기술 보고서

ISBN:

Kokkos 프로그래밍 모델

2015.11

국가슈퍼컴퓨팅연구소
슈퍼컴퓨팅응용실
권오경

목 차

1. 개요

샌디아연구소에서 개발한 멀티/매니코어 기반의 프로그래밍 모델로서 C++기반의 API를 제공함. Kokkos가 포함하는 디바이스는 다음 세가지를 포함함: (1) 멀티코어 프로세서, (2) 매니코어 프로세서(대역폭 위주의 컴퓨팅을 위한 중규모 수준의 컴퓨팅 유닛들), (3) GPGPU(general-purpose graphics processor unit) 프로세서 (보다 긴 latency를 가지지만 보다 많은 적은 성능의 컴퓨팅 유닛들을 통해 많은 수준의 병렬화를 통해서 성능을 향상이 가능한).

이러한 디바이스들에서 실행하기 위한 프로그래밍 모델은 기본적으로 멀티/매니코어 프로세서를 위해서 OpenMP, Clik+, Thread Building Blocks(TBB), Linux p-threads와 같은 shared memory 기반의 접근방법이 제공됨. 그리고 GPGPU까지 지원되기 위해서 OpenMP, OpenACC, OpenCL와 같은 프로그래밍 모델도 사용이 될 수도 있음. GPGPU에서 최적성능을 내기 위해서 CUDA가 선택이 될수도 있음.

Kokkos는 위와 같이 다양한 프로그램 모델에 대한 컴퓨팅 및 데이터 할당/메모리 레이아웃에 대한 추상화를 제공함. 이를 통한 이기종 환경에서 이식성 및 고성능을 동시에 추구함

Kokkos는 이를 위해서 두가진 주요 기능을 제공함. 1) 멀티/매니 코어에서 병렬 작업 실행, 2) 다차원 배열을 통한 효율적인 메모리 관리. 효율적인 메모리 관리를 통해 아키텍쳐에 따른 array of structures (AoS)냐 structure of arrays (SoA)에 대한 선택을 따로 할 필요가 없음

2. 머신(Machine) 모델

여기서 소개되는 해당 머신 모델을 통해 이식성 및 고성능이 가능함.
머신 모델은 다음 두가지 요소로 구성됨
- 메모리 영역: 데이터 구조가 할당되는 영역
- 실행 영역: 메모리 영역의 데이터를 이용해서 병렬 연산이 수행되는 영역

2.1 동기

다음 두가지 측면을 살펴볼 수 있음. 하나는 이식성과 고성능을 위한 기본 개념에 대한 추상 머신 모델. 다른 하나는 C++로 구현된 모델에 대해서 프로그래밍할 수 있게 제공. 해당 두가지 측면을 통해서 다른 프로그래밍언어로 개발되어도 개발된 알고리즘이 그대로 사용할 수 있게됨.

2.1.1 Kokkos 추상 머신 모델

shared memory 기반의 컴퓨팅 아키틱처를 가정함. 또한 해당 모델은 Figure 2.1에 있는 것처럼 하나의 컴퓨팅 노드에 여러 계산 유닛이 가능함을 가정함. 그림처럼 두가지 종류의 계산 유닛을 선정함. 하나는 현대의 프로세서 코어와 유사한 여러개의 코어이고 하나는 가속기에 포함된 코어임. 프로세서와 가속기는 별도의 메모리를 사용함. 각 메모리는 개별 성능 특징을 보유하고 노드를 넘어서 접근이 가능함.

