

ВВЕДЕНИЕ В КОККОС

Курникова Анастасия
группа 381803-2, кафедра МОСТ

Нижний Новгород
2021

О Kokkos

- Kokkos реализует модель программирования на C++ для написания высокопроизводительных переносимых приложений, ориентированных на все основные платформы НРС. Он предоставляет абстракции как для параллельного выполнения кода, так и для управления данными
- Kokkos предназначен для сложных архитектур узлов с N-уровневыми иерархиями памяти и несколькими типами ресурсов выполнения...

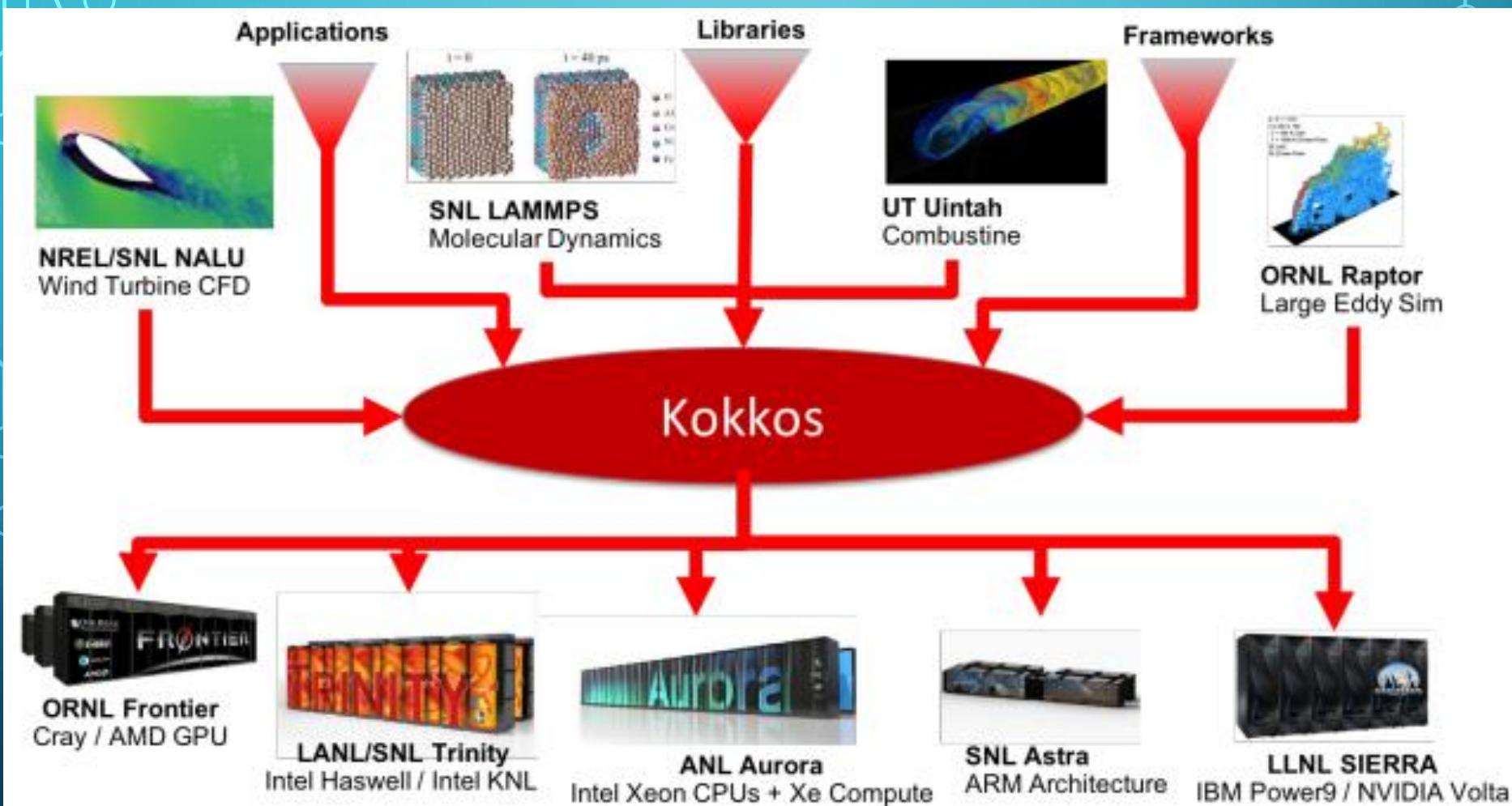


О Kokkos

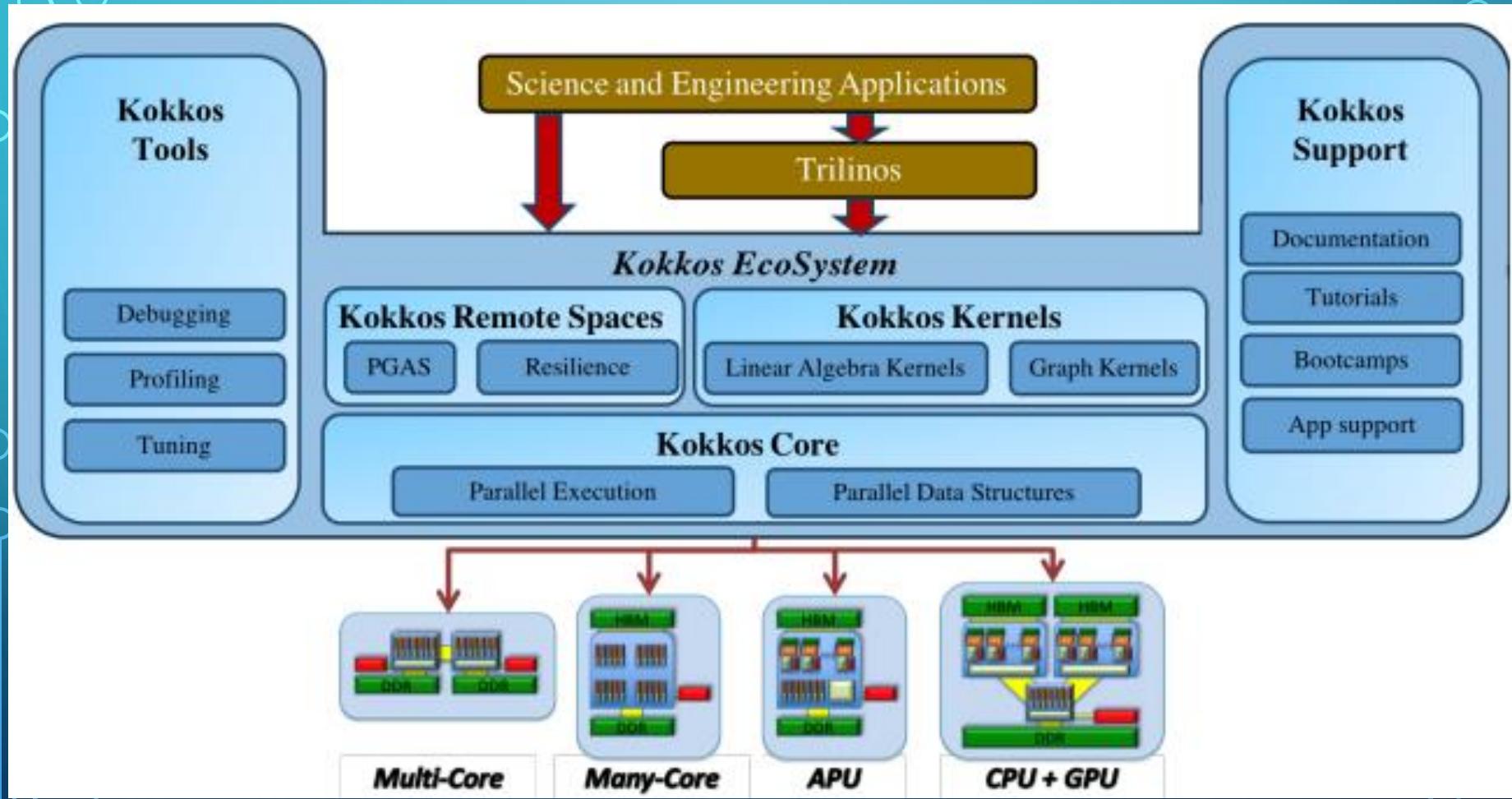
- В настоящее время он может использовать CUDA, HPX, OpenMP и PThreads в качестве моделей программирования серверной части с несколькими другими процессами разработки
- Kokkos C++ Performance Portability Programming EcoSystem предоставляет математические ядра, а также инструменты профилирования и отладки



Перспективы использования



Экосистема Kokkos



Возможности Kokkos

Базовые:

- простые модели параллельных вычислений с одномерными данными
- принятие решений о месте запуска кода и месте расположения данных
- управление шаблонами доступа к данным для повышения производительности
- многомерный параллелизм данных

Расширенные:

- безопасность и масштабируемость потоков и атомарные операции
- иерархические шаблоны для максимизации параллелизма
- программирование на основе конкретной задачи...

