 상세하게 분석하기 위하여, 각 계층의 캐쉬 크기를 다양하게 하여 실험을 수행한다.

표 3. 계층적 캐쉬 구조 변수

L1 캐쉬 구성 요소				
	세트 수	블록 크기	웨이 수	교체 정책
NO L1	없음			
8KB L1	32	64bytes	4	LRU
16KB L1	64	64bytes	4	LRU
32KB L1	128	64bytes	4	LRU
64KB L1	256	64bytes	4	LRU
L2 캐쉬 구성 요소				
	세트 수	블록 크기	웨이 수	교체 정책
NO L2	32KB L1; 없음			
128KB L2	256	64bytes	8	LRU
256KB L2	512	64bytes	8	LRU
512KB L2	1024	64bytes	8	LRU

일반적으로 캐쉬의 크기는 세트 수와 캐쉬 내 블록의 크기, 그리고 웨이 수에 따라 증가 또는 감소한다. 본 논문에서는 데이터 캐싱에 따른 성능을 분석하고자 하기 때문에, 캐쉬의 크기 변화에 따른 성능을 평가한다. 그러므로 지역성에 크게 영향을 받는 캐쉬 내 블록의 크기나 충돌 미스와 직접적인 연관이 있는 웨이 수는 고정하고 세트 수만을 변화시킨다. 뿐만 아니라, 본 논문에서는 교체 정책을 포함하여 캐쉬 성능에 큰 영향을 미치는 기타 모든 요소들을 동일하게 설정한다. 계층적 캐쉬 구조에 대한 상세한 내용은 [표 3]에서 보여주고 있다.

1. 계층적 캐쉬 구조에 따른 성능 변화

각 벤치마크 별로 다양한 계층적 캐쉬 구조에 따른 GPU의 성능 변화를 평가하기 위하여, 성능 평가의 기준으로 IPC(Instruction per Cycle)를 측정한다. 본 장에서는, 실험에 사용된 벤치마크들을 Breadth-First Search는 BFS, Black - Scholes는 BLK, 3D Laplace Solver는 LPS, MUMmerGPU는 MUM, Neural Network는 NN, N-Queens Solver는 NQU, Ray Tracing은 RAY로 각각 표기한다.

L1 캐쉬의 구성 요소에 따른 성능 변화 양상은 [그림 5]에서 보이는 바와 같다. 그림에서 NO L1은 L1 캐쉬가 없는 GPU 기본 구조를 나타내며, 8KB L1, 16KB L1, 32KB L1, 64KB L1 등은 [표 3]에서 보이는 캐쉬 구조를 나타낸다. 각 벤치마크 프로그램마다 성능의 절대 값의 차이가 너무 크기 때문에, 가독성을 높이기 위하여 NO L1을 기준으로 정규화하여 나타낸다. 실험 결과, L1 캐쉬가 없는 경우와 비교하여 L1 캐쉬를 추가한 경우, BFS, MUM, NN은 성능이 향상된 반면에 BLK, LPS, NQU, RAY의 성능은 감소된 것을 볼 수 있다. 가장 높은 성능 향상을 보이는 NN의 경우에는 8KB L1은 약 115%, 16KB L1은 약 144%, 32KB L1은 약 148%의 성능향상을 보여준다. 가장 큰 성능 저하를 보이는 RAY의 경우, 8KB L1은 약 8.1%, 16KB L1은 약 4.5%, 32KB L1은 약 3.1%의 성능이 저하된다. 분석 결과에 따르면, 성능이 저하되는 이유는 데이터의 쓰기 작업이 많은 벤치마크 프로그램의 경우(특히, RAY)에는 데이터 일관성을 유지하기 위하여 메모리 트랜잭션이 급격하게 증가하기 때문이다. 다시 말하면, 데이터 캐싱으로 인해 얻을 수 있는 이익보다 데이터 일관성 유지를 위해 소모되는 비용이 큰 경우 성능이 오히려 저하될 수 있다는 것이다. 이와는 달리, L1 캐쉬가 추가된 경우만을 비교해 보았을 경우에는 모든 벤치마크 프로그램에서 L1 캐쉬의 크기가 증가함에 따라 성능이 향상되는 것을 볼 수 있다.

