Нижегородский государственный университет им. Н.И. Лобачевского

Параллельные численные методы

Обзор возможностей библиотеки Intel® Array Building Blocks

При поддержке компании Intel

Кустикова В.Д., кафедра математического обеспечения ЭВМ, факультет ВМК

Содержание

- Библиотека Intel® Array Building Blocks
- Компонентный состав Intel® ArBB
  - Среда исполнения потоков (Threading Runtime)
  - Менеджер памяти (Memory Manager)
  - Динамический компилятор (Just-in-Time Compiler)
- Основные типы и структуры данных библиотеки ArBB
- Некоторые функции работы с контейнерами
- Объявление и вызов функций при использовании библиотеки ArBB
- Пример

Библиотека Intel® Array Building Blocks (1)

- Intel® Array Building Blocks – библиотека параллельного программирования в системах с общей памятью, которая устраняет зависимость разрабатываемых приложений от конкретных механизмов низкоуровневого параллелизма и от аппаратных архитектур.

- Intel® ArBB – развитие технологии Intel® Ct (2007).

Библиотека Intel® Array Building Blocks (2)

- ***Модель программирования***. ArBB позволяет выразить параллелизм по данным с последовательной семантикой (простейшие операции над коллекцией, а не над отдельными элементами).
- ***Язык***. Разработчики ArBB предоставляют новые типы данных и операции над ними, а также конструкции управления исполнением.
- ***Абстрактная машина***. Использование средств ArBB освобождает разработчика от написания аппаратно зависимого кода с целью повышения производительности.

Компонентный состав Intel® ArBB (1)

- ***Среда исполнения потоков (Threading Runtime)***.
  - Обеспечивает мелкозернистую модель данных и потоков задач.
  - Обрабатывает комплекс моделей синхронизации.
  - Служба решает низкоуровневые задачи, а потому динамически адаптируется к конкретной архитектуре.
- ***Менеджер памяти (Memory Manager)***.
  - Основные задачи менеджера:
    - выделение памяти,
    - форматирование данных,
    - разделение данных для выполнения параллельных операций.
  - В состав менеджера памяти входит сборщик мусора.

Компонентный состав Intel® ArBB (2)

- ***Динамический компилятор (Just-in-Time Compiler)***.
  - Промежуточное представление вычислений.
  - Выполняет оптимизации.
  - Генерирует исполняемый код.
  - Компиляция состоит из трех фаз:
    - *высокоуровневая* (архитектурно независимые оптимизации кода с целью снижения количества обращений к памяти и излишних вычислений),
    - *низкоуровневая* (оптимизации, зависимые от среды исполнения, в частности, за генерацию кода с использованием архитектурно независимых встроенных средств векторизации (таких, как SSE, например)),
    - *фаза генерации кода уровня векторных инструкций* (Instruction Set Architecture – ISA) под конкретную архитектуру.

Преимущества библиотеки ArBB

- Параллелизм достигается как на уровне ядер за счет создания потоков, так и на уровне данных (SIMD).
- Приложения работают на всех Intel-совместимых архитектурах без необходимости модификации исходных кодов и перекомпиляции проектов.
- По умолчанию ArBB предотвращает такие наиболее распространенные ошибки параллельного программирования, как гонки данных и взаимные блокировки.

Подключение библиотеки ArBB

- Указать путь до заголовочных файлов библиотеки **Configuration Properties→C/C++→General→Additional Include Directories**
- Указать путь до lib-файлов **Configuration Properties→Linker→General→Additional Library Directories**.
- Указать библиотеку **arbb.lib**, в которой должен собираться проект, в **Configuration Properties→Linker→Input→Additional Dependencies**.
- Определить константу препроцессора **NOMINMAX** в Debug и Release режимах компиляции (**Configuration Properties→C/C++→Preprocessor→Preprocessor Definition**).

Базовые синтаксические конструкции ArBB

- Группы синтаксических конструкций:
  - Скалярные типы (аналоги примитивных типов C++).
  - Векторные типы (параллельные коллекции скалярных данных).
  - Скалярные и векторные операторы.
  - Конструкции управления исполнением (условия, циклы и др.).

