공학석사 학위논문

Xeon Phi를 활용한 병렬 프로그래밍

- 하드웨어·소프트웨어적 특징과 병렬화 모델 -

2015 년 5 월

서울대학교 대학원

전기·컴퓨터공학부

구 동 훈

Xeon Phi를 활용한 병렬 프로그래밍

- 하드웨어·소프트웨어적 특징과 병렬화 모델 -

지도교수 윤 성 로

이 논문을 공학석사 학위논문으로 제출함
2015 년 5 월

서울대학교 대학원
전기·컴퓨터공학부
구 동 훈

구동훈의 석사 학위논문을 인준함
2015 년 5 월

위 원 장 _____ (인)

부위원장 _____ (인)

위 원 _____ (인)

국문초록

본 논문은 Intel의 병렬연산을 위한 보조 프로세싱 장치(Co-Processor)인 Xeon Phi를 이용한 가속화 알고리즘 설계 및 구현상 주요 이슈와 Xeon Phi로 구현 가능한 병렬화 모델을 제시한다. 효과적인 병렬프로그래밍을 위해서는 사용하고자하는 가속기의 구조적인 특징과 프로그래밍 모델에 대하여 개발자가 구체적인 이해를 할 필요가 있다. 그리고 가속기로 구현이 가능한 병렬화 모델을 파악하여 대상 어플리케이션에 효과적인 적용을 하여야 한다.

따라서 본 논문은 먼저 효과적인 프로그래밍을 위해 알아두어야 할 Xeon Phi의 구조적 특징, 프로그래밍 모델을 설명한다. 더불어 프로그래머 관점에서 GPGPU (General-Purpose computing on Graphics Processing Units)와의 차이도 언급하여 이미 GPGPU에 익숙해진 개발자도 Xeon Phi에 쉽게 적응할 수 있도록 하였다. 다음으로 Xeon Phi로 구현할 수 있는 병렬화 모델을 정의하고 구체적인 사례와 함께 제시한다. 기존에 멀티코어 CPU (Central Processing Unit)를 위해 구현이 되었던 Strassen-Winograd 알고리즘과 SHRINK (SHaRed-memory SLINK)알고리즘에 Xeon Phi를 적용하여 구현한 뒤 실험을 통해 가속화 효과를 보였고, 이를 통해 Xeon Phi는 GPGPU와 동일한 형태의 병렬화 모델을 구현할

수 있을 뿐 아니라 GPGPU로는 효과적으로 구현하기 힘든 병렬화 모델도 구현할 수 있음을 확인하였다.

이러한 구체적인 사례와 함께 본 논문에 소개되지 않은 어플리케이션에도 가속화를 위해 적용할 수 있는 두 가지 병렬화 모델을 정의해 개발자가 주어진 어플리케이션의 성능 향상을 위해 효과적으로 Xeon Phi를 적용할 수 있도록 하였다.

주요어 : Xeon Phi, 가속기, Parallel computing, Parallel architecture, GPGPU, Strassen, Hierarchical Clustering
학 번 : 2013-20744

목 차

표 목 차

그 림 목 차

제 1 장 서 론

하드웨어의 클럭 주파수 향상에 따라 큰 수고를 들이지 않고도 어플리케이션의 성능 향상을 얻을 수 있었던 “Free Lunch”의 시대가 끝나고 (Sutter & Herb, 2005) 멀티코어 CPU, GPU (Graphics Processing Unit), FPGA (Field Programmable Gate Arrays) 등을 활용한 병렬 프로그래밍이 어플리케이션의 성능 향상을 위한 해결책으로 떠올랐다. 이러한 흐름에 맞추어 주요한 프로세서 제조사 중 하나인 인텔은 새로운 Many Integrated Core (MIC) 구조를 지닌 Xeon Phi 코프로세서를 출시하였다 (Chrysos et al., 2012).

Xeon Phi는 기존의 CPU에 비해 증가한 코어의 개수, 확장된 벡터 레지스터 등 병렬프로그래밍에 적합한 구조를 지니고 있다 (Chrysos & George, 2014). 하지만 GPGPU, FPGA 등과 마찬가지로, 그 성능을 효과적으로 이끌어 내기 위해서는 사용자가 기본적인 구조와 프로그래밍 모델 숙지와 더불어 어떠한 방식으로 병렬화에 적용할 수 있는 지를 반드시 알아야할 필요가 있다. 따라서 본 논문은 Xeon Phi를 활용한 효과적인 프로그래밍을 위해 이러한 사항들을 담고 있으며 GPGPU에 비해 아직 널리 사용되고 있지 않는 Xeon Phi에 대한 이해를 돕고자하는 바람을 담고 있다. 논문은 아래와 같이 구성되어 있다.

2장에서는 Xeon Phi에 관한 기본적인 설명을 한다. Xeon Phi의 등장과 더불어 프로그래머의 입장에서 효과적인 개발을 위해 알아두어야 할

구조적인 특징, 프로그래밍 모델 등을 언급한다. 또한 프로그래머 관점에서 GPGPU와의 비교를 통해 두 장치를 이용한 어플리케이션 개발 시 차이점을 알 수 있도록 하였다.

3장에서는 Xeon Phi를 활용한 가속화에 대해 실제 사례를 들어 설명을 한다. Simple 모델과 Complex 모델 두 가지 병렬화 모델을 정의하여 설명을 진행하며 병렬 프로그래밍을 적용할 어플리케이션에 따라 Xeon Phi를 다양한 방식으로 사용할 수 있음을 보였다.

이후 4장에서는 2장과 3장의 내용을 종합한 결론을 내린다.

제 2 장 Xeon Phi 소개

제 1 절 개요 및 등장 배경

1980년대 중반부터 2004년에 이르기까지 컴퓨터의 성능을 향상시키기 위한 주요한 방법은 무어의 법칙 (Schaller & Robert R, 1997)에 따라 증가하는 트랜지스터 집적도에 기반하여 프로세서의 클럭 주파수를 증가시키는 것이었다. 하지만 계속된 주파수의 증가에 따른 전력 소비와 열의 생성이 문제가 되기 시작하였다. 그로 인해 코어의 주파수를 증가시키는 것이 아니라 코어의 개수를 늘리는 전략을 택한 멀티코어 아키텍처(Architecture)가 관심을 받기 시작하였고 (Akhter et al., 2006), 이를 활용한 병렬 컴퓨팅이 프로그램의 성능을 향상시키기 위한 필수방안으로 부상하였다 (Michael & J. Quirm, 2004).

그 이후 멀티코어 프로세서가 컴퓨터 아키텍처의 주된 패러다임이 되었으며, 나아가 더욱 향상된 컴퓨팅 성능을 갖추기 위해 기존의 그래픽 카드 (GPU)를 일반적인 연산처리에 사용하는 GPGPU의 개념도 등장하였다. 특히 GPGPU를 활용할 수 있는 플랫폼 중 하나인 CUDA (Compute Unified Device Architecture) (Sanders et al, 2010)는 기존의 RNA 3중 구조의 예측 알고리즘을 멀티코어 CPU와 CUDA를 적용해 가속화한 (Jeon et al., 2013), 다이나믹 프로그래밍을 CUDA를 사용해 병렬화한 (Lee et al., 2012)을 비롯한 다양한 사례에 적용되어 가속화를 위

한 효과적인 수단임이 밝혀졌다.

Xeon Phi는 인텔 Xeon 프로세서 시리즈의 전력 소비를 줄일 방안을 찾는 연구에서부터 시작되었다. 인텔의 연구진은 많은 전력을 소비하지 않는 새로운 아키텍처를 개발하고자 하였다. 더불어 이 새로운 아키텍처는 다양한 산업 분야에서 요구하는 복잡한 연산을 빠르게 처리하기 위해 기존 멀티코어 아키텍처보다 더 향상된 컴퓨팅 성능을 지녀야 했다. 개발 과정에서 새로운 아키텍처는 저 전력을 위하여 스케줄링 방식을 아웃-오브-오더(out-of-order)가 아닌 인-오더(in-order) (Hennessy et al., 2012)방식이 채택되었으며 기존의 프로세서 코어보다 짧은 파이프라인과 낮은 클럭 주파수를 갖추었다. 인-오더 방식을 택하였을 경우 발생할 수 있는 데이터 해저드 (Data Hazard) (Hennessy et al., 2012)는 한 번에 네 개의 스레드가 동시에 스케줄링되는 방식을 택함으로서 방지하였다. 그리고 컴퓨팅 성능을 향상시키기 위해 확장된 하드웨어 멀티스레딩과 벡터 연산 레지스터 (Levinthal et al., 1984)를 갖추었다. 이러한 Many Integrated Core (MIC) 아키텍처 개발을 위한 연구의 첫 번째 성과는 2005년에 GPGPU인 Larrabee를 개발한 것이다. 하지만 Larrabee는 양산되지 못하였고 그 하드웨어 스펙은 Knights Ferry라고 하는 고성능 컴퓨팅 코프로세서 (Coprocessor)를 만들기 위한 프로젝트에 계승되었다 (Rahman & Rezaur, 2013). 여기서 코프로세서란 주 프로세서(호스트 CPU)에서 특정한 연산 작업을 오프로딩 (offloading)하여 그 작업을 빠르게 수행할 수 있도록 디자인된 마이크로프로세서 칩을 말한다. GPGPU와 Xeon Phi 모두 코프로세서에 해당한다.

이후 계속된 개발과정을 통해 새로운 하드웨어에 리눅스 운영체제를

지원할 수 있게 하는 등 소프트웨어 측면의 보완이 이루어졌다. 또한 멀티스레딩이 가능한 어플리케이션이 개발될 수 있도록 Message Passing Interface (MPI)와 OpenMP도 지원할 수 있게 되었다. 더불어 C, C++, Fortran 등의 프로그래밍 언어와 Amplifier XE, Math Kernel Library (MKL) 등 기존 인텔 프로세서에서 사용되었던 개발 툴 및 라이브러리 또한 사용가능하게 되었다. 이렇게 테크니컬 컴퓨팅 분야에서 요구되는 고성능 연산을 위한 하드웨어와, 개발자가 그것을 원활히 사용할 수 있도록 하는 소프트웨어 기반을 갖추게 된 새로운 아키텍처를 Kngiths Corner (KNC)라 명하게 되었으며 이후 Xeon Phi라는 브랜드 명을 갖추게 된다 (Jeffers & Reinders, 2013).

제 2 절 구조적 특징

2015년 4월까지 출시된 인텔 Xeon Phi 코프로세서는 최대 61개의 코어를 지니고 있으며 모든 코어는 그림 1과 같이 링 인터페이스 (Jeffers & Reinders, 2013)로 연결되어있다. 링 인터페이스는 코어 간 통신을 가능하도록 하며 L2캐시를 공유할 수 있도록 해준다.

