기술 보고서

계산가속기 프로그래밍 모델별 비교분석서

2014.11

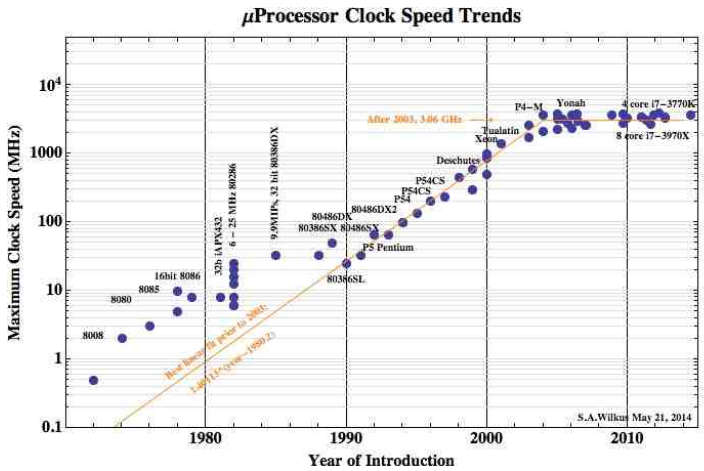
슈퍼컴퓨팅사용자지원실

목 차

1. ····· 1
2. 계산가속기 역사 ····· 1
3. 연구의 필요성 ····· 2
4. 선진국 동향 ····· 5
5. 국내동향 ····· 6
6. Xeon Phi 프로그래밍 모델 분석 ····· 8
7. NVIDIA GPU 프로그래밍 모델 분석 ····· 14
8. 실제 응용 코드의 컴퓨팅 모델별 테스트 ····· 19
<부록: Xeon Phi에서의 병렬프로그래밍 모델에 따른 성능 비교> ····· 23

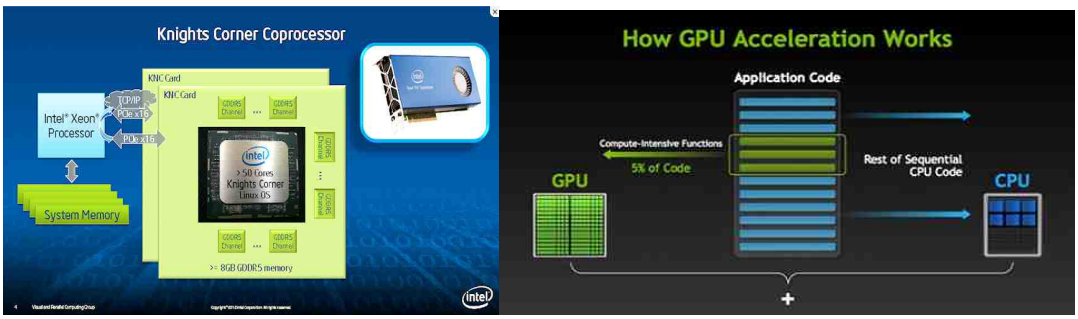
계산가속기 프로그래밍 모델별 비교분석서

- 1. 개요
  - 가속기 컴퓨팅 모델별 이식성 및 성능을 파악함으로서 사용자 코드를 가속기 모델로 이식하여 성능을 향상할 수 있는 기술력 습득을 통해 관련 사용자 지원 역량 강화
- 2. 계산가속기 역사
  - 1972년 최초의 8비트 MPU8008이 IBM에 의해 개발된 이후, Intel에 의해 16비트로 처리할 수 있는 중앙처리장치 CPU가 개발됨으로써 개인용 컴퓨터 시대가 시작
  - 1982년 최초의 멀티태스킹의 개념이 적용되었고, 2개의 정수 실행용 파이프라인과 확장형 부동소수점 프로세서인 펜티엄이 1993년 출시되었다. 현재는 0과 1의 구분을 초당 40억번 이상할 수 있는 4GHz 성능의 CPU가 출시되었다.



마이크로프로세스 클럭 속도 경향

  - CPU는 싱글 코어 구조에서 클록 속도를 높여 성능을 향상 시키려는 노력을 해왔으나 한계에 도달하자 하나의 칩에 코어를 여러 개 둔 멀티코어 형태로 발전
  - CPU의 성능 향상을 위해 이제는 3차 그래픽을 연산처리하기 위해 만들어진 GPU와 결합하기 되었음
  - CPU와 GPU의 결합은 CPU간의 결합보다 훨씬 더 좋은 성능을 보였고 전력의 사용량도 더 적었으며 비용면에서도 경제적이라는 장점을 가지고 있음. 이런 이유로 세계 주요 슈퍼컴퓨팅센터들에서는 NVIDIA의 GPU나 Intel의 Xeon Phi를 탑재한 이기종 컴퓨팅 환경을 구축하거나 구축할 계획을 가지고 있음



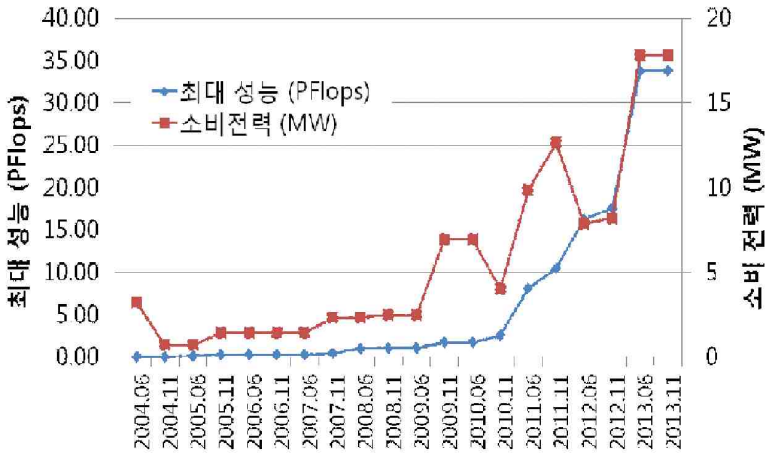
Intel의 Xeon Phi와 NVIDIA의 GPU 가속 컴퓨팅 개념도

• 3. 연구의 필요성

- 고성능컴퓨팅 (HPC: High Performance Computing) 분야에서 초고성능컴퓨터의 성능 향상에 의한 소비 전력의 증가가 큰 문제가 되고 있음

  * 현재의 초고성능컴퓨터는 계산노드를 네트워크로 연결하여 구성되기 때문에 시스템의 코어 수가 증가함으로써 성능을 향상시킬 수 있으나, 동시에 소비 전력이 코어수에 비례하여 증가함

  * 초고성능컴퓨터의 성능을 평가하는 TOP 500 순위에 의하면, 시스템의 성능은 10년 전인 2004년과 비교하여 900배 이상 증가하였으며 소비 전력은 5배에서 18배 이상 증가함

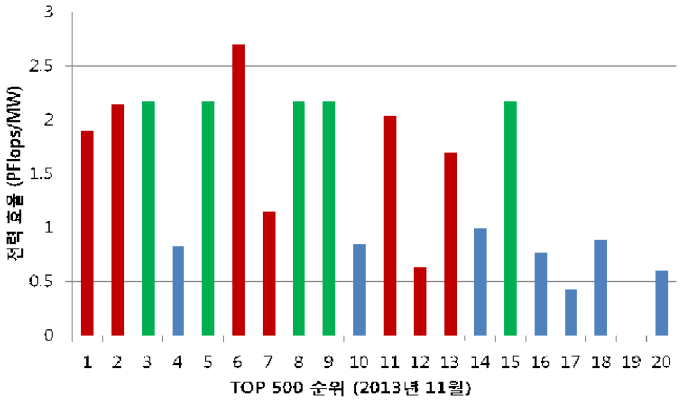


초고성능컴퓨터의 성능과 소비전력

- 초고성능컴퓨터의 설계 규모가 소비전력에 의하여 제한됨에 따라 시스템의 전력 효율 (단위 와트에 대하여 달성 가능한 성능) 이 큰 주목을 받게 됨

  * 기존의 초고성능컴퓨터는 Intel Xeon 프로세서와 같은 상용 CPU를 사용하여 왔으나 보다 전력 효율이 높은 연산 장치가 등장하여 계산 시스템에 사용되고 있음

* IBM BlueGene 시리즈는 기존의 프로세서보다 처리 능력이 낮으나 전력 효율이 높은 프로세서를 다수 네트워크로 연결하여 사용함으로써 전력 효율이 높은 슈퍼컴퓨터를 구축함
* 최근에는 프로세서 코어의 기능을 보다 단순화하여 많은 수의 코어를 한 장의 카드 안에 구현함으로써 전력 효율을 극대화한 가속기(accelerator)가 등장하여, 고성능컴퓨팅 분야에 이용되고 있음



2013년 11월 초고성능컴퓨팅 TOP 500 순위

- 제한된 소비 전력량을 이용하여 높은 성능을 달성하기 위하여 페타플롭스급의 초고성능컴퓨터를 중심으로 가속기를 채용하는 사례가 증가하고 있음
  * TOP 500 순위에 따르면 2010년 6월 당시의 초고성능컴퓨터의 1.8%가 가속기를 탑재하여 전체 성능의 9%를 차지하였으나, 지난 3년간 가속기의 도입이 활발히 이루어져 2013년 6월 현재 500위 이내의 10.8%의 시스템이 가속기를 탑재하였으며 이는 전체 성능의 33.7%를 차지함
  * 많은 해외 센터에서 현재 가속기 기반 초고성능컴퓨터의 도입을 검토 중이며, 가속기 기반 시스템의 구축과 활용 기술에 관한 연구개발 경쟁이 활발하게 이루어지고 있음

현재 서비스 중인 페타플롭스급 가속기 기반 초고성능컴퓨터

국가명	소속기관	이름(제작사)	이론성능 (가속기와CPU 비율)	가속기 모델	TOP500	서비스
중국	NUDT	Tianhe-2(NUDT)	55 PF(7.13:1)	Intel Xeon Phi	1	'13.6
미국	ORNL	Titan (Cray)	27 PF (9.42:1)	NVIDIA K20x	2	'12.11
스위스	CSCS	Piz Daint (Cray)	7.8PF	NVIDIA K20x	6	'13.11
미국	ACC	Stampede(DELL)	6.5 PF (2.96:1)	Intel Xeon Phi	7	'12.11

- 5호기 이후의 초고성능 슈퍼컴퓨터에서 가속기 도입이 불가피함
  * 현행 4호기를 기준으로 5호기의 목표 성능치 30PFLOPS를 달성하기 위해서는 124,481kW의 전력이 필요함
  * 2013년 GREEN 500 순위의 상위 10위가 모두 GPU 가속기 기반 시스템이었으며, 이를 이용함으로서 전력을 효율적으로 이용하는 초고성능컴퓨터 구축이 가능함

2013년 11월 초고성능컴퓨팅 GREEN 500 순위

국가명	소속기관	이름(제작사)	MFLOPS /W	이론성능	가속기 모델	GREEN 500 순위	전체 전력소모
일본	도쿄공대	TSUBAME-KFC	4,503	125TF	NVIDIA K20x	-	27.78kW
스위스	CSCS	Piz Daint (Cray)	3,186	7800TF	NVIDIA K20x	4	2448kW
한국	KISTI	Tachyon2	215	307TF	-	331	1275kW

  * GPU 가속기를 사용할 경우 30PFLOPS의 시스템을 6662kW의 전력으로 달성 가능할 것으로 예상 (TSUBAME-KFC 시스템 기준)
  * 4호기 시스템과 동등한 전력 예산에서 페타스케일급 성능을 달성하기 위해서는 5호기 이후의 시스템에 가속기의 탑재가 불가피할 것으로 예상되며, KISTI 뿐만 아니라 전국의 대학, 연구 기관 등에서 가속기를 채용하는 사례가 늘어날 것으로 전망

- 가속기를 채용한 시스템은 높은 전력 대비 성능을 달성하는 반면, 기존의 CPU와 소스 코드의 호환성이 없기 때문에 전용 프로그래밍 모델을 이용한 프로그램의 변경이 필요함
  * 기존의 CPU를 대상으로 한 프로그래밍 코드는 가속기를 이용할 수 없으며, OpenCL, CUDA와 같은 병렬 프로그래밍 모델을 사용하여 병렬화 작업을 수행하여야 함
  * 병렬화 작업에 많은 시간이 소요됨 (Oak Ridge National Laboratory (ORNL)의 경우, CPU 기반의 Jaguar 시스템에서 GPU 가속기 기반의 Titan 시스템으로 코드를 이식하는데 2-4명의 작업이 필요)
  * 가속기의 성능을 효과적으로 끌어내기 위해서는 대상 아키텍처에 최적화된 수치 라이브러리의 이용과 전용 프로그래밍 모델을 이용한 성능 최적화 등의 작업이 필요하며 이는 장기간에 걸친 노하우 축적이 필요함