[그림 6]은 32KB의 L1 캐쉬 하위에 다양한 L2 캐쉬를 추가한 경우의 성능을 보여준다. 앞서 설명한 바와 같이, 각 벤치마크 프로그램마다 성능의 절대 값이 너무 상이하기 때문에 NO L2를 기준으로 정규화한 값으로 성능을 표현한다. 그림에서 NO L2은 32KB의 L1 캐쉬만을 가진 GPU 구조를 나타내며, 128KB L2, 256KB L2, 512KB L2 등은 [표 3]에서 보이는 L2 캐쉬 구조로 구성된 캐쉬를 포함한 GPU 구조를 나타낸다. 실험 결과, NO L2에 비해 128KB L2, 256KB L2, 512KB L2는 MUM(128KB L2: 약 5.06%, 256KB L2: 약 20.73%, 512KB L2: 약 29.61%)을 제외하고는 성능이 거의 변화

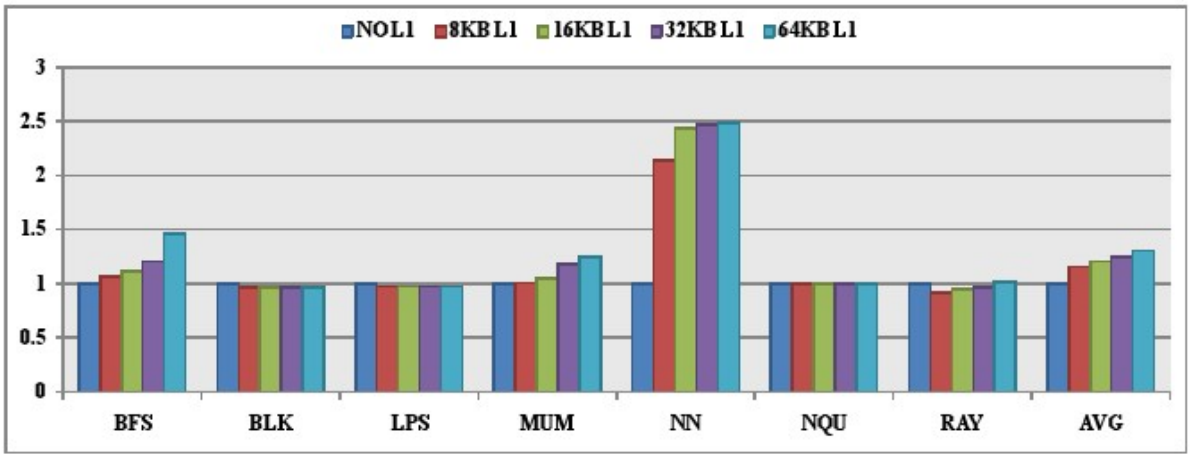

그림 5. L1 캐쉬 추가에 따른 GPU의 성능 변화

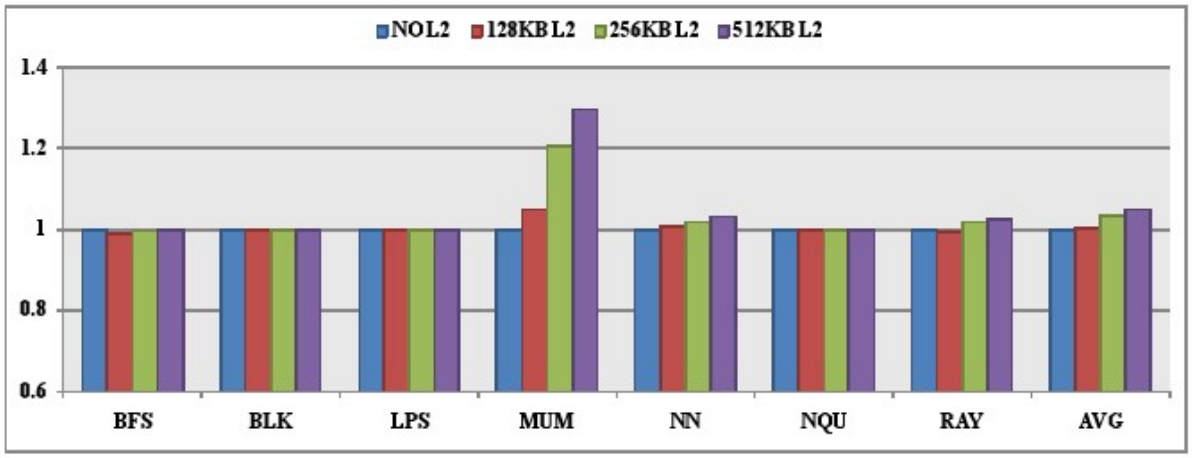

그림 6. L2 캐쉬 추가에 따른 GPU의 성능 변화

가 없다. 특히, BFS의 경우에는 L2 캐쉬를 추가하는 경우 성능이 오히려 감소(128KB L2: 약 0.92%, 256KB L2: 약 0.24% 512KB L2: 약 0.14%)하는 것을 볼 수 있다. 분석 결과에 따르면, BFS를 수행하는 경우 L2 캐쉬에서 상당한 미스(miss)가 발생하여 평균 메모리 접근 지연 시간을 증가시킨 것으로 분석된다. 그리고 MUM의 경우 L1 캐쉬 추가에 비해 L2 캐쉬를 추가할 때 성능이 상대적으로 많이 향상되는 것을 볼 수 있다. 