Xeon Phi는 인텔 Xeon 프로세서 (호스트)와 PCI 익스프레스를 통해 통신할 수 있다. 2015년 4월까지 출시되어있는 Xeon Phi 시리즈의 사양은 표 1과 같다. 표 1은 Xeon Phi의 제품 번호에 따른 보드 TDP (Thermal Design Power), 코어의 개수, 클럭 주파수, 배정밀도 (Double

Precision) 성능과 메모리 대역폭, 메모리 크기를 보여준다.

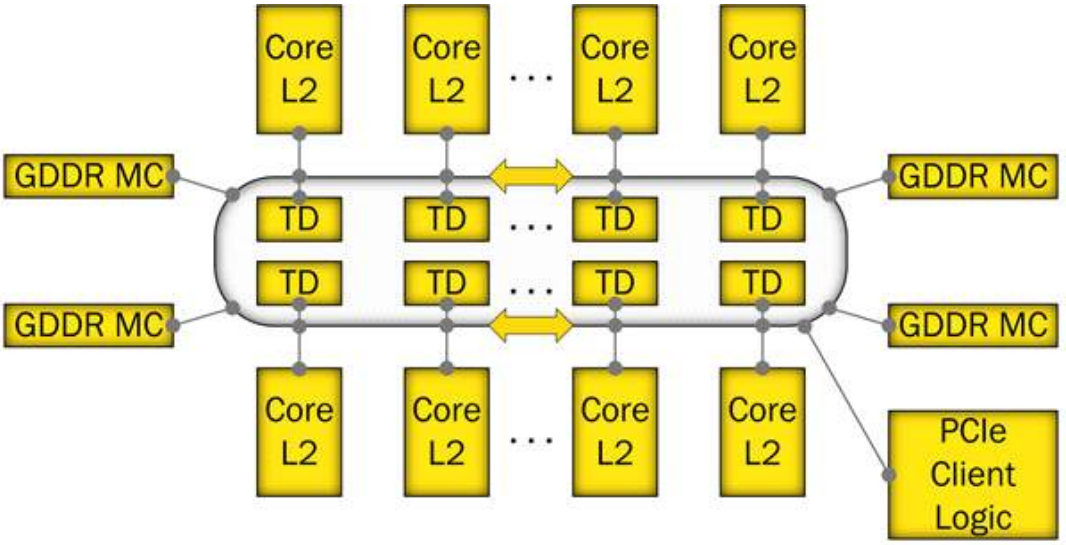


그림 1. 링 인터페이스로 연결된 Xeon Phi의 코어들

표 1. 인텔 Xeon Phi 시리즈 사양 (Jeffers & Reinders, 2013)

제품 번호	보드 TDP (WATTS)	코어 개수	주파수 (GHz)	배정밀도 성능 (GFLOP)	메모리 대역폭 (GB/s)	메모리 크기 (GB)
3120P	300	57	1.1	1003	240	6
3120A	300	57	1.1	1003	240	6
5110P	225	60	1.053	1011	320	8
5120D	245	60	1.053	1011	352	8
7110P	300	61	1.238	1208	352	16
7120X	300	61	1.238	1208	352	16

Xeon Phi에서 효과적인 어플리케이션 개발을 위해 알아두어야 할 MIC 아키텍처의 주요한 특징은 표 2와 같이 정리할 수 있다.

표 2. MIC 아키텍쳐 특징 (Weinberg, V. "Best Practice Guide - Intel Xeon Phi)

항 목	특 징
Core	• 인-오더 방식의 Scalar Unit 보유. • 각 코어는 Scalar Unit과 더불어 Vector Processing Unit 보유. • 코어 당 4개의 스레드 생성 가능. • 라운드로빈 방식의 스레드 스케줄링. • 기존 인텔의 SIMD 명령어 셋 사용 불가 (MME, SSE, AVX 등). • 각 코어는 Xeon Phi만의 512비트 벡터 명령어 셋 지원. • L1 I캐시와 D캐시 보유. • 모든 코어는 Core Ring Interface를 통해 통신.
Vector Processing Unit (VPU)	• 32개의 512비트 레지스터 보유. • 단정밀도 부동소수점 (Single-precision floating point)과 정수(integer)의 경우 한 사이클에 16개의 연산을 동시에 수행 가능. • 배정밀도 부동소수점 (Double-precision floating point)의 경우 한 사이클에 8개의 연산을 수행 가능.
Core Ring Interface (CRI)	• 코어 간 통신을 가능하도록 하며 L2 cache를 공유할 수 있도록 함.
SBOX	• 2세대 PCI-E logical control unit. • Xeon Phi와 외부 디바이스 (호스트 CPU포함) 사이의 연결을 담당.

표 2는 Xeon Phi의 개별 코어와 각 코어가 보유한 VPU (Vector Processing Unit)와 더불어 Xeon Phi 내외부의 통신을 가능하게 하는 CRI (Core Ring Interface)와 SBOX에 대해 설명한다.

캐시 계층구조의 주요한 특징은 표 3과 같다. Xeon Phi는 2차의 캐시 구조를 갖추고 있다. 각 코어는 32KB의 L1 명령어 캐시와 데이터 캐시를 가진다. 또한 모든 코어는 512KB의 L2 캐시를 보유하고 있는데 이 L2캐시는 링 인터페이스를 통해 일관성 (Coherency)을 제공한다. 그로인해 각각의 코어는 다른 코어의 L2 캐시를 사용가능하다. 개발자의 입장에서는 특히 표 3에서 Xeon Phi의 Line Size가 64 바이트라는 사실에 주목할 필요가 있다. 이에 맞춰 데이터가 메모리에 정렬될 수 있도록 해주는 것이 Vectorization의 빈도를 높아지게 하여 전체 성능향상에 도움이 되기 때문이다. 그리고 이러한 데이터 정렬(alignment)은 __declspec(align(64))와 _mm_malloc() 등의 지시어를 통해 가능하다 (Jeffers & Reinders, 2013).

표 3. Xeon Phi의 캐시 계층구조 특징 (Weinberg, V. "Best Practice Guide - Intel Xeon Phi)

파라미터	L1 캐시	L2 캐시
Coherence	Mesi	Mesi
Size	32KB (I) + 32KB (D)	512 KB
Associativity	8-way	8-way
Line Size	64 bytes	64 bytes
Banks	8	8
Access Time	1 cycle	11 cycles
Policy	pseudo LRU	pseudo LRU

제 3 절 프로그래밍 모델

Xeon Phi는 활용의 측면에서 다양한 옵션을 제공한다. 이는 전체 프로그램에서 데이터 병렬성이 있는 부분의 구동을 단순히 오프로딩해서 사용하는 GPGPU와 비교되는 점이다. 다양한 활용 옵션은 성능을 향상시키고자 하는 어플리케이션의 특성에 맞는 해결책을 좀 더 유연하게 제공할 수 있다는 점에서 커다란 장점이 된다. 개발자는 대상이 되는 어플리케이션의 특성과 구현 방식을 고려해 Xeon Phi를 어떻게 활용할 것인지 정할 수 있다. Xeon Phi의 프로그래밍 모델과 그에 해당하는 설명은 그림 2와 같다.

Xeon Phi는 기본적으로 인텔 프로세서의 아키텍처에서 유래하였으므로 기존 인텔 멀티코어 기반의 병렬프로그래밍을 하였을 때 사용하였던 라이브러리나 도구를 그대로 사용할 수 있다

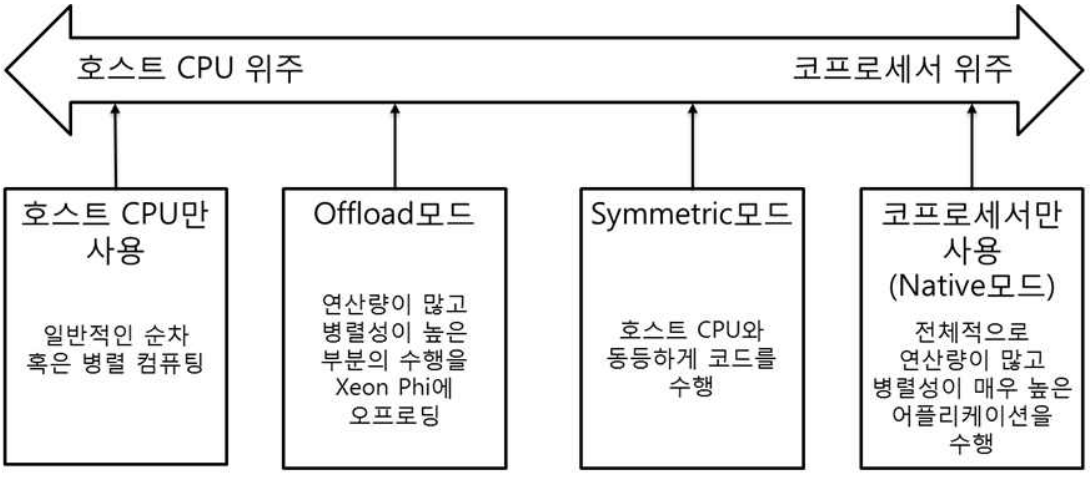


그림 2. Xeon Phi의 프로그래밍 모델 (Jeffers & Reinders, 2013, Newburn et al., 2013)

개발자는 이미 최적화가 잘 되어있는 라이브러리를 활용할 수도 있고 필요에 따라 Pthreads나 C/C++ Vector Classes 같은 하위 단계에서 직접 코드를 작성해 하드웨어의 성능을 더 이끌어낼 수도 있다. 그림 3은 이러한 개발시의 여러 옵션을 보여준다. 스레딩과 벡터연산의 활용측면에서 개발자가 선택 가능한 여러 개발 옵션이 존재한다. 스레딩 옵션은 멀티스레딩을 적용하고자 할 때 활용할 수 있는 수단들이며 벡터 옵션은 벡터 연산을 사용할 때 활용할 수 있는 수단들이다. 스레딩 옵션의 경우 사용 시 상대적인 난이도에 따라 MPI, 인텔 Cilk Plus, OpenMP, Pthreads 순으로 나열할 수 있다. MPI나 인텔 Cilk Plus를 통해 간단하게 Xeon Phi의 250여개에 달하는 스레드를 구동시킬 수 있다. 하지만 이러한 간단한 방법은 세밀한 스레드의 조작을 어렵게 한다. 따라서 보다 정교하게 멀티스레딩을 조작하고 싶다면 하위 단계에서 프로그래밍이 가능한 OpenMP나 Pthreads를 고려해야 한다. 벡터 옵션 또한 인텔 Math Kernel Library를 비롯한 상위 단계의 방법은 간단한 벡터 연산의 적용을 가능하게 해주나 정교한 작업의 수행은 어려우므로 세밀하게 벡터 연산을 적용하고자 한다면 보다 하위 단계의 방법을 적용해야 한다.