- 5호기 이후에 도입될 가속기 기반 시스템에 대비하여 계산기의 구축과 운영, 프로그래밍 기술에 관한 노하우 축적이 필요함
  * 가속기를 탑재한 병렬 고성능 계산기를 구축하여 MPI와 같은 병렬 라이

브러리 및 벤치마크에 관한 연구를 수행함으로써 5호기 이후에 등장할 가속기 기반 초고성능컴퓨터의 구축과 운영에 관한 노하우를 축적할 수 있으며, 이를 통하여 새롭게 도입될 시스템의 성능 평가와 운영상의 문제 해결을 원활히 수행하는 점에 기여할 수 있음

* 4호기 슈퍼컴퓨터까지의 계산시스템은 CPU로만 구성되어 있기 때문에 사용자 코드, 수치 라이브러리, 벤치마크 등이 CPU를 대상으로 설계되어 있으며, 이를 가속기 기반 초고성능슈퍼컴퓨터에서 사용하기 위해서는 소프트웨어를 가속기 병렬 프로그래밍 모델을 사용하여 성능최적화를 하는 등의 재구축이 필요함

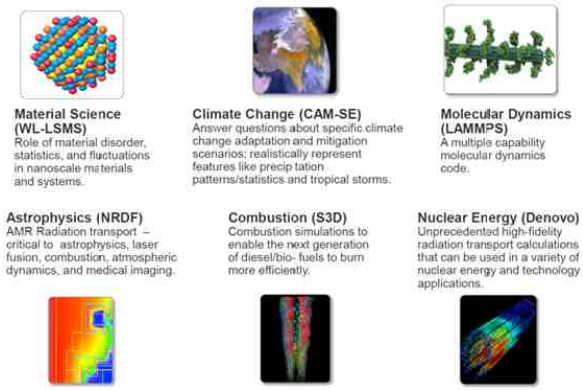
* 사용자가 필요로 하는 수치 라이브러리 및 프로그래밍 모델의 확보와 다양한 유형의 사용자 코드의 병렬화 작업을 원활히 수행할 수 있는 내부 역량의 강화를 통하여 이기종 슈퍼컴퓨터 도입에 의한 급격한 아키텍처의 변화에 즉각적으로 대응할 수 있도록 준비할 수 있음

• 4. 선진국 동향

- 미국 ORNL(Oak Ridge National Laboratory) 슈퍼컴퓨팅 연구소에서는 GPU 기반 초고성능컴퓨터 Titan 시스템을 구축하여 운영중임

  * 가속기 활용을 극대화하기 위하여 가속기 대응 센터인 CARR(Center for Accelerated Application Readiness)를 설립함

  * 가속기 활용 대표 사례 6개를 중심으로 연구를 진행중임



ORNL/CAAR 6개 대표 응용 사례

- 미국 TACC(Texas Advanced Computing Center는 계산 가속기인 Intel Xeon Phi, NVIDIA GPU를 탑재한 초고성능컴퓨터 Stampede를 보유하고 있으며, 2013년 현재 의학, 지질학 등을 포함한 100여 분야의 사용자를 대상으로 1000개 이상의 프로젝트가 진행중임

  * 미국 Stanford University와 미국 The University of Texas at Austin의 연구 그룹은 GPU 가속기를 사용하여 물 분자의 동작을 재현하는 연구를 수행하고 있으며, GPU 가속기가 가지는 높은 성능을 활용하여 물이 가지는

성질의 규명과 이를 바탕으로 한 신약의 개발 속도가 크게 단축될 것으로 기대됨

* 미국 UIUC(University of Illinois at Urbana-Champaign) 대학의 연구 그룹은 차세대 바이오 연료 개발을 위한 '계산 현미경'으로서 Stempede를 사용하고 있으며, 이를 위하여 Intel Xeon Phi 가속기를 사용한 고성능 분자동력학(MD: Molecular Dynamics) 코드를 작성하여 이용 중임

- 미국 University of Tennessee의 Jack Dongarra 교수의 연구 그룹은 가속기를 사용하는 수치 라이브러리의 개발과 클러스터 성능 최적화에 관한 연구를 진행 중임
  * GPU, Intel Xeon Phi와 같은 가속기와 CPU를 효율적으로 사용하는 고성능 수치 라이브러리 MAGMA를 개발하여 배포하고 있으며, 이는 다수의 수치 시뮬레이션 코드에 사용되고 있음
  * 수치 라이브러리의 개발과 클러스터 병렬 최적화에 관한 연구를 바탕으로 Intel Xeon Phi 가속기를 탑재한 Beacon 계산시스템을 이용하여 2013년 6월의 Green 500 순위에서 3위를 달성하였으며, 이를 통해 Intel Xeon Phi 가속기가 가지는 높은 전력대비 성능을 증명하였음

- 중국은 정부에 의하여 가속기 활용 기술에 관한 집중 투자가 이루어지고 있으며, 하드웨어와 소프트웨어의 양면에서 높은 수준의 결과를 달성하고 있음
  * 2010년 11월, NVIDIA GPU 가속기를 탑재한 NUDT의 Tianhe-1A 계산시스템이 Top 500 순위에서 1위를 달성하였으며, 2013년 6월에는 Intel Xeon Phi 가속기를 탑재한 후속 시스템 Tianhe-2가 다시 1위를 달성함
  * GPU 가속기를 사용한 수치 라이브러리 프로그램인 DGEMM의 개발에 있어서 기존의 NVIDIA 회사보다 높은 성능을 달성하는 프로그램을 구현하여 고성능컴퓨팅 분야의 정상급 컨퍼런스인 Supercomputing Conference에 논문을 발표함

- 일본 동경공업대학의 Satoshi Matsuoka 교수의 연구 그룹은 GPU 가속기를 사용한 저전력 클러스터의 연구 개발과 이를 이용한 대규모 고성능 시뮬레이션에 관한 연구를 진행중임
  * TSUBAME 2.0은 NVIDIA GPU 가속기를 탑재한 가속기 기반 초고성능컴퓨터이며, 이를 이용한 합금의 구조 시뮬레이션 계산이 2.0 PFLOPS의 성능을 달성하여 2011년 고성능컴퓨팅 분야에서 가장 권위있는 상중에 하나인 골든 벨(Gordon Bell Prize) 특별상을 수상함
  * 40 노드 (노드 당 4장의 GPU)로 구성된 고밀도 GPU 가속기 기반 클러

스터인 TSUBAME-KFC는 순환식 오일 냉각액을 사용하여 높은 전력 성능비를 달성하였으며 2013년 11월 Green 500 순위에서 1위에 랭크됨

• 5. 국내 동향

- 한국과학기술정보연구원 (KISTI)에서는 Intel Many Integrated Core Architecture Program을 통하여 Intel Xeon Phi 가속기 기반의 클러스터 구축과 병렬화 연구를 수행함
  * Intel로부터 아시아 태평양 지역 최초로 IPCC(Intel Parallel Computing Center)사업 선정되어 Intel Xeon Phi기반의 대용량 슈뢰딩거 방정식 계산의 최적화 및 이를 이용한 차세대 반도체소재 특성 계산 연구를 수행 중에 있음
  * Intel로부터 발매 이전의 가속기를 공급받아 8 노드의 클러스터를 구축, 각 노드에 2장의 Intel Xeon Phi 가속기를 장착하여 총 16 TFLOPS의 성능을 달성함
  * QMC(Quantum Monte Carlo) 모사와 K-Molecular Dynamics (KMD) 솔버의 Intel Xeon Phi 가속기 기반의 병렬화를 수행하여 CPU 대비 각각 20배와 34배의 성능 향상을 달성함

- 서울대학교의 이재진 교수의 연구 그룹은 GPU 가속기를 이용한 초고성능 컴퓨터의 개발과 이를 이용하는 프로그래밍 모델에 관한 연구를 진행 중임
  * GPU 클러스터 천둥 시스템은 HPL(High Performance Linpack) 벤치마크 Suite을 이용하여 106.8FLOPS의 성능을 달성하여 2012년 11월에 TOP 500 순위에 277위로 등재됨 (천둥 시스템의 TOP 500 등재는 GPU 클러스터로서 한국 최초 사례)
  * 각 계산 노드에 4개의 AMD GPU 가속기 카드를 탑재하여 2013년 6월의 Green 500 순위에서 36위를 달성하여 높은 전력 대비 성능을 증명함
  * 가속기 병렬 프로그래밍 모델인 OpenCL을 가속기 기반 초고성능컴퓨터에서 사용 가능하도록 확장한 프로그래밍 모델 SnuCL을 개발중임

- 한국전자통신연구원 (ETRI)에서도 가속기 기반 초고성능컴퓨터 마하(MAHA) 시스템을 도입
  * 53대의 계산노드로 구성되어 있으며, 각 노드에 2장의 NVIDIA GPU (M2090) 가속기, 혹은 2장의 Intel Xeon Phi 가속기를 탑재함
  * GPU 가속기 기반의 계산 노드 성능이 53.2 TFLOPS, Intel Xeon Phi 가속기 기반 노드 성능이 51.3 TFLOPS이며, 이를 합한 시스템 총성능이 104 TFLOPS를 달성함
  * CPU, GPU 가속기, Intel Xeon Phi 가속기와 같은 다양한 계산 자원을 이용하여 단백질, 유전체 분석을 수행

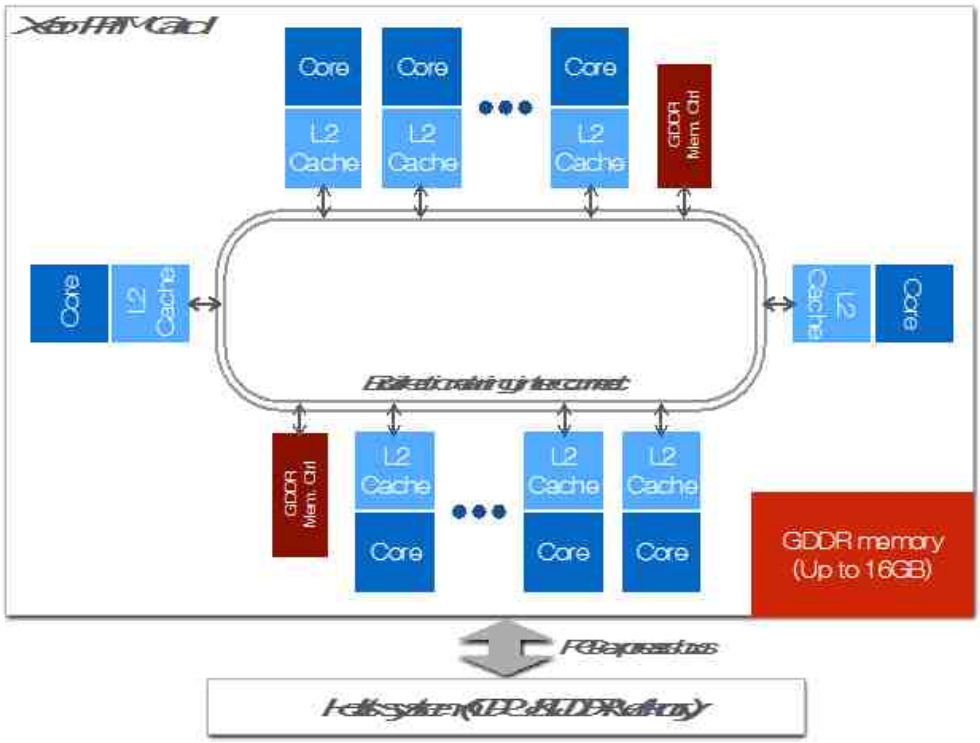
• 6. Xeon Phi 프로그래밍 모델 분석

- 하드웨어 개요

Xeon Phi 가속기와 범용 CPU의 하드웨어 사양 비교

	Xeon Phi 가속기	범용 CPU
명칭	Intel Xeon Phi Coprocessor 7120P	Intel Xeon E5-2670 v2
코어수	61개	20개
동작주파수	1,238 GHz	2.5 GHz
최대성능	1208 Gflops (Double Precision) 코어 당 19.8 Gflops	400 Gflops (Double Precision) 코어 당 20 Gflops
메모리 용량	16 GB	768 GB (최대)
메모리 성능	352 GB/s	59.7 GB/s
소비전력	300 W 전력 대비 성능: 4 Gflops/W	115 W 전력 대비 성능: 3.48 Gflops/W