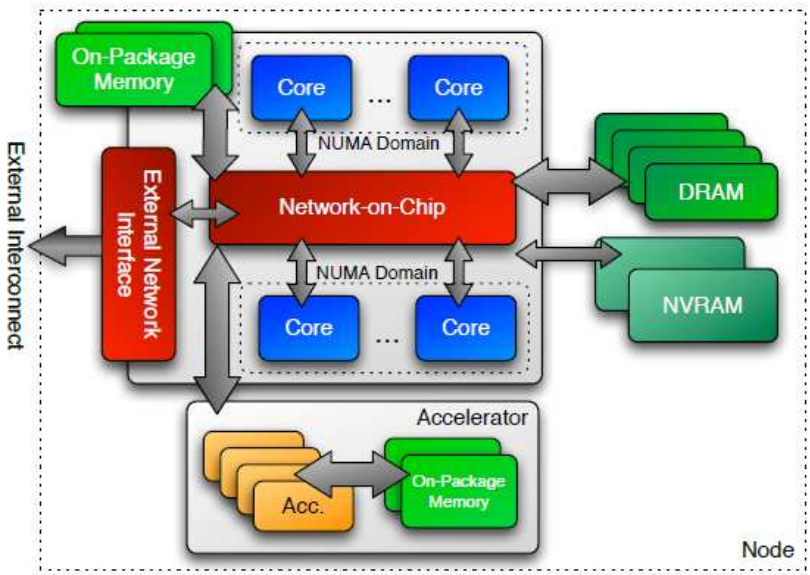


Figure 2.1: Conceptual Model of a Future High Performance Computing Node

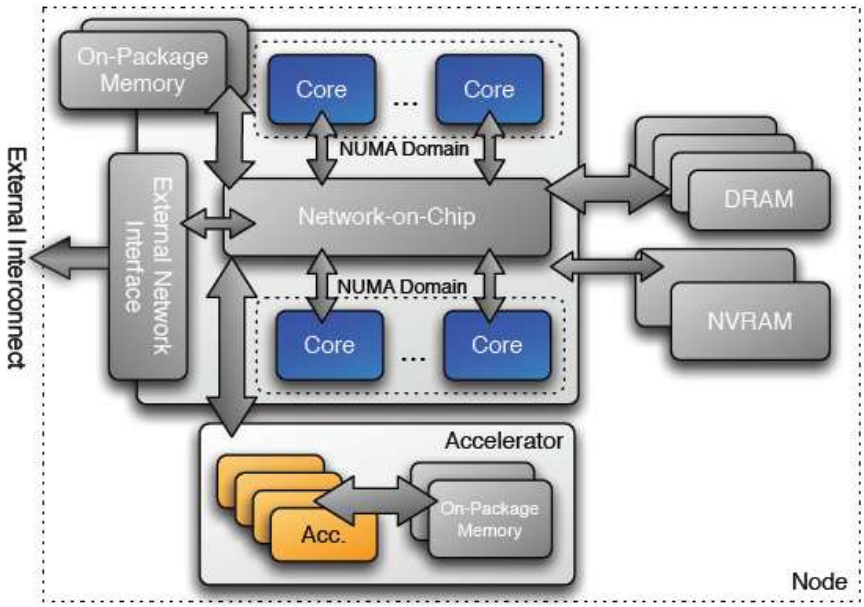


Figure 2.2: Example Execution Spaces in a Future Computing Node

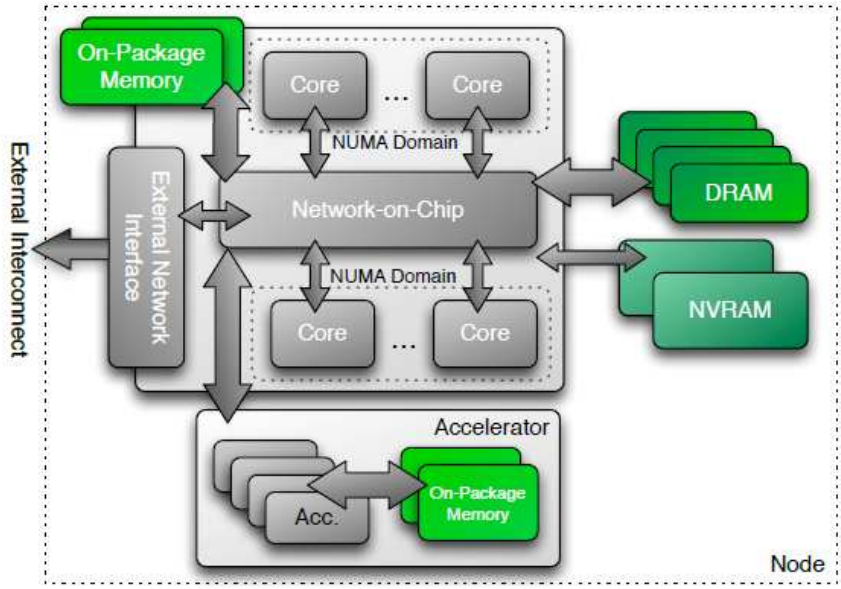


Figure 2.3: Example Memory Spaces in a Future Computing Node

2.2 실행 영역

실행 영역은 동일한 성능 특징을 보유하고 있는 계산 유닛의 논리적 그룹을 위해서 사용됨. 여러 병렬 명령어를 통해서 프로그래머에 의해서 사용가능한 병렬 실행 자원 집합을 제공함.

2.2.1 실행 영역 인스턴스

인스턴스는 프로그래머가 병렬 태스크에 대한 실행 영역에 대한 하나의 생성. 첫 번째 예제로서 실행 영역은 멀티코어 프로세스를 묘사하는데 사용될 수도 있다. 이때 멀티코어 프로세서는 동종(homogeneous)의 코어임. Kokkos 모델로 작성된 프로그램내에서 실행 영역 인스턴스는 병렬 커널로 실행됨. 두 번째 예제로서 GPU를 멀티코어 프로세서에 부착된 형태로 되어 있으면, 프로그래머가 선택해야할 두가지 형태의 인스턴스들이 존재하게 됨. 여기서 중요한 점은 다른 실행 영역에 대해서 프로그래머가 일일이 신경쓸 필요가 없다는 점임.

2.2.2 메모리 영역

메모리의 여러 종류는 메모리 영역에 의해서 추상화 됨. 각 메모리 영역은 유한한 사이즈내에서 의미가 있으며, 데이터 구조가 할당되고 사용됨. 이식성 있는 할당 및 성능향상을 위해서, Kokkos는 데이터를 k-차원 배열로 할당해야 함(k는 0이상). 배열의 각 차원은 컴파일 시에 고정되거나, 실행시에 동적으로 결정될 수 있음

2.2.3 메모리 영역의 인스턴스

위에서 실행 영역 인스턴스와 마찬가지로 메모리 영역에 대한 인스턴스는 사용됨. 메모리 영역의 인스턴스는 프로그래머가 데이터 할당에 대한 요청시 제공함. 실행 영역의 인스턴스의 예제로 돌아가서, 멀티코어 프로세서는 온패키지(on-package) 메모리를 포함한 다중의 메모리 영역을 가지고 있고, GPU는 로컬 온패키지 메모리 내에 별도의 메모리 영역을 가지고 있다. 프로그래머는 해당 인스턴스에 대해서 메모리 할당 및 관리에 대한 적절한 추상화된 방법을 선택할 수 있음.

- 메모리에 대한 Atomic한 접근

하나의 메모리 주소에 여러 쓰레드가 접근할 시, 각 쓰레드가 계산을 끝내고 동일 주소에 작성시에 충돌이 발생할 수 있음. 레이스 컨디션(race conditions)이라고 불리는 해당 현상은 병렬 환경에서 발생가능함. 레이스 컨디션이 발생하지 않게하기 위해서 lock, critical region, atomic operation등의 방법이 적용될 수 있음.
실행 영역은 atomic operation 제공에 대해서 보장이 되지 않지만, atomic 접근을 제공하지 않는 영역에 대해서 사용하는 부분에 대해서는 컴파일시 에러가 발생함.

- 메모리 일관성

메모리 일관성은 복잡한 문제임. 하드웨어 캐쉬나 메모리 접근에 관한 연산에 의존함. 하드웨어안에 캐쉬될 필요는 없고, weak memory 일관성 모델을 가정함. 프로그래머는 메모리 연산에 있어서 순서를 가정하지 않음. 이로인한 레이스 컨디션이 발생할 수 있음. 이를 위해 계산이 메모리 연산에 관련해서 완전히 끝날 수 있도록 fence 연산을 제공함.

2.3 프로그램 실행

프로세서가 실제로 코드를 실행하는 것에 대한 형식적인 정의. 실행영역에서 병렬로 연산을 하는 것을 가정함. 이때 parallel for, reduce, scan, View allocation, initialization을 포함함. 이러한 연산자들을 parallel dispatch로 칭함. 다음 세가지 형태의 코드가 있음.

- Kokkos 병렬 연산내에서의 실행하는 코드
- Kokkos에게 무엇인가를 요청하는 Kokkos 병렬 연산 밖에서의 코드 (예, parallel dispatch)
- Kokkos와 아무런 관련이 없는 코드

Reproducible reductions and scans

Kokkos는 병렬 루프내에서 발생하는 순서 관련해서는 보장하지 않지만 똑같은 병렬 reduction이나 scan을 같은 하드웨어 내에서 수행하면 동일한 결과를 보장함.

Asynchronous parallel dispatch

parallel dispatch는 비동기적으로 실행함. 실제로 완료되기 전에 함수는 일찍 return하고, 이에 대해서 fence를 통해서 동기화를 해야함

3. 이식성 및 고성능

문제점: 개별 아키텍처에 맞게 메모리 접근 패턴을 구현해야 좋은 성능이 가능함

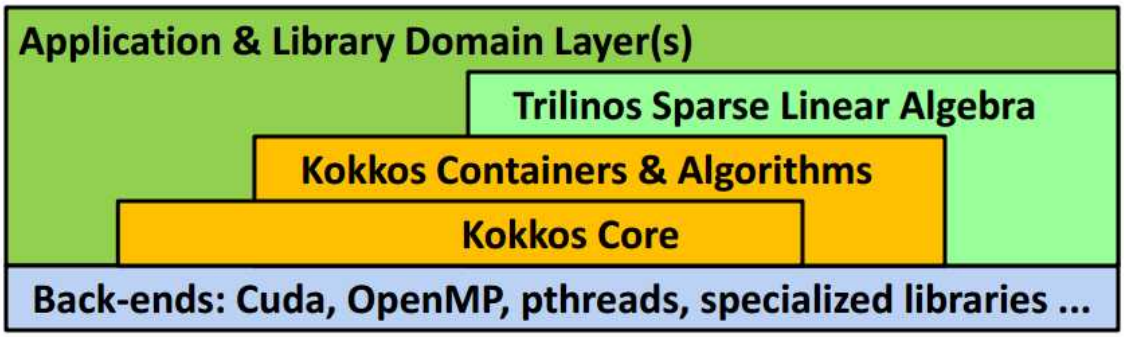
- CPU(xeon phi)
  * Core-data affinity: NUMA에 맞게 구현해야 함
  * 캐쉬와 벡터 사이즈에 맞는 alignment
  * Hyperthreads' cooperative use of L1 cache
- GPU
  * Thread-data affinity: coalesced access with cache-line alignment
  * Temporal locality and special hardware (texture cache)