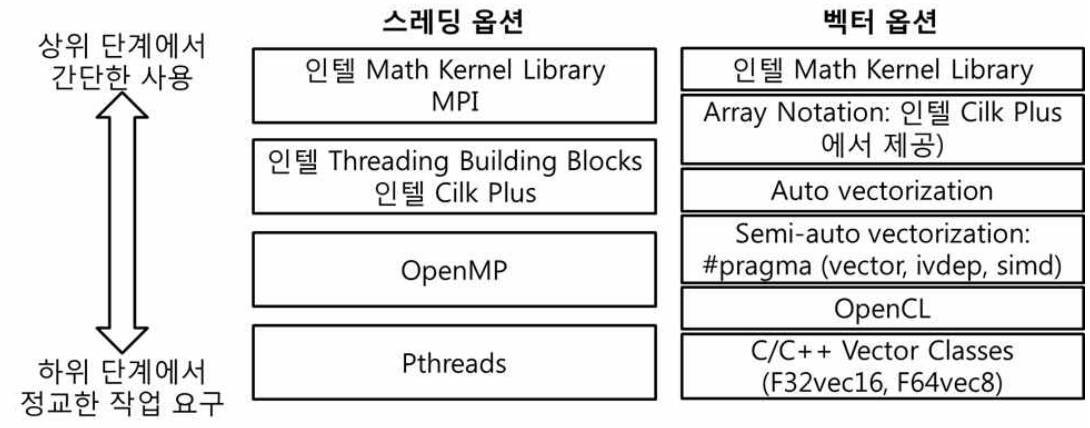


그림 3 Xeon Phi에서 사용가능한 개발 옵션 (Jeffers & Reinders, 2013)

제 4 절 GPGPU의 대비 특징

Xeon Phi와 GPGPU 모두 호스트 CPU에서 복잡한 연산이 있는 코드 일부의 수행을 오프로딩하여 사용하는 코프로세서의 역할을 수행할 수 있다. 또한 많은 스레드를 동시에 구동시킬 수 있는 점과 라이브러리를 통한 프로그래밍이 가능한 점 등의 공통점을 지니고 있다. 하지만 이 둘은 하드웨어 특성 뿐 아니라 프로그래밍 모델과 성능에 있어서도 차이를 보인다. 기존에 둘 사이의 차이에 주목한 (Jeong et al., 2013), (Teodoro et al., 2013), (Jang et al., 2014), (Saule et al., 2014), (Fang et al., 2014)의 연구가 있었다. 하지만 하드웨어 스펙과 프로그래밍 모델, 성능의 차이를 모두 언급해 놓은 사전 연구는 없었으며 본 논문에서는 이에 대해 논의한다. 2013년 초에 출시된 두 모델(Xeon Phi의 경우 5100P, GPGPU의 경우 GTX TITAN)을 비교한다. 기본적인 하드웨어 사양의

차이는 표 4와 같다.

표 4에 나와 있듯이 코어의 수는 GPGPU가 많지만 각 코어의 클럭 주파수는 Xeon Phi가 높다. 그리고 SP TFLOPS (Single Precision Tera FLoating-point Operations Per Second)와 DP TFLOPS (Double Precision Tera FLoating-point Operations Per Second) 같은 이론적인 성능은 GPGPU가 우세하지만 메모리 측면(크기, 대역폭)에서는 Xeon Phi가 더 높은 수치를 보인다. 호스트 CPU와 코프로세서를 연결하는 PCI 익스프레스는 Xeon Phi가 오직 2.0버전만을 사용할 수 있는 데 반해 GPGPU는 3.0버전을 지원하므로 데이터 전송측면에서는 GPGPU가 우수할 것임을 알 수 있다.

표 4. Xeon Phi와 GPGPU의 하드웨어 특성 비교

	Intel Xeon Phi	NVDIA GPU
모델 명	5110P	GTX TITAN
코어 개수	60	2688
코어 클럭 주파수 (MHz)	1053	837
SP TFLOPS	2.02	4.5
DP TFLOPS	1.01	1.3
메모리 크기 (GB)	8	6
메모리 대역폭 (GB/s)	320	288.4
PCI-E 버전	2.0	3.0
가격 (USD)	2,162	2,054
출시일	2013년 1월 28일	2013년 2월 21일

Xeon Phi와 GPGPU는 프로그래밍 관점에서도 차이를 보인다. 제 2장 3절에서 언급했듯이 Xeon Phi는 그림 2의 Symmetric모드나 Native모드

도 지원하는 반면 GPGPU는 기본적으로 오직 Offload모드만을 지원한다. CPU 용 코드와 이와 동일한 기능을 수행할 수 있는 GPGPU 용 코드를 별개로 준비하여 GPGPU에서도 Symmetric모드를 구현할 수는 있으나 이러한 방법은 비효율적이고 CPU의 코드와 동일한 기능의 GPGPU 코드를 구현하는 데에도 한계가 있다.

또한 GPGPU의 스레드는 Warp 단위로 묶여 최소한 하나의 Warp가 같은 명령을 수행할 때 그 성능을 잘 발휘할 수 있게 설계되어있다. 이와 달리 Xeon Phi의 스레드는 이러한 SIMD 단위 개념을 따르지 않고도 멀티스레딩을 성능 저하 없이 활용가능하다. 따라서 많은 스레드를 GPGPU보다 유연하게 사용할 수 있다. 이러한 특성은 제 3장에서 언급할 Xeon Phi가 GPGPU에 비해 보다 다양한 병렬한 모델을 구현할 수 있다는 사실과 관련이 된다.

두 가속기의 프로그래밍 관점에서 대응되는 개념은 표 5와 같다. GPGPU의 경우 가장 널리 사용되는 CUDA의 개념을 적용하였다. 표 5을 통해 기존에 GPGPU (CUDA)만을 사용해왔던 개발자도 GPGPU (CUDA)의 개념과 연관지어 Xeon Phi에 대해 쉽게 이해할 수 있을 것이다.

Xeon Phi와 GPGPU는 표 4에 나와 있듯이 하드웨어 특성 뿐 아니라 표 5와 같이 프로그래밍 관점에서도 차이를 보인다. 또한 Xeon Phi는 그림 2와 같이 GPGPU에 비해 더욱 다양한 프로그래밍 모델을 지원한다. 알고리즘에서 특정 SIMD연산을 오프로딩하여 사용하는 경우라면 Xeon Phi에 비해 GPGPU의 성능이 더 우수할 가능성이 크다. Xeon Phi가 GPGPU의 많은 코어에 대응해 SIMD lane을 갖추고는 있다고는 하

나 GPGPU의 코어 개수가 워낙 많을뿐더러 PCI 익스프레스 지원 또한 Xeon Phi에 비해 우수하기 때문이다. 하지만 단순히 SIMD연산을 적용하기 힘든 경우라면 Xeon Phi의 멀티스레딩을 활용하는 것이 가속화를 위한 훌륭한 방안이 될 것이다. Warp와 같은 스레드 구동의 단위 제약이 없다는 것과 더 다양한 프로그래밍 모델 지원 등에서 Xeon Phi는 GPGPU에 비해 더 넓은 응용 범위와 더 많은 적용의 융통성을 발휘하게 해준다. 본 논문의 제 3장을 통해 이와 같은 사실을 확인할 수 있다.

표 5. Xeon Phi와 GPGPU의 프로그래밍 관점에서의 비교

Xeon Phi	GPGPU (CUDA)
• Phi 코어	• Symmetric Multiprocessor
• SIMD lane	• CUDA 코어
• 하나의 코어에 속한 스레드들	• 스레드 block
• SIMD 연산	• Warp 단위 연산
• Coherent, automatic L2 cache와 hardware prefetching	• Automatic local caching과 manual local caching
• OpenMP • Offload 위해 제공되는 Language Extensions, OpenCL. • 라이브러리 (MKL 등)	• CUDA • OpenCL, OpenACC offloads • 라이브러리 (CUBLAS 등)
• 호스트 CPU와 독립적으로 Native 모드로 구동 가능	• 호스트 CPU 필요

제 3 장 Xeon Phi를 통한 가속화

이번 장에서는 Xeon Phi를 통해 어플리케이션을 가속화하는 방법을 실제 사례를 들어 설명한다. Fynn's Taxonomy에 따른 멀티프로그래밍 관점의 컴퓨터 구조는 표 6과 같이 분류될 수 있다 (Duncan & Ralph, 1990). MIMD는 하나의 program이 결국 여러 instruction으로 이루어진다는 점에서 SPMD와 MPMD를 포함할 수 있다.

표 6. Fynn's Taxonomy에 따른 컴퓨터 구조 분류

	Single instruction	Multiple instruction	Single program	Multiple program
Single data	SISD	MISD		
Multiple data	SIMD	MIMD	SPMD	MPMD

Xeon Phi는 표 6의 MIMD 구조를 지니고 있다 (Reinders & Jeffers, 2014). Xeon Phi는 각 스레드가 어떻게 동작하고 통신할 것인지에 대한 제약이 없기 때문에 어플리케이션의 특성에 따라 다양한 병렬화 방법을 제시할 수 있다. 반면 GPGPU의 경우 SPMD의 구조를 지니므로 Xeon Phi에 비해 응용 범위는 좁을 수 있다 (Reinders & Jeffers, 2014).

Xeon Phi를 활용한 가속화 설명의 편의를 위해 병렬화 모델에 대해 표 7과 같이 정의를 할 수 있다. Xeon Phi는 제 2장 3절에서 언급했듯이 다양한 프로그래밍 모델을 제공한다. 본 논문은 Xeon Phi를 이용할

때 가장 널리 사용되는 방식이기도 하면서 GPGPU와 직접적으로 비교할 수 있는 Offload모드로 범위를 한정해 설명을 진행한다.

논문의 이어지는 부분에서는 표 7에서 정의한 두 가지 병렬화 모델의 실제 적용사례를 소개한다. 개발자는 논문에서 소개하는 두 병렬화 모델의 구현방식을 수행시간 단축이 필요한 어플리케이션에 적용해 가속화 효과를 얻을 수 있을 것이다.

표 7. 병렬화 모델의 분류

병렬화 모델	설명
Simple 모델	• 코프로세서가 수행하는 대상 코드가 data-level 혹은 loop-level 병렬성을 보유. • GPGPU와 같이 동작 시 스레드의 기본 단위(Warp 등)가 존재할 경우 비효율성을 초래할 수 있는 조건문 등을 보유하지 않음.
Complex 모델	• 코프로세서가 수행하는 대상 코드가 task-level 병렬성을 보유. • Simple 모델에서 언급한 스레드의 기본 단위가 존재할 경우 비효율성을 가져올 수 있는 조건문 등을 포함.