* Intel Xeon Phi 제품군은, Intel에서 개발된 Many-Integrated Core(MIC) 아키텍처를 사용하는 보조연산장치(coprocessor)
* 보조연산장치는 중앙처리장치(CPU)의 계산량의 일부를 분배받아 계산을 수행하는 장치들을 통칭. 그래픽 카드 등과 같은 독립적인 카드 형태로 구현되어 있으며 PCI Express 인터페이스를 통하여 메인보드에 접속됨
* 일반적인 범용 CPU와 비교하여 성능이 낮으나 전력 효율이 높은 계산 코어를 다수 탑재하여 시스템 전체의 성능을 향상시키는 매니코어(many-core) 아키텍처. Pentium아키텍처에 기반한 수십(현재 출시된 제품군의 경우 61개) 개의 CPU 코어를 탑재
* 계산 코어는 Intel Pentium 프로세서를 기반으로 설계되어 있기 때문에 명령 체계 및 하드웨어 구조에서 유사성을 가짐. 코드들의 전면적인 수정 없는 사용이 가능하며, 많은 수의 코어를 활용한 병렬 계산 속도에서의 이득을 얻을 수 있을 것이라 기대할 수 있음
* 사용하고자 하는 코드의 성격에 따라서, 몇 가지의 서로 다른 병렬화 구현 모델을 사용할 수 있다는 것 또한 Xeon Phi가 기존의 다른 보조연산장치들에 비해 갖는 이점임
* 카드 내부에 독립적인 메모리를 가지며 호스트 메모리를 가속기가 직접 참조하는 것은 불가능함
* 범용 CPU와 비교하여 우수한 소비전력 대비 성능을 달성



개략적인 Xeon Phi위에서의 코어와 메모리, 그리고 host 시스템과의 연결 구조

* Intel Xeon Phi coprocessor 7120P 제품
  . 1.24GHz의 클럭 속도로 구동되는 CPU 코어를 61개 탑재하고, 이들은 서로 16GB의 메모리를 공유
  . 각 코어는 advanced vector extension (AVX) 명령어 아키텍처를 통해 SIMD(Single Instruction Multiple Data) VPU(Vector Processing Unit)을 갖고 있기 때문에 클럭 사이클 한번에 16개의 single-precision(SP) 또는 8개의 double-precision(DP) 부동소수점 값을 동시에 연산할 수 있고, 따라서 61개의 코어를 모두 사용할 경우 single-precision의 경우 2.4 teraflops의 이론 성능
  . 수십 개의 코어와 메모리 사이의 효율적인 통신을 위해서, 위의 그림에서 보이는 것처럼 코어들과 메모리들은 양방향으로 연결되는 고리 모양의 네트워크를 통해 효율적으로 데이터를 주고 받게 되며, 또한 그래픽 연산장치들과 같이 PCI-express 버스를 통해 host 시스템의 주 메모리 및 CPU와 통신함

- 소프트웨어 환경
  * Intel 컴파일러, 디버거 등의 개발 도구를 Xeon Phi 가속기에서도 그대로 사용 가능하기 때문에 습득이 용이하며, 범용 CPU와 소스 코드 레벨의 호환성을 가짐

* Xeon Phi 가속기의 내부에서 CPU와 독립적으로 Linux OS가 동작
  . 기존 Linux와 동일한 방법으로 설정, 관리가 가능함
  . CPU 측에서 Xeon Phi의 Linux OS에 *ssh*를 이용한 네트워크 접속이 가능.
* 소프트웨어 실행 모델
  . Offload 모델: 가속기를 CPU가 통제하는 가속기로 취급하는 실행 모델. 코드는 CPU에서 실행 되며, 계산의 일부를 가속기 카드에 전송하여 고속화
    - 컴파일러는 자동으로 또는 사용자가 소스 코드에 지정한 부분을 Xeon Phi 카드에서 실행
    - 자동 offload(AO: automatic offload): 계산 크기가 큰 일부 LAPACK/BLAS 루틴들 (BLAS: ?GEMM, ?TRSM, ?TRMM, LAPACK: ?GETRF, ?GEQRF, ?POTRF)을 사용자의 지시 없이도 자동으로 offload를 수행. 사용자가 코드를 직접 수정하지 않아도 되기 때문에 Xeon Phi에서의 병렬화가 용이하지만, 호스트와 Xeon Phi간 전환시 메모리 값 복사에 따른 추가 시간이 소모되어 효율이 떨어질 수 있음. 추후 FFT등 자동 offload을 지원하는 루틴들이 확대될 수 있음. 사용자가 직접 offload되는 부분을 지시할 경우, 좀 더 병렬화 성능을 끌어올릴 수 있다는 장점이 있음
  . Native 모델: 가속기를 CPU와 네트워크를 통하여 접속된 독립된 계산기로 취급하는 실행 모델. CPU는 계산에 참여하지 않으며, 가속기가 필요로 하는 I/O 등의 기능만을 제공함
    - 기존에 출시된 다른 보조연산장치들과는 달리, Xeon Phi는 보다 일반적인 연산이 가능한 CPU 코어들을 탑재하고 있음. 장치 내 자체적인 리눅스 OS를 구동할 수 있으며, 따라서 Xeon PhiTM카드를 수십개의 코어를 탑재한 독립적인 하나의 컴퓨터로서 다룰 수 있음
    - 따라서, 구동하고자 하는 코드와 라이브러리들을 모두 Xeon Phi 위로 복사해 온 후, 호스트와의 통신 없이 Xeon Phi 카드 위에서만 계산을 수행할 수 있음
    - 호스트와 Xeon Phi 사이의 통신에 따르는 추가 시간을 없앨 수 있기 때문에 계산의 효율을 끌어올릴 수 있지만, 일반적인 CPU용으로 작성된 코드가 Xeon Phi 환경에서 효율적으로 작동하기 위해서는 벡터화등의 추가적인 최적화 작업이 필요함
  . Symmetric 모델: CPU와 가속기가 동시에 계산에 참여하는 실행 모델. 가속기에서는 Native 모델과 동등한 기능을 사용 가능함
    - 파일 입출력이 많은 경우에 CPU에서만 파일 입출력을 수행하게 하면 성능향상이 가능함 (Xeon Phi에서는 파일 입출력이 느림)

- 프로그래밍 특징

  * 가속기의 성능을 충분히 끌어내기 위하여 다음의 성능최적화 및 병렬화 작업이 필요함

    . 벡터화: 단일 계산코어의 성능을 충분히 달성하기 위하여 성능이 우수한 벡터 명령을 사용하도록 유도

    . 다중쓰레딩: 계산을 다수의 스레드로 분할하여 계산 코어에 배분함으로써 장치 내부의 모든 코어를 활용

    . 오프로딩: Offload 실행 모델의 경우, PCI Express를 통한 CPU와 가속기간 통신을 적절하게 이용하여 통신 성능을 최적화

  * Xeon Phi 가속기는 Intel CPU와 유사한 아키텍처를 가지기 때문에 Intel 프로세서의 성능 최적화 및 병렬화에 관한 지식과 경험이 Xeon Phi에서 공유될 수 있음

  * Xeon Phi는 기존의 CPU 또는 GPU와는 다른 내부 구조를 가지고 있으며, 그 성능을 최대한 끌어내기 위해서는 기존의 프로그래밍 모델과는 다른 종류의 최적화 방법론을 사용해야 함

  * Xeon Phi는 CPU에 비해 낮은 클럭 동작속도를 보이며, 또한 GPU에 비해 적은 수의 연산 단위를 갖고 있다. 여기에서 오는 불리함을 만회하기 위해서는, 위에서 말한 것과 같은 벡터 연산이 효율적으로 작동하도록 데이터들을 최적화하는 작업이 필요

  * 하드웨어에서의 상이함에 따르는 코드 작성 및 최적화에서의 어려움을 덜기 위해서, 최근 제공되는 Intel 컴파일러 제품군들은 (Intel® Composer XE 2013 이후) Xeon Phi를 위한 자동 최적화 옵션들을 제공함

- 프로그래밍 모델별 비교

Xeon Phi 프로그래밍 모델별 비교

	MKL	Cilk Plus	OpenMP/ OpenACC	OpenCL
설명	수치 라이브러리	Intel 독자적인 병렬 모델	directive를 이용한 모델	CUDA와 유사한 프로그래밍 모델
벡터화	내부 구현	수동	수동/컴파일러 지원	수동
다중쓰레딩	내부 구현	수동	수동	수동
오프로딩	자동	자동	수동	수동
장점	- 높은 성능 - 가속기 지식 필요 없음	- 추상도 높은 범용 모델	- 컴파일러 간의 호환성 보장 - 코드 수정이 적음	- 성능 최적화의 자유도가 가장 높음
단점	- 이용 범위 제한적	- Intel 컴파일러 독자 기능	- Directive 사용의 타당성을 사용자가 보장	- 프로그래밍 부담이 가장 큼
이용 난이도	하	중	중	상

* Math Kernel Library (MKL)

. 행렬, 벡터의 연산 등에 관한 루틴을 가지는 수치 라이브러리

. 벡터화, 다중쓰레딩 등의 작업이 적용된 버전이 제공됨

. 가속기 프로그래밍 지식 없이 전문가에 의하여 고도로 최적화된 코드를 이용 가능함

. 특정한 연산을 대상으로 하기 때문에 적용 범위가 제한적임

* Cilk Plus

. C/C++를 독자적인 키워드로 확장한 Intel의 독자적인 프로그래밍 모델

. 벡터화, 다중쓰레딩은 Cilk Plus의 독자적인 언어 구문을 이용하여 기술

. 오프로딩 실행 모델의 경우, 소프트웨어가 자동으로 CPU-가속기 간의 메모리 전송을 수행

. 현재 Intel 컴파일러만이 Cilk Plus의 언어 구문을 지원하고 있기 때문에 소스 코드의 호환성이 저하됨

* OpenMP/OpenACC
  . C/C++, Fortran 코드에 주석문과 유사한 directive를 추가하여 컴파일러에게 가속기 병렬화를 지시하는 프로그래밍 모델
  . 벡터화는 컴파일러에 의하여 자동으로 수행되거나 유저가 명시적으로 지시할 수 있으며, 기타 기능은 해당 directive를 소스코드에 기술하여 구현
  . OpenMP/OpenACC의 directive 사양을 따르는 컴파일러 사이에서 소스코드 레벨의 호환성이 보장됨
  . MKL과 같은 라이브러리 이용을 제외하면, 소스 코드의 수정이 다른 프로그래밍 모델과 비교하여 제일 작음
  . Directive가 올바르게 사용되었는지에 유저가 그 타당성을 보장하여야 하며, 이를 위하여 가속기와 프로그래밍 모델에 관한 이해와 지식이 필요함.
* OpenCL
  . 매니코어 아키텍처의 병렬화를 기술하기 위한 프로그래밍 API
  . C/C++에서 이용한 라이브러리와 C언어 기반의 커널 프로그래밍 언어로 구성됨
  . 다중쓰레딩과 오프로딩은 OpenCL 라이브러리를 이용하여 기술
  . 벡터화는 독자적인 커널 프로그래밍 언어로 기술
  . Intel, NVIDIA, AMD 등의 업체가 OpenCL를 지원하고 있기 때문에 소스코드 레벨의 호환성이 보장됨
  . 모든 작업이 수동으로 수행되어야 하기 때문에 성능 최적화의 자유도가 가장 높으나, 동시에 필요한 지식과 프로그래밍 부담도 제일 큼