==> 해결 방안: 병렬 계산과 고차원 배열 데이터를 매니코어 아키텍처에 맞게끔 매핑

- 사용자의 병렬 계산을 쓰레드에 매핑
  * Parallel pattern: foreach, reduce, scan, task-dag, ...
  * Parallel loop/task body: C++11 lambda or C++98 functor
- 사용자의 데이터들을 메모리로 매핑
  * 고차원 배열
  * 메모리 레이아웃 : multi-index (i,j,k,...) ↔ memory location
  * 아키텍처에 적합한 메모리 접근 패턴은 Kokkos가 선택
  * Polymorphic 고차원 배열
- 사용자 데이터를 특정 하드웨어로 접근
  * GPU texture cache to speed up read-only random access patterns
  * Atomic operations for thread safety

4. C++ 라이브러리

- 표준 C++ 라이브러리
  * 해당언어처럼 언어확장은 아님: OpenMP, OpenACC, OpenCL, CUDA
  * Intel's TBB, NVIDIA's Thrust & CUSP, MS C++AMP에 영향을 받음
- C++ template meta-programming
  * 이전에는 C++1998 standard
  * 현재는 lambda functionality로 인해 C++2011이 필요
  * Supported by Cuda 7.0; full functionality in Cuda 7.5
  * Participating in ISO/C++ standard committee for core capabilities



5. Spaces, Policies, Patterns

- 메모리 영역: 데이터가 있는 영역
  * Differentiated by performance; 예) size, latency, bandwidth
- 실행 영역: 함수가 실행하는 곳
  * 하드웨어 자원을 표현 예) cores, GPU, vector units, ...
  * 접근가능 메모리 영역
- 실행 정책: how (and where) a user function is executed
  * 예) data parallel range : concurrently call function(i) for i = [0..N)
  * User's function is a C++ functor or C++11 lambda
- Pattern: parallel_for, parallel_reduce, parallel_scan, task-dag, ...
- Compose: pattern + 실행 정책 + 사용자 함수; e.g.,
  * parallel_pattern(Policy<Space>, Function);
  * Pattern과 정책에 따라 함수를 실행
- Extensible spaces, policies, patterns

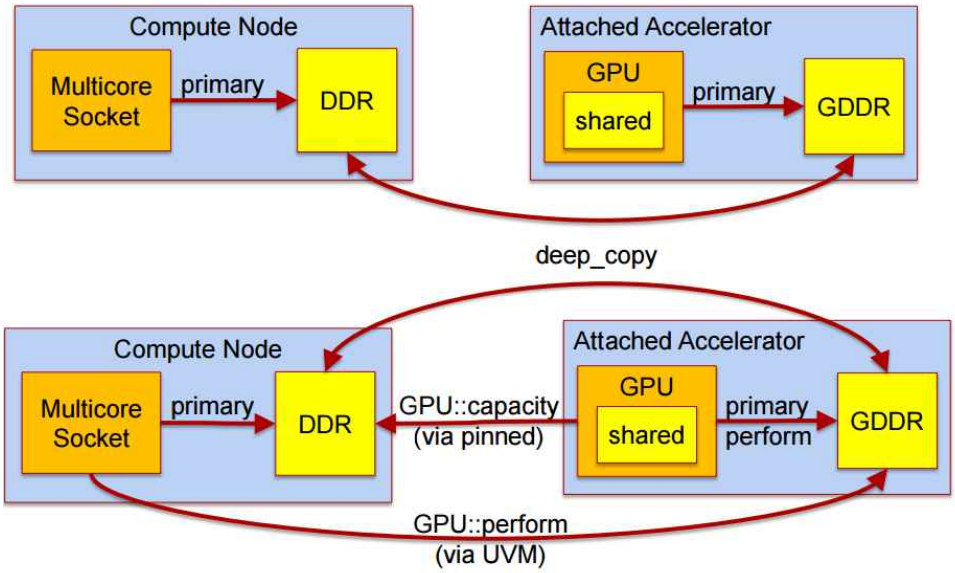


Figure 5.1 실행 영역과 메모리 영역의 예

6. 다차원 배열

계산 코드들이 배열의 데이터를 처리하는데 많은 시간을 소요. 가능한 최적의 성능을 내기 위해서 노력하는데 하드웨어 아키텍처, 런타임 환경, 언어, 프로그래밍 모델에 따라 달라짐. Kokkos는 다차원 배열에 대해서 프로그래머가 이런 노력을 줄이게 제공함. 최적의 메모리 레이아웃을 만들고 캐쉬 alignment를 위한 padding을 함

- View< double**[3][8], LayoutRight, DeviceType > a("a",N,M);
  * C++ template 클래스인 View를 통해서 구현되어 있음, 메모리 영역에 다음 배열을 할당: dimensions [N][M][3][8]
  * RowMajor data storage(즉, 4th index is stride-one access)
  * allocated in memory space of DeviceType
  * C++17 improvement to allow View<double[ ][ ][3][8],Space>
  * Kokkos View는 an array of zero or more dimensions
- a(i,j,k,l) : 배열 데이터에 접근시
  * "Space" accessibility enforced; 예) GPU 코드는 CPU 메모리에 접근 불가 * * Optional array bounds checking of indices for debugging
- View Semantics: View<double**[3][8],Space> b = a ;
  * A shallow copy: 'a' and 'b' are pointers to the same allocated array data
  * Analogous to C++11 std::shared_ptr
- deep_copy(destination_view , source_view);
  * 'source_view'에서 'destination_view'로 복사
  * Kokkos policy: never hide an expensive deep copy operation
- View<const int**[8][3], LayoutRight, Device, RandomRead> b = a;
  * const + RandomRead => use Texture fetches on GPUs
- 메모리 deallocation은 자동으로 됨

- Layout mapping : a(i,j,k,l) → memory location
  * 최적의 메모리 레이아웃은 컴파일시 결정됨
  * Kokkos는 "Space"에 적합한 기본 레이아웃을 결정
  * 예) row-major, column-major, Morton ordering, dimension padding, ...
- Layout을 지정 : View< ArrayType, Layout, Space >
  * Override Kokkos' default choice for layout

* Why? For compatibility with legacy code, algorithmic performance tuning, ...