Обзор возможностей библиотеки Intel® Array Building Blocks

СКАЛЯРНЫЕ ТИПЫ

Скалярные типы

Тип в ArBB	Тип в C/C++
`f32` `f64`	`float` `double`

`i8`
`i16`
`i32`
`i64`
	`char`
	`short`
	`int`

`u8`
`u16`
`u32`
`u64`
	`unsigned char`
	`unsigned short`
	`unsigned int`

`boolean`	`bool`
`usize, isize`	`size_t`

Особенности работы со скалярными типами

- Работает явное приведение типов для переменных скалярных типов ArBB (от меньшего размера к большему, как в C/C++).

- Недопустимо явное приведение скалярного ArBB типа к типу C/C++:

```
f32 fp_scalar;
float f = (float)fp_scalar; // не поддерживается
```

- Работает копирование значения переменных встроенных типов C/C++ в скалярные типы библиотеки ArBB:

```
f32 fp_scalar2(f);  // создание копии
f32 fp_scalar3 = f; // создание копии
```

Обзор возможностей библиотеки Intel® Array Building Blocks

ВЕКТОРНЫЕ ТИПЫ

Векторные типы (контейнеры)

- ***Регулярные контейнеры***
  - Синтаксис: **dense<T>**, где **T** – скалярный тип данных.
  - Предназначены для хранения векторов и многомерных матриц.
  - По своей сути очень похожи на шаблоны библиотеки STL.
- ***Нерегулярные контейнеры***
  - Синтаксис: **nested<T>**, где **T** – скалярный тип данных.
  - Такие структуры необходимы для представления, например, набора векторов, содержащих разное количество элементов.

Создание регулярных контейнеров

```
// создание пустого одномерного массива
dense<f32> a;

// создание пустого двумерного массива
dense<f32, 2> a;

// одномерный массив из 10 элементов
dense<f32> b(10);

// новый одномерный массив из 10 элементов
dense<f32> c(b);
```

Операции работы с регулярными контейнерами (1)

- Доступ к элементу контейнера:
  - Операция []. Так же, как к элементам массива.
- Получение числа элементов регулярного контейнера:
  - Метод length() объекта контейнера.
  - Пример:

```
dense<i32> a(3); // создание контейнера из
                 // трех элементов
int n = value(a.length());// получение значения и
                          // числа элементов
```

- Различные векторно-скалярные операции (сложение, умножение, взятие синуса и т.п.).

Операции работы с регулярными контейнерами (2)

- Связывание области памяти с конкретным контейнером:
  - Функция bind().
  - Пример:

```
int n = 5;
double *a = new double[n];
dense<f64> b;
for (int i = 0; i < n; i++) a[i] = double(i + 1);
bind(b, a, n);
for (int i = 0; i < value(b.length()); i++) {
    printf("%f\n", value(b[i]));
}
delete []a;
```

Обзор возможностей библиотеки Intel® Array Building Blocks

СКАЛЯРНЫЕ И ВЕКТОРНЫЕ ОПЕРАТОРЫ

Некоторые функции работы с контейнерами

- Функция **shift** обеспечивает сдвиг элементов контейнера вправо или влево на заданное количество элементов.
- Функция **gather** изымает из исходного контейнера элементы с некоторым набором индексов.
- Функция **repeat** дублирует элементы исходного контейнера заданное количество раз.
- Функция **reshape_nested_offsets** преобразует регулярный контейнер в нерегулярный.

Оператор shift

```
void TestShiftOperator()
{
    int n = 5;
    size_t *colptr = new size_t[n + 1];
    dense<usize> colptr_arbb;
    colptr[0] = 0; colptr[1] = 1; colptr[2] = 4;
    colptr[3] = 6; colptr[4] = 7; colptr[5] = 9;
    bind(colptr_arbb, colptr, n + 1);
    // -1 – сдвиг влево на 1 элемент (1 – сдвиг вправо)
    // 0 - заполнитель
    dense<i32> shift_colptr = shift(colptr_arbb,-1,0);
    for (int i=0; i<value(shift_colptr.length()); i++)
        printf("%i\n", value(shift_colptr[i]));
    delete []colptr;
}
```

Оператор gather (1)

```
void TestGatherOperator()
{
    int n = 3;
    dense<i32> res_arbb, vec_arbb, rows_arbb;
    int *vec = new int[n];
    // индексы элементов массива vec
    int rows[] = {0, 2, 0, 1, 2};
    // vec = {1,2,3}
    for (int i = 0; i < n; i++)
    {
        vec[i] = i + 1;
    }
    // продолжение на следующем слайде
```