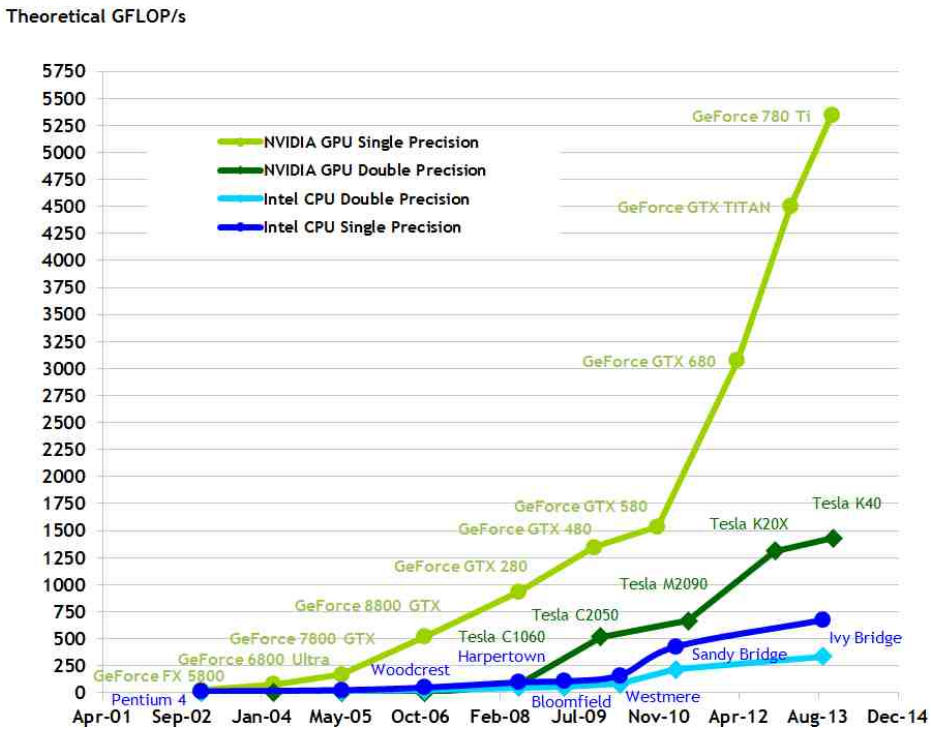
· 7. NVIDIA GPU 프로그래밍 모델 분석

- 하드웨어 개요

NVIDIA GPU와 Xeon Phi 가속기의 하드웨어 사양 비교

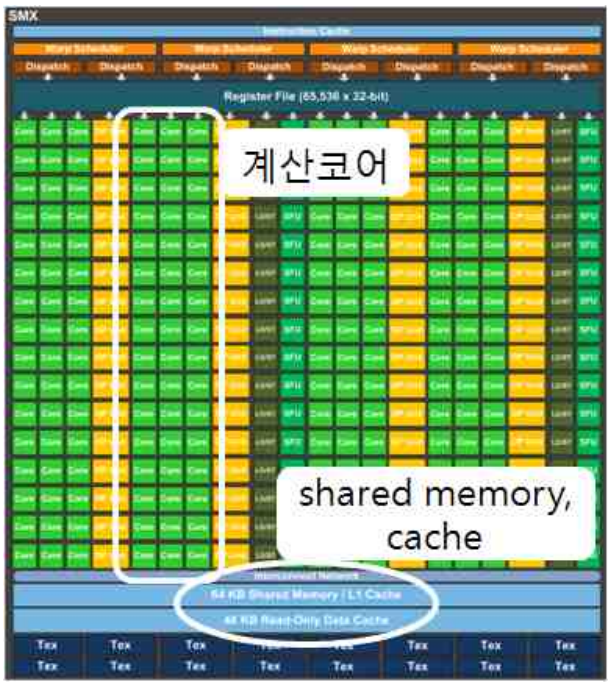
	NVIDIA GPU	Xeon Phi 가속기
명칭	NVIDIA Tesla K40m	Intel Xeon Phi Coprocessor 7120p
코어수	2880개	61개
최대성능	1.43 Tflops	1.208 Tflops
메모리 용량	12 GB	16 GB
메모리 성능	288 GB/s	352 GB/s
소비전력	235 W	300 W

* 기존의 Graphic Processing Unit (GPU)는 화상처리를 전담하는 장치로 개발되었으며, 2D/3D 도형의 좌표 계산 및 변환, 표시 등의 작업을 수행하는데 주로 사용되어 왔음. CPU의 경우, 과거에 개발된 프로그램과의 호환성 유지를 위하여 아키텍처의 개선이 제한되나, GPU 아키텍처는 이와 같은 제한이 적기 때문에 세대 간의 획기적인 구조 개선에 의하여 성능이 비약적으로 발전되어 왔으며, 현재도 CPU와의 성능 차이가 점점 벌어지고 있음



각 세대별 CPU와 GPU의 성능 차이

* 화상 처리 용도로만 사용되던 GPU의 성능이 CPU와 큰 차이를 보이며 발전함에 따라서, GPU를 계산 용도로 이용하고자 하는 요구가 증가하였으며, 이에 대응하여 NVIDIA, AMD 등의 GPU 제조사에서 GPU를 이용한 범용 프로그래밍 모델을 제공하게 되었음. 범용 GPU 프로그래밍 모델을 이용하여 화상처리 이외의 분야에서 GPU가 가지는 높은 성능을 이용할 수 있음
* Intel Xeon Phi와 유사하게 그래픽 카드 등과 같은 독립적인 카드 형태로 구현되어 있으며 PCI Express 인터페이스를 통하여 메인보드에 접속됨
* 카드 내부에 독립적인 메모리를 가지며 CPU의 메모리를 가속기가 직접 참조하는 것은 불가능함
* 범용 CPU와 비교하여 우수한 소비전력 대비 성능을 달성
* 현대의 CPU가 2~16개의 계산 코어를 가지는 것과는 달리, GPU는 CPU보다 단순한 구조를 가진 수십~수백개의 계산 코어를 가지고 있으며 이를 many-core 아키텍처라고 부른다.
* 다음 그림은 NVIDIA GPU의 기본 계산 단위를 나타낸 것이며 계산 코어의 집합체인 SMX의 아키텍처를 표시. 1개의 SMX는 192개의 계산 코어를 가짐



NVIDIA Kepler SMX 아키텍처

* 같은 SMX 내부에 존재하는 계산 코어는 64KB의 메모리를 공유하며, 이

를 shared memory. 사용자가 데이터 배치와 이동을 자유롭게 프로그래밍 가능하며 일부를 CPU의 cache와 같이 사용하는 것도 가능

* 그림에서도 알 수 있듯이, GPU는 메인보드에 PCI Express Interface로 접속되는 카드 형태로 제공되며, 카드 내부에 CPU 메모리와 독립적으로 존재하는 GPU 메모리를 가짐. GPU에서 계산을 수행하기 위해서는 CPU 메모리에 저장되어 있는 데이터를 GPU에 전송하여 계산을 수행한 후, 다시 CPU메모리에 전송하는 작업이 필요함

- 소프트웨어 환경
  * CPU와 GPU는 기본적으로 다른 하드웨어 구조를 가지기 때문에, GPU 가속 코드 안에는 CPU 이진 코드와 GPU 이진 코드가 포함되어 있음.
  * GPU는 컴퓨터 시스템에 장착된 부가 장치이므로, 시스템에서 관리 및 제어를 하는 형태.
  * 소프트웨어 실행 모델
    . CPU가 응용프로그램의 모든 제어권을 가지고 있고 필요에 따라 GPU에 계산을 넘겨 수행시키는 Offload 모델이나, 최근에 소개된 Dynamic Parallelism 기술을 이용하면, 대부분의 계산을 GPU에서 수행하는 형태가 가능함.
- 프로그래밍 특징
  * 가속기의 성능을 충분히 끌어내기 위하여 다음의 성능최적화 및 병렬화 작업이 필요함
    . 워프(Warp) 특성 고려: NVIDIA GPU는 32개의 쓰레드 단위인 워프가 동일 명령어를 수행하므로, 한 워프가 동일한 제어 흐름(Control Flow)을 갖도록 프로그래밍 하는 것이 중요함
    . 계층적 메모리 구조: 워프안의 쓰레드들이 연속적인 메모리를 참조할 때에 메모리 접근 비용을 최소화 할 수 있으므로, 이를 위한 자료구조 설계 및 알고리즘 개선이 필요. 또한 latency가 작은 cache 메모리 및 shared memory를 잘 활용하여 데이터 접근 비용을 최소화할 수 있음
    . CPU-GPU간의 데이터 통신: CPU-GPU간의 데이터 통신을 비동기적으로 하여, GPU의 계산과 데이터 통신을 동시에 수행하여 통신 비용을 최소화하는 것이 중요

- 프로그래밍 모델별 비교

NVIDIA GPU 프로그래밍 모델별 비교

	Drop-in 라이브러리	CUDA	OpenACC	OpenCL
설명	코드 수정없이 GPU 가속 라이브러리를 사용 가능	NVIDIA에서 제안한 C/Fortran 확장 형태의 언어	Directive를 이용한 모델	CUDA와 유사한 프로그래밍 모델
오프로딩	자동	수동	자동	수동
장점	- 높은 성능 - 가속기 지식 필요 없음 - 코드 수정없이 GPU 가속 가능	- GPU를 저수준으로 프로그래밍 가능 - 높은 수준의 성능최적화 가능	- 코드 이식성 높음 - 시간적인 코드 개발 비용 낮음	- 대부분의 매니코어 아키텍처에서 사용가능 - 런타임시에도 코드 수정 및 적용가능
단점	- NVIDIA GPU에서만 가능 - 이용 범위 제한적	- NVIDIA GPU에서만 가능 - 프로그래밍 난이도 높음	- Directive 사용의 타당성을 사용자가 보장	- 프로그래밍 난이도가 높음 - CUDA에 비해 성능효과는 낮음
이용 난이도	하	상	중	상

* Drop-in Library

. CPU 라이브러리 호출을 사전에 메모리에 로드되어 있던 GPU 라이브러리가 가로채 수행하는 형태

. 사용자의 코드 수정 없이 간단한 환경변수 설정으로 GPU 가속이 가능

. 현재 NVIDIA에서 제공하는 BLAS, FFT 수치 라이브러리가 해당 기능을 제공함

. 가속기 프로그래밍 지식 없이 전문가에 의하여 최적화된 코드를 이용 가능함

. 특정한 연산을 대상으로 하기 때문에 적용 범위가 제한적임

. FFTW 라이브러리를 사용한 C로 작성된 코드에서는 코드 수정없이 cuFFT의 header파일을 추가하고 컴파일시 라이브러리 링크만으로

cuFFT 사용가능

. 하지만 Drop-in 사용시 성능을 올리기 위해서는 데이터 전송, 타입 등에 대한 최적화 필요

. cuFFT 라이브러리는 C만 지원하므로 Fortran에서는 C-Fortran bind기능을 이용해서 사용가능

* CUDA C/Fortran

. 기존의 C/Fortran을 GPU 가속이 가능하도록 확장한 형태

. GPU를 저수준으로 프로그래밍할 수 있기 때문에 강력하나, 프로그래밍 난이도가 높고 시간적 개발 비용이 많이 소요

. CUDA C는 NVIDIA에서 배포하는 CUDA 툴킷에 컴파일러가 포함되어 있음. CUDA Fortran은 PGI 컴파일러에서 지원

* OpenACC

. C/C++, Fortran 코드에 주석문과 유사한 directive를 추가하여 컴파일러에게 가속기 병렬화를 지시하는 프로그래밍 모델

. 소스코드 레벨의 호환성이 보장되고, CUDA에 비해 플랫폼의 영향을 받지 않아 이식성이 높음

. CUDA에 비해 프로그래밍 난이도가 낮고 시간적 개발 비용이 적음

. 현재 CAPS, PGI, Cray 컴파일러에서 OpenACC를 지원

. Directive의 타당성 유무를 사용자가 보장하여야 하며, 이를 위하여 가속기와 프로그래밍 모델에 관한 이해와 지식이 필요함

* OpenCL

. 매니코어 아키텍처의 병렬화를 기술하기 위한 프로그래밍 API

. C/C++에서 이용한 라이브러리와 C언어 기반의 커널 프로그래밍 언어로 구성됨

. 런타임 상에서 OpenCL 코드를 컴파일하기 때문에 실시간 코드 수정 및 수행이 가능함

. Intel, NVIDIA, AMD 등의 벤더가 OpenCL를 지원하고 있기 때문에 소스코드 레벨의 호환성이 보장됨

. 대부분의 매니코어 아키텍처에서 사용하기 위해 표준화 및 추상화되어 있으며 동일한 프로그래밍 난이도에 불구하고 CUDA에 비해 성능최적화로부터 얻는 기대효과는 낮은 편임