- Example Tiling Layout
  * View<double**,Tile<8,8>,Space> m("matrix",N,N);
  * Tiling layout transparent to user code : m(i,j) unchanged
  * Layout-aware algorithm extracts tile subview

- Array subview of array view (new)
  * Y = subview(X , ...ranges_and_indices_argument_list...);
  * View of same data, with the appropriate layout and index map
  * Each index argument eliminates a dimension
  * Each range [begin,end) argument contracts a dimension

- Access intent Attributes
  View< ArrayType, Layout, Space, Attributes >
  * How user intends to access datum
  * Example, View with const and random access intension
    - View< double ** , Cuda > a("mymatrix", N, N);
    - View< const double **, Cuda, RandomAccess > b = a ;
      - ➢Kokkos implements b(i,j) with GPU texture cache

7. Parallel Patterns

parallel_pattern(Policy<Space> , UserFunction)

- Easy example BLAS-1 AXPY with views

```
parallel_for( N , KOKKOS_LAMBDA( int i ){ y(i) = a * x(i) +  y(i); } );
```

  * Default execution space chosen for Kokkos installation
  * Execution policy "N" => RangePolicy<DefaultSpace>(0,N)
  * #define KOKKOS_LAMBDA [=] /* non-Cuda */
  * #define KOKKOS_LAMBDA [=]__device__ /* Cuda 7.5 candidate feature */

- More verbose without lambda and defaults:

```
struct axpy_functor {
 View<double*,Space> x , y ; double a ;
 KOKKOS_INLINE_FUNCTION
 void operator()( int i ) const { y(i) = a * x(i) +  y(i); }
 // ... constructor ...
};
parallel_for( RangePolicy<Space>(0,N) , axpy_functor(a,x,y) );
```

- Example: DOT product reduction

```
parallel_reduce( N , KOKKOS_LAMBDA( int i , double & value )
                  { value += x(i) * y(i); }
                   , result );
```

  * Challenges: temporary memory and inter-thread reduction operations
  * Cuda shared memory for inter-warp reductions
  * Cuda global memory for inter-block reductions
  * Intra-warp, inter-warp, and inter-block reduction operations

- 사용자가 reduction type과 operations를 정의 가능

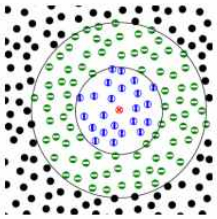
```
struct my_reduction_functor {
typedef ... value_type ;
KOKKOS_INLINE_FUNCTION void join( value_type&, const value_type&) const ;
KOKKOS_INLINE_FUNCTION void init( value_type& ) const ;
};
```

  * 'value_type': 실행시간에 결정되는 일차원 배열
  * 'join' and 'init' plugged into inter-thread reduction algorithm

8. 메모리 접근 패턴

- 병렬 실행
  * 함수를 쓰레드별로 매핑
  * GPU: iw = threadIdx + blockDim * blockIds
  * CPU: iw ∈[begin,end)Th ; contiguous partitions among threads
- 다차원 배열의 메모리 레이아웃
  * Leading dimension is parallel work dimension
  * Leading multi-index is 'iw' : a(iw , j, k, l)
  * 아키텍처에 맞는 적절한 배열 선정
  * 예) AoS for CPU and SoA for GPU
- Fine-tune 메모리 레이아웃
  * 예) 캐쉬에 적합한 메모리 패딩
- 접근 패턴에 따른 성능 영향
  * miniMD(MD(Molecular dynamics) 코드)를 이용
  * Lennard Jones force model
  * N*N computations을 회피하기 위해서 다음과 같이 Atom neighbor list 구성

```
pos_i = pos(i);
for( jj = 0; jj < num_neighbors(i); jj++) {
  j = neighbors(i,jj);
  r_ij = pos_i - pos(j); //random read 3 floats
  if (|r_ij| < r_cut) f_i += 6*e*((s/r_ij)^7 - 2*(s/r_ij)^13)
}
f(i) = f_i;
```



  * 테스트한 문제
    - 864k atoms, ~77 neighbors
    - 2D neighbor 배열
    - Different layouts CPU vs GPU
    - Random read 'pos' through GPU texture cache

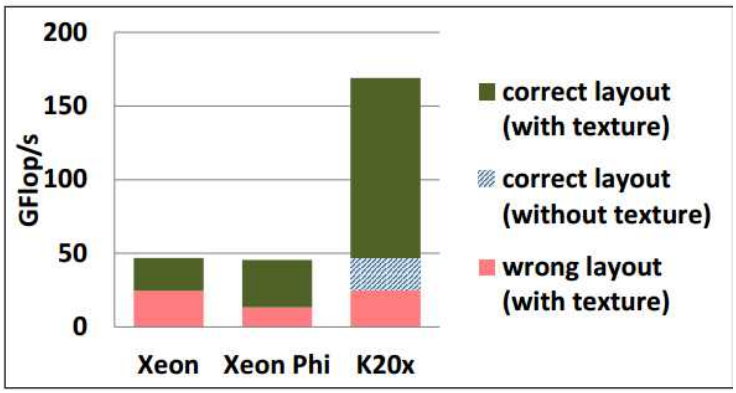


Figure 8.1. 잘못된 메모리 레이아웃으로 인한 대규모의 성능 손실

9. 병렬 실행

- 병렬 수행을 위한 다음 명령어가 제공됨
  * parallel_for: 독립적인 iterations에 대한 loop를 구현
  * parallel_reduce: reduction
  * parallel_scan: prefix scan(1차원 배열에 대한 reduction과 같음)을 구현
- 병렬 루프의 body를 정의하기 위해서 두가지 방법 제공: functors, lambdas
  * functor 메소드를 명시하기 위해 KOKKOS_INLINE_FUNCTION macro를 사용
  * lambda를 사용하기 위해 KOKKOS_LAMBDA macro를 사용
- 실행에 대한 policy또한 명시 가능

9.1 병렬 루프

9.1.1 Functors

병렬 루프의 body를 정의하기 위한 한가지 방법. public operator() instance method를 가진 class나 struct임. method의 인자는 실행하고자 하는 명령어(for, reduce, scan)와 정책(range, team)에 따라 다름. 가장 일반적인 parallel_for를 살펴보면 루프의 인덱스를 위한 정수 인자를 받음.

operator() method는 다음과 같이 상수(const)여야 하며, KOKKOS_INLINE_FUNCTION macro로 표시되어야 함(CUDA에서 실행되면 CUDA용으로 대체됨)

- 예제

```
class MaxPlus {
public:
// Kokkos reduction functors need the value_type typedef.
// This is the type of the result of the reduction.
typedef double value_type;
// Just like with parallel_for functors , you may specify
// an execution_space typedef. If not provided , Kokkos
// will use the default execution space by default.
// Since we're using a functor instead of a lambda ,
// the functor's constructor must do the work of capturing
// the Views needed for the reduction.
MaxPlus (const View<double*>& x) : x_ (x) {}
// This is helpful for determining the right index type,
// especially if you expect to need a 64-bit index.
typedef View<double*>::size_type size_type;
KOKKOS_INLINE_FUNCTION void
operator() (const size_type i, value_type& update) const
```

```
{ // max-plus semiring equivalent of "plus"
if (update < x_(i)) {
update = x_(i);
}
}
// "Join" intermediate results from different threads.
// This should normally implement the same reduction
// operation as operator() above. Note that both input
// arguments MUST be declared volatile.
KOKKOS_INLINE_FUNCTION void
join (volatile value_type& dst,
const volatile value_type& src) const
{ // max-plus semiring equivalent of "plus"
if (dst < src) {
dst = src;
}
}
// Tell each thread how to initialize its reduction result.
KOKKOS_INLINE_FUNCTION void
init (value_type& dst) const
{ // The identity under max is -Inf.
// Kokkos does not come with a portable way to access
// floating -point Inf and NaN. Trilinos does, however;
// see Kokkos::ArithTraits in the Tpetra package.
#ifdef __CUDA_ARCH__
return -CUDART_INF;
#else
return strtod ("-Inf", (char**) NULL);
#endif // __CUDA_ARCH__
}
private:
View<double*> x_;
};
// This example shows how to use the above functor:
const size_t N = ...;
View<double*> x ("x", N);
// ... fill x with some numbers ...
```

```
// Trivial reduction in max-plus semiring is -Inf.
double result = strtod ("-Inf", (char**) NULL);
parallel_reduce (N, MaxPlus (x), result);
```