그 이유는 MUM이 필요로 하는 데이터 크기가 크기 때문에 작은 L1 캐쉬 메모리에서는 미스가 상당히 많이 발생하여 평균 메모리 접근 지연 시간을 증가시킨다. 하지만 L2 캐쉬를 추가하면, L1 캐쉬에서 발생된 미스로 인해 발생한 메모리 접근을 상당히 줄여 성능을 향상시킨 것으로 분석된다.

2. 메모리 요청 합병에 따른 성능 변화

본 장에서는 2.2.2에서 기술한 메모리 요청 합병의 두 기법인 내부 코어 합병과 상호 코어 합병에 따른 GPU의 성능 변화를 분석하고자 한다. [그림 7]은 내부 코어 합병 기술만이 적용된 경우 대비 두 합병 기술이 모두 적용된 경우의 성능을 보여준다. 그림에서 Intra는 내부 코어 합병 기술만 적용된 경우를 나타내며, Inter-Intra는 두 합병 기술이 모두 적용된 경우를 나타낸다.

그림에서 보이듯이, 내부 코어 합병 기술만 적용된 경우에 비해 두 합병 기술이 모두 적용된 경우 성능이 평균 8.15% 향상되었다. 그리고 BLK, NQU, RAY를 제외하고는 모든 벤치마크 프로그램에서 성능이 향상됨을 볼 수 있다. BFS는 약 4.41%, LPS는 약 8.82%,

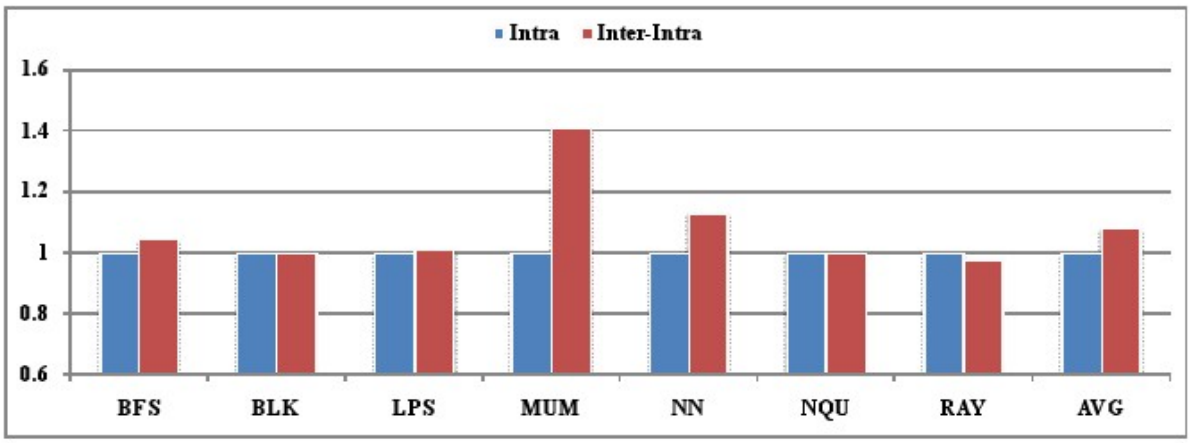

그림 7. 메모리 병합 기술에 따른 GPU의 성능 변화

MUM은 약 40.88%, 그리고 NN은 약 12.94%의 성능이 향상되었다. MUM과 같이 높은 지역성을 가진 벤치마크 프로그램들에서는 동일한 메모리를 요청하기 때문에 상호 코어 합병 기술을 사용하는 경우 메모리 요청 합병이 자주 발생하게 되므로 성능이 크게 향상된다.