• 8. 실제 응용 코드의 컴퓨팅 모델별 테스트

- CPU 기반 계산시스템에서 수행하는 in-house 사용자코드를 가속기 기반 시스템으로 이식 및 성능테스트

프로그램	NVIDIA GPU		Xeon Phi	
	이식성	성능평가	이식성	성능평가
Particle-In-Cell 기반 Magnetron Sputtering	CUDA로 개발	원본코드 대비 40배 향상	-	-
천체물리 N-Body 모사코드 GOTPM	CUDA로 개발	GPU 병렬화 부분 97배, 전체 실행 시간 4.3배 향상	-	-
비등방성 다상장모델 코드	CUDA로 개발	단일노드에서 CPU 대비 4.16배 향상	Native로 수행	단일노드에서 CPU 대비 1.04배 향상
Lattice QCD 코드	CUDA로 개발	단일노드에서 CPU 대비 147.53배 향상, 노드 8개까지 병렬확장성 보장	Native로 수행	단일노드에서 CPU 대비 9.3배 향상, 노드 2개부터 병렬확장성 보장 안됨
모바일센서데이터 분석코드	CUDA로 개발	단일노드에서 CPU 대비 약 3배, 12노드의 CPU대비 약 2배 향상	OpenCL로 Offload 수행	단일노드에서 CPU대비 3배, 12노드 CPU대비 약 2배 향상

- 가속기 기반 환경에서 범용 수치라이브러리 이식성 및 성능테스트

프로그램	NVIDIA GPU		Xeon Phi	
	이식성	성능평가	이식성	성능평가
FFT 라이브러리	cuFFT(CUDA)	15.48배 향상 (1D FFT, 사이즈 16,777,216)	Native/Offload	Offload에서 1.13배 성능저하, Native에서는 3.09배 향상 (1D FFT, 사이즈 16,777,216)

- 4호기 슈퍼컴퓨터에서 빈번하게 사용하는 응용코드의 BMT(Benchmark Test)

순위	프로그램명	분야	비고	가속기 지원 테스트
1	VASP	나노물리/양자화학	상용	Xeon Phi: Native/AO NVIDIA GPU: CUDA 구현물 (비공개)
3	Amber	분야동역학	상용	NVIDIA GPU: CUDA 구현물 ※ Xeon Phi: 14버전부터 Offload지원
4	Gromacs	분야동역학	오픈소스	Xeon Phi: Native NVIDIA GPU: CUDA 구현물
5	Lammps	분야동역학	오픈소스	Xeon Phi: Offload NVIDIA GPU: CUDA 구현물
8	Quantum Espresso(QE)	나노물리/양자화학	오픈소스	Xeon Phi: Native/AO NVIDIA GPU: CUDA 구현물
10	WRF	대기과학	오픈소스	Xeon Phi: Native/Symmetric NVIDIA GPU: OpenACC 구현물

- 응용 코드 및 in-house 이식성 및 성능테스트 요약

응용	NVIDIA GPU		Xeon Phi	
	이식성	성능평가	이식성	성능평가
VASP	CUDA 개발중 (비공개)	단일노드에서 CPU대비 1.77배 향상	Native(단일노드), AO(실패)	CPU대비 9배 성능저하(원자수 52개)
AMBER	CUDA 지원 (GPU자원만)	K40카드 한 장이 120 CPU cores(6 nodes) 대비 1.34배 향상 (단백질 접힘, ECC off)	Amber 14부터 Offload 지원	-
GROMACS	CUDA 지원	단일노드에서 2.15배 향상(512k 물분자)	5.0-rc1 버전부터 Native 지원	단일노드에서 Native시 1.29배 저하, Symmetric시 1.2배 향상 (512k 물분자)
LAMMPS	CUDA 지원 (GPU/CUDA 패키지)	- GPU 패키지: CPU 계산(20cores)대비 1GPU 추가시 약 2.26배 향상 - CUDA 패키지: CPU 계산(20cores)대비 1GPU추가시 약 1.55배 향상	15 Aug. 2014버전부터 Offload 지원	CPU 계산(20cores)대비 20ppn+1MIC는 약2.93배 향상
Quantum Espresso	CUDA 지원	단일노드에서 CPU(20cores)대비 1.94배 성능저하	Native 지원	단일노드에서 Xeon Phi Native 사용시 CPU(20cores)대비 2.35배 성능감소
WRF	CUDA/	단일노드에서	Native/	- 단일노드에서 CPU대비

	OpenACC 개발중 (비공개)	CPU(2cores) 대비 GPU(OpenACC, MPS 동작, 1GPU추가) 사용 1.36배 향상 (CONUS 18km)	Symmetric 수행가능	Native는 1.27배 감소, Symmetric은 1.52배 향상 (CONUS 12km) - CPU대비 Symmetric시 4노드 1.7배 향상 (CONUS 2.5km)
Magnetron Sputtering(in-house)	CUDA로 개발	원본코드 대비 40배 향상	-	-
GOTPM (in-house)	CUDA로 개발	GPU 병렬화 부분 97배, 전체 실행 시간 4.3배 향상	-	-
다상장 모델 코드(in-house)	CUDA로 개발	단일노드에서 CPU 대비 4.16배 향상	Native로 수행	단일노드에서 CPU 대비 1.04배 향상
Lattice QCD (in-house)	CUDA로 개발	단일노드에서 CPU 대비 147.53배 향상, 노드 8개까지 병렬확장성 보장	Native로 수행	단일노드에서 CPU 대비 9.3배 향상, 노드 2개부터 병렬확장성 보장 안됨
모바일센서데이터 (in-house)	CUDA로 개발	단일노드에서 CPU 대비 약 3배, 12노드의 CPU대비 약 2배 향상	OpenCL로 Offload 수행	단일노드에서 CPU대비 3배, 12노드 CPU대비 약 2배 향상
FFT	cuFFT(CUDA)	15.48배 향상 (사이즈 16,777,216)	Native/Offload	Offload에서 1.13배 성능저하, Native에서는 3.09배 향상 (사이즈 16,777,216)

- BMT를 통한 컴퓨팅 모델별 이식성 및 성능을 평가함으로서 차세대 컴퓨팅시스템에서의 사용자지원 용이성 파악

	NVIDIA GPU	Xeon Phi
성능	**○CUDA 구현물이 성능면에서 강점이 많음** - QE를 제외하고 CPU대비 성능 강점 - VASP, GROMACS는 Xeon Phi대비 각각 15.93배, 1.79배 빠름 - 노드 내에서 다중 코어에서 하나의 가속기 카드로 동시실행시 MPS 사용시 성능향상: WRF, GROMACS, GOTPM(in-house) - 멀티 노드에서 GPU간의 데이터 통신시 GPUDirect 사용 성능향상	**○벡터화가 잘 되는것이 관건 => Offload 방법으로 새로 구현하는 구현물은 강점을 보임** - LAMMPS(offload)는 CUDA 대비 1.3배 빠름 (Charmm, Gayberne 포텐셜만 지원) - GROMMACS, QE는 Offload 방법으로 개발 중이고 LAMMPS, AMBER, NAMD가 개발되어 있음
이식성	**○CUDA로 이식된 구현물이 더 많으며, 기능도 많이 구현** **○CUDA 구현자체는 아직 구현이 어렵지만, 이식성과 편의성에 초점을 맞춘 솔루션(OpenACC, Drop-in) 제공** - WRF에서 OpenACC 구현물에 대해서 CPU대비 1.36배 향상 - Drop-in 라이브러리인 경우는 성능보다는 편의성 위주로 제공됨	**○Native/Symmetric 방법으로 수정없이 실행 가능** - 단일 노드에서 벡터화가 잘 되지 않고, 파일입출력이 많으면 성능저하 발생 (in-house,WRF,FFT를 제외하고 성능저하) - 멀티 노드에서 Inter-connect 및 PCI-express 오버헤드로 인한 성능저하 발생 - Symmetric 사용시 파일입출력이 호스트에서만 발생하면 CPU대비 성능향상 (WRF: 1.7배 향상) **○AO방법으로 제한적으로 실행 가능** - Drop-in보다도 편하게 사용가능하지만 BLAS 라이브러리의 함수들에서 제한적으로 사용가능하고, 성능 향상을 기대하기 어려움 (QE)

<부록: Xeon Phi에서의 병렬프로그래밍 모델에 따른 성능 비교>

※ C/C++언어로 병렬 프로그램 작성시 가장 흔하게 사용하는 세가지 프로그래밍 모델인 OpenMP, Intel Cilk Plus, Intel TBB 이용해서 Xeon Phi에서의 성능 비교를 수행

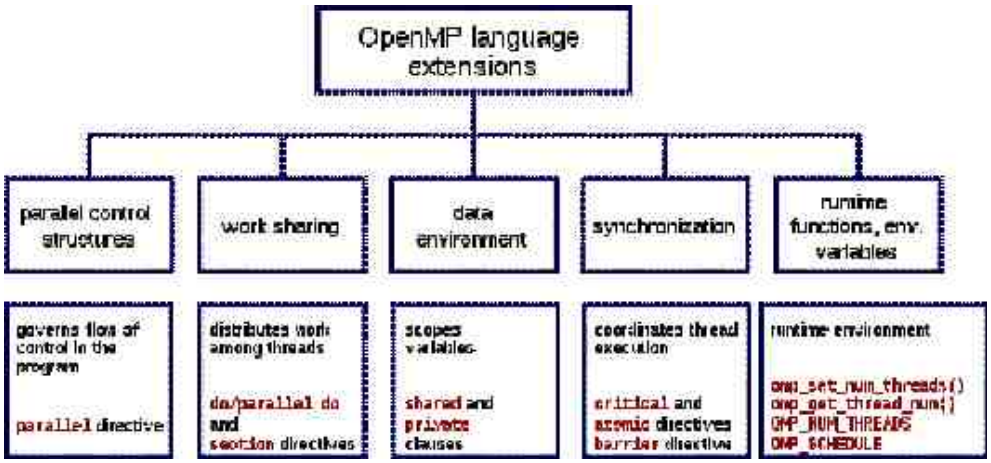
1. 3개 프로그래밍 모델 소개

1.1 OpenMP

OpenMP는 여러개의 프로세스가 공유된 메모리를 참조하는 환경에서 다중 스레드 병렬 프로그래밍을 위한 표준 스펙으로OpenMP 3.0의 출시 이후에는 작업의 관리 기능 또한 제공하고 있다. OpenMP의 주요 동기요인은 성능, 확장성, 이식성, 표준성을 목표로 한다. 그리고 C, C++(#으로 시작), Fortran(90에서는 !$로 시작)을 지원하고 있다.

OpenMP를 주로 사용하는 방법은 프로파일링을 통해 시간을 많이 차지하는 부분을 식별하고 해당 프로그램에서 사용되는 의존성, 데이터의 유효 범위를 예측한 다음 지시어의 설정을 통해 병렬 처리를 하여 성능 향상을 얻는 방식으로 진행된다.

. OpenMP 구조



OpenMP의 구조

-Directives(컴파일러 지시자)

쓰레드 사이의 작업 분담, 통신, 동기화를 담당을 한다. 컴파일러가 지시자를 참고하여 다중 쓰레드 코드를 생성하게 된다. Work-Sharing 지시자는 상세히 설명할 필요가 있어서 아래에 새로운 단락으로 빼서 설명하고 여기서는 동기화, 데이터 유효 범위 지시자를 설명한다. OpenMP 프로그래밍은 컴파일러가 모두 알아서 처리하는 게 아니라 동기화 및 의존성 제거 등의 작업은 사용자가 직접 소스코드에 기술해

주어야 하는 어려움을 지니고 있다.

-Runtime Library

병렬화에 직접 참여는 하지 않지만 실행 환경에서 병렬 매개 변수(참여 스레드의 개수, 번호 등)의 설정과 조회를 통해 병렬화 정보를 변경 조회할 수 있다.

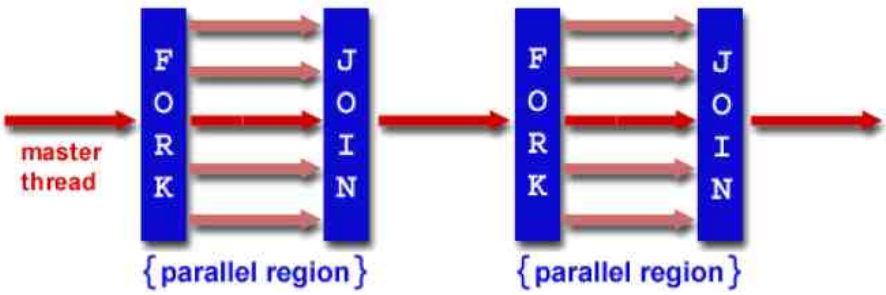
-Environment Variables

Runtime Library와 중복되는 부분이 많지만 우선순위는 동일한 지시자를 반복 사용했을 경우 Runtime Library가 우선된다. 주요한 일은 실행 시스템의 병렬 매개 변수(스레드 개수 등)를 설정해 실행시에 제어를 한다.