9.1.2 Lambdas

2011 version of the C++ standard (“C++11”)부터 lambda라는 새로운 construct를 제공함. anonymous 함수로써 functor와 같은 역할을 함. 현재의 상태 자체를 포인터가 아닌 값으로 가져감. 간단한 루프 body에서 편리함. 좀더 복잡한 형태는 Functor를 사용하는 것이 나음.

- 예제

```
const size_t N = ...;
View<double*> x ("x", N);
// ... fill x with some numbers ...
double sum = 0.0;
// KOKKOS_LAMBDA macro includes capture -by-value specifier [=].
parallel_reduce (N, KOKKOS_LAMBDA (const int i, double& update) {
update += x(i); }, sum);
```

10. 다단계 병렬화 (Hierarchical Parallelism)

여기서는 멀티 레벨의 병렬화 방법에 대해서 논함. 이러한 수준(level)은 쓰레드 팀(team), 팀내에서의 쓰레드, vector lane이 있음. 문법은 실행 정책에 따라서 다름.

10.1 개요

단계에서 각 수준은 하드웨어 자원에 따라 결정됨. 높은 수준은 낮은 수준의 자원에 접근이 가능함. 예를 들어 CPU기반 클러스터는 다수의 멀티코어 CPU들로 구성되어 있음. 각 코어는 하나 이상의 하이퍼쓰레드를 지원하고, 각 하이퍼쓰레드는 벡터 연산을 수행가능한데 이는 4개 수준의 병렬화를 나타냄.

- CPU 소켓들은 같은 메모리와 네트워크 자원에 접근을 공유함
- 각 소켓안의 코어들은 하나의 공유하는 last level cache(LLC)를 가지고 있음
- 같은 코어안의 하이퍼쓰레드들은 공유하는 하나의 L1 cache에 접근하고 동일한 실행 유닛들로 명령어를 제출함
- 벡터 유닛들은 다중 데이터들을 공유 명령어로 실행함

GPU기반 클러스터도 다음과 같은 4개 수준 병렬화를 나타냄

- 같은 노드내의 여러 GPU들은 같은 호스트 메모리와 네트워크 자원에 대한 접근을 공유함
- 코어 클러스터(NVIDIA GPU에서의 SM들)는 공유 캐쉬를 가지며 단일 GPU내에서 동일한 높은 대역폭의 메모리에 대한 접근을 가짐
- 같은 코어 클러스터안에 동작하는 쓰레드들은 같은 L1 캐쉬와 스크래취 메모리에 접근
- 쓰레드내에서 WARP나 WAVE FRONT다 불리는 그룹들은 항상 동기화되고 direct register swapping을 통해서 협력이 가능함

10.2 쓰레드 팀(team)

Kokkos에서 다단계 병렬화에서 가장 기본적인 개념으로 서로 동기화가능하고 “scratch pad” 메모리를 공유하는 쓰레드 그룹임

하드웨어 자원에 1-D 범위의 인덱스를 매핑하는 대신에, 2-D 인덱스 영역을 매핑함. 첫 번째 인덱스는 league아고 두 번째 인덱스는 하나의 팀내에 쓰레드 인덱스임. CUDA에서 1-D 블락들의 1-D 그리드를 수행하는것과 같음. 팀 사이즈는 하드웨어 조건에 맞아야 함. CUDA에서는 제한된 숫자의 팀만 active하며, 끝나야지만 새로운 팀이 active함.

10.2.1 정책 인스턴스 생성

Kokkos::TeamPolicy를 이용해서 쓰레드 팀에 정책 인스턴스 생성이 가능. 생성자는 league 사이즈 및 팀 사이즈를 이용해서 만듬. Kokkos::RangePolicy를 이용해서 특정 실행 태그 및 실행 공간을 설정할 수 있음

```
// Using default execution space and launching
// a league with league_size teams with team_size threads each
Kokkos::TeamPolicy <>
policy( league_size , team_size );
// Using a specific execution space to
// run a n_worksets x team_size parallelism
Kokkos::TeamPolicy <ExecutionSpace >
policy( league_size , team_size );
// Using a specific execution space and an execution tag
Kokkos::TeamPolicy <SomeTag , ExecutionSpace >
policy( league_size , team_size );
```

10.2.2 기본 커널

team 정책의 member_type은 병렬 커널내의 팀을 사용하기 위해 필요한 기능을 제공. 이를 통해서 league 랭크 및 사이즈와 같은 쓰레드 구분자와 팀 랭크와 사이즈에 접근 가능함. 팀 barrier, reduction, scan와 같은 동기화 기능도 가능함.

```
using Kokkos::TeamPolicy;
using Kokkos::parallel_for;
typedef TeamPolicy <ExecutionSpace >::member_type member_type;
// Create an instance of the policy
TeamPolicy <ExecutionSpace > policy (league_size , team_size);
// Launch a kernel
parallel_for (policy , KOKKOS_LAMBDA (member_type team_member) {
// Calculate a global thread id

int k = team_member.league_rank () * team_member.team_size () +
team_member.team_rank ();
// Calculate the sum of the global thread ids of this team
int team_sum = team_member.reduce (k);
// Atomicly add the value to a global value
a() += team_sum;
});
```

“TeamPolicy”는 명시적으로 팀에 대해서 그것을 사용하는 커널에 대해서 병렬로 구성하는 것을 말함

10.3 팀 “scratch pad” 메모리

각 팀은 “scratch pad” 메모리를 가지고 있음. 각 팀내의 쓰레드에 접근가능한 메모리 주소의 인스턴스임. “scratch pad”는 알고리즘을 공유된 공간내에서 로딩해서 같은 팀 메머간에 동작하게 함. 라이프타임은 팀과 같음. 팀 수준의 “scratch pad”는 CUDA내의 블락당 공유 메모리와 같은 개념임

scratch pads를 특별한 메모리 영역(execution_space::scratch_memory_space)을 통해 실행 영역을 연결시킴. TeamPolicy 멤버 타입을 통해서 메모리를 할당할 수 있음. 또한 동시에 다수의 메모리 할당도 가능함.

```
template <class ExecutionSpace >
struct functor {
typedef ExecutionSpace execution_space;
typedef execution_space::member_type member_type;
KOKKOS_INLINE_FUNCTION
void operator() (member_type team_member) const {
size_t double_size = 5*team_member.team_size()*sizeof(double);
// Get a shared team allocation on the scratch pad
double* team_shared_a = (double*)
team_member.team_shmem().get_shmem(double_size);
// Get another allocation on the scratch pad
int* team_shared_b = (int*)
team_member.team_shmem().get_shmem(160*sizeof(int));
// ... use the scratch allocations ...
}
// Provide the shared memory capacity.
// This function takes the team_size as an argument ,
// which allows team_size dependent allocations.
size_t team_shmem_size (int team_size) const {
return sizeof(double)*5*team_size +
sizeof(int)*160;
}
};
```