Возможности Kokkos

- Kokkos обеспечивает переносимость производительности (**Single Source Performance Portable Codes**)
- Сложность работы не сильно выше, чем при работе с OpenMP
- Расширенные возможности оптимизации производительности **проще в использовании** с Kokkos, чем, например, с CUDA или HIP
- Kokkos предоставляет важные для производительности абстракции данных, **недоступные в других моделях программирования**
- Kokkos **включает инструменты** (профилировщик, отладчик, настройка, математические библиотеки и т.д.), необходимые для разработки приложений в профессиональной среде

Разница между «переносимостью» и «переносимостью производительности»

Реализации могут быть нацелены на определенные архитектуры и могут не быть масштабируемым потоком (например, блокировки ЦП (locks) не масштабируются до 100 000 потоков на GPU)

Цель – написать единую реализацию, которая

- компилируется и запускается на нескольких архитектурах
- получает эффективные шаблоны доступа к памяти через архитектуры
- может использовать особенности конкретной архитектуры, где это возможно

Kokkos обеспечивает переносимость производительности на многоядерные архитектуры

Пример распараллеливания цикла

Serial

```
for (int64_t i = 0; i < N; ++i) {  
    /* loop body */  
}
```

OpenMP

```
#pragma omp parallel for  
for (int64_t i = 0; i < N; ++i) {  
    /* loop body */  
}
```

Kokkos

```
parallel_for(N, [=] (const int64_t i) {  
    /* loop body */  
});
```

- Простое применение Kokkos концептуально не сложнее, чем OpenMP, отличие в расположении частей кода

Терминология

Шаблон – структура вычислений (for, reduction, scan, task-graph, ...)

Политика исполнения – как выполняются вычисления
(статическое планирование, динамическое планирование,
группы/команды потоков, ...)

Тело вычислений - код, выполняющий единицу работы
(например, тело цикла)

- **Шаблон и политика исполнения управляют телом вычислений**

Последовательный разбор

Kokkos распределяет работу по ресурсам исполнения:

- каждая итерация тела вычислений – это единица работы
- индекс итерации идентифицирует конкретную единицу работы
- диапазон итераций определяет общий объем работы

Планирование работы: указывается диапазон итераций и тело вычислений (ядро), и Kokkos принимает решение, как распределить эту работу по исполняющим ресурсам

- Вычислительные тела передаются как функторы или функциональные объекты (так же, как и в C++)

Распределение работы: общее количество рабочих элементов передается шаблону Kokkos, и рабочие элементы назначаются функторам one-by-one...

Последовательный разбор

Kokkos

```
parallel_for(N, [=] (const int64_t i) {  
    /* loop body */  
});
```

- Тело параллельного функтора должно иметь доступ ко всем необходимым данным через элементы данных функтора
- Функторы объемны, поэтому для краткости применяются **лямбды (C++11)**. Они используются как автокомпиляторы функторов

Пример: умножение матрицы на вектор

```
//OpenMP

#pragma omp parallel for
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        res[i * N + j] += matrix[i * N + j] * vector[i];

//Kokkos

Kokkos::parallel_for(N, KOKKOS_LAMBDA(const int i) {
    for (int j = 0; j < N; j++)
        res[i * N + j] += matrix[i * N + j] * vector[i];
```

MDRangePolicy

```
#pragma omp parallel for collapse(3)
for (int64_t i = 0; i < N0; ++i) {
    for (int64_t j = 0; j < N1; ++j) {
        for (int64_t k = 0; k < N2; ++k) {
            /* loop body */
        }
    }
}
```