. 프로그래밍 모델

-Fork-Join 모델

스레드가 병렬 구문를 만나면 스레드는 그 자신을 포함한 추가적인 스레드(0개 이상의)로 이루어진 쓰레드 팀을 만든다(Fork). 병렬 구문를 만난 스레드는 팀의 마스터 쓰레드를 호출 하게 되고(Join) 다른 스레드들은 팀의 slave 스레드라고 불리어진다.



Fork-Join 방식의 개념도

-Work-Sharing 모델

같은 작업을 쓰레드별로 실행하는 것이 아니라 작업을 분할해서 쓰레드별로 나누어서 실행하는 것을 말한다.

• for

바로 뒤에 오는 for 루프의 반복 실행을 쓰레드에 분배한다. 그리고 루프 끝에 암시적 장벽(동기화)이 존재한다. 이를 피할려면 nowait 사용하면 된다.

데이터 병렬화시에 활용된다.

• sections

독립된 여러 개 작업을 각 스레드에 분산 실행한다. 그리고 sections 구문 끝에 암

시적 장벽(동기화)이 존재하고 장벽을 피하기 위해서는 nowait를 사용한다. 주로 기능적 분할에 이용된다.

• single

병렬 영역 내부에서 하나의 스레드만을 이용해 작업 수행한다. single 지시어자 가장 먼저 접근한 스레드에 작업 할당된다. single 구문 끝에 암시적 장벽(동기화)이 존재한다.마찬가지로 이를 피하기 위해서는 nowait를 사용하면 된다. 병렬 영역 내에서의 입/출력 수행에 주로 사용된다.

• Task

OpenMP 버전 3.0 이상부터 추가된 지시자며, 수행할 작업을 한번의 호출로 진행되는 작업이나 한번의 구문으로 실행 가능한 작업 단위 나눈다. 태스트 실행은 작업 스케쥴링이 적용된다.

1.2 Intel Cilk Plus

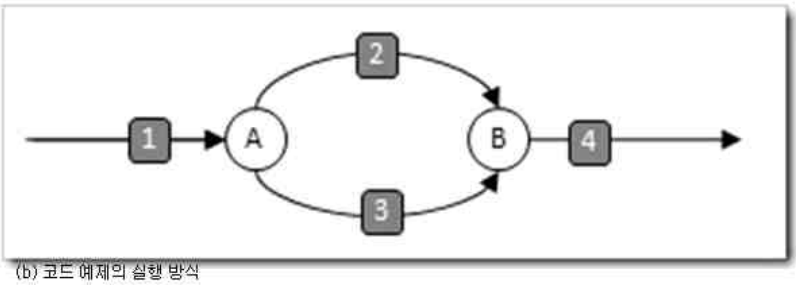
인텔 Cilk Plus는 C/C++ 기반의 병렬 프로그래밍 언어이다. 3개의 키워드로 태스크/루프 병렬성을 기술할 수 있으며, 벡터 병렬성도 배열 표기법을 확장하여 표현한다. 무엇보다 Cilk Plus의 런타임 스케줄러는 효율적으로 병렬 작업을 CPU를 할당한다.

. Cilk Plus의 태스크 병렬성 지원: cilk_for/spawn/sync

Cilk Plus는 크게 태스크 (혹은 루프) 수준 병렬성과 데이터 수준 병렬성(여기서는 벡터 형태로)을 지원하는 것으로 구분할 수 있다. 먼저, 태스크 수준 병렬성 지원에 대해 알아보자. Cilk Plus는 cilk_for, cilk_spawn, cilk_sync라는 3개의 키워드로 태스크 병렬성을 표현한다.

```
1: do_1();                  //
2: cilk_spawn do_2();  // 2번 작업을 spawn (A)
3: do_3();                  // 3번 작업을 계속 실행.
4: cilk_sync;               // 2, 3번 작업 완료를 기다림 (B)
5: do_4();
```

(a) Cilk 구문을 이해하기 위한 간단한 코드 예제



(b) 코드 예제의 실행 방식

Cilk 구문 이해를 위한 예제: cilk_spawn으로 두 병렬 작업을 생성하고 이들이 완료될 때까지 기다린 뒤 다음으로 진행

-cilk_spawn

함수 호출 구문 앞에 이 키워드를 쓰면, 뒤따르는 함수 호출이 호출자와 함께 동시에 실행될 수 있음을 런타임 시스템에게 알린다. 그런데 반드시 항상 병렬 수행이 된다는 보장은 하지 않는다. 이 부분은 런타임 스케줄러가 알아서 판단한다. 보다 쉽게 설명하면 논리적인 병렬 작업이 생성됨을 알리는 것으로 프로그램의 논리적인 실행 흐름을 두 갈래로 나누는 것이다. 포크(fork)에 비유될 수 있다. 그렇지만 반드시 물리적인 스레드가 생성되는 것은 아니다.

-cilk_sync

배리어(barrier) 연산으로 현재 수행중인 함수에서 cilk_spawn으로 생성한 병렬 작업들이 모두 종료될 때까지 기다리게 한다. 그런 뒤에 다음 구문을 실행한다.

-cilk_for

for 구문의 병렬 처리 버전으로 for 구문의 순환이 서로 병렬로 처리될 수 있음을 기술한다.

위의 그림 예제는 cilk_spawn으로 두 병렬 작업을 생성하고 이들이 완료될 때까지 기다린 뒤 다음으로 진행한다.

1.3 Intel TBB

인텔 스레딩 빌딩 블록(Intel Threading Building Blocks)은 C++로 병렬 프로그램을 구 현할 때 사용할 수 있는 일종의 라이브러리이며, 풍부하고 완벽한 방법론을 제공함으로써 스레드 처리에 대한 전문 지식이 없이도 멀티코어 프로세서 시스템의 성능을 향상시킬 수 있도록 도와준다. 스레딩 빌딩 블록은 단순히 기존의 스레드를 대체할 수 있는 라이브러리 를 제안하는 정도가 아니라 '태스크(task)'개념을 기반으로 하는 높은 수준의 병렬처리 기술이라고 할 수 있으며 성 능(performance)과 확장성(scalability)을 목표로 한다.

Intel TBB의 특징

인텔 TBB 태스크 스케줄러는 사용자를 위해 로드 밸런싱을 수행한다. TBB의 스케줄러는 프로그램을 여러 작은 태스크로 나눈 후 최상의 확장성을 구현할 수 있도록 태스크를 스레드에 고르게 할당하며 구현하는 방식은 C++에서 이미 사용 중인 방법을 확장하는 방식으로 구현된다. Intel TBB는 parallel_for, parallel_ while, parallel_reduce, pipeline, parallel_sort, parallel_scan 등 일부 함수와 템플릿을 컨커런트 컨테이너 (concurrent container)와 함께 제공해 코드에서 병렬성을 개발할 때 생산성을 향상시킨다.

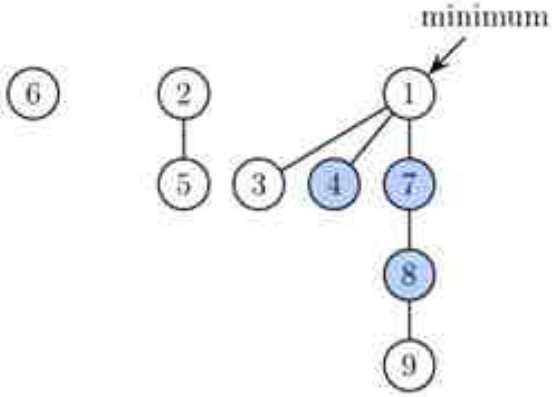
2. 3가지 프로그래밍 모델을 사용한 Intel Xeon Phi Coprocessor의 성능 벤치마크

성능평가를 위한 코드는 모두 C++로 작성되었고 실행파일은 MIC에서 native 방식으로 가동한다. 각각의 코드를 실행시키려면 우선 ssh를 통해서 Xeon Phi에 각각의 실행 파일과 실행에 필요한 라이브러리를 함께 복사하는데 OpenMP 코드를 위해서는 libiomp5.so, Cilk Plus 코드를 위해서 libcilkrts.so.5, TBB를 위해서 libtbb.so.2 라이브러리를 MIC에 복사하며 라이브러리들의 위치는 export LD LIBRARY PATH=./:$LD LIBRARY PATH에 미리 설정되어 있어야 한다. 프로그램들을 컴파일할 때, TBB 코드는 –ltbb, 오픈엠피는 –openmp 컴파일 옵션과 함께 컴파일 되어야 하며 제온파이에서 네이티브 모드로 프로그램을 실행시키기 위해서 icpc(icc) 14.0.2 버전에서 -O2 –mmic –no-offload 컴파일 옵션을 사용하였다. 모든 가속화 비율은 코드의 실행시간에 대비해서 계산되었다.
각 프로그래밍 모델의 병렬화 성능 비교를 위해서 세 가지 단순한 프로그램을 사용했으며 각각은 1) 피보나치 수열 계산 알고리즘 2) Merge Sort 알고리즘 3) 행렬 곱셈 알고리즘 과 같다.
이 실험에서 중요한 부분 중 하나는 각각의 프로그래밍 모델로 제작된 어플리케이션을 MIC에서 단독으로 실행하는 경우(싱글프로그래밍)와 동시에 실행하는 경우(멀티프로그래밍)의 두가지의 결과를 비교하고 병렬화 성능의 지표중 어느 것이 멀티프로그래밍 상황에서의 성능을 예측하기에 적합한지 확인하는 것으로 이를 확인하기 위해 각각의 프로그램은 단독으로 실행되기도 하였고 멀티프로그램 환경에서 실행되기도 하였다.

2.1 싱글 프로그래밍 환경

2.1.1 피보나치 수열 계산 알고리즘

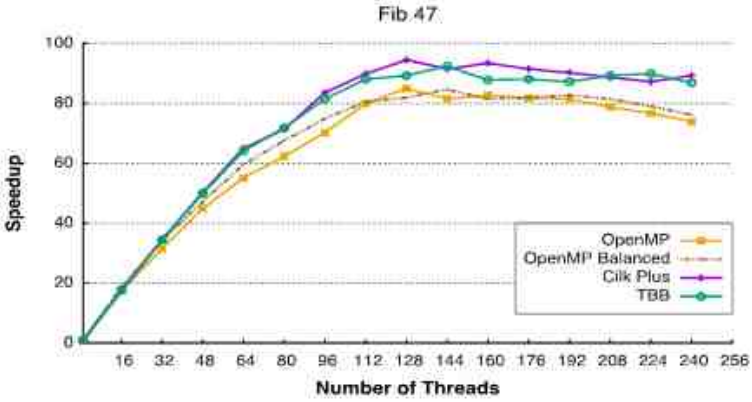
첫 번째 테스트는 병렬 컴퓨팅 예제로 자주 사용되는 병렬 피보나치 수열 코드이다. 병렬화 성능을 테스트 함에 있어서 피보나치 수열의 단순 재귀 패턴은 병렬화하기 쉽고 부하 불균형이나 작업 부하등을 테스트 하기에 적합하다. 이 연구에서는 피보나치 수열의 47번째 숫자까지를 계산하기 위하여 피보나치 힙 방식의 수열 계산 알고리즘을 사용하였다.