직접 메모리를 할당하는 대신, 사용자가 Views를 scratch 메모리로부터 바로 할당이 가능. View 생성자의 첫 번째 인자로 공유메모리 핸들을 제공하는 것에 의해서 가능함. 또한 Views는 공유메모리 사이즈 요구사항을 반환하는 정적 메모리 함수를 가지고 있음

```
tyepdef Kokkos::DefaultExecutionSpace::scratch_memory_space
ScratchSpace;
// Define a view type in ScratchSpace
typedef Kokkos::View<int*[4],ScratchSpace ,Kokkos::MemoryTraits
<Kokkos::Unmanaged >> // Get the size of the shared memory allocation
size_t shared_size = shared_int_2d::shmem_size(team_size);
Kokkos::parallel_for(Kokkos::TeamPolicy <>(league_size ,team_size),
KOKKOS_LAMBDA ( member_type team_member) {
// Get a view allocated in team shared memory.
// The constructor takes the shared memory handle and the
// runtime dimensions
shared_int_2d A(team_member.team_shmem(), team_member.team_size());
...
});
```

10.4 예제: Sparse Matrix-Vector Multiplication

```
  * Traditional serial compressed row storage (CRS) 알고리즘
for ( int i = 0 ; i < nrow ; ++i )
for ( int j = irow(i) ; j < irow(i+1) ; ++j )
     y(i) += A(j) * x( jcol(j) );
  * Thread team algorithm, using C++11 lambda
typedef TeamPolicy<Space> policy ;
parallel_for( policy( nrow /* #leagues */ ),
 KOKKOS_LAMBDA( policy::member_type const & member ) {
 double result = 0 ;
 const int i = member.league_rank();
 parallel_reduce( TeamThreadRange(member,irow(i),irow(i+1)),
 [&]( int j , double & val ) { val += A(j) * x(jcol(j));},
 result );
 if ( member.team_rank() == 0 ) y(i) = result ;
 }
);
```

11. 성능 측정

11.1 miniMD

MD프로그램인 miniMD를 LJ(Lennard Jones) force calculation를 사용해서 성능 측정함.

아래 그림과 같이 LJ-kernel은 atom에 대해서 루프를 수행하고, cutoff rcut보다는 적은 거리 dij의 atom간에 이웃 pair에 대해서 힘계산을 함. 각 atom의 이웃 j들에 대해서 미리 계산하고 커널에서 사용됨.

```
// Parallel iteration of all atoms in the system
for(i=0;i<natoms;i++) {
  double x_i[3], f_i[3];
  x_i[0..2] = x(i,0..2);
  f_i[0..2] = 0;
  // Iterating the precomputed list of neighbors
  for(jj=0;jj<num_neighbors(i);jj++) {
    int j = neighbors(i,jj);
    double d_ij[3] , d ;
    d_ij[0..2] = x_i[0..2] - x(j,0..2);
    d = norm(d_ij);
    if(d<r_cut) {
      const double sr2 = 1.0 / (d*d);
      const double sr6 = sr2 * sr2 * sr2;
      const double force = 48.0 * sr6 * (sr6 - 0.5) * sr2;
      f_i[0..2] += force * d_ij[0..2];
    }
  }
  f(i,0..2) = f_i[0..2];
}
```

그림. Pseudo code for the thread safe Lennard Jones molecular dynamics kernel (LJ-kernel) in MiniMD.

평균 77개의 이웃들에 대해서 테스트를 수행. 여기서는 1408Flops와 atom i당 311 메모리 접근. neighbors(i,jj)에서 일반적인 메모리 접근 속성, x(j,0-2)에서 random한 메모리 접근 속성을 가짐. 여기서 atom들이 듬성하게 메모리를 접근하더라도, x(j,0-2)에 대해서는 높은 캐쉬 재사용이 가능.

이러한 메모리 레이아웃은 이웃 인덱스인 j(j = neighbors(i,jj))의 사용에 나타남. Xeon와 Xeon Phi에서는 LayoutRight(row majaor order)이어야 하는데 이는 jj에 대해서 inner loop의 다음 iteration 단계에서 값을 읽을 때 캐쉬 라인안에 있게끔 하기 위해서임. Kepler GPU에서는 LayoutLeft(column major order)이어야 하는데, 다른 atom i에서 각 쓰레별로 연속적으로 메모르 접근이 이루어지기 때문임. Kepler GPU에서는 x(j,0-2) random accesses을 위해서 RandomRead 속성을 통해서 texture fetches을 사용하는 것이 중요.

Name	Compton	Shannon
Nodes	32	32
CPU	2x Intel E5-2670 HT-on	2x Intel E5-2670 HT-off
Co-Processor	2x Intel Xeon Phi 57c 1.1GHz	2x K20x
Memory	64 GB	128 GB
Interconnect	QDR IB	QDR IB
OS	RedHat 6.1	RedHat 6.2
Compiler	ICC 13.1.2	GCC 4.4.6 + CUDA 5.5 RC
MPI	IMPI 4.1.1.036	MVAPICH2 1.9

아래 그림은 각 테스트베드에서 GFlops/s에 대해서 성능을 보여주고 있음. 실제로 메모리 레이아웃이 잘못되었을 대비 성능향상으로 동시에 보여줌. 실제로 메모리 레이아웃 때문에, Xeon에서는 1.9x, Xeon Phi에서는 3.4x, Kepler GPU에서는 6.6x차이가 남. 그리고 Kepler GPU에서 texture fetch를 사용하지 않으면 3.6x의 성능 저하가 발생

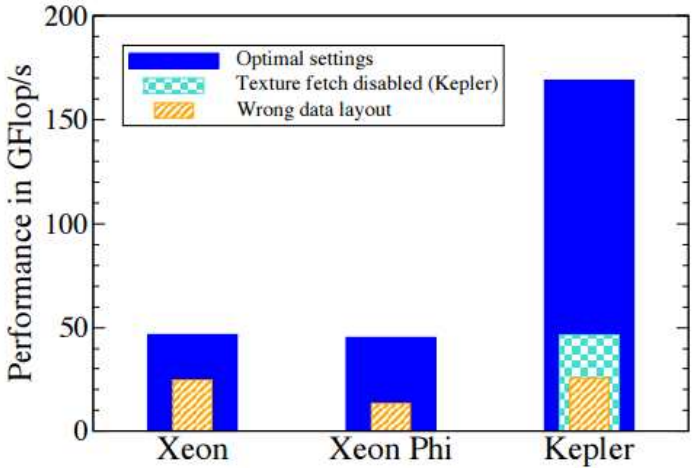


그림. LJ-kernel performance in miniMD on a dual Intel Xeon (Sandy Bridge), a pre-production Intel Xeon Phi co-processrs (57 cores), and a NVIDIA Kepler GPU (K20x) for the miniMD default test problem with 864,000 atoms

11.2 miniFE

하이브리드 병렬(MPI+X) finite element 프로그램

3D 열 diffuction 문제를 푸는 linear system 방정식이고 CG 솔버를 풀기 위해서 200번의 iterations를 수행 (implicit 병렬 finite element의 중요한 특징들을 포함함)

NVIDIA GPU에서 수행하는 프로그램은 linear algebra 함수들이 cuBLAS와 cuSparse함수들도 대치됨.

miniFE는 노드당 8M 요소의 weak-scaling 문제. 노드당 3.3GB메모리가 필요함.