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
    KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
        /* loop body */
});
```

- До этого были рассмотрены распараллеливания только в одном измерении, оставляющие неиспользованным потенциал общего параллелизма...

MDRangePolicy

- Указывается размерность цикла с помощью **Rank<DIM>**
- Указываются **списки инициализаторов** для начальных и конечных значений
- Функтор или лямбда принимают соответствующее число индексов
- MDRangePolicy может распараллеливать плотно вложенные циклы от 1 до 6 измерений. Правила параллельной редукции не меняются
- Добавляется локальный аргумент потока...

MDRangePolicy

- В приложениях со структурированной сеткой для помощи в кэшировании часто используется стратегия листов.
MDRangePolicy использует стратегию листов для пространства итераций
- Указывается как третий список инициализаторов
- Для графических процессоров страница обрабатывается однопоточным блоком
- Обеспечивается контроль времени компиляции над шаблонами итераций
- По сути схоже с collapse() из OpenMP

Пример: умножение матриц (3 цикла)

```
//OpenMP
```

```
#pragma omp parallel for collapse(3)
for (int i = 0; i < size; i++) {
    for (int k = 0; k < size; k++) {
        for (int j = 0; j < size; j++) {
            res[i * size + j] += a[i * size + k] * b[k * size + j];}}}
```

```
//Kokkos
```

```
Kokkos::parallel_for(Kokkos::MDRangePolicy<Kokkos::Rank<3>>({0,0,0},
{size,size,size}), KOKKOS_LAMBDA(const int i, const int k, const int j) {
    res[i * size + j] += a[i * size + k] * b[k * size + j];}
```

Пример: умножение матриц (блоки)

//OpenMP

```
#pragma omp parallel for collapse(3)
for (int ib = 0; ib < xsize; ib++)
for (int jb = 0; jb < xsize; jb++)
for (int kb = 0; kb < ysize; kb++) {
    int iEnd = min(xsize, ib + 1);
    int jEnd = min(xsize, jb + 1);
    int kEnd = min(ysize, kb + 1);
    for (int i = ib; i < iEnd * sizex; i++)
    for (int k = kb; k < kEnd * sizey; k++)
    for (int j = jb; j < jEnd * sizex; j++)
        res[i * xsize + j] += a[i * xsize + k] * b[k * ysize + j];}
...
}
```

Пример: умножение матриц (блоки)

```
//Kokkos
```

```
Kokkos::parallel_for(Kokkos::MDRangePolicy<Kokkos::Rank<3>>({0,0,0},  
{xsize,xsize,ysize}), KOKKOS_LAMBDA(const int ib, const int jb, const int  
kb) {  
    int iEnd = min(xsize, ib + 1);  
    int jEnd = min(xsize, jb + 1);  
    int kEnd = min(ysize, kb + 1);  
    for (int i = ib; i < iEnd * sizex; i++)  
        for (int k = kb; k < kEnd * sizey; k++)  
            for (int j = jb; j < jEnd * sizex; j++)  
                res[i * xsize + j] += a[i * xsize + k] * b[k * ysize + j];}
```

Результаты экспериментов

- Эксперименты проводились на кластерных узлах со следующими характеристиками:
2xAMD EPYC7742x64,
512GB DDR4 3200MGz,
8xNvidiaA100 40GB,
InfiniBand EDR100Gbit/s
- Время указано в секундах

Threads	OpenMP	Kokkos	MKL
4	12.51	12.11	1.62
8	6.11	5.82	0.89
16	3.16	3.09	0.47
32	1.66	1.59	0.23
n = 100 000			

Таблица 1. Умножение матрицы на вектор

Threads	OpenMP	Kokkos	MKL
4	11.11	10.92	1.31
8	5.18	5.10	0.69
16	3.03	2.97	0.41
32	1.55	1.49	0.21
n = 1 500			

Таблица 2. Умножение матриц (3 цикла)

Threads	OpenMP	Kokkos	MKL
4	1.90	1.86	1.31
8	0.93	0.90	0.69
16	0.52	0.48	0.41
32	0.30	0.27	0.21
n = 1 500			

Таблица 3. Умножение матриц (блоки)

Ссылки

- Kokkos C++ Performance Portability Programming EcoSystem: The Programming Model - Parallel Execution and Memory Abstraction
<https://github.com/kokkos/kokkos>
- Tutorials for the Kokkos C++ Performance Portability Programming EcoSystem <https://github.com/kokkos/kokkos-tutorials>
- Kokkos C++ Performance Portability Programming EcoSystem: Profiling and Debugging Tools <https://github.com/kokkos/kokkos-tools>
- Kokkos C++ Performance Portability Programming EcoSystem: Math Kernels - Provides BLAS, Sparse BLAS and Graph Kernels
<https://github.com/kokkos/kokkos-kernels>