제 1 절 Simple 모델 가속화

Xeon Phi와 GPGPU를 비롯한 코프로세서에게 프로그램 내의 계산 집약적이고 데이터 병렬성이 풍부한 영역의 수행을 오프로딩해서 전체 프로그램의 성능을 향상 시킬 수 있다. 이때 오프로딩하는 영역에서 행렬의 합이나 곱을 구하는 등 많은 스레드가 동시에 단순한 명령을 수행할 때 Simple 모델 가속화를 적용할 수 있다. GPGPU는 구조적으로 (Sanders et al, 2010) 이러한 방식의 가속화에 특화되었으며 그 사례 또한 (Harish et al., 2007), (Suda & Reiji, 2010), (Thibault et al., 2009), (Hong-tao et al., 2009) 등으로 매우 많다. Xeon Phi 역시 240여개의 스레드와 확장된 벡터 연산을 활용해 Simple 모델 가속화에 활용할 수 있다. (Caballero et al., 2013), (Pennycook et al., 2013), (Fang et al., 2013), (Wu et al., 2013)가 그 예이며 특히 (Wu et al., 2013)는 멀티코어 CPU로 구현된 K-means 알고리즘에서 여러 데이터 간의 거리를 구하는 부분을 Xeon Phi를 통해 가속화한 예로써 Xeon Phi의 넓은 벡터 라인을 활용하기 위해 데이터가 메모리에 배치되는 방식(data layout)을 바꾸는 등의 기법을 적용하였다. 이는 Xeon Phi의 성능을 이끌어내기 위해 하드웨어의 특성을 고려한 최적화 기법을 도입한 사례로써 참고할 만하다.

Strassen-Winograd 알고리즘은 기존 행렬 곱셈의 시간복잡도를 낮춘 알고리즘으로 구현 과정에서 Simple 모델 가속화가 가능한 구간을 갖추고 있다. 이미 (구동훈 외., 2013 a)에서 Strassen 알고리즘에 멀티코어 CPU를 적용해 싱글코어 CPU에 비해 성능향상을 이루었으며 (나병국

외., 2014)에서는 Strassen-Winograd 알고리즘의 구현에 멀티코어 CPU와 GPGPU를 모두 활용하여 더욱 가속화를 하였다. 본 논문에서는 Simple 모델 가속화의 사례로 이 Strassen-Winograd 알고리즘을 택하여 그 효과를 살펴보았다.

1. Strassen-Winograd 알고리즘 소개

기존 행렬곱셈은 $O(n^3)$으로 시간복잡도가 매우 크다. 이를 해결하기 위해 Strassen 알고리즘은 시간복잡도를 높이는데 많은 영향을 미치는 곱셈 연산의 수를 줄이는 방법을 적용하였다 (Strassen & Volker, 1969). 이를 통해 시간복잡도를 $O(n^{2.807})$로 낮추었다. 나아가 Winograd 변형 알고리즘은 Strassen 알고리즘에서 덧셈과 뺄셈 연산의 횟수를 줄여 시간복잡도는 동일하지만 실제 수행시간은 더욱 단축되도록 하였다.

Strassen-Winograd (Winograd & Shmuel., 1968) 알고리즘은 기본적으로 2의 배수를 만족하는 M개의 행과 N개의 열을 지니는 행렬 A와 B의 곱셈 결과를 구하는 알고리즘이다. 가장 먼저 행렬 A와 B 각각을 같은 크기의 정사각 행렬 4개로 나눈다. <식 1>에서 행렬 A와 B는 각기 $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, $A_{2,2}$ 그리고 $B_{1,1}$, $B_{1,2}$, $B_{2,1}$, $B_{2,2}$와 같이 4개의 정사각 행렬로 나누어짐을 확인할 수 있다.

$$\begin{bmatrix} A_{1,1} A_{1,2} \\ A_{2,1} A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} B_{1,2} \\ B_{2,1} B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} C_{1,2} \\ C_{2,1} C_{2,2} \end{bmatrix} \quad <식\ 1>$$

$O(n^3)$으로 높은 시간복잡도를 보이는 기존 행렬곱셈은 <식 2>와 같이 8번의 곱셈과 4번의 덧셈이 필요하다.

$$
\begin{aligned}
C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{aligned}
\qquad \text{<식 2>}
$$

반면 Strassen-Winograd 알고리즘은 <식 3>과 같이 <식 1>에서 구한 정사각 행렬들로 곱셈과 덧셈, 뺄셈을 수행해 M_1, M_2,.., M_7의 7개의 행렬을 정의하고, <식 4>와 같이 덧셈과 뺄셈을 수행해 최종적으로 구하고자 하는 행렬 C를 이루는 부분 행렬들을 구할 수 있다. 이 과정에서 총 7번의 곱셈과 15번의 덧셈과 뺄셈을 진행하게 된다. <식 3>에서 모든 M_k (k = 1, 2, .., 7)행렬은 정사각 행렬 간의 곱으로 이루어지기 때문에 M_k를 구하는 과정에서 다시 Strassen-Winograd 방식이 적용된다. 재귀적으로 진행되는 과정 중에 만약 행렬의 크기가 미리 정해놓은 임계값(Threshold)보다 같거나 작아지게 되면 일반적인 행렬곱셈을 적용하게 된다.

$$
\begin{aligned}
M_1 &:= (A_{2,1} + A_{2,2} - A_{1,1})(B_{2,2} - B_{1,2} + B_{1,1}) \\
M_2 &:= A_{1,1}B_{1,1} \\
M_3 &:= A_{1,2}B_{2,1} \\
M_4 &:= (A_{1,1} - A_{2,1})(B_{2,2} - B_{1,2}) \\
M_5 &:= (A_{2,1} + A_{2,2})(B_{1,2} - B_{1,1}) \\
M_6 &:= (A_{1,2} - A_{2,1} - A_{2,2} + A_{1,1})B_{2,2} \\
M_7 &:= A_{2,2}(B_{2,2} - B_{1,2} + B_{1,1} - B_{2,1})
\end{aligned}
\qquad \text{<식 3>}
$$

$$C_{1,1} = M_1 + M_3$$
$$C_{1,2} = M_1 + M_2 + M_5 + M_6$$
$$C_{2,1} = M_1 + M_2 + M_4 - M_7$$
$$C_{2,2} = M_1 + M_2 + M_4 + M_5$$

<식 4>

이렇게 Strassen-Winograd 알고리즘은 시간 소모가 더 많은 곱셈의 수를 줄이고 덧셈과 뺄셈의 수를 늘리는 방식을 택하였으며 이를 재귀적으로 반복해 총 $7n^{\log_2 7} - 6n^2$의 연산을 수행하게 되며 전체 시간복잡도는 $O(n^{2.807})$이 된다 (Neapolitan et al., 2010).

하지만 일반적인 행렬곱셈의 시간복잡도를 낮춘 Strassen-Winograd 알고리즘 역시 큰 사이즈의 행렬을 이용한 곱셈을 수행하기에는 여전히 느리다. 더불어 행렬곱셈은 기계학습이나 데이터마이닝을 비롯한 많은 학문 분야에서 기반이 되는 연산으로 사용되는 바 그 활용도가 매우 높다. 따라서 멀티코어를 비롯한 GPGPU, Xeon Phi 등의 가속기를 활용한 병렬화가 절실히 요구된다.

2. 알고리즘 가속화

Strassen-Winograd 알고리즘의 수행속도를 높이기 위해 여러 시도가 있었다. (구동훈 외., 2013, a)에서는 Task Queue라는 자료 구조와 더불어 멀티코어 CPU를 활용해 성능을 향상시켰으며, (구동훈 외., 2013, b)는 (구동훈 외., 2013, a)에 OpenMP의 태스크 병렬처리와 더불어 다른 최적화 요소를 도입해 멀티코어 환경에서 더욱 향상된 성능을 이끌어 내었다. 이에 더해 (나병국 외., 2014)는 GPGPU를 도입해 이기종 환경에서 효과적으로 Strassen-Winograd 알고리즘을 구현하였다. 본 논문에서

는 멀티코어 CPU와 더불어 <식 3>에서 행렬의 크기가 임계 값 보다 작아졌을 때 수행하게 되는 행렬곱셈의 수행, 즉 SIMD 연산의 수행에 Xeon Phi를 사용하고 그 성능을 측정해 보았다.

3. Xeon Phi의 적용

앞서 알고리즘의 소개에서 언급하였듯이 Strassen-Winograd 알고리즘은 재귀적으로 행하여진다. 설명과 구현의 편의성을 위해 한 번의 알고리즘 수행을 하나의 태스크로 설정하였다.

하나의 태스크는 3단계로 나눌 수 있다. 단계 1은 곱셈을 수행할 행렬 A와 B 각각을 정사각 행렬 4개로 나누어 <식 3>의 덧셈과 뺄셈을 하는 것이다. 단계 2는 단계 1에서 구한 결과들로 <식 3>과 같이 곱셈을 수행해 7개의 행렬 M_k (k = 1, 2,..., 7)를 구하는 것이다. 이 과정에서 재귀적으로 Strassen-Winograd 알고리즘을 수행하게 되는데, 만약 행렬의 크기가 임계 값 이하가 되면 일반적인 행렬 곱셈을 적용한다. 이때 일반적인 행렬 곱셈을 Xeon Phi만을 사용하거나 CPU만을 사용해 수행하게 된다. 그리고 마지막 단계에서는 앞 단계에서 구한 M_k 행렬로 <식 4>와 같이 $C_{1,1}$, $C_{1,2}$, $C_{2,1}$, $C_{2,2}$를 계산한다. 그림 4는 단계 1부터 단계 3을 나타낸다. 그림 4와 같이 단계 1, 2, 3은 순차적으로 실행되며 세 단계가 하나의 태스크를 구성한다. 전체 수행 과정은 그림 5와 같다. 그림 5에서 최초 생성된 태스크가 처리하는 행렬의 크기는 임계 값 보다 큰 상황이다.

일반적인 행렬곱셈을 수행할 경우는 행렬의 크기에 따라 Xeon Phi를

사용할 것인지, CPU를 사용할 것인지가 결정된다. 행렬의 크기가 너무 작은 경우는 오히려 Xeon Phi를 적용했을 때 생기는 전송 오버헤드로 인해 성능이 떨어지는 현상이 발생한다. 따라서 전송 오버헤드로 인한 손실보다 Xeon Phi를 통해 얻을 수 있는 속도 이득이 더 클 경우에만 Xeon Phi를 사용하게 하였다.