밑의 그림들은 각 테스트베드에서 수행한 결과임. weak-scaling 그래프이며 12번

수행중 가장 성능이 좋은 것을 선택함. Kokkos 수행내용이 native 구현물과 비슷하다고 결론지을 수 있음. 특히 Kepler에서는 CUDA보다 13% 나은 성능을 보여줌. Xeon CPU에서는 느리고, Xeon Phi에서는 10%정도 느림. Xeon와 Kepler의 경우에 32 MPI 랭크에 대해서 95%의 병렬 효율성을 보여줌. Xeon Phi에서의 병렬확장성 문제는 테스트베드에 설치된 MPI 자체의 문제로 보임.

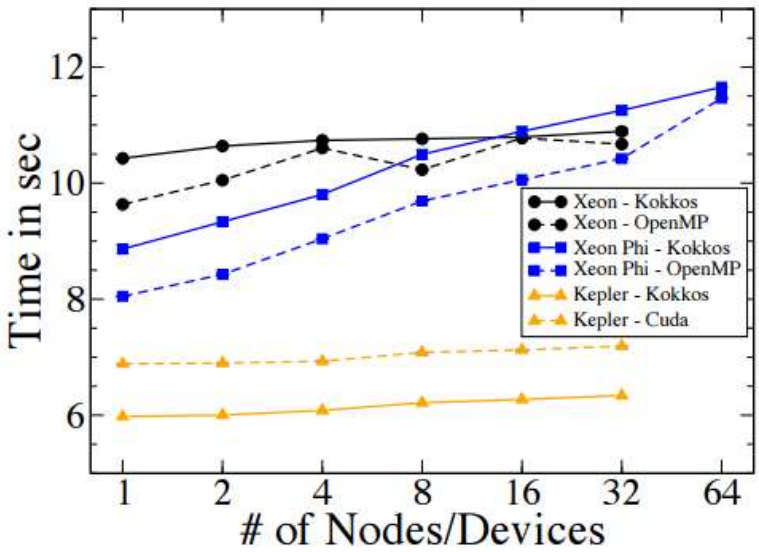


그림. Time for 200 iterations of a CG-solve with miniFE variants on different testbeds. The solid lines represent runs using miniFE-Kokkos, while the dashed lines show results with peer variants

11.3 TeaLeaf 2D

샌디아 연구소에서 개발한 Mantevo benchmark suite의 일부인 mini-app

메모리 바운드 응용이고 다음 세가지 sparse 매트릭스 솔버를 사용함: Conjugate Gradient (CG), Chebyshev, Preconditioned Polynomial CG (PPCG).

여기서는 Lawrence Livermore National Laboratories에서 개발한 Raja 프로그래밍 모델도 같이 비교함

CPU와 Xeon Phi 테스트는 University of Bristol, GPU는 Cray에서 소유하고 있는 Swan supercomputer에서 테스트 수행

인텔 컴파일러(15.0), 16개 코어에서 테스트. 4096*4096 매쉬에서 수행시 CG, PPCG에 대해서 Raja는 OpenMP 대비 15% 성능 저하 발생, Kokkos는 5% 성능 저하 발생

CG 솔버에서 다단계 병렬화(hierarchical parallelism)를 적용시 Xeon Phi에서도 5% 성능향상, GPU에서 10% 성능향상. 하지만 반대로 해당 솔루션 적용시 GPU에서 Chebyshev와 PPCG 솔버는 30% 성능 감소. (여러 단계의 병렬화를 통해서 복잡성이 증가함)

NVIDIA GPU에서 Kokkos는 Chebyshev, PPCG 솔버에 대해서는 직접 CUDA코드 포팅을 하는 것만큼의 성능 향상이 있음.