Оператор gather (2)

```
    // связывание массивов с контейнером
    bind(vec_arbb, vec, n); // [1 2 3]
    bind(rows_arbb, rows, 5); // [0 2 0 1 2]
    // формирование нового контейнера
    // res_arbb = [1 3 1 2 3]
    res_arbb = gather(vec_arbb, rows_arbb);
    for (int i = 0; i < value(res_arbb.length()); i++)
    {
        printf("%i\n", value(res_arbb[i]));
    }
    delete []vec;
}
```

Оператор repeat (1)

```
void TestRepeatOperator()
{
    int n = 5;
    int *vval = new int[n + 1];
    size_t *colptr = new size_t[n + 1];
    size_t *diff = new size_t[n + 1];
    dense<i32> vval_arbb;
    dense<usize> colptr_arbb;
    // заполнение массива vval = {1,2,3,4,5,0}
    for (int i = 0; i < n; i++) vval[i] = i + 1;
    vval[n] = 0;

    // продолжение на следующем слайде
```

Оператор repeat (2)

```
// заполнение массива colptr = {0,1,4,6,7,9}
colptr[0] = 0; colptr[1] = 1; colptr[2] = 4;
colptr[3] = 6; colptr[4] = 7; colptr[5] = 9;
// связывание массивов с контейнерами
bind(vval_arbb, vval, n + 1);
bind(colptr_arbb, colptr, n + 1);
// сдвиг элементов colptr вправо на 1
// результат: shift_ptr = {0,0,1,4,6,7}
dense<usize> shift_colptr = shift(colptr_arbb, 1, 0);
// поэлементное вычитание
// diff_arbb = {0,1,3,2,1,2}
dense<usize> diff_arbb = colptr_arbb - shift_colptr;

// продолжение на следующем слайде
```

Оператор repeat (3)

```
    // сдвиг влево на 1 элемент (diff_ptr = {1,3,2,1,2,0})
     diff_arbb = shift(diff_arbb, -1, 0);
     // дублирование элементов контейнера vval_arbb
     // результат: vvalnew_arbb = {1,2,2,2,3,3,4,5,5}
     dense<i32> vvalnew_arbb=repeat(vval_arbb,diff_arbb);
     for (int i=0; i < value(vvalnew_arbb.length()); i++)
     {
         printf("%i\n", value(vvalnew_arbb[i]));
     }
     delete []vval;
     delete []colptr;
     delete []diff;
}
```

Оператор reshape_nested_offsets (1)

```
void TestReshapeNestedOffsets()
{
    int n = 9;
    int *a = new int[n];
    size_t *rowidx = new size_t[n - 4];
    // заполнение массивов
    // a = {2,2,10,14,9,27,16,30,30}
    // rowidx = {0,1,4,7,8}
    a[0] = 2; rowidx[0] = 0;
    a[1] = 2; rowidx[1] = 1;
    a[2] = 10; rowidx[2] = 4;
    a[3] = 14; rowidx[3] = 7;
    a[4] = 9;  rowidx[4] = 8; //последний индекс в ‘a’
    // продолжение на следующем слайде
```

Оператор reshape_nested_offsets (2)

```
a[5] = 27;
a[6] = 16;
a[7] = 30;
a[8] = 30;
dense<i32> a_arbb;
dense<usize> rowidx_arbb;
// связывание массивов с контейнерами
bind(a_arbb, a, n);
bind(rowidx_arbb, rowidx, n - 4);
// результат разбиения: [2][2,10,14][9,27,16][30][30]
// каждый вектор с индексом i в нерегулярном
// контейнере

// продолжение на следующем слайде
```

Оператор reshape_nested_offsets (3)

```
// содержит (rowidx[i+1] - rowidx[i]) элементов,
// начиная с элемента с индексом rowidx[i]
nested<i32> prod = reshape_nested_offsets(a_arbb,
                                          rowidx_arbb);
// коллективная операция суммирования элементов
// векторов в нерегулярном контейнере
// результат: sum = {2,26,52,30,30}
dense<i32> sum = add_reduce(prod);
for (int i = 0; i < value(sum.length()); i++)
{
    printf("%i\n", value(sum[i]));
}
delete []a; delete []rowidx;
}
```

Обзор возможностей библиотеки Intel® Array Building Blocks

КОНСТРУКЦИИ УПРАВЛЕНИЯ ИСПОЛНЕНИЕМ

Управляющие конструкции ArBB

- Управляющие конструкции библиотеки ArBB:
  - циклы (_for, _while),
  - условия (_if),
  - вызов функций (call, map).