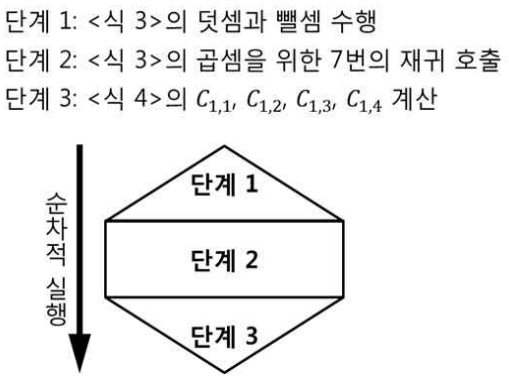


그림 4. Strassen-Winograd 알고리즘 태스크 구성

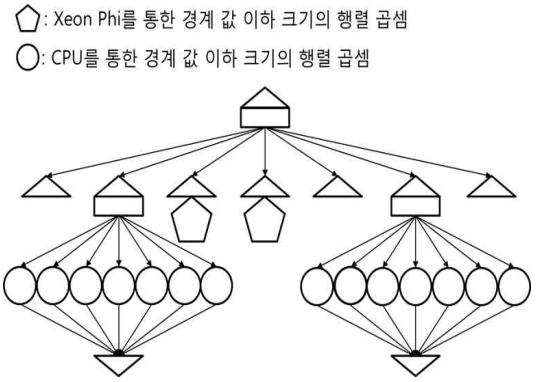


그림 5. Strassen-Winograd 알고리즘 전체 수행 과정

제 2 절 Complex 모델 가속화

GPGPU와 달리 Xeon Phi는 Complex 모델의 구현도 가능하다. 이 경우 Xeon Phi의 많은 스레드는 각기 다른 데이터를 지니고 동시에 서로 다른 명령어를 수행할 수 있다. 혹은 여러 스레드가 각기 단순히 하나 혹은 소수의 명령어가 아니라 많은 명령어를 지닌 하나의 함수나 프로그램을 동시에 실행할 수도 있다. 앞서 제 2장 4절의 Xeon Phi와 GPGPU

와의 비교에서 언급했듯이 GPGPU의 스레드는 Warp 단위로 동작한다. 이럴 경우 특정한 조건의 스레드만이 어떠한 명령어를 실행하게끔 하는 조건문을 만나면 대상 스레드가 동작할 동안 그 스레드가 속한 Warp의 다른 스레드 모두는 유휴(idle)상태가 되어 버리고 만다. 따라서 GPGPU는 이러한 조건문을 포함하는 경우에 구현이 불가능하거나 매우 비효율적이다 (Sanders et al., 2010). 따라서 Xeon Phi를 Complex 방식의 가속화에도 사용할 수 있다는 것은 그 활용도에 있어서 큰 장점이 된다. 예를 들어 GPGPU를 활용한 방식의 가속화가 불가능한 구현 방식의 어플리케이션 성능을 향상시키고자 할 때, 더욱이 소켓의 한계 상 더 이상 CPU를 늘리기는 힘든 상황에서 Xeon Phi는 훌륭한 해결책이 될 수 있다.

이후에 소개할 Hybrid SHRINK (Hybrid SHaRed-memory SLINK)는 이러한 Xeon Phi의 Complex 모델 가속화 특성을 적용해 기존 멀티코어로 구현되어있던 SHRINK (SHaRed-memory SLINK) (Hendrix et al., 2012) 알고리즘을 더욱 가속화한 경우이다.

1. SHRINK 알고리즘 소개

계층 군집화 (Hierarchical Clustering) (Ward Jr & Joe H, 1963)는 문서 군집화나 Bioinformatics 분야에서의 종 (species)간 계통수 (Phylogenetic Tree) 파악 등을 포함한 많은 경우에 사용된다. 군집의 개수를 미리 알려줄 필요가 없고 전체 데이터의 구조나 관계를 잘 표현해 주는 점 등의 장점이 있어 널리 사용되고 있다. 하지만 Bottom-up 방식

으로 행해지는 기본적인 계층 군집화의 경우 시간복잡도가 $O(n^3)$으로 매우 크다. 따라서 큰 데이터에 계층 군집화를 사용하면 너무나 많은 시간이 걸리므로 병렬화를 적용하여 그 수행시간을 단축시키는 것이 필요하다. 그러나 계층 구조를 파악해나가는 과정에서 특정 단계의 작업을 수행하기 위해선 앞 단계의 정보가 필요하게 된다. 이러한 알고리즘 상의 데이터 의존성 때문에 단순히 여러 스레드가 데이터를 나눠가져서 계층 군집화를 진행하는 방법은 적용할 수 없다. SHRINK는 이러한 한계를 극복하고 공유 메모리 (Shared Memory)환경에서 SHC (Single-linkage Hierarchical Clustering) (Sibson & Robin, 1973)를 멀티코어 CPU를 이용해 병렬화한 알고리즘이다. 핵심 아이디어는 주어진 데이터의 거리 정보를 활용해 MST (Minimum Spanning Tree) (Cormen et al., 2001)를 만들고 이를 이용해 SHC의 결과를 구하는 것이다 (Gower et al., 1969). 대부분의 수행 시간을 소비하는 MST의 형성 과정에 병렬화를 적용하여 빠르게 MST를 구하고, 이 MST를 SHC의 결과 표현 방식인 계통수 (Dendrogram)로 치환해줌으로써 향상된 수행 속도를 얻었다.

주된 병렬화 방법은 그림 6과 같다. 우선 전체 데이터를 중복된 부분 집합으로 나누어 여러 스레드(그림 6에서 t_0, t_2,...,t_5 에 해당)에 분배한다. 그리고 각 스레드는 자신이 지니고 있는 데이터로 PRIM 알고리즘 (Prim & Robert Clay, 1957)을 적용해 local MST를 형성하게 된다 (그림 6에서 ξ_P 에 해당).

모든 스레드가 local MST를 만들게 되면 KRUSKAL 알고리즘 (Kruskal & Joseph B, 1956)을 적용해 그것들을 합쳐나가 최종적인 MST를 얻게 된다 (그림 6에서 ξ_K 에 해당). 그리고 마지막에 얻어진

MST를 SHC의 결과 계통수로 바꾸어줌으로써 우리는 원하는 결과를 구할 수 있게 된다.

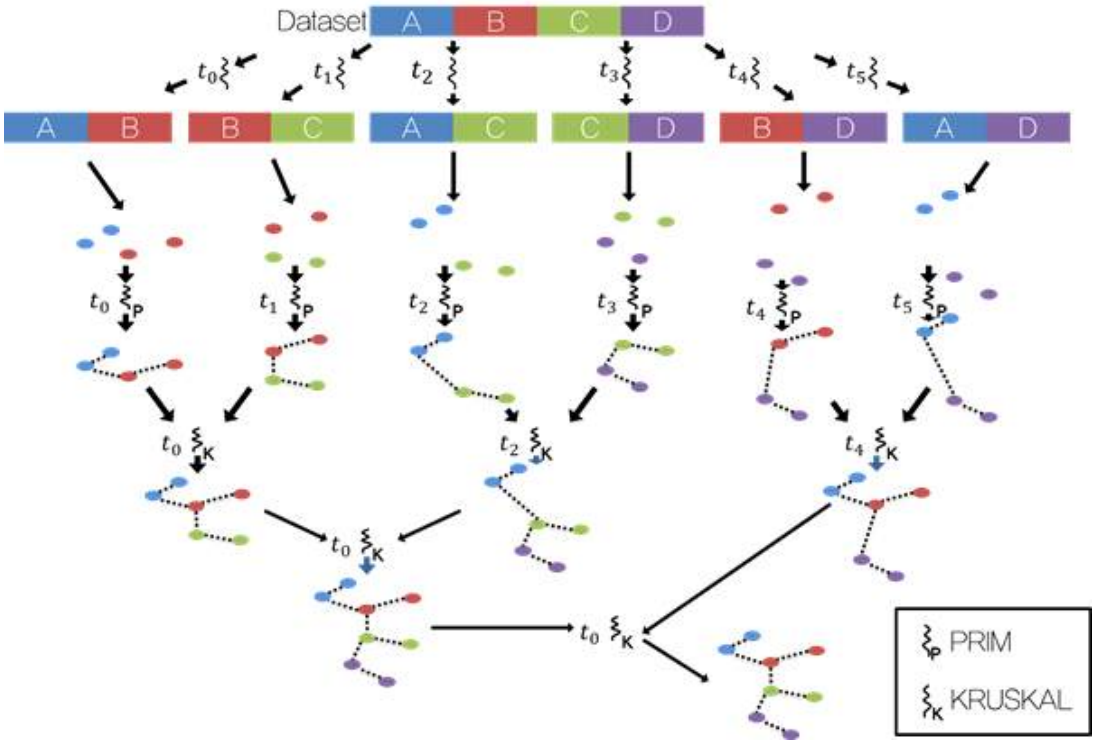


그림 6. SHRINK 수행 과정

2. Xeon Phi의 적용

Xeon Phi를 사용해 SHRINK 알고리즘의 수행 속도를 더욱 빠르게 할 수 있다. 기존의 SHRINK의 구현은 여러 스레드가 각기 다른 데이터를 가지고 조건문 등을 포함한 많은 명령어로 구현된 알고리즘을 수행하는 형태이다. 즉, Complex 모델의 형태이므로 이를 GPGPU에 적용하기에는 어려움이 따른다. 반면 Xeon Phi는 이러한 경우에도 적용이 가능하므로

효과적인 해결책이 될 수 있다. Xeon Phi만을 사용하여 SHRINK를 수행할 수도 있지만 이 경우 호스트 CPU와 Xeon Phi 간의 데이터 전송 시간이 오버헤드가 될 뿐 아니라 호스트 CPU는 아무런 연산을 수행하지 않게 되어 시스템 측면에서 비효율적인 자원사용을 하게 된다. 따라서 호스트 CPU와 Xeon Phi를 모두 사용하는 hybrid 방식이 선호된다. 이에 따라 본 논문은 CPU와 Xeon Phi를 모두 활용하는 hybrid 방식의 SHRINK 알고리즘, 즉 Hybrid SHRIK를 개발하였다. Hybrid SHRINK는 Xeon Phi의 Complex 모델 가속화를 적용하였고 이를 통해 제 4장 2절 3과 같이 멀티코어 CPU를 위한 SHRINK에 비해 수행 시간을 최대 약 2.5배 단축시킬 수 있었다.

3. Hybrid SHRINK 소개

Hybrid SHRINK의 pseudocode는 알고리즘 1과 같다.