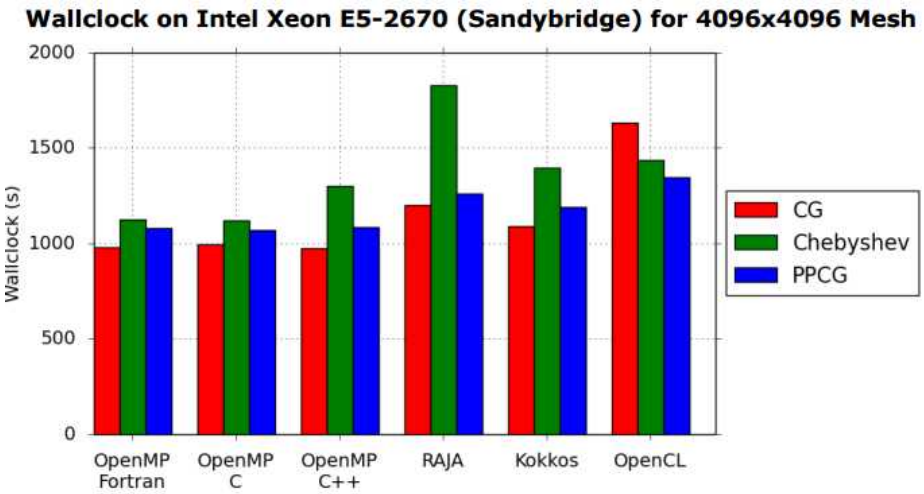


그림. 4096*4096 메쉬 사이즈에서 Xeon 테스트

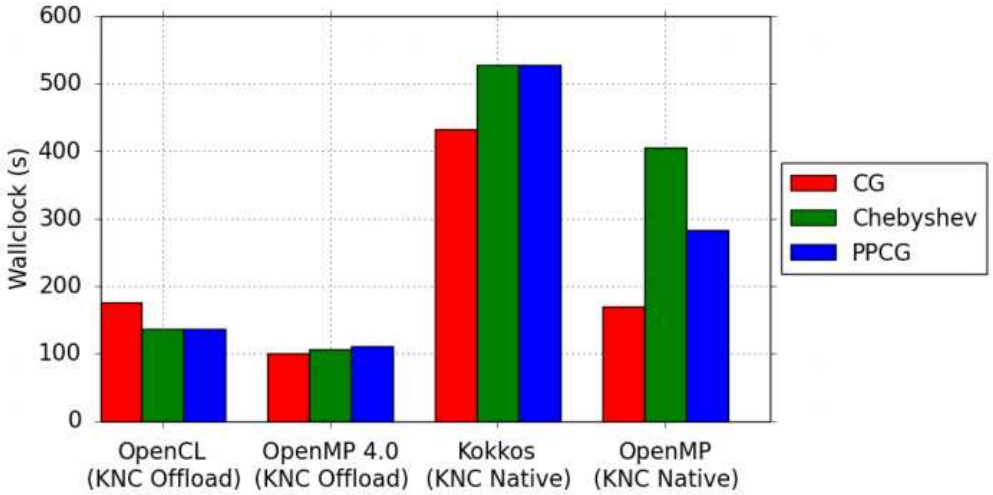


그림. 2048*2048 메쉬 사이즈에서 Xeon Phi 테스트

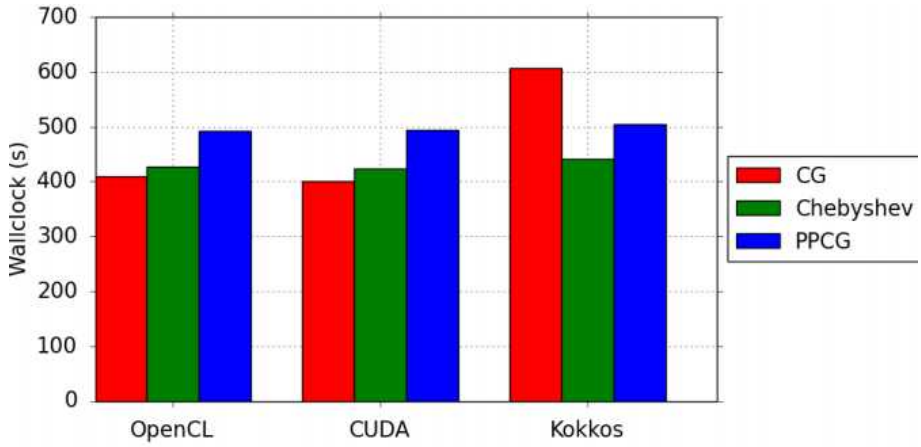


그림. 4096*4096 매쉬에서 NVIDIA K20X 테스트

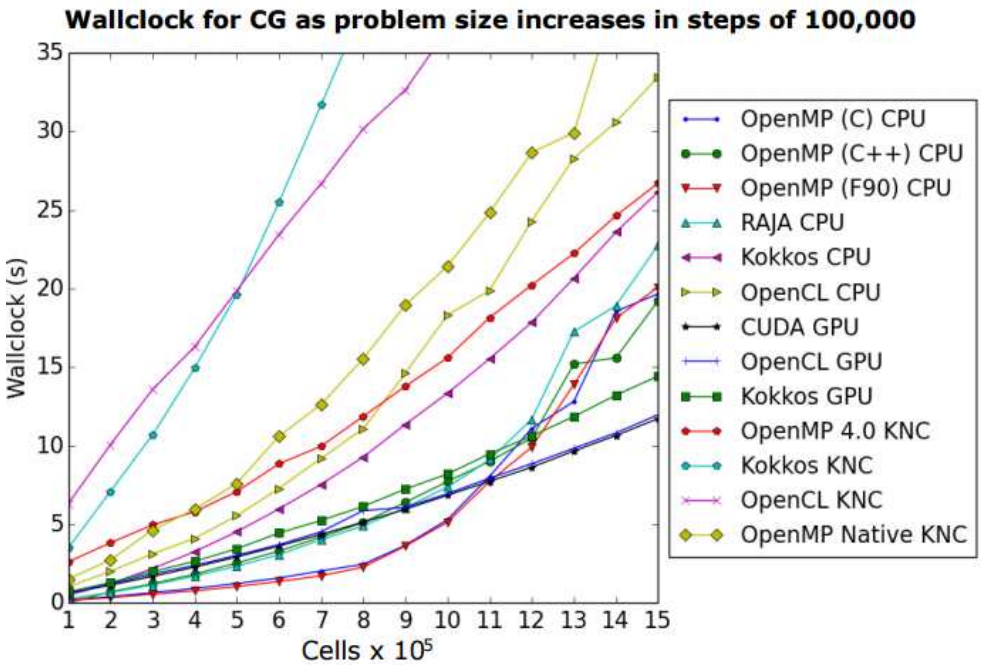


그림. CG에서 100,000 스텝내에서 문자 사이즈를 증가시킴

11.4 LAMMPS

샌디아 연구소에서 개발한 오픈소스 MD 프로그램

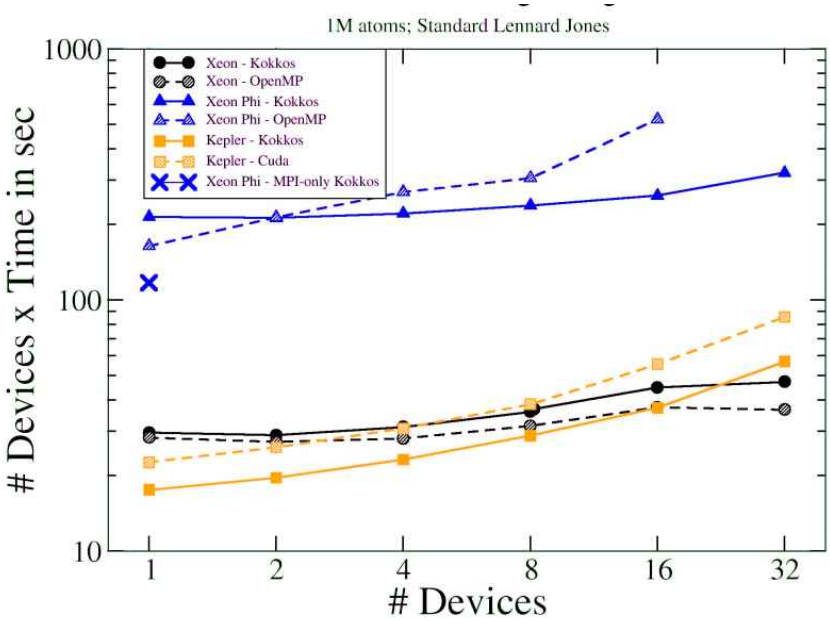


그림. Lammps에서의 Strong 병렬확장성

12. 참고문헌

[1] Kokkos Progamming Guide version 1.0, May, 2015

[2] Kokkos: Enabling performance portability across manycore architectures. XSCALE13, https://www.xsede.org/documents/271087/586927/Edwards-2013-XSCALE13-Kokkos.pdf

[3] Kokkos, Manycore Device Performance Portability for C++ HPC Applications, http://on-demand.gputechconf.com/gtc/2015/presentation/S5166-H-Carter-Edwards.pdf

[4] A Performance Evaluation of Kokkos & RAJA using the TeaLeaf Mini-App, http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/poster_files/post206s2-file2.pdf

[5]. "A next generation LAMMPS: preparing for the many-core future with Kokkos", Christian Trott,
http://lammps.sandia.gov/workshops/Aug13/Trott/LAMMPS-Kokkos3.pdf

[6]. "Towards Performance-Portable Applications through Kokkos: A Case Study with LAMMPS", Christian Trott,
Carter Edwards, and Simon Hammond, http://ondemand.gputechconf.com/supercomputing/2013/presentation/SC3103_Towards-Performance-PortableApplications-Kokkos.pdf