Цикл _for

- Синтаксис:

```
_for (start, end, step)
{
  // вычисления
} _end_for;
```

- Пример:

```
_for (usize i = 0, i < n, i++)
{
  // вычисления
} _end_for;
```

Цикл _while

- Синтаксис:

```
_while (условие)
{
  // вычисления
} _end_while;
```

- Пример:

```
_while (i < N)
{
  // вычисления
} _end_while;
```

Особенности циклических конструкций

- Все циклические конструкции **не обеспечивают параллелизма**.
- Циклические конструкции ArBB используются, чтобы выразить последовательную логику управления ArBB-типами.

- Замечание: в ArBB реализованы собственные операторы принудительного перехода

  _break (выход из цикла),

  _continue (переход к следующей итерации),

  _return (выход из функции).

Условия

```
_if (условие)
{…}
_end_if;
```

```
_if  (условие)
{…}
_else
{…}
_end_if;
```

```
_if (условие)
{…}
_else_if
{…}
_else
{…}
_end_if;
```

- Используются при написании кода, исполнение которого зависит от значения скаляра.

Объявление и вызов функций

- ***Вызов функций call()***
  - Синтаксис:

```
call(function_ptr)(arg1, arg2, …, argn)
```

  - Характерные особенности:
    - Управление передается от вызывающей и к вызываемой функции (равносилен вызову обычной функции в C).
- ***Вызов функций map()***
  - Синтаксис:

```
map(function_ptr)(arg1, arg2, …, argn)
```

  - Характерные особенности:
    - Конвертирует скалярную функцию в параллельную операцию на элементами контейнера.

Механизм исполнения call()

- При первом вызове функций с использованием call во время исполнения программы ArBB формирует так называемое ***соглашение*** – представление закодированных вычислений.
- Соглашения компилируются JIT компилятором в машинный язык векторных команд и далее могут выполняться многократно с различным набором параметров.
- После формирования соглашения осуществляется компиляция, кэширование и непосредственный вызов с переданными параметрами.

Пример использования call в задаче умножения плотной матрицы на вектор (1)

```
void arbbMultiplication(const dense<f64> &A,
                        const dense<f64> &x,
                        dense<f64> &b)
{
    usize n = x.length();
    dense<f64> row, multi;
    _for (usize i = 0, i < n, i++)
    {
        row = section(A, i * n, n);
        multi = row * x;
        b[i] = add_reduce(multi);
    } _end_for;
}
```

Пример использования call() в задаче умножения плотной матрицы на вектор (2)

```
void Multiplication(double *A, double *x,
                    int n, double *b)
{
    dense<f64> A_arbb, x_arbb, b_arbb;
    bind(A_arbb, A, n * n); bind(x_arbb, x, n);
    bind(b_arbb, b, n);
    try {
        call(arbbMultiplication)(A_arbb, x_arbb, b_arbb);
    }
    catch (const std::exception& e) {
        printf("%s\n", e.what());
    }
    catch (...) {
        printf("Unknown error\n");
    }
}
```

Механизм исполнения map()

- Ядро вызова – это операция со скалярными аргументами.
- Разбиение пространства индексов регулярного контейнера.
- Параллельное исполнение функции ядра над соответствующими элементами регулярных контейнеров.

Пример использования map в задаче умножения плотной матрицы на вектор

```
void arbbMulti(const f64 a, const f64 b, f64& c) {
    c = a * b;
}
void arbbMultiplication(const dense<f64> &A,
                        const dense<f64> &x, dense<f64> &b)  {
    usize n = x.length();
    dense<f64> row, multi;
    _for (usize i = 0, i < n, i++) {
        row = section(A, i * n, n);
        map(arbbMulti)(row, x, multi);
        b[i] = add_reduce(multi);
    } _end_for;
}
```

Заключение

- Разработчики Intel® ArBB обеспечивают параллелизм данных уровня SIMD.
- Сохранение последовательной семантики.
- Отсутствие зависимости от низкоуровневых механизмов векторизации.
- Переносимость приложений.

Вопросы

- ???

Авторский коллектив

- Кустикова Валентина Дмитриевна,
  ассистент кафедры
  Математического обеспечения ЭВМ факультета ВМК ННГУ.
  valentina.kustikova@gmail.com