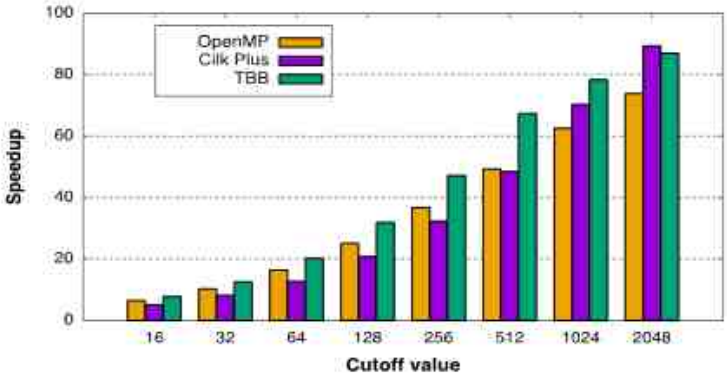
피보나치 힙 방식의 예: tree depth가
0,1,3인 세개의 나무가 있고 3개의 꼭지점이
파란 원으로 표시되어 있다. 이 힙의
잠재력은 3 tree + 2*vertice-number = 9

좋은 성능을 얻기 위해서는 병렬화 하는 작업의 크기가 너무 작거나 반복문이 지나치게 많이 반복되는 것을 피해야 하기 때문에 컷오프 기준을 설정해야 하는데 생성되는 피보나치 수열의 tree-depth에 따라 $2^{tree-depth}$개의 작업이 생성되는데 컷오프 기준에 따라 tree-depth가 제한된다.

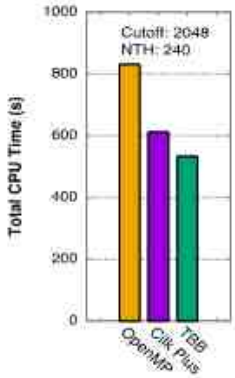
아래 그림에 각각의 프로그래밍 모델에 대한 성능 테스트 결과가 있다. (a) 는 컷오프를 2048로 고정한 후 실행한 결과(tree-depth = 11)이며 그래프에서 볼 수 있듯이 Cilk Plus와 TBB는 거의 유사한 결과를 보여주고 OpenMP의 경우 스레드의 갯수가 늘어날 수록 성능의 저하가 현저하다. KMP AFFINITY=balanced를 설정하여 테스크를 분산시켜 OpenMP는 성능의 개선을 확인하기 어렵다. (b)는 컷오프 기준을 각각 16, 32, 64, 128, 256, 512, 1024, 2048로 바꾸어 가며 성능을 확인한 것으로 테스크의 갯수가 늘어날 수록 부하 조절의 성능의 개선에 기여할 수 있는 가능성 또한 함께 커지는 것을 알 수 있다.



(a) Speedup, cutoff 2048, varying numbers of threads



(b) Speedup, 240 threads, varying cutoffs



(c) Total CPU Time

(d) OpenMP, CPU balance

(e) Cilk Plus, CPU balance

(f) TBB, CPU balance

47개의 피보나치 수열을 계산하기 위한 병렬 프로그램의 성능 분석: 가장 훌륭한 성능은 Cilk Plus나 TBB를 사용하여 얻을 수 있었다. 적절한 컷오프 값을 설정하는 것이 좋은 성능을 얻는 중요한 부분임을 (b) 의 결과를 통해 알 수 있는데 그 이유는 적당히 많은 숫자의 테스그가 시스템에 있어야 병렬화의 부하 조절 능력이 효과적으로 발휘되어 속도 향상에 영향을 미치게 된다. 계산에 소요된 전체 CPU의 사용 시간이 (c)~(f)에 나타나 있고 TBB가 가장 적은 CPU 시간을 소모하면서 적절한 성능을 보이고 Cilk Plus가 가장 좋은 성능을 보이고 있다. (c) ~ (f)의 그래프의 x축은 Xeon Phi의 각각의 논리적 코어(총 240개)를 나타내며 y축은 초당 0에서 지정된 최대의 숫자를 논리적 코어 당 시간으로 표시하였다.

. 전체 CPU 사용 시간

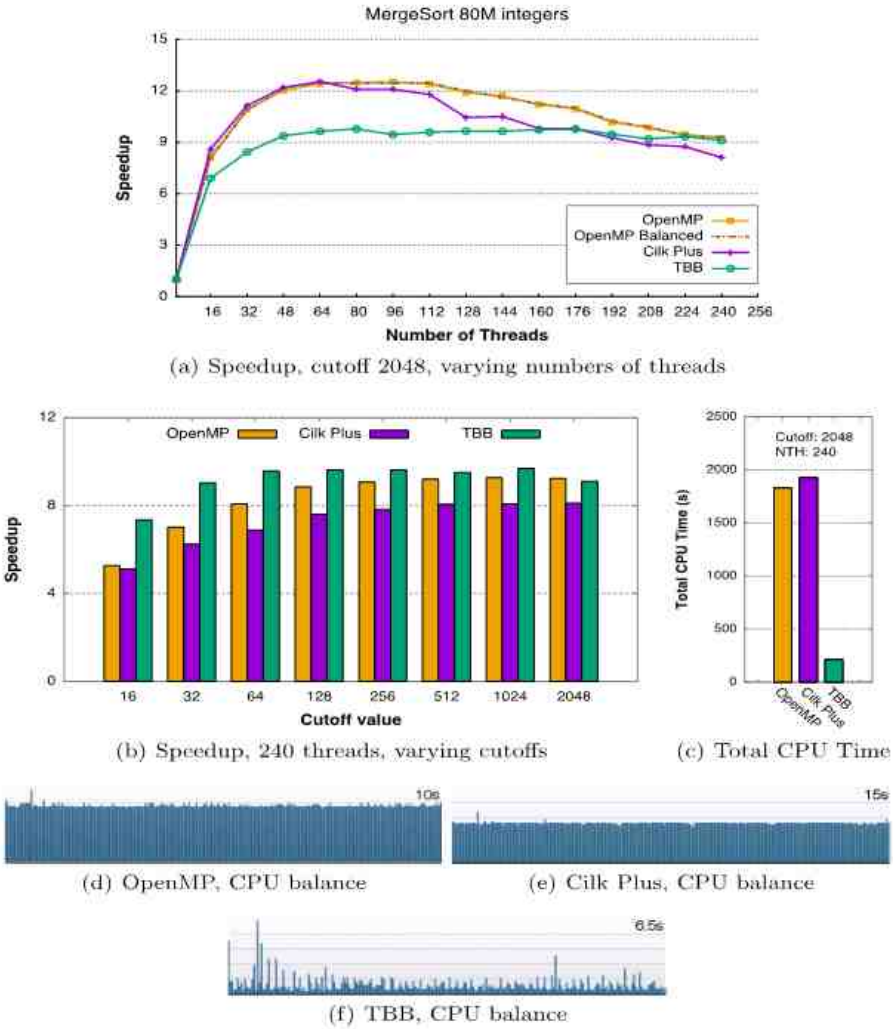
이 그래프는 시스템이 프로그램의 작업을 완료할 때까지 소모한 CPU시간을 나타낸 것으로 작을수록 좋은 것이다. 이 메트릭과 CPU 시간이 소모된 세부 사항은 인텔의 VTune Amplifier XE 2013의 성능 분석기를 이용하여 얻어졌다. 그림 (d)~(f)는 VTune Amplifier의 화면을 바로 보여주는 것으로 각각의 프로그램이 Xeon Phi에서 네이티브 모드로 동작하는 상황이다. X축은 Xeon Phi의 논리적인 코어를 나타내고 (이 경우 240개) y축은 각각의 코어가 소모하는 CPU 시간을 나타낸다. 피보나치 수열 계산의 경우 OpenMP가 가장 많은 CPU시간을 소모하고 성능은 가장 낮음으로서 세가지 병렬화 방법 중 가장 비효율 적인 방법이라 할 수 있다.

2.1.2 합병 정렬(Merge Sort) 알고리즘

이 벤치마크는 합병 정렬을 사용해서 8천만개의 숫자를 정렬하는 프로그램이다. 이 배열의 i번째 숫자는i∗((i%2)+2)로 초기화 되어 있다. 컷오프 값은 한번에 연속적으로 정렬되는 숫자의 지점을 정하는데 예를 들어 컷오프가 2048이라면 병렬로 8000만개의 1/2048개의 숫자를 연속적으로 정렬한 후 병렬로 두개씩 묶어서 다시 정렬하는 방식으로 완벽히 정렬된 배열을 완성한다. 합병정렬 프로그램의 경우 모든 작업이 동등한 것이 아니라 부모와 자식 관계의 작업이 생성된다. 앞선 피보나치 수열 계산의 경우에서도 부모 자식 관계의 작업 구조가 있었지만 부모 작업은 그저 숫자를 더하는 역할만 할 뿐 병렬화의 성능에는 영향을 미치지 않았었다. 하지만 이 프로그램의 경우 부모 작업에 걸리는 부하가 상당히 크기 때문에 앞선 프로그램과는 구별되는 결과를 보여준다.

아래 그림 (a)에서 볼 수 있는 바와 같이 스레드의 갯수를 늘려도 TBB는 눈에 띄는 성능의 변화가 없는 반면에 Cilk Plus나 OpenMP는 오히려 성능의 저하가 눈에 띈다. 스레드의 Affinity를 조절해 보아도 OpenMP의 경우는 성능에 큰 차이를 보이지 않는다.

그림 (c)에서 (f)는 240 스레드에서 2048 컷오프를 가질경우 Vtune Amplifier에서 캡쳐한 화면을 보여주고 있다. 이 프로그램의 경우 부하에 대한 조정 옵션을 사용하지 않았음에도 불구하고 TBB가 소모하는 CPU 시간이 명백히 적다. 즉 TBB는 부하 조절 없이도 좋은 성능을 보여준다.

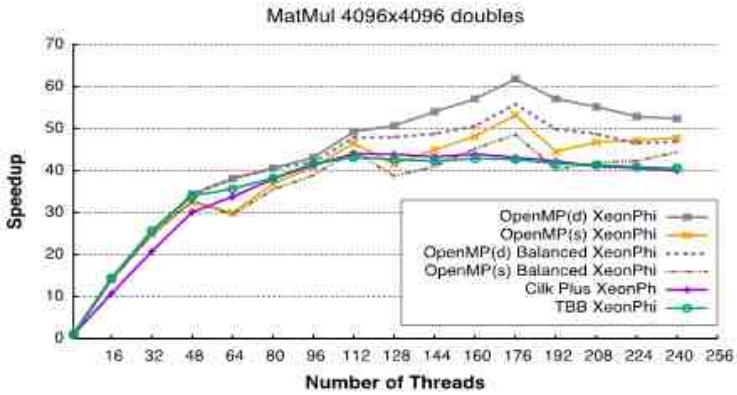


8000만개의 숫자 배열에 대한 병렬화 된 합병정렬 알고리즘의 성능 테스트: 이 테스트의 경우 가장 좋은 성능은 OpenMP와 Cilk Plus를 사용하여 얻을 수 있었다. 이것과 같이 메모리 부하가 큰 프로그램의 경우 스레드의 갯수가 충분히 많다고 할 때 컷오프 값은 64 이상만 되어도 충분히 좋은 결과를 보여준다. (c)의 결과를 보면TBB가 다른 모델들에 비해서 현저하게 CPU시간을 적게 소모하는 것으로 나타났다. 대신 (a)에서 볼 수 있듯이 스레드의 갯수가 적을 경우는 반대로 Cilk Plus와 OpenMP가 TBB보다 더 좋은 성능을 발휘했다.