BLK, NQU는 성능에 변화가 없는 반면에, RAY는 약 2.05%의 성능 저하가 발생한다. RAY 벤치마크 프로그램은 워크로드 분배가 균등하게 이루어지지 못하여 내부 코어 합병 기술로 인해 요구된 메모리의 응답시간이 오히려 늦어짐에 따라 성능이 저하되는 경우이다. CUDA SDK의 모든 실험 결과에 따르면, 대부분의 벤치마크 프로그램들은 BLK와 NQU의 경우와 유사하게 성능의 변화가 없거나 또는 거의 없다. 그 이유는 CUDA SDK의 벤치마크 프로그램들의 경우, 최적화된 지역성을 가지도록 프로그래밍 되어있기 때문에 내부 코어 합병으로 인한 이득을 얻기 힘들기 때문이다. 지역성 최적화는 모든 스레드들간의 높은 지역성을 의미하는 것이 아닌, 하나의 스트리밍 멀티프로세서 내부의 스레드들이 생성한 메모리 요청들의 높은 지역성을 의미한다.

표 4. GPU 성능에 미치는 주요 요소들의 영향

프로그램 특성		계층적 캐쉬 구조의 수준	
		높음	낮음
프로그램의 지역성	높음	√	
	낮음	√	
지역성 최적화	높음		√
	낮음	√	
캐쉬 일관성 요청	다수		√
	소수	√	

[표 4]는 계층적 캐쉬 구조와 벤치마크 프로그램의 특성이 GPU의 성능에 미치는 영향을 요약하여 보여주고 있다. 표에서 계층적 캐쉬 구조의 수준의 높음은 다중 레벨 캐쉬 구조를 의미하며, 낮음은 단일 레벨 캐쉬 구조를 나타낸다. 표에서 보이듯이, 프로그램의 지역성과는 무관하게 계층적 캐쉬 구조의 수준이 높을수록 성능이 향상된다. 특히, 프로그램의 지역성이 낮을수록 성능 향상의 이익을 크게 볼 수 있다. 지역성이 최적화되어 있으면, 하위 수준의 캐쉬로의 접근이 적을 뿐만 아니라, 적중률(hit rate)이 낮기 때문에 계층적 캐쉬 구조로 인한 성능 향상을 기대하기 어렵다. 뿐만 아니라, 앞서 설명한 바와 같이 캐쉬 일관성 요청이 많을수록 일관성 유지를 위한 오버헤드가 커지기 때문에 낮은 수준의 계층적 캐쉬 구조가 적합하다.

V. 결 론

본 논문에서는 다양한 GPU의 메모리 구조에 따른 성능을 평가 및 분석 하고자 메모리 요청 병합 기법과 계층적 메모리 구조에 대한 실험을 수행하였다. 실험 결과, 메모리 요청 병합 기법에 따른 성능 변화는 크게 나타나지는 않았으나 이는 메모리 요청 병합 기법 자체의 낮은 효율성이 아니라 실험에 사용된 벤치마크 프로그램의 최적화 때문으로 분석된다. 달리 말하면, 지역성이 최적화되지 않는 프로그램을 수행하는 경우에는 메모

리 요청 병합 기법이 GPU 성능을 크게 증가시킬 수 있다는 것을 나타낸다. 그리고 L1 캐쉬와 L2 캐쉬의 추가에 따른 분석 결과에 따르면, 기본적으로 캐쉬의 용량이 커질수록, 다중 레벨의 계층적 캐쉬 구조를 가질수록 성능이 향상됨을 볼 수 있다. 하지만, 계층적 캐쉬 구조의 경우 데이터 일관성을 유지하기 위하여 메모리 트랜잭션이 증가되어 오히려 성능이 저하되기도 한다. 뿐만 아니라, 많은 레벨의 계층적 캐쉬 구조를 사용함에 따라 메모리 요청에 따른 평균 메모리 접근 지연 시간을 증가시켜 성능을 저하시키기도 하는 것을 실험을 통해 확인 할 수 있었다. 그러므로 데이터 일관성 유지를 위한 메모리 트랜잭션을 많이 발생시키는 쓰기 작업과 데이터 공유 연산이 많이 발생하는 응용프로그램의 경우에는 낮은 수준의 계층적 캐쉬 구조를 사용하는 메모리 구조가 적합하다. 우리는 향후 본 논문의 실험 결과를 기반으로 GPU에서 적합한 메모리 구조를 개발하고자 데이터 일관성 문제와 평균 메모리 접근 지연 시간을 감소시킬 수 있는 연구를 진행하고자 한다.