알고리즘 1: Hybrid SHRINK의 Pseudocode

1. 전체 데이터를 k (>2)개의 부분 집합 D_1, D_2,…,D_k로 나눈다
2. 한 쌍의 부분집합(D_i, D_j)를 CPU와 Xeon Phi로 분배한다.
3. CPU와 Xeon Phi는 D_i와 D_j의 합집합을 가지고 SHRINK 알고리즘을 수행해 각각의 local MST를 구한다.
4. **repeat**
5. 　　한 번에 MST 두 개씩을 합쳐나간다.
6. **until** 모든 MST가 합쳐질 때까지 수행한다.
7. **return** 합쳐진 MST를 결과로 내놓는다.

Hybrid SHRINK는 기존 SHRINK 알고리즘과 같이 중복을 허용해 데이터를 분산하므로 CPU와 Xeon Phi가 $\binom{k}{2}$개의 데이터 집합(D_i와 D_j의 합집합에 해당)을 나누어가져 SHRINK 알고리즘을 수행하게 된다. Hybrid SHRINK의 전체 수행과정은 그림 7과 같다. 실험환경의 제약 상황과 설명의 편의성을 위해 k = 3으로 고정하였고 두 개의 Xeon Phi를 사용하는 경우를 고려하였다.

Hybrid SHRINK는 우선 전체 데이터를 3개의 부분집합으로 나눈 다음 부분집합의 모든 경우의 쌍을 CPU 혹은 Xeon Phi에게 전송한다. 데이터의 부분집합을 할당받은 CPU나 Xeon Phi는 독립적으로 SHRINK 알고리즘을 수행해 MST를 생성한다. 이때 CPU와 Xeon Phi는 비동기적으로 구동하게 된다. Xeon Phi는 MST가 완성되면 그것을 호스트 CPU로 전송하게 된다. CPU는 자신의 MST가 완성되면 #offload wait 명령어를 실행해 Xeon Phi로부터 MST를 받기 전까지 다음 단계를 진행하지 않도록 한다. 모든 MST가 모이게 되면 호스트 CPU는 KRUSKAL 알고리즘을 적용하여 자신의 것을 포함한 여러 MST를 합쳐 최종적인 MST를 완성한다.

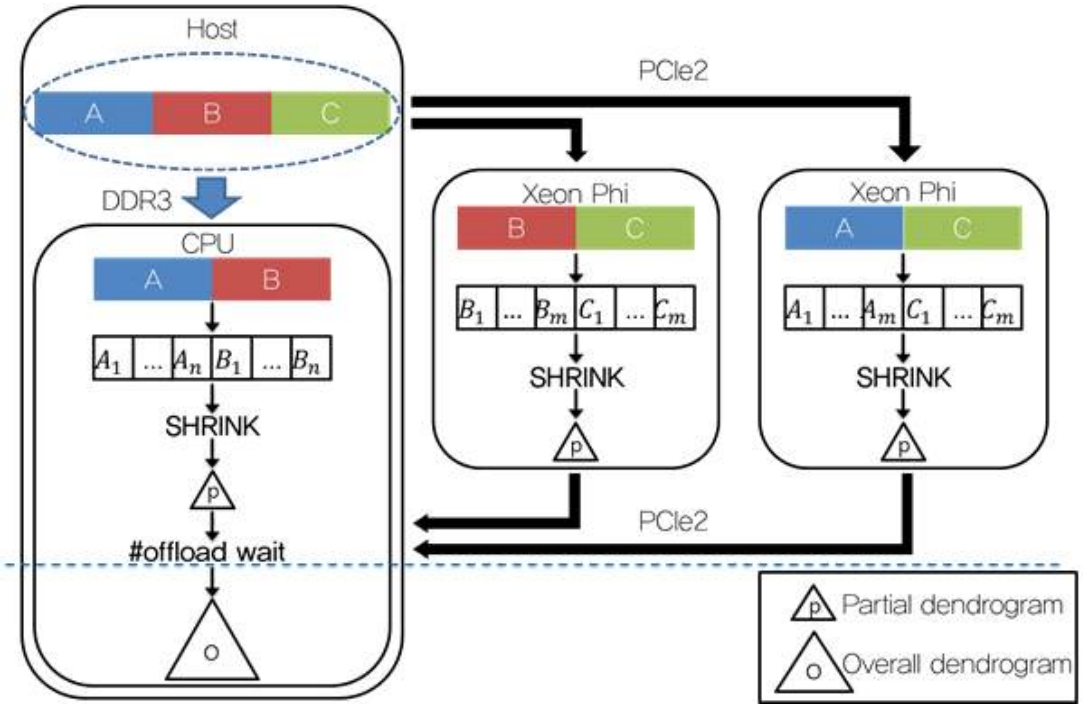


그림 7. Hybrid SHRINK 전체 수행 과정

제 4 장 실 험

제 1 절 Strassen-Winograd 알고리즘 가속화

Strassen-Winograd 알고리즘에 대한 실험은 기존의 멀티코어 CPU로 구현된 연구 (구동훈 외., 2013, b)와 GPU와 CPU를 함께 사용한 연구 (나병국 외., 2014) 등과 그 성능을 비교하였다. 실험은 행렬의 크기를 변화시켜가면서 수행 시간을 측정하는 것으로 진행되었다. 본 논문에서 제시하는 Xeon Phi를 이용한 Strassen-Winograd 알고리즘은 진행 중 일반 곱셈을 수행할 시점에 Xeon Phi를 쓸 것인지 여부를 행렬 크기에 따라 결정한다. 본 논문에서는 이때 결정의 근거가 되는 행렬 크기를 반복적인 실험을 통해 정하였다. 예를 들어 Strassen-Winograd 알고리즘에 입력된 행렬의 크기가 8192×8192 일 때 일반 Xeon Phi를 이용해 일반 행렬곱셈을 수행하는 크기 기준은 2048×2048 이상이 되도록 하였다.

1. 실험 환경

실험에 사용한 시스템은 표 8과 같다.

표 8. Strassen-Winograd 알고리즘 실험 환경

	호스트 프로세서	Xeon Phi	GPGPU
CPU	Intel® Xeon E5-2650 @ 2.00GHz	5110P	GTX TITAN
코어	소켓 당 8개 (스레드 8개)	60개 (스레드 240개)	2688개
메모리	256 GB	8GB	6 GB
운영체제	CentOS 6.6, kernel 2.6.32-504	Linux kernel 2.6.38.8 MPSS3.1.4	없음
개수	2	1	1

또한 실험에서 사용한 모든 행렬의 데이터 타입은 double precision이며 행렬의 크기는 1024×1024, 2048×2048, 4096×4096, 8192×8192, 16384×16384의 총 5개로 구성되었다.

2. 실험 결과

행렬의 크기를 변화시켜가면서 수행 시간을 비교해보았고 결과는 그림 8과 같다.

그림 8에서 Sequential은 CPU 싱글코어를 사용한 Strassen-Winograd 알고리즘의 수행시간이다. OMP (CPU)는 멀티코어 CPU를 활용한 연구 (구동훈., 2013, b)의 수행시간를 말하며 Hetero (GPU)는 멀티코어 CPU와 GPU를 함께 활용한 연구 (나병국 외., 2014)의 수행시간이다. Hetero (PHI)는 본 논문에서 제시하는 Xeon Phi와 CPU를 함께 활용한 Strassen-Winograd 알고리즘의 수행시간이다.

그림 8에서 OMP (CPU)와 Hetero (PHI)를 비교해보면 4096×4096 이하의 크기에선 두 경우의 수행시간이 거의 같음을 알 수 있다. 이는 행

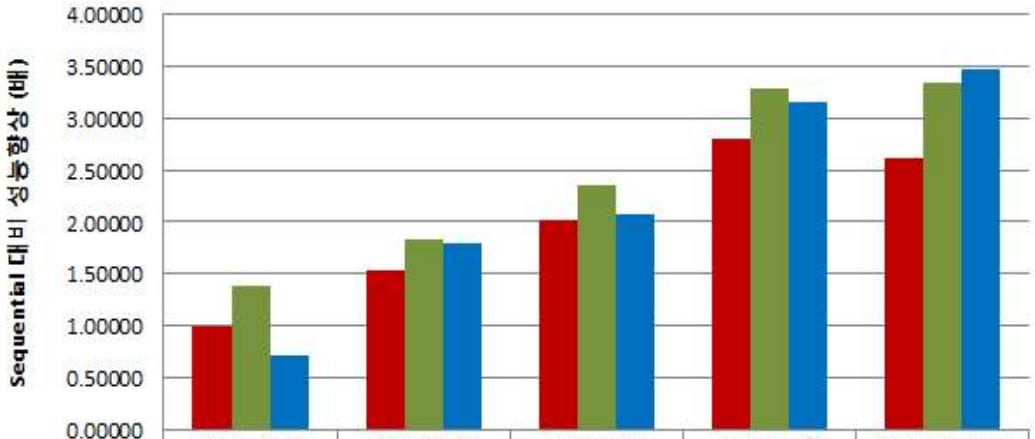
렬의 크기가 작을 경우 Xeon Phi에 의존하지 않고 CPU가 모든 작업을 처리하기 때문이다. 예를 들어 4096×4096 크기의 행렬로 Hetero (PHI)를 실행하면 전체 수행과정이 CPU에서만 이루어지게 된다. 이 경우 행렬의 크기가 1024×1024 이하가 되면 일반적인 행렬 곱셈을 수행하는데 이때 Xeon Phi를 사용한다면 오버헤드로 인한 전체 수행시간 저하가 초래되기 때문이다. 반면 행렬의 크기가 8192×8192 이상부터는 Xeon Phi가 수행하는 작업의 비중이 커지게 된다. 8192×8192 크기의 행렬을 처리할 경우 알고리즘 수행 중 행렬의 크기가 2048×2048 이하가 되면 일반적인 행렬 곱셈 연산을 하게 된다. 이때 총 49개의 일반 행렬 곱셈을 수행하는데 그 중 약 9개를 Xeon Phi가 처리하게 된다. 나아가 16384×16384 크기의 행렬을 처리할 때는 총 47개의 4096×4096 크기의 일반 행렬 곱셈 중 12개 이상을 Xeon Phi가 처리하게 된다. 이처럼 처리하는 행렬의 크기가 커짐에 따라 Xeon Phi의 작업 비중이 커지면서 Hetero (PHI)가 OMP (CPU)보다 우수한 성능을 보이는 것을 확인할 수 있다.

	1024x1024	2048x2048	4096x4096	8192x8192	16384x16384
Hetero (PHI)	0.1730343	0.1983429	0.7651791	3.806802	20.82169
OMP (CPU)	0.12380238	0.230717	0.7841735	4.301179	27.76116
Hetero (GPU)	0.08899129	0.1929026	0.6702904	3.649093	21.72688
Sequential	0.12221121	0.354344	1.579891	12.01543	72.48286

그림 8. Strassen-Winograd 알고리즘 수행 결과

Hetero (GPU)와 Hetero (PHI)는 성능에서 큰 차이를 보이지 않는다. 이를 통해 Xeon Phi 또한 GPGPU와 마찬가지로 Simple 모델 병렬화에 성공적으로 적용될 수 있음을 확인할 수 있다.

그림 9는 OMP (CPU)와 Hetero (GPU) 그리고 본 논문에서 제시한 Hetero (PHI)가 Sequential 대비 얼마만큼의 성능 향상이 있었는지를 나타낸 것이다. 앞에서 언급했던 것과 같이 행렬의 크기가 커질수록 Hetero (PHI)는 더 큰 성능 향상을 보이게 된다. 그리하여 Hetero (PHI)는 행렬의 크기가 16384×16384 일 때 Sequential 대비 약 3.5배의 성능 향상을 이루었다.



	1024x1024	2048x2048	4096x4096	8192x8192	16384x16384
OMP (CPU)	0.98715	1.53584	2.01472	2.79352	2.61094
Hetero (GPU)	1.37329	1.83691	2.35702	3.29272	3.33609
Hetero (PHI)	0.70628	1.78652	2.06473	3.15631	3.48112

그림 9. 싱글코어 CPU 대비 성능 향상

제 2 절 SHRINK 알고리즘 가속화

기존 SHRINK 대비 Hybrid SHRINK의 성능 향상을 확인하기 위한 실험을 진행하였다.

1. 실험 환경

호스트 프로세서와 Xeon Phi 코프로세서가 사용되었으며 데이터의 차원은 동일하나 객체의 개수가 다른 데이터 10종류가 사용되었다. 실험에 사용된 시스템의 사양은 표 9와 같다.

표 9 Hybrid SHRINK 실험 환경

	호스트 프로세서	코프로세서
CPU	Intel® Xeon E5-2650 @ 2.00GHz	Intel Xeon Phi 5110P
코어	소켓 당 8개의 물리 코어(스레드 8개)	코프로세서 당 60개의 물리 코어 (스레드 240개)
메모리	256 GB	코프로세서 당 8GB
운영체제	CentOS 6.6, kernel 2.6.32-504	Linux kernel 2.6.38.8 MPSS3.1.4
개수	2	2

또한 사용한 데이터는 표 10과 같다.

표 10 Hybrid SHRINK 실험 데이터

이름	객체 개수	차원
50K	50,000	10
100K	100,000	10
150K	150,000	10
200K	200,000	10
250K	250,000	10
300K	300,000	10
350K	350,000	10
400K	400,000	10
450K	450,000	10
500K	500,000	10

2. 실험 결과

기존 멀티코어로 수행하는 SHRINK (CPU_only), Xeon Phi 하나로만 실행한 SHRINK (Phi_only), CPU와 Xeon Phi 하나를 사용한 Hybrid

SHRINK (Hybrid_1Phi), 그리고 CPU와 Xeon Phi 두 개를 사용한 Hybrid SHRINK (Hybrid_2Phi) 등 총 4개의 실험이 진행되었다. 실험 환경과 알고리즘의 제한 사항으로 데이터의 부분집합 개수 k는 3으로 고정하였다. 표 10의 모든 데이터에 대해 수행시간을 측정하였으며 CPU와 Xeon Phi는 성능을 저하시키지 않는 최대치의 스레드를 사용하기 위해 각각 16개, 236개의 스레드를 생성하였다 (Xeon Phi의 경우 1개의 코어는 스케쥴링을 위해 사용하지 않았다). 수행 시간이 구해진 다음에는 Phi_only, Hybrid_1Phi, Hybrid_2Phi에 대해 멀티코어 SHRINK 대비 성능 향상을 구하였다. 4개의 실험에 대한 수행시간은 그림 10과 같다.

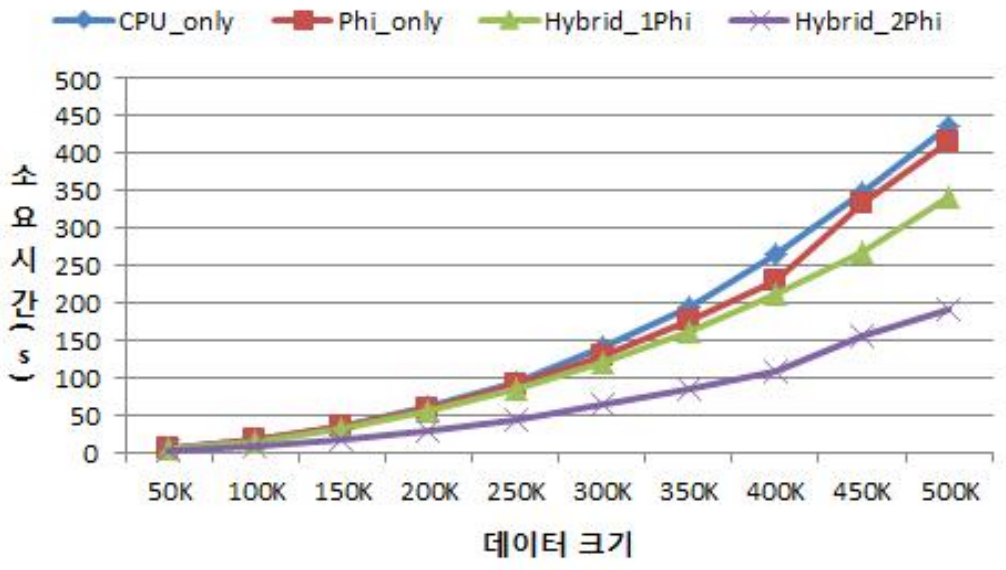


그림 10 SHRINK 및 Hybrid SHRINK 수행 속도

그림 10을 통해 CPU_only에 비해 Phi_only, Hybrid_1Phi, Hybrid_2Phi 모두 수행 속도가 향상되었음을 확인할 수 있다. 데이터 크

기가 500K일 경우 CPU_only 대비 Phi_only는 약 1.1배, Hybrid_1Phi는 약 1.3배, Hybrid_2Phi는 약 2.3배의 속도 향상을 보인다. 또한 데이터 크기가 증가함에 따라 수행 속도가 증가하는 양상의 형태가 비슷하다는 사실도 확인할 수 있다. 그림 11은 16개의 스레드를 사용한 멀티코어 SHRINK 대비 성능 향상을 나타낸 것이다.

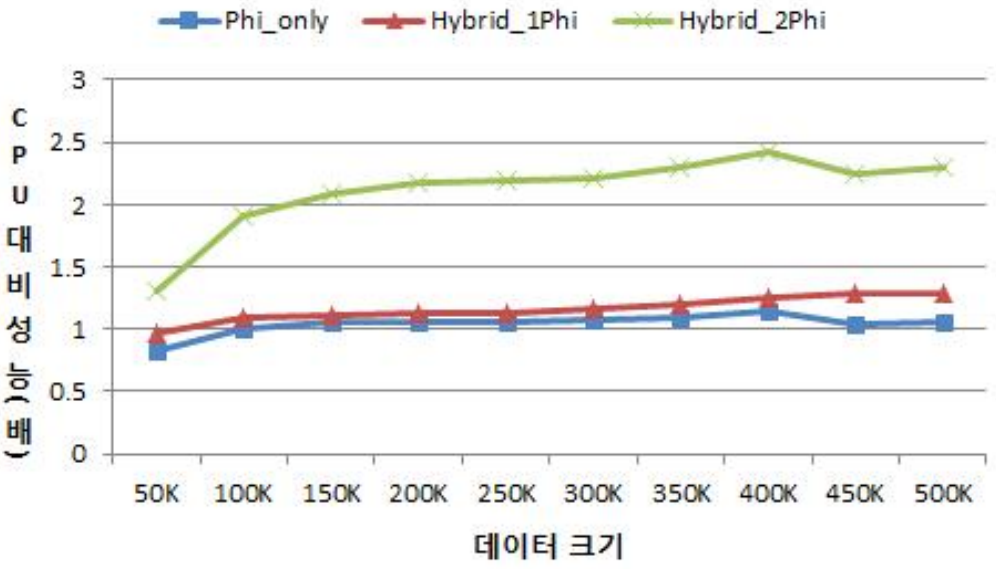


그림 11 멀티코어 CPU (16 스레드) 대비 성능 향상

데이터 크기가 100K 이상부터는 Phi_only, Hybrid_Phi1, Hybrid_2 모두 CPU_only에 비해 더 나은 성능을 보인다는 사실을 알 수 있다. 또한 그래프를 통해 성능향상 비가 어느 정도 일정하게 유지됨을 알 수 있다. 성능향상 비는 Phi_only와 Hybrid_1Phi는 약 1.1배이며 Hybrid_2Phi의 경우는 약 2.3배이다. Hybrid_1Phi의 경우는 CPU 혹은 Xeon Phi로 3개의 데이터 집합 중 2개의 데이터 집합이 몰려 균등한 작업 분배가 일어

나지 않는다. 그로 인해 CPU_only에 비해 큰 성능향상을 이룰 수가 없었다. 반면 Hybrid_2Phi의 경우 CPU와 2개의 Xeon Phi에 균등한 작업 분배가 이루어지나 각기 형성된 3개의 MST를 하나로 합치는 과정 등의 선후처리 작업이 더해져 이론적인 성능 향상치(3배)에는 미치지 못하는 결과를 보이게 되었다. 향후 연구에서 작업 분배 방식과 선후처리 작업의 개선이 이루어진다면 더욱 향상된 결과를 얻을 수 있을 것으로 보인다.

이처럼 Hybrid SHRINK의 구현과 실험 결과를 통해 Xeon Phi는 Complex 모델에서도 효과적으로 구현될 수 있음을 알 수 있다.

제 4 장 결 론

본 논문은 Xeon Phi를 활용한 효과적인 병렬프로그래밍을 위해 소프트웨어 측면에서 고려해야 할 사항을 담고 있다. Xeon Phi의 구조적인 특징과 프로그래밍 모델에 대해 설명하고 있으며 기존에 고성능 컴퓨팅을 위해 사용되고 있던 GPGPU와의 비교 사항도 언급하고 있다. 또한 Xeon Phi를 이용하여 구현 가능한 병렬화 모델도 제시해 Xeon Phi를 통해 어플리케이션의 성능을 높이고자 할 때 참고가 될 수 있도록 하였다. Xeon Phi를 통한 병렬화 모델은 크게 Simple 모델과 Complex 모델로 구분하였고 각 경우에 해당하는 구체적인 사례와 해당하는 실험 결과를 제시하였다. Simple 모델의 경우 Strassen-Winograd 알고리즘을, Complex 모델의 경우 Hybrid SHRINK 알고리즘을 예로 들고 있다. 각 사례에서는 Xeon Phi를 해당하는 병렬화 모델에 맞게 적용하고 실험을 진행하였다. 실험 결과를 통해 기존 GPGPU가 쓰이던 방식인 Simple 모델을 Xeon Phi도 성공적으로 구현할 수 있음을 보였고, 나아가 GPGPU는 효율적인 구현이 어려운 Complex 모델의 경우도 Xeon Phi는 훌륭하게 구현할 수 있음을 보였다.

물론 Xeon Phi는 이제 1세대가 출시되었으므로 PCI 익스프레스 2.0만을 지원한다거나 전송해 줄 수 있는 데이터 구조의 제약이 있다는 것 등의 단점이 존재한다. 하지만 본 논문의 구체적인 사례와 실험을 통해 살펴보았듯이 Xeon Phi는 기존의 GPGPU가 수행하던 기능 뿐 아니라

그 이상의 역할도 소화해낼 수 있다. 따라서 어플리케이션의 성능을 향상시키고자하는 개발자는 Xeon Phi의 사용 또한 반드시 고려해 보아야 한다. Strassen-Winograd와 같은 일반적인 알고리즘 뿐 아니라 SHRINK를 비롯한 Large-Scale의 데이터마이닝 같은 특수한 알고리즘에 이르기까지 다양한 알고리즘에 본 논문에서 제시한 것과 같은 Simple 모델, Complex 모델의 형태로 Xeon Phi를 적용한다면 효과적인 가속화를 이루어낼 수 있을 것이다.

참 고 문 헌

Akhter, Shameen, and Jason Roberts. Multi-core programming. Vol. 33. Hillsboro: Intel press, 2006.

Byunghan Lee, Joonhong Park, and Sungroh Yoon. "Rapid and Robust Denoising of Pyrosequenced Amplicons for Metagenomics." In ICDM, pp. 954-959. 2012.

Caballero, Diego, Alejandro Duran, and Xavier Martorell. "An OpenMP* barrier using SIMD instructions for Intel® Xeon PhiTM coprocessor." In OpenMP in the Era of Low Power Devices and Accelerators, pp. 99-113. Springer Berlin Heidelberg, 2013.

Chandra, Rohit, ed. Parallel programming in OpenMP. Morgan Kaufmann, 2001.

Chrysos, George, and Senior Principal Engineer. "Intel xeon phi coprocessor (codename knights corner)." In Proceedings of the 24th Hot Chips Symposium, HC. 2012.

Chrysos, George. "Intel® Xeon Phi™ Coprocessor-the Architecture." Intel Whitepaper, 2014.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. Vol. 2. Cambridge: MIT press, 2001.

Dongarra, Jack, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov. "Portable HPC programming on intel many-integrated-core hardware with MAGMA Port to Xeon Phi." In Parallel processing and applied mathematics, pp. 571-581. Springer Berlin Heidelberg, 2014.

Duncan, Ralph. "A survey of parallel computer architectures." Computer 23, no. 2 (1990): 5-16.

Fang, Jianbin, Ana Lucia Varbanescu, Baldomero Imbernon, Jose M. Cecilia, and Horacio Perez-Sanchez. "Parallel Computation of Non-Bonded Interactions in Drug Discovery: Nvidia GPUs vs. Intel Xeon Phi." In Proceedings of the 2nd International Work-Conference on Bioinformatics and Biomedical Engineering (IWBBIO'14). 2014.

Fang, Jianbin, Ana Lucia Varbanescu, Henk Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. "An Empirical Study of Intel Xeon Phi." arXiv preprint arXiv:1310.5842, 2013.

Gower, John C., and G. J. S. Ross. "Minimum spanning trees and single linkage cluster analysis." Applied statistics, 1969: 54-64

Harish, Pawan, and P. J. Narayanan. "Accelerating large graph algorithms on the GPU using CUDA." In High performance computing - HiPC 2007, pp. 197-208. Springer Berlin Heidelberg, 2007.

Hendrix, William, M. M. Ali Patwary, Ankit Agrawal, Wei-keng Liao,

and Alok Choudhary. "Parallel hierarchical clustering on shared memory platforms." In High Performance Computing (HiPC), 2012 19th International Conference on, pp. 1-9. IEEE, 2012.

Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2012.

Hong-Tao, Bai, He Li-li, Ouyang Dan-tong, Li Zhan-shan, and Li He. "K-means on commodity gpus with cuda." In Computer Science and Information Engineering, 2009 WRI World Congress on, vol. 3, pp. 651-655. IEEE, 2009.

Hwancheol Jeong, Weonjong Lee, Jeonghwan Pak, Kwang-jong Choi, Sang-Hyun Park, Jun-sik Yoo, Joo Hwan Kim, Joungjin Lee, and Young Woo Lee. "Performance of Kepler GTX Titan GPUs and Xeon Phi System." arXiv preprint arXiv:1311.0590 (2013).

Jeffers, James, and James Reinders. Intel Xeon Phi coprocessor high-performance programming. Newnes, 2013.

Kruskal, Joseph B. "On the shortest spanning subtree of a graph and the traveling salesman problem." Proceedings of the American Mathematical society 7, no. 1 (1956): 48-50.

Levinthal, Adam, and Thomas Porter. "Chap-a SIMD graphics processor." In ACM SIGGRAPH Computer Graphics, vol. 18, no. 3, pp. 77-82. ACM, 1984.

Michael, J. Quirm. "Parallel Programming in C with MPI and

OpenMP." (2003).

Neapolitan, Richard, and Kumarss Naimipour. Foundations of algorithms. Jones & Bartlett Publishers, 2010.

Newburn, Chris J., Serguei Dmitriev, Ravi Narayanaswamy, John Wiegert, Ravi Murty, Francisco Chinchilla, Rajiv Deodhar, and Russell McGuire. "Offload Compiler Runtime for the Intel® Xeon Phi Coprocessor." In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, pp. 1213-1225. IEEE, 2013.

Pennycook, Simon J., Chris J. Hughes, M. Smelyanskiy, and Stephen A. Jarvis. "Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors." In Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pp. 1085-1097. IEEE, 2013.

Prim, Robert Clay. "Shortest connection networks and some generalizations." Bell system technical journal 36, no. 6 (1957): 1389-1401.

Rahman, Rezaur. Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers. Apress, 2013.

Reinders, James, and James Jeffers. High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches. Morgan Kaufmann, 2014.

Sanders, Jason, and Edward Kandrot. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, 2010.

Saule, Erik, Kamer Kaya, and Ümit V. Çatalyürek. "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi." In Parallel Processing and Applied Mathematics, pp. 559-570. Springer Berlin Heidelberg, 2014.

Schaller, Robert R. "Moore's law: past, present and future." Spectrum, IEEE 34, no. 6 (1997): 52-59.

Sibson, Robin. "SLINK: an optimally efficient algorithm for the single-link cluster method." The Computer Journal 16, no. 1 (1973): 30-34.

Strassen, Volker. "Gaussian elimination is not optimal." Numerische Mathematik 13, no. 4 (1969): 354-356.

Suda, Reiji. "Investigation on the power efficiency of multi-core and gpu processing element in large scale simd computation with cuda." In Proceedings of the International Conference on Green Computing, pp. 309-316. IEEE Computer Society, 2010.

Sutter, Herb. "The free lunch is over: A fundamental turn toward concurrency in software." Dr. Dobb's journal 30, no. 3 (2005): 202-210.

Teodoro, George, Tahsin Kurc, Jun Kong, Lee Cooper, and Joel Saltz.

"Comparative Performance Analysis of Intel Xeon Phi, GPU, and CPU." arXiv preprint arXiv:1311.0378 (2013).

Thibault, Julien C., and Inanc Senocak. "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows." In Proceedings of the 47th AIAA Aerospace Sciences Meeting, pp. 2009-758. 2009.

Trapnell, Cole, and Michael C. Schatz. "Optimizing data intensive GPGPU computations for DNA sequence alignment." Parallel computing 35, no. 8 (2009): 429-440.

Ward Jr, Joe H. "Hierarchical grouping to optimize an objective function." Journal of the American statistical association 58, no. 301 (1963): 236-244.

Weinberg, V. "Best Practice Guide - Intel Xeon Phi."

Winograd, Shmuel. "A new algorithm for inner product." Computers, IEEE Transactions on 100, no. 7 (1968): 693-694.

Wu, Fuhui, Qingbo Wu, Yusong Tan, Lifeng Wei, Lisong Shao, and Long Gao. "A Vectorized K-Means Algorithm for Intel Many Integrated Core Architecture." In Advanced Parallel Processing Technologies, pp. 277-294. Springer Berlin Heidelberg, 2013.

Yong-Chull Jang, Hwancheol Jeong, Jangho Kim, Weonjong Lee, Jeonghwan Pak, and Yuree Chung. "Code Optimization on Kepler GPUs and Xeon Phi." arXiv preprint arXiv:1411.2223

(2014).

Yongkweon Jeon, Eesuk Jung, Hyeyoung Min, Eui-Young Chung, and Sungroh Yoon. "GPU-based acceleration of an RNA tertiary structure prediction algorithm." Computers in biology and medicine 43, no. 8 (2013): 1011-1022.

구동훈, 전용권, 윤성로, "OpenMP 태스크 병렬처리를 활용한 행렬 곱셈 가속화." 대한전자공학회 학술대회 (2013): 867-870, a

구동훈, 전용권, 유승학, 윤성로, "Task Queue 기반 병렬처리를 통한 행렬 곱셈 가속화." 대한전자공학회 학술대회 (2013): 1321-1324, b

나병국, 전용권, 구동훈, 윤성로, "이기종 컴퓨팅 환경에서의 동적 스케쥴링을 통한 행렬곱셈 가속화." 한국정보과학회 학술발표논문집 (2014): 51-53.

Abstract

Parallel Programming with Intel Xeon Phi

Donghoon Koo

Department of Electrical and Computer Engineering

The Graduate School

Seoul National University

The Intel Xeon Phi is the high-performance coprocessor for highly parallelized computation based on its many-integrated core (MIC) architecture including over 50 processing cores. A full understanding of hardware features and programming models of Xeon Phi coprocessor would provide developers with the knowledge to implement various parallelized computation effectively. In this paper, I describe the details of Xeon Phi features and programming models. Moreover, I elaborate the differences between GPGPU (General-Purpose computing on Graphics Processing Units) and Xeon Phi from developer's

perspective. Next, I present plausible(possible) parallel models which can be implemented with Xeon Phi, including the Strassen-Winograd algorithm and the Hybrid SHRINK algorithm. The output from running the two algorithms on Xeon Phi shows improved performance in comparison to previous work using GPGPU. In conclusion, my work demonstrates that Xeon Phi can be employed more extensively than GPGPU for various algorithms.

keywords : Xeon Phi, Accelerator, Parallel programming, GPGPU, Strassen, Hierarchical Clustering
Student Number : 2013-20744