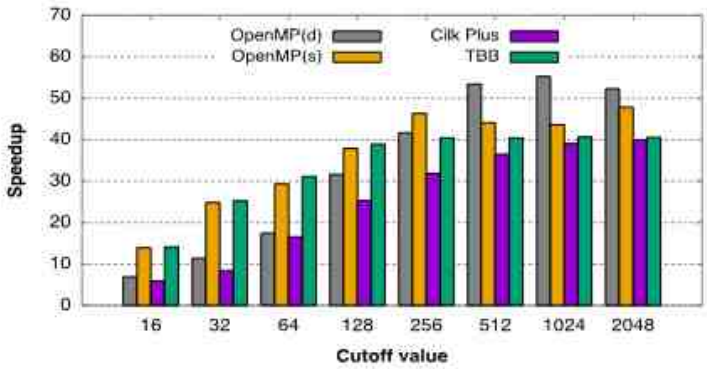
2.1.3 NxN 행렬 곱셈(Mat mul)알고리즘

이 벤치마크는 i,j,k의 세개의 반복문으로 구성된 NxN 크기의 배정밀도 부정소수점 숫자(double- precision floating point numbers)로 구성된행렬간의 곱셈을 다룬다. 이런 구조의 프로그램은 OpenMP의 for방식 작업분담에 적절하다.
이 벤치마크에도 컷오프 개념의 작업이 들어가있는데 각 스레드가 처리해야 하는 반복문의 조각의 크기를 제한하는 방식이다. OpenMP의 경우#pragma omp for

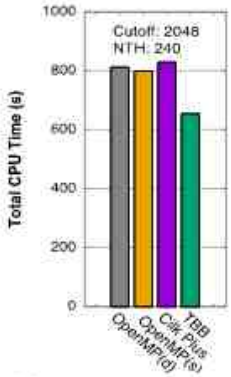
schedule(dynamic, N/cutoff)의 방식으로 구현하는데 dynamic은 static으로 대체해도 무방하다. Cilk Plus의 경우 쓰레드가 처리하는 반복문의 조각의 크기는#pragma cilk grainsize = N/cutoff로 설정한다. TBB의 경우는 parallel_for()를 호출할때simple partitioner()를 함께 호출하는 것으로 조각의 크기를 설정할 수 있다.
결과를 살펴보기 전에 기술적인 부분을 조금 더 살펴보기로 한다. Xeon Phi의 자동 벡터화를 사용하기 위해서는 TBB와 OpenMP는-ansi-alias 플래그와 함께 컴파일 되어야 한다. OpenMP의 for에서 schedule 구문은 관련된 반복문에서 정적 내지는 동적으로 반복횟수를 나누어서 일련의 스레드들에게 어떻게 배분할지를 결정한다. 이 벤치마크에서 OpenMP 의 for구조는 지정된 크기로 나누어서 라운드로빈 방식으로 실행 테스크에 배분된다. Xeon Phi에 있어서 컷오프 값과 스레드의 갯수, 스드의 affinity의 관계에 대해서 더 잘 이해하기 위해 다음의 예제를 확인한다.
-예제
#pragma omp for schedule(static, N/cutoff)를 사용할 때 a의 실행시간이 Xeon Phi에서 b보다 3배이다.
a) omp set num threads(32), cutoff=32, KMP AFFINITY=balanced
스레드는 물리적으로 32개의 코어에 배분되고 affinity설정에 의해 칩을 건너서 한 코어에 한 스레드를 배분한다. 결과적으로 모든 나뉘어진 작업은 물리적으로 분할된 코어에서 실행된다.
b) omp set num threads(240), cutoff=32, KMP AFFINITY=balanced
쓰레드는 물리적으로 60개의 코어에 배분되고 32개의 하드웨어 쓰레드는 8개의 코어 사이에서 적절하게 배분된다. 240개의 쓰레드를 사용함으로서 논리적인 코어에 하나씩의 스레드가 배분되고 컷오프가 32이기 때문에 스레드 아이디가 0~31인 스레드는 N/32의 크기의 작업을 배정받는다.
이러한 지식을 바탕으로 수행된 행렬의 곱셈 결과는 아래 그림에 자세히 설명되어 있다



(a) Speedup, cutoff 2048, varying numbers of threads



(b) Speedup, 240 threads, varying cutoffs



(c) Total CPU Time

(d) OpenMP (dynamic), CPU balance

(e) OpenMP (static), CPU balance

(f) Cilk Plus, CPU balance

(g) TBB, CPU balance

병렬화된 배정밀도 숫자로 구성4096x4096행렬간의 곱셈 계산 알고리즘의 성능 테스트: 가장 좋은 성능은 OpenMP방식을 사용하여 얻을 수 있었다. 만약 컷오프 기준이 256보다 크다면 OpenMP의 동적 스케쥴링이 다른 방식들에 비해 가장 좋은 조정능력을 갖는다. 프로그램이 소비한 총 CPU시간은 TBB가 다른 방법들에 비해 가장 적은 CPU시간을 소모한 것으로 나타났는데 특히 (d)와 (e)를 비교해 보면 OpenMP의 부하 조정 능력이 활성화 되어 있으면 그렇지 않은 상태에 비해서 훨씬 좋은 성능을 보여줌을 알 수 있다.

2.1.4 싱글프로그래밍 환경에 대한 결과 분석

이러한 병렬 프로그래밍 모델 사이의 차이를 추론하는 방법 중 하나는 그들의 런타임 라이브러리에 의해 소비되는 전체 CPU 시간을 비교하는 것이다. 그러므로 우리는 각각의 경우에 공유 라이브러리에 소요되는 시간의 비율로 결과를 요약했다.

런타임 라이브러리에 의해 소모된 전체 CPU 시간의 퍼센트

Benchmark	OpenMP (libiomp5.so)	Cilk Plus (libcilkrts.so.5)	TBB (libtbb.so.2)
Fibonacci	50%	16%	5%
MergeSort	78%	81%	3%
MatMul	22% (Dynamic) 20% (Static)	6%	1%

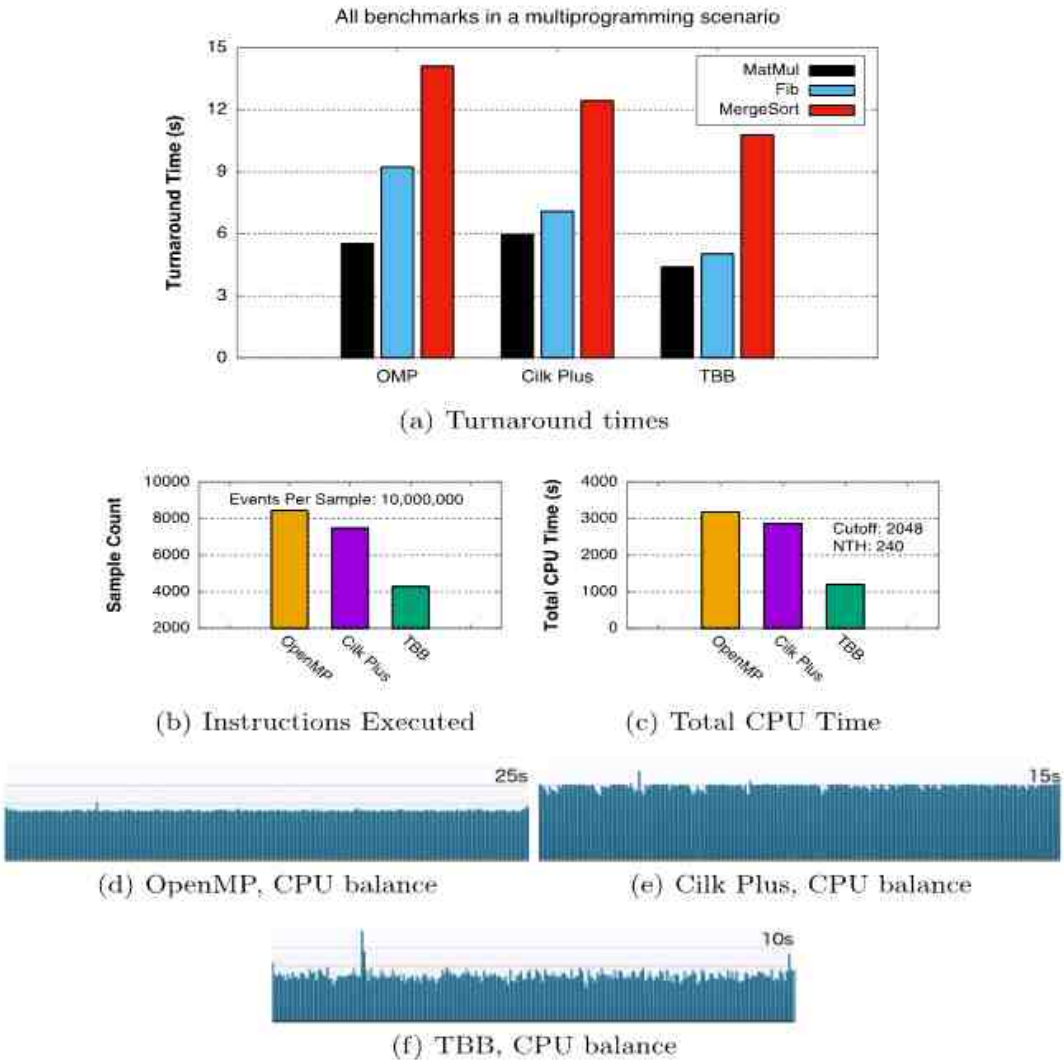
위의 표는 CPU시간이 어디서 소모되는지를 더 잘 나타내준다. 예를들어 OpenMP의 런타임 라이브러리의 경우는 소모된 CPU시간을 크게 두가지로 분류할 수 있는데 그 첫번째는 마스터 스레드가 직렬 코드를 실행하고 노예 스레드가 회전할때. 두번째는스레드는 병렬 영역을 완료하고, 그 동기 지점에 도달하기 위해 모든 다른 스레드를 기다리는 장벽에 돌고있는 경우이다. 때때로 프로그램의 솔로 실행에 이러한 추가 CPU 사이클이 실행 시간 (벽에서의 시간)에 미치는 영향은 무시할 수를 가지고 있지만, 우리는 그들이 멀티 프로그램 실행에서 다른 프로그램에 미치는 영향을, 다음 에서 설명하도록 한다.

2.2 멀티 프로그래밍 환경

이 절에서는 앞서 설명한 세가지 병렬화 프로그래밍 모델로 작성된 어플리케이션을 같은 MIC에 동시에 실행하는 환경(멀티프로그래밍)을 설정하여 성능을 테스트 한다. 비교를 위해 사용한 메트릭은 멀티프로그램 시스템에서 작업을 제출하고 완료하는데 까지 걸리는 시간을 측정한 것을 사용한다.

실험을 위해 세가지의 벤치마크 모두 단독 실행의 경우와 같은 입력의 크기와 2048의 컷오프와 Xeon Phi의 논리적 코어의 갯수와 같은 240개의 스레드에서 동작하도록 하였다 . 모든 프로그램을 동시에 실행하지는 않았고 오히려 병렬화단계가 동시에 시작하도록 조절하였다. 때문에 합병 정 벤치마크가 가장 먼저 실행되고 2초후 행렬 곱셈 벤치마크가, 0.5초후 피보나치 수열 계산 벤치마크가 실행되도록 조절하였다.

프로그램 단독 실행의 결과에 따라 세가지 벤치마크 모두에서 가장 적은 CPU 시간을 보였던 TBB가 이경우에도 가장 좋은 성능을 보일 것으로 예상되었는데 그 이유는 프로그램 하나를 단독실행 할 경우 CPU시간을 독점할 수 있기 때문에 소모되는 CPU시간이 성능에 큰 영향을 미치지 않지만 여러프로그램이 자원을 두고 경쟁해야 하는 상황이라면 CPU시간이 성능에 큰 영향을 미칠 것이기 때문이다. 이 실험의 결과는 아래 그림에 잘 나와있다.



세가지 벤치마크에 대한 멀티프로그래밍 시나리오에서의 성능 테스트: (a)는 가장 훌륭한 성능은 TBB를 통해 얻을 수 있음을 보여준다. (b)는 VTune Amplifier를 통해 수집한 명령 구문의 실행이라는 하드웨어 이벤트의 갯수를 보여주는데 역시 TBB가 다른 두가지 방법에 비해 월등히 좋은 성능을 보여둔다. 소모된 전체 CPU 시간은 (c)에 나타나 있는데 (b)와 유사하게 TBB가 가장 좋은 성능을 보이는 것으로 나타냈다. (d)~(f)의 결과를 통하여 OpenMP가 가장 많은 CPU시간을 소모하고 그러므로 가장 나쁜 성능을 보여준 다는 것을 알 수 있다.

3 멀티 프로그래밍 환경

이 보고서에 CPU 균형, 그리고 총 CPU시간, 가속화 세가지의 관점에서 잘 알려진 병렬 프로그래밍 접근 방식인 OpenMP와 Cilk Plus 그리고 TBB에 대해서 Xeon Phi에 대한 성능을 보였다. 실험을 위하여 피보나치와 합병 정렬, 행렬 곱셈의 세가지 다른 벤치마크를 사용하였고 각각의 세가지 벤치마크는 우리가 보고자한 세가지의 관점을 잘 나타내었다. 멀티프로그래밍 시나리오는 한 시스템에서 세가지 벤치마크를 모두 실행하여 이런 상황에서 프로그래밍모델들이 각기 어떻게 반응하는지를 살펴보기 위함이었다.

각각의 프로그램을 독립적으로 실행했던 경우의 결과 중에 특히 총 CPU시간을 기반으로 보았을 때 TBB 방식의 병렬화가 멀티프로그래밍 환경에 가장 적합할 것으로 예측하였고 우리의 실험 결과가 이것을 입증하였다.