참 고 문 헌

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: the end of the road for conventional microarchitectures," In Proceedings of the 27th International Symposium on Computer Architecture, pp.248-259, 2000.

[2] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-Chip Multiprocessor," In Proceedings of 7th Conference on Architectural Support for Programming Languages and Operating Systems, pp.2-11, 1996.

[3] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger, "Dark Silicon and the End of Multicore Scaling," In Proceedings of International Symposium on Computer Architecture, pp.365-376, 2011.

[4] http://www.isuppli.com/

[5] Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," In Proceedings of 31th Annual Conference on Computer Graphics, pp.777-786, 2004.

[6] H. J. Choi and C. H. Kim, "Performance Evaluation of the GPU Architecture Executing Parallel Applications," Journal of the Korea Contents Association, Vol.12, No.5. pp.10-21, 2012.

[7] H. J. Choi and C. H. Kim, "Analysis of Impact of Correlation Between Hardware Configuration and Branch Handling Methods Executing General Purpose Applications," Journal of the Korea Contents Association, Vol.13, No.3. pp.9-21, 2013.

[8] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," Euro-graphics 2005, State of the Art Reports, pp.21-51, 2005.

[9] http://www.gpgpu.org

[10] http://www.khronos.org/opencl/

[11] http://www.amd.com/stream

[12] http://developer.nvidia.com/object/cuda_3_1_downloads.html

[13] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," In Proceedings of 37th International Symposium on Computer Architecture, pp.235-246, 2010.

[14] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," In

Proceedings of 40th Microarchitecture, pp.407-420, 2007.

[15] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch : Enabling Energy Optimizations in GPGPUs," In Proceedings of the 27th International Symposium on Computer Architecture, pp.487-498, 2013.

[16] N. B. Lakshminarayana and H. S. Kim, "Effect of Instruction Fetch and Memory Scheduling on GPU Performance," Workshop on Language, Compiler, and Architecture Support for GPGPU(in conjunction with HPCA/PPoPP 2010), 2010.

[17] http://www.nvidia.com/object/product_quadro_fx_5800_us.html

[18] W. W. L. Fung and T. M. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," In Proceedings of the 17th International Symposium on High Performance Computer Architecture, pp.25-36, 2011.

[19] E. Lindholm, J. Nickolls, S.Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE MICRO, Vol.28, No.2, pp.39-55, 2008.

[20] 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 9th International Symposium on Performance Analysis of Systems and Software, pp.163-174, 2009.

[21] D. C. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," Computer Architecture News, Vol.25, No.3, pp.13-25, 1997.

[22] http://nocs.stanford.edu/booksim.html

[23] http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html

저 자 소 개

최 홍 준(Hongjun Choi) 정회원

- 2009년 : 전남대학교 전자컴퓨터공학부 공학사
- 2011년 : 전남대학교 전자컴퓨터공학과 공학석사
- 2011년 ~ 현재 : 전남대학교 전자컴퓨터공학과 박사과정

<관심분야> : 컴퓨팅 시스템, 병렬처리, 컴퓨터구조

김 종 면(JongMyon Kim) 정회원

- 1995년 : 명지대학교 전기공학사
- 2000년 : University of Florida ECE 석사
- 2005년 : Georgia Institute of Technology ECE 박사
- 2005년 ~ 2007년 : 삼성종합기술원 전임연구원
- 2007년 ~ 현재 : 울산대학교 전기공학부부 교수

<관심분야> : 임베디드 SoC, 컴퓨터 구조, 프로세서 설계, 병렬처리

김 철 홍(Cheolhong Kim) 정회원

- 1998년 : 서울대학교 컴퓨터공학사
- 2000년 : 서울대학교 컴퓨터공학부 공학석사
- 2006년 : 서울대학교 전기컴퓨터공학부 공학박사
- 2005년 ~ 2007년 : 삼성전자 반도체총괄 SYS.LSI 사업부 책임 연구원
- 2007년 ~ 현재 : 전남대학교 전자컴퓨터공학부 교수

<관심분야> : 임베디드시스템, 컴퓨터구조, SoC 설계, 